Jan Jürjens

Secure Systems Development with UML

12th July 2004

Springer
Berlin Heidelberg New York
Hong Kong London
Milan Paris Tokyo
to Li
Foreword

Those who spend their professional life developing and deploying – or observing – new information technology systems may believe that the security issues raised in this process are a direct consequence of developments in technology. This is not quite so. New technologies create opportunities for new applications, sometimes not even foreseen when the technology was first fielded. Pertinent examples are e-mail, the first major application of the Internet and its precursors (rather than the “serious” scientific collaborations originally intended), or the success of SMS (Short Messaging Service), which was initially perceived as a minor addition to the services offered by second-generation mobile telecommunications systems.

At the same time, new technologies and the applications they facilitating also open up new opportunities for creating mischief of various hues, which in turn trigger a demand for “security technologies” that should prevent – or at least reduce – unwelcome use of those new applications. To stay with the example of e-mail, spam has today become a major nuisance, to the extent that some see success in the battle against spam as essential for e-mail to survive as a useful service.

For software systems, the release of the Internet for commercial use in the early 1990s was an incisive event, whose implications have still not been fully digested. It first led to the development of distributed applications in closed environments that use the Internet as an open communications network. In this domain, security requirements are mainly, but not exclusively, related to communications security. Virtual Private Networks may serve as an example. However, today we are also dealing with open environments without central points of control or authority, which require novel ways of approaching security. Indeed, the fact that in different applications fundamentally different security requirements have to be met is one of the reasons why the design of security protocols is difficult and error prone.

All of which brings us to security. Security professionals like to state that security must not be treated as an add-on feature, and that systems cannot be made secure by adding some so-called security features in the later design
stages. To the extent that security requirements depend on the application, it is then the task of the application designer to include those requirements in the specification early on, and the task of the design process to make sure that adequate protection mechanisms are implemented. There is thus an obvious demand for design methodologies that help in specifying security requirements, and in making sure that suitable security mechanisms are implemented.

To add a second general statement on security, there are hardly ever correct answers to security challenges, only answers that are better or worse than others. When proposing design methodologies for security, we are walking a tightrope if security-unaware application writers are asked to decide on matters of security. In application areas where security requirements are well understood and met by a fairly standardized set of security mechanisms, we may justifiably hope that such methodologies can be put to good use. However, particularly in novel kinds of applications, we will not always know the security requirements in advance, and prudent engineering practices may change over time. As an example, robustness against denial-of-service attacks and identity protection (plausible deniability) have become new aspects in the design of key establishment protocols in recent years, as witnessed in the discussions about a successor to the Internet Key Exchange protocol (IKE).

This book makes valuable contributions towards the development of well-founded design methodologies for security engineering. By building on a widely adopted specification language like UML, consideration of security aspects fits into the design process in a natural way. The proposed methodology has solid theoretical foundations so that it is possible to verify in a precise setting whether a design has its desired security properties. The definition of these foundations would in itself constitute a substantial piece of work, but the book goes further. For any design methodology striving to have practical impact, the proverbial saying that “the proof of the pudding is in the eating” applies. The book does not fall short on this count either, covering several case studies the methodology has been applied to, and presenting the tools that have been developed to support this approach.

To say that this book is a first step in a promising direction would thus seriously underrate what has already been achieved. The reader may treat this book as an exemplary demonstration of how formal methods for the design of secure systems could be made accessible to application software designers in general, and wait with interest for further developments as the methodology matures.

Hamburg,  

Dieter Gollmann 
May 2004
Preface

Attacks against computer networks, which modern society and modern economies rely on for communication, finance, energy distribution, and transportation, can threaten the economical and physical well-being of people and organizations. Due to the increasing interconnection of systems, such attacks can be waged anonymously and from a safe distance. Thus networked computers need to be secure.

The high-quality development of security-critical systems is difficult. Many systems are developed, deployed, and used that do not satisfy their criticality requirements, sometimes giving rise to spectacular attacks.

Part of the difficulty of secure systems development is that "the goal of correctness is often in conflict with that of low development cost". Where thorough methods of system design pose high cost through personnel training and use, they are all too often avoided.

The Unified Modeling Language (UML) offers an unprecedented opportunity for high-quality and cost- and time-efficient secure systems development:

- As the de facto standard in industrial modeling, a large number of developers are trained in UML.
- Compared to previous notations with a user community of comparable size, UML is relatively precisely defined.
- A variety of tools exist that provide the basic functionality required to use UML (such as the drawing of UML diagrams).

To exploit this opportunity, however, some challenges remain which include the following:

- Adapting UML to the application domain of security-critical systems.
- Advancing the correct use of UML in this application domain.
- Dealing with conflicts between flexibility and unambiguity in the meaning of UML models.
- Developing advanced tool-support for secure systems development with UML (such as automatic analysis of UML specifications with respect to security requirements).
This book aims to contribute to overcoming these challenges.

We present the extension UMLsec of UML for secure systems development, using the standard UML extension mechanisms. One can thus evaluate UML specifications for vulnerabilities using a formal semantics of a (restricted and simplified) fragment of UML version 1.5 which we also provide. One may also encapsulate established rules of prudent security engineering and make them available to developers. The possibility of a high degree of abstraction, and diagrams offering different views of a system, allow the modeling of security-critical components in the system context. Our method thus aims to be useful both to security experts and to developers who may not be experts in security.

We demonstrate the adequacy of UMLsec by using it in several case studies. For example, we develop a secure channel specification and uncover flaws in a published variant of the Internet protocol TLS and in the Common Electronic Purse Specifications, propose corrections, and verify them. We use UMLsec in the context of banking applications and of Java security. We present the concepts and technologies needed for constructing tool support for analyzing UML models for sophisticated requirements, such as the constraints included in UMLsec specifications. The tool support is based on an XML dialect called XMI which allows interchange of UML models.

This book is based on a PhD thesis, several invited talks, summer school lectures, a series of more than 20 tutorials at international conferences (and feedback from many of their participants), and about 30 articles in international journals and at conferences by the author, and on feedback from projects with industrial partners (including a major German bank and a major German telecommunications company), and on discussions at international workshops organized on this topic, as well as the supervision of about 15 Master’s and Bachelor theses on related topics and five University courses given at the University of Oxford and TU München which included part of the topics covered in this book. Additional material is given on the website [Web] associated with this book which is continuously being updated. It includes the following material:

- Slides and audio recordings from the tutorials and courses mentioned above.
- Other learning and teaching materials, including exercises and answers.
- A web interface for a tool which analyzes UMLsec models written using an industrial UML modeling tool (which one can upload over the Internet) for security requirements.

Note that although the UML extension proposed in this book aims to also offer assistance to developers who are not security experts (for example, by enabling them to use security mechanisms in a secure way), parts of the book are concerned with advanced applications (such as cryptoprotocol analysis) for which background knowledge in security would be helpful.

I would like to express my sincerest gratitude to all of the people involved in some way or another with the above undertakings, and with the
compilation of this book in particular. These include my advisor for the PhD thesis on which this book is based, Samson Abramsky (for his insights and advice, encouragement, and patience), as well as Manfred Broy, head of my subsequent affiliation; the Software & Systems Engineering group at TU Munich (for interesting discussions, for sharing his profound experience in formal methods and software engineering, and for providing a very stimulating working environment), various people who provided encouragement to pursue the idea to write a book based on the thesis, my coauthors and colleagues (for fruitful collaborations, and inspiring discussions on security or UML), my students (for helpful collaborations on tool support and for questions on dubious parts of the material), several people reading various portions of the draft and offering useful comments and advice, as well as the 'many' reviewers of the papers on which this book is based, altogether several hundred participants of my tutorials, as well as the audiences of my other talks related to security or UML, many of whom contributed comments and questions. I would also like to thank the members of different organizations in which I am involved (including the working group for Formal Methods and Software Engineering for Safety and Security (FoMSESS) within the German Society for Informatics (GI), the Division of Safety and Security within the GI, the Bavarian Competence Center for Safety and Security, the working group on e-Security of the Bavarian regional government, and the IFIP Working Group 1.7 "Theoretical Foundations of Security Analysis and Design") for interesting discussions about security, the technical support at TU Munich (including the system administrators and the student assistants), and last but not least my editor at Springer-Verlag, Ralf Gerstner, for his interest in the book project and his patience. Apologizing to those who currently manage to escape my mind, I would like to name in particular the following: Martin Abadi, Eweny Alter, Axelle Apvrille, David Basin, Egon Börger, Peter Braun, Ruth Breu, Pierpaolo Degano, Martin Deubler, Eduardo B. Fernandez, Robert France, Onno Garms, Geri Georg, Dieter Gollmann, Roberto Gorrieri, Johannes Grünbauer, Joshua Guttmann, Sebastian Höhn, Hella Holmann, Siv Hilde Houmb, Anna Ioshepe, Gergely Kokavec, Thomas Kuhn, Britta Liebscher, Volkmar Lotz, Gavin Lowe, Frank Marschall, Shasha Meng, Carlo Montanaro, Gerhard Popp, Jan Romberg, Bernhard Rupme, Robert Sandner, Robert Schmidt, Marilynn Schwaiger, Stephan Schwarzmüller, Brani Selic, Pasha Shabalina, Shunwei Shen, Oscar Slotosch, Perdita Stevens, Martin Strecker, Guido Wimmel, and Bo Zhang. Finally, I particularly thank my parents and my brother for their continued 'moral support.

This work was financially supported in part by the Studienstiftung des deutschen Volkes, Laboratory for Foundations of Computer Science (University of Edinburgh), Bell Laboratories (Lucent Technologies), Computing Laboratory (University of Oxford), Software & Systems Engineering (TU Munich), the FairPay project (German Ministry of Economy), the HypoVereinsbank (Munich), and the Verisoft project (German Ministry of Education and Research).
Comments or questions regarding the content of this book are always welcome, and can be made through the book’s website [Web].

München,  
October 2003  

Jan Jürjens

Two roads diverged in a wood, and  
I took the one less traveled by,  
And that has made all the difference.  

Robert Frost, The Road Not Taken
## Contents

### Part I Prologue

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>3</td>
</tr>
<tr>
<td>1.1</td>
<td>Overview</td>
<td>9</td>
</tr>
<tr>
<td>1.2</td>
<td>Outline</td>
<td>14</td>
</tr>
<tr>
<td>1.3</td>
<td>How to Use this Book</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td>Walk-through: Using UML for Security</td>
<td>17</td>
</tr>
<tr>
<td>2.1</td>
<td>Security Requirements Capture with Use Case Diagrams</td>
<td>18</td>
</tr>
<tr>
<td>2.2</td>
<td>Secure Business Processes with Activity Diagrams</td>
<td>18</td>
</tr>
<tr>
<td>2.3</td>
<td>Physical Security Using Deployment Diagrams</td>
<td>19</td>
</tr>
<tr>
<td>2.4</td>
<td>Security-Critical Interaction with Sequence Diagrams</td>
<td>20</td>
</tr>
<tr>
<td>2.5</td>
<td>Secure States Using Statechart Diagrams</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>Background</td>
<td>23</td>
</tr>
<tr>
<td>3.1</td>
<td>Security Engineering</td>
<td>23</td>
</tr>
<tr>
<td>3.2</td>
<td>Unified Modeling Language</td>
<td>25</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Use Case Diagrams</td>
<td>26</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Class Diagrams</td>
<td>27</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Statechart Diagrams</td>
<td>28</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Sequence Diagrams</td>
<td>29</td>
</tr>
<tr>
<td>3.2.5</td>
<td>Activity Diagrams</td>
<td>30</td>
</tr>
<tr>
<td>3.2.6</td>
<td>Deployment Diagrams</td>
<td>31</td>
</tr>
<tr>
<td>3.2.7</td>
<td>Subsystems</td>
<td>32</td>
</tr>
<tr>
<td>3.2.8</td>
<td>UML Extension Mechanisms</td>
<td>32</td>
</tr>
<tr>
<td>3.3</td>
<td>Analyzing UML Models</td>
<td>34</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Notation</td>
<td>34</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Outline of Formal Semantics</td>
<td>35</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Modeling Cryptography</td>
<td>36</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Security Analysis of UML Diagrams</td>
<td>37</td>
</tr>
<tr>
<td>3.3.5</td>
<td>Important Security Properties</td>
<td>40</td>
</tr>
</tbody>
</table>
Part II Developing Secure Systems

4 Secure Systems Development with UML ......................... 47
  4.1 UMLsec Profile .................................................. 47
    4.1.1 Requirements on a UML Extension for Development of Security-Critical Systems .................. 47
    4.1.2 The Extension ................................................. 48
    4.1.3 Addressing the Requirements ............................... 63
  4.2 Design Principles for Secure Systems ......................... 66
  4.3 Applying Security Patterns .................................... 68
  4.4 Notes .......................................................... 69
  4.5 Discussion ..................................................... 71

5 Applications .......................................................... 73
  5.1 Secure Channels ..................................................... 73
  5.2 A Variant of the Internet Protocol TLS ....................... 78
  5.3 Common Electronic Purse Specifications ...................... 86
    5.3.1 Purchase Transaction ...................................... 87
    5.3.2 Load Transaction .......................................... 98
  5.4 Developing Secure Java Programs .............................. 114
    5.4.1 Access Control in Java ................................... 114
    5.4.2 Design Process ............................................ 116
    5.4.3 Example: Financial Application ......................... 118
  5.5 Further Applications ............................................ 121
    5.5.1 Modeling and Verification of a Bank Application ...... 121
  5.6 Notes .......................................................... 123
  5.7 Discussion ..................................................... 123

Part III Tool Support

6 Tool support for UMLsec ................................................. 127
  6.1 Extending UML CASE Tools with Analysis Tools .............. 127
    6.1.1 Meta-Object Facility (MOF) ............................... 128
    6.1.2 XML-Based Data-Binding with MDR ....................... 130
  6.2 Automated Tools for UMLsec .................................. 131
    6.2.1 Tool Functionality ....................................... 132
    6.2.2 Implementation Details ................................... 134
  6.3 UMLsec Versus Runtime Data: SAP R/3 Permissions ........... 135
    6.3.1 Automated Analysis of Security Rules ................... 137
    6.3.2 Instance Data ............................................. 140
    6.3.3 Evaluating Rules .......................................... 144
  6.4 Linking Models to Code ......................................... 148
    6.4.1 Test-Sequence Generation ................................ 149
6.4.2 Code Generation and Code Analysis ...............152
6.5 Notes ...........................................153
6.6 Discussion .....................................154

7 A Formal Foundation ..................................157
7.1 UML Machines ..................................157
7.2 UML Machine Systems .........................165
7.3 Refinement ..................................168
7.4 Rely-Guarantee Specifications .................171
7.5 Reasoning About Security Properties ..........172
7.5.1 Secrecy ..................................175
7.5.2 Integrity ..................................178
7.5.3 Authenticity ................................179
7.5.4 Freshness ..................................180
7.5.5 Secure Information Flow ..................181
7.6 Notes ...........................................182
7.7 Discussion .....................................183

8 Formal Systems Development with UML ............185
8.1 Formal Semantics for a Fragment of UML ......185
8.1.1 General Concepts ............................189
8.1.2 Class Diagrams ..............................196
8.1.3 Statechart Diagrams .......................197
8.1.4 Sequence Diagrams .......................208
8.1.5 Activity Diagrams .........................213
8.1.6 Deployment Diagrams ......................216
8.1.7 Subsystems .................................216
8.2 Development with UML .........................223
8.2.1 Refinement ..................................223
8.2.2 Rely-Guarantee Specifications .............227
8.3 Notes ...........................................227
8.4 Discussion .....................................228

Part IV Epilogue

9 Further Material ..................................233
9.1 More on UMLsec .................................233
9.2 Other Approaches to Security Engineering ....234
9.2.1 Formal Methods Applied to Security ....234
9.2.2 Software Engineering and Security ....236
9.2.3 Other Approaches Using UML ..........236
9.2.4 Other Non-functional Requirements ..238

10 Outlook .........................................239
# Part V Appendices

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Towards UML 2.0</td>
<td>243</td>
</tr>
<tr>
<td>B</td>
<td>The Semantics of UML Machine Rules</td>
<td>245</td>
</tr>
<tr>
<td>C</td>
<td>Proofs</td>
<td>249</td>
</tr>
<tr>
<td>C.1</td>
<td>Secure Channels</td>
<td>249</td>
</tr>
<tr>
<td>C.2</td>
<td>A Variant of the Internet Protocol TLS</td>
<td>251</td>
</tr>
<tr>
<td>C.3</td>
<td>Common Electronic Purse Specifications</td>
<td>255</td>
</tr>
<tr>
<td>C.3.1</td>
<td>Purchase Transaction</td>
<td>255</td>
</tr>
<tr>
<td>C.3.2</td>
<td>Load Transaction</td>
<td>260</td>
</tr>
<tr>
<td>C.4</td>
<td>UML Machines</td>
<td>263</td>
</tr>
<tr>
<td>C.5</td>
<td>Refinement</td>
<td>263</td>
</tr>
<tr>
<td>C.6</td>
<td>Rely-Guarantee Specifications</td>
<td>266</td>
</tr>
<tr>
<td>C.7</td>
<td>Reasoning About Security Properties</td>
<td>266</td>
</tr>
<tr>
<td>C.8</td>
<td>Formal Systems Development with UML</td>
<td>271</td>
</tr>
<tr>
<td></td>
<td>References</td>
<td>273</td>
</tr>
<tr>
<td></td>
<td>Index</td>
<td>295</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Model-based development ........................................... 8

2.1 Use case diagram for business application ....................... 18
2.2 Purchase activity diagram ........................................... 19
2.3 Example secure links usage ......................................... 19
2.4 Key exchange protocol .............................................. 21
2.5 Customer account data object ....................................... 22

3.1 Use case diagram ..................................................... 27
3.2 Class diagram ...................................................... 28
3.3 Statechart diagram .................................................. 29
3.4 Sequence diagram ................................................... 30
3.5 Activity diagram .................................................... 31
3.6 Deployment diagram ................................................ 31
3.7 Subsystem ............................................................ 33

4.1 UMLsec stereotypes .................................................. 49
4.2 UMLsec tags .......................................................... 50
4.3 Use case diagram for business application ....................... 51
4.4 Purchase activity diagram ......................................... 52
4.5 Role-based access control example ................................. 54
4.6 Threats from the default attacker ................................... 56
4.7 Threats from the insider attacker card issuer ....................... 56
4.8 Example secure links usage ......................................... 57
4.9 Key generation subsystem instance ................................. 58
4.10 Bank account data object ........................................ 59
4.11 TLS protocol variant ............................................... 61
4.12 Financial application specification ................................ 64
4.13 Security pattern example: sender and receiver .................... 69
4.14 Security pattern example: secure channel ......................... 70
5.1 Example subsystem: sender and receiver .......................... 74
5.2 Example subsystem: secure channel .............................. 76
5.3 Variant of the TLS handshake protocol ........................... 79
5.4 Repaired variant of the TLS handshake protocol ............... 83
5.5 POS device overview ............................................. 87
5.6 Specification for the CEPS purchase transaction ............... 89
5.7 Modified part of CEPS purchase specification ................... 94
5.8 Specification for load transaction ................................ 97
5.9 Load transaction class diagram .................................. 99
5.10 Load transaction: load acquirer .................................. 99
5.11 Load transaction: card ............................................. 100
5.12 Load transaction: card issuer ..................................... 100
5.13 Sequence diagram for load transaction ......................... 101
5.14 Values exchanged in the load specification ................... 102
5.15 Specification for modified load transaction .................... 109
5.16 Modified load transaction class diagram ....................... 111
5.17 Modified load transaction: load acquirer ....................... 111
5.18 Modified load transaction: card issuer ......................... 112
5.19 Sequence diagram for modified load transaction ............... 113
5.20 Guard object example ............................................. 117
5.21 Guard object message exchange ............................... 118
5.22 Guard object security flaw ..................................... 118
5.23 Financial application specification ............................. 120
5.24 Authentication protocol ......................................... 122

6.1 MOF framework: meta-levels ................................. 128
6.2 MOF framework: example ........................................ 129
6.3 Using the MDR library ........................................... 131
6.4 UML tools suite ................................................. 133
6.5 Tool interfaces ................................................... 135
6.6 Overview of the tool’s architecture ............................ 138
6.7 Simple role-based access control ............................... 140
6.8 Snippet from an instance file ................................. 141
6.9 The graph after model and information are inserted .......... 141
6.10 Class diagram showing the structure of rules ................ 144
6.11 Small separation of duty example ............................ 146
6.12 Sample configuration for using the analyzer ................ 149

7.1 UML Machine rules .............................................. 159
7.2 Behavior of UML Machine ...................................... 163
7.3 Example UML Machine ......................................... 164
7.4 Behavior of UMS ............................................... 167
7.5 Behavior of UMS (only asynchronous messages) .............. 168
7.6 Simple scheduler ................................................. 168
8.1 Example: parallel invocations .............................................. 202
8.2 Statechart rule ............................................................... 204
8.3 Event execution rule .......................................................... 205
8.4 Exit state rule ................................................................. 206
8.5 Execute event rule ............................................................. 206
8.6 Enter state rule ............................................................... 207
8.7 Example: sender statechart .................................................. 207
8.8 Example interpretation ....................................................... 207
8.9 UML Machine rule for sequence diagram ......................... 212

B.1 The semantics of UML Machine rules ............................ 246
Part I

Prologue
Introduction

A Need for Security

Modern society and modern economies rely on infrastructures for communication, finance, energy distribution, and transportation. These infrastructures depend increasingly on networked information systems. Attacks against these systems can threaten the economical or even physical well-being of people and organizations. There is widespread interconnection of information systems via the Internet, which is becoming the world’s largest public electronic marketplace, while being accessible to untrusted users. Attacks can be waged anonymously and from a safe distance. If the Internet is to provide the platform for commercial transactions, it is vital that sensitive information (like credit card numbers or cryptographic keys) is stored and transmitted securely.

Problems

Developing secure software systems correctly is difficult and error-prone. Many flaws and possible sources of misunderstanding have been found in protocol or system specifications, sometimes years after their publication or use (for example, the observations in [Low95] were made 17 years after the well-known protocol concerned had been published in [NS78]). Many vulnerabilities in deployed security-critical systems have been exploited, sometimes leading to spectacular attacks. For example, as part of a 1997 exercise, an NSA hacker team demonstrated how to break into US Department of Defense computers and the US electric power grid system, among other things simulating a series of rolling power outages and 911 emergency telephone overloads in Washington, DC, and other cities [Sch99]. While there are of course many more recent examples of security breaches, this particular example also shows that there is more to be concerned about than website defacements and creditcard misuse.

Computer breaches do significant damage, as a study by the Computer Security Institute shows:
• Ninety percent of the respondents detected computer security breaches within the last 12 months.
• Forty-four percent of them were willing and able to quantify the damage. These 223 firms reported $455,848,000 in financial losses [Ric03].

Causes

Firstly, enforcing security requirements is intrinsically subtle, because one has to take into account the interaction of the system with motivated adversaries that act independently. Thus security mechanisms, such as security protocols, are notoriously hard to design correctly, even for experts. Also, a system is only as secure as its weakest part or aspect.

Secondly, risks are very hard to calculate: security-critical systems are characterized by the fact that the occurrence of a successful attack at one point in time on a given system dramatically increases the likelihood that the attack will be launched subsequently at another system. For some attacks (for example, against websites), this problem is made worse by the existence of the Internet as a mass communication medium that is currently largely uncontrolled and enables fast and anonymous distribution of exploited information.

Thirdly, many problems with security-critical systems arise from the fact that their developers, who employ security mechanisms, do not always have a strong background in computer security. This is problematic since, in practice, security is compromised most often not by breaking dedicated mechanisms such as encryption or security protocols, but by exploiting weaknesses in the way they are being used [And01]:

• According to A. Shamir, the Israeli state security apparatus is not hampered in its investigations by the fact that suspects may use encryption technology that may be virtually impossible to break. Instead, other weaknesses in overall computer security can be exploited [Sha99].
• As another example, the security of Common Electronic Purse Specifications (CEPS) [CEP01] transactions depends on the fact that in the immediately envisaged scenario (use of the card for purchases in shops) it is not feasible for the attacker to act as a relay between an attacked card and an attacked terminal. However, this is not explicitly stated, and it is furthermore planned to use the CEPS over the Internet, where an attacker could easily act as such a relay (this is investigated in Sect. 5.3).
• [Wal00] attributes failures in the security of the mobile phone protocol GSM to:
  - the failure to acknowledge limitations of the underlying physical security (misplaced trust in terminal identity, possibility to create false base stations),
  - an inadequate degree of flexibility to upgrade security functions over time, and
  - a lack in the user interface to communicate security-critical information (no indication to the user whether encryption is on).
Thus it is not enough to ensure correct functioning of security mechanisms used; they cannot be “blindly” inserted into a security-critical system, but the overall system development must take security aspects into account [And94]. In the context of computer security, “an expansive view of the problem is most appropriate to help ensure that no gaps appear in the strategy” [SS75]. In other words, “those who think that their problem can be solved by simply applying cryptography don’t understand cryptography and don’t understand their problem” (mutually attributed by B. Lampson and R. Needham to each other). In fact, according to [Sch99], 85% of Computer Emergency Response Team (CERT) security advisories [CERT] could not have been prevented just by making use of cryptography. Thus, with the current state of software security, “using encryption on the Internet is the equivalent of arranging an armored car to deliver credit card information from someone living in a cardboard box to someone living on a park bench”, as Gene Spafford puts it [VM02]. Building trustworthy components does not suffice, since the interconnections and interactions of components play a significant role in trustworthiness [Sch99].

Lastly, while functional requirements are generally analyzed carefully in systems development, security considerations often arise after the fact. Adding security as an afterthought, however, often leads to problems [Gas88, And01]. Also, security engineers get little feedback about the secure functioning of their products in practice, since security violations are often kept secret for fear of harming a company’s reputation.

It has remained true over the last 25 years that “no complete method applicable to the construction of large general-purpose systems exists yet” [SS75] that would ensure security, in spite of very active research and many useful results addressing particular subgoals [Sch99]. Ad hoc development has led to many deployed systems that do not satisfy relevant security requirements. Thus a sound methodology supporting secure systems development is needed.

**Traditional Approaches**

In practice, the traditional strategy for security assurance has been “penetrate and patch”: it has been accepted that deployed systems contain vulnerabilities; whenever a penetration of the system is noticed and the exploited weakness can be identified, the vulnerability is removed. Sometimes this is supported by employing friendly teams trained in penetrating computer systems (so-called “tiger teams” [Wei95, McG98]).

For many systems, this approach is not ideal:

- Each penetration using a new vulnerability may already have caused significant damage, before the vulnerability can be removed.
- For systems that offer strong incentives for attack (such as financial applications), the prospect of being able to exploit a discovered weakness only once may already be enough motivation to search for such a weakness.
- System administrators are often hesitant to apply patches, especially in critical systems (since applying the patch may disrupt the service) [And01].
Having to create and distribute patches costs money and leads to loss of customer confidence.

- Patches may contain security threats themselves (an example is a virus in a Microsoft patch released in 2001).

It would thus be preferable to consider security aspects more seriously in earlier phases of the system life-cycle, before a system is deployed, or even implemented, because late correction of requirements errors costs up to 200 times as much as early correction [Boe81].

The difficulty of designing security mechanisms correctly, such as authentication protocols, has motivated quite successful research using mathematical concepts and tools to ensure correct design of small security-critical components such as security protocols (for example, [MCF87, BAN89, Mea91, Low96, AG99, Pau98b]). In formal methods, one makes use of modeling and analysis based on a mathematically precise foundation. They are beneficial in application areas with a need to provide rigorous evidence of system properties, such as security-critical systems. A goal behind the use of formal methods is to establish crucial requirements on the specification level after formalization and proof (which may be mechanical). Note that it is not possible to actually prove a system secure in an absolute sense; Proofs can only be performed with respect to models which are necessarily abstractions from reality. Attackers can always try to go beyond the limitations of a given model to still attempt an attack. Nevertheless, a model-based security analysis is useful, because certain attacks can be prevented and the required effort for successful attacks increased. Also, often problems with a specification are detected just by trying to make it sufficiently precise for formal analysis [Gol03a].

Unfortunately, due to a perceived high cost in personnel training and use, formal methods have not yet been employed very widely in industrial development [Hoe96, Hei99, Sch00, AR00]. To increase industry acceptance in the context of security-critical systems, it would be beneficial to integrate security requirements analysis with a standard development method, which should be easy to learn and to use [CW96]. Also, security concerns must inform every phase of software development, from requirements engineering to design, implementation, testing, and deployment [DS00b].

Some other challenges for using sound engineering methods for secure systems development remain:

- Currently a large part of effort both in verifying and in implementing specifications is wasted since these are often formulated imprecisely and unintelligibly, if they exist at all [Pau98a]. If increased precision by use of a particular notation brings an additional advantage (such as assistance with regard to security aspects), this may be sufficient incentive for providing it.
- In particular, a formal specification of the system is usually not available. Constructing it (from informal specifications that can easily consist of sev-
eral hundred pages or more, such as the CEPS) requires expert knowledge and can be very time consuming.

- It is sometimes only feasible to construct the specification of a small security-critical part of the system. The boundaries of these components with the rest of the system need to be carefully examined, for example with respect to implicit assumptions on the system context [Gol00, Aba00].

- Stepwise development (where one starts with an abstract specification and refines it in several steps to a concrete specification which is implemented) allows mistakes to be detected early in the development cycle, leading to considerable savings. Stepwise development of security-critical systems is hindered by the fact that many security properties proposed in the literature are not preserved under standard notions of refinement (the so-called refinement problem [RSG+01]). For such properties, developing secure systems in a stepwise manner requires one to redo the security analysis after each refinement step. More worryingly, since an implementation is necessarily a refinement of its specification, an implementation of a secure specification may not be secure. Hence, one needs to develop precise formulations of security requirements that are indeed preserved under refinement.

The present work aims to make some contribution toward addressing some of these challenges.

Model-Based Security with UML

Towards a solution of the problems mentioned in the previous sections, we propose to use an approach of model-based development using the Unified Modeling Language (UML) [RJB99, Obi03].

Generally, in model-based development (see Fig. 1.1), the idea is to first construct a model of a system, which should be as close to human intuition as possible and is typically relatively abstract. In a second step, the implementation is derived from the model (either automatically using code generation, or manually, in which case one can still generate test sequences from the model to establish conformance of the code regarding the model). The goal is to increase the quality of the implemented code while keeping the implementation cost and the time-to-market bounded.

For security-critical systems, this approach allows one to consider security requirements:

- from early on in the development process,
- within the development context, and
- in a seamless way and taking an expansive view on the problem.

Using the model-based approach, one can, firstly, establish that the system fulfills the relevant security requirements on the design level, by analyzing the model. Secondly, one can check that the code is also secure by generating test sequences from the model.
UML now offers unprecedented opportunities as a notation for a high-quality model-based development of security-critical systems that is feasible in an industrial context:

- As the de facto standard in industrial modeling notations, a large number of developers are trained in UML (and this number is still growing because UML is widely taught at universities). Thus, a UML specification may already be available for security analysis, or less difficult to obtain than other notations.
- Compared to previous notations with a user community of comparable size, UML is relatively precisely defined.
- A variety of tools exist that provide the basic functionality required to use UML (such as the drawing of UML diagrams).

To exploit this opportunity, however, some challenges remain which include the following:

- Adapting UML to the application domain of security-critical systems.
- Advancing the correct use of UML in this application domain.
- Dealing with conflicts between flexibility and unambiguity in the meaning of UML models.
- Developing advanced tool support for secure systems development with UML (such as automatic analysis of UML specifications with respect to security requirements).

This book aims to contribute to overcoming these challenges.

More specifically, it presents the UML extension UMLsec for secure systems development. UMLsec:

- allows one to evaluate UML specifications for security weaknesses on the design level,
- encapsulates established rules of prudent security engineering in the context of a widely known notation, and thus makes them available to developers who may not be experts in security;  
- allows the developer to consider security requirements from early on in the system development process, and  
- involves little additional overhead, since the UML diagrams can serve as system documentation, which is always desirable to have, and sometimes even strictly required (for example, for security certifications).

Note that although the UML extension proposed in this book aims to also offer assistance to developers who are not security experts (for example, by enabling them to use security mechanisms in a secure way), parts of the book are concerned with advanced applications (such as cryptoprotocol analysis) for which background knowledge in security would be helpful.

1.1 Overview

UMLsec

We present an extension of the UML [Obj03] for secure systems development, called UMLsec. UML is the de facto industry standard in object-oriented modeling providing graphical, intuitive description techniques with multiple views of a system through different kinds of diagrams. UML offers standard extension mechanisms (stereotypes, tags, constraints, profiles) which we use to define UMLsec. Note that although UML was developed to model object-oriented systems, one may use UMLsec just as well to analyze systems that are not object-oriented, by thinking of objects as components and not making use of object-oriented features, such as inheritance (in that case, however, underlying assumptions, such as controlled access to data, still need to be ensured).

Recurring security requirements (such as secrecy, integrity, and authenticity) are offered as specification elements by the UMLsec extension. The properties are used to evaluate diagrams of various kinds and to indicate possible vulnerabilities. One can thus verify that the stated security requirements, if fulfilled, enforce a given security policy. One can also ensure that the requirements are actually met by the given UML specification of the system. UMLsec encapsulates knowledge on prudent security engineering and thereby makes it available to developers who may not be experts in security.

The extension is given in form of a UML profile using the standard UML extension mechanisms. Stereotypes are used together with tags to formulate security requirements and assumptions on the system environment; constraints give criteria that determine whether the requirements are met by the system design (by referring to the precise semantics mentioned above).
Note that an extension of UML to an application domain such as security-critical systems that aims to include requirements from that application domain as stereotypes (as opposed to just adding specific architectural primitives) can probably never feasibly be fully complete, because it would then have to incorporate all existing design knowledge on security-critical computing systems, which would fill countless books. Therefore, here we focus on providing a core profile that includes the main security requirements, and we expect this to be extended with additional, more specific concepts (for example, from sub-application domains such as mobile security), also in the course of the intended striving for standardization.

We list the requirements on a UML extension for secure systems development and discuss how far our extension meets these requirements. We explain the details of the extension by means of examples.

We show how to use UMLsec in order to apply security patterns. We demonstrate how to employ the extension for enforcing established rules of secure systems design.

Applications

To validate our approach using UMLsec for secure systems development, we investigate the degree to which it is suitable for enforcing established rules of prudent security engineering.

We consider several case studies:

- We demonstrate stepwise development of a security-critical system with UMLsec as the example of a secure channel design, together with a formal verification (where, in the context of this book, by “formal” we mean “mathematically precise”, but not necessarily involving formal derivations in logic).
- We uncover a flaw in a variant of the handshake protocol of the Internet protocol TLS proposed in [APS99], suggest a correction, and verify the corrected protocol.
- We apply UMLsec to a security analysis of CEPS, a candidate for a globally interoperable electronic purse standard. We discover flaws in the two central parts of the specifications (the purchase and the load protocol), propose corrections, and give a verification of the corrected versions.
- We show how to use UMLsec to correctly employ advanced Java 2 security concepts such as guarding objects in a way that allows formal verification of the specifications.
- We also report on a project with a major German bank, where we applied our ideas about model-based development of security-critical systems to a web-based banking application.
Tool Support

For the ideas that we present in this book to be of benefit in practice, it is important to have advanced tool support to assist in using them. We present the necessary background and some results achieved so far toward this goal with the UMLsec project at TU Munich [JSA+03]. The developed tools can be used to check the constraints associated with UMLsec stereotypes mechanically, based on XMI output of the diagrams from the UML drawing tool in use. For this, the developer creates a model using a UML drawing tool capable of XMI export and stores it as an XMI file. The file is imported by the UMLsec analysis tool (for example, through its web interface) which analyses the UMLsec model with respect to the security requirements that are included. The results of the analysis are given back to the developer, together with a modified UML model, where the weaknesses that were found are highlighted.

We also explain a framework for implementing verification routines for the constraints associated with the UMLsec stereotypes. The goal is that advanced users of the UMLsec approach should be able to use this framework to implement verification routines for the constraints of self-defined stereotypes. In particular, the framework includes the UMLsec tool web interface, so that new routines are also accessible over this interface. The idea behind the framework is thus to provide a common programming framework for the developers of different verification modules. A tool developer should be able to concentrate on the implementation of the verification logic and not be required to implement the user interface.

Furthermore, we present research on linking the UMLsec approach with the automated analysis of security-critical data arising at runtime. Specifically, we present joint research with Sebastian Höhn on the construction of a tool which automatically checks the SAP R/3 configuration for security policy rules (such as separation of duty). The permissions are given as input in an XML format through an interface from the SAP R/3 system, the rules are formulated as UML specifications in a standard UML CASE tool and output as XMI (as part of the UMLsec framework mentioned above), and the tool checks the permissions against the rules using an analyzer written in Prolog. Because of its modular architecture and its standardized interfaces, the tool can be adapted to check security constraints in other kinds of application software (such as firewalls or other access control configurations).

As noted, for example, in [FS97], the ultimate benefit in software development is not "pretty pictures", but the running implementation of a system. Therefore, we also present some approaches for linking UML models to implementations (for example, model-based testing, in joint work with Guido Wimmel). The aim is to ensure that the benefits gained from the model-based approach on the level of the system model (such as increased confidence in satisfaction of critical requirements) actually carries over to the implemented system, as one would hope.
To provide tool support for analyzing UMLsec models with respect to the security properties included as stereotypes, they need to be formulated in a mathematically precise way. This is only possible if the UML specification they refer to also has a mathematically precise meaning, in particular regarding the behavioral aspects (since many security requirements refer to the system behavior). For this goal, we provide a precise execution semantics for a (restricted and simplified) part of UML using so-called UML Machines. These are based on Abstract State Machines which give a mathematically rigorous yet rather flexible framework for modeling computing systems (see [Gur95, BS03a]). We thus have a mathematically sound foundation which can be coded up relatively straightforwardly into the UMLsec tool support mentioned above. To keep a mechanical analysis of these requirements feasible, the present work focusses on a specific way of using a restricted and simplified part of UML for critical systems development.

UML Machine Systems allow us then to build up UML Machine specifications in a modular way and to treat external influences on the system beyond the planned interaction (such as attacks on insecure communication links). We define notions of refinement and rely-guarantee specifications for UML Machines to support stepwise and modular development methodologies for reactive systems. The role of UML Machines is thus to provide the conceptual framework for formulating the concepts we use, which are close in spirit to the models of data-flow (such as [Bro86, Abr90]).

We show how to use UML Machine Systems to specify security-critical systems (that may employ cryptographic operations). We give definitions for secrecy, integrity, authenticity, and secure information flow. We give equivalent internal characterizations of secrecy, integrity, and authenticity which allow easier verification. We show secrecy, integrity, authenticity, and secure information flow to be preserved under refinement.

The notion of UML Machine Systems allows a rather natural modeling of potential adversary behavior. We can model specific types of adversaries that can attack different parts of the system in a specified way. For example, an attacker of type insider may be able to intercept the communication links in a company-wide local area network. We model the actual behavior of the adversary by defining a class of UML Machines that can access the communication links of the system in a specified way. To evaluate the security of the system with respect to the given type of adversary, we consider the joint execution of the system with any UML Machine in this class. This way of reasoning allows an intuitive formulation of many security properties. Since the actual verification is rather indirect this way, we also give alternative intrinsic ways of defining security properties, which are more manageable, and show that they are equivalent to the earlier ones.

We provide a precise semantics for a (restricted and simplified) part of UML that allows one to use UML subsystems to group together several diagrams. The statechart semantics which is part of it is based on part of the statechart semantics from [BCR00]. The precise semantics for a restricted
version of subsystems incorporates the precise semantics of the diagrams contained in a subsystem in a specific way. Actions and internal activities are modeled explicitly (rather than treating them as atomic given events). In particular, objects, and more generally system components, can communicate by exchanging messages with parameters, which can be used in the subsequent execution. One may compose subsystems by including them in other subsystems.

We emphasize that we only consider a simplified fragment of the UML syntax. In particular, the notion of subsystem considered here is restricted, for example in the kinds and numbers of diagrams that may be contained. Also, an occurrence of a message is only created and consumed once in a given execution of a specification. The motivation is to concentrate on a core of UML for which it is feasible to construct and use advanced tool support. A more detailed discussion of the simplifications and their motivation can be found in the relevant sections. To demonstrate that our choice of a subset of UML is reasonable and our semantics of sufficient interest, we present several case studies (as mentioned above). Some of these are taken from industrial applications and of a size that goes beyond that of the examples usually considered in the academic literature (which is often limited [Go03a]); they were chosen to demonstrate that the fragment of UML used in our work is sufficient for our needs.

Via UML Machines and UML Machine Systems we make use of the presented treatment of security-critical systems. In particular, UML specifications can be evaluated using the attacker model, which incorporates the possible attacker behaviors, to find vulnerabilities. For the trivial kind of adversary who is not able to access any part of the system, our approach gives us the usual (simplified) UML semantics.

We give some conditions for consistency among different diagrams in the kind of UML specification considered here that offers different views on the specified system. We define a notion of behavioral equivalence between the UML specifications considered here. This can be used for example to verify consistency of two subsystem specifications that are supposed to describe the same behavior, for example one of which uses statecharts to specify object behavior, and the other sequence diagrams. We define two kinds of refinement for our UML specifications. The first of these, property refinement, provides full behavioral conformance, and thus preserves all safety properties. The second, interface refinement, allows some control over the extent to which structure and behavior of the system is preserved. Since these notions are derived from the corresponding notions for UML Machines, they enjoy the same structural properties (such as preservation of the considered security properties). Finally, we define rely-guarantee specifications for UML and prove some useful results regarding them.
1.2 Outline

Here is an outline of the following chapters:

Chapter 2: For a short “walk-through” to highlight the UMLsec approach, we consider a (fictitious and simplified) model of an Internet-based business application as a running example.

Chapter 3: We recall some background information needed in the remainder of the book.

Chapter 4: We explain the UML extension mechanisms. We discuss requirements on a UML extension for secure systems development and present the UMLsec profile. We show how to formulate security requirements on a system and security assumptions on the underlying layer in UMLsec. We explain how to evaluate the system specification against the security requirements, by referring to the precise semantics sketched in Chap. 3. We explain the details of the extension by means of examples. We show how to use UMLsec in order to apply security patterns. We demonstrate how to employ the extension for enforcing established rules of secure systems design.

Chapter 5: We demonstrate stepwise development of a security-critical system with UMLsec as the example of a secure channel design. We uncover a flaw in a variant of the handshake protocol of the Internet protocol TLS proposed in [APS99], suggest a correction, and verify the corrected protocol. We use UMLsec for a security analysis of CEPS, a candidate for a globally interoperable electronic purse standard. We discover flaws in the two central parts of the specifications, propose corrections, and give a verification. We show how to use UMLsec to correctly employ advanced Java 2 security concepts such as guarded objects.

Chapter 6: We explain the necessary background for developing tool support for UMLsec. We present a tool which automatically checks a UMLsec model with respect to the security requirements associated with the UMLsec stereotypes, based on XML output of industrial UML drawing tools. We present a framework which allows advanced users to conveniently include verification routines for the constraints of self-defined stereotypes. We present a tool which links the UMLsec approach with the automated analysis of security-critical data arising at runtime (such as permissions in SAP R/3 systems). We present approaches for linking UML models to implementations (such as model-based testing).

Chapter 7: We introduce UML Machines and UML Machine Systems. We define notions of refinement and rely-guarantee specifications for UML Machines. We explain how we use UML Machines to specify security-critical systems. We give definitions for secrecy, integrity, 'authenticity', and secure information flow, and give equivalent internal characterizations to simplify verification. We show secrecy, integrity, 'authenticity', and secure information flow to be preserved under refinement.
Chapter 8: We use UML Machines and UML Machine Systems to give a precise semantics for a simplified part of UML. We give consistency conditions for different diagrams in a UML specification. We define notions of refinement and behavioral equivalence, and investigate structural properties (such as substitutivity). We consider rely-guarantee properties for UML specifications and their structural properties.

Chapter 9: We give an account of other soundly-based approaches to security engineering, some of which also use UML.

Chapter 10: We conclude with a critical evaluation of the material presented and an outlook on future developments.

Appendices: We give the formal definition of UML Machine rules and the proofs for the statements from Chaps. 5, 7, and 8.

1.3 How to Use this Book

Being the first book on the topic of secure systems development with UML, this book was written with two audiences in mind:

- researchers and graduate students interested in UML, computer-aided software engineering or formal methods, and IT security, who may use the book as background reading for their own research in using UML for critical systems development, or in building advanced tool support for UML
- advanced software developing professionals as the intended users of the approach proposed in this book.

For the benefit of the second group, we deferred the material on the semantics of UML (which the first group needs to know but the second needs not) to the end of the book in Chaps. 7 and 8. These can then be left out by people who are not interested in constructing advanced tool support for UML by themselves; the information in Sect. 3.3 about the used semantics of UML is sufficient to understand the remaining chapters. Note that although the UML extension proposed in this book aims to also offer assistance to developers who are not security experts (for example, by enabling them to use security mechanisms in a secure way), parts of the book are concerned with advanced applications (such as cryptoprotocol analysis) for which background knowledge in security would be helpful.

The material in this book has been used extensively for teaching both of these audiences. For example, full-day tutorials for practitioners have been delivered based on the material in Chaps. 3 and 4 and Sects. 6.2 and 6.4. For a two-day course, one can also include Chap. 5. A Masters-level student course could also cover Chaps. 7 and 8.

Additional material is given on a website [Web] associated with this book which is continuously being updated. It includes the following material:

- Slides and audio recordings from the tutorials and courses based on this book.
1 Introduction

- Other learning and teaching material, including exercises and answers.
- A web interface for a tool which analyzes UMLsec models written using an industrial UML modeling tool (which one can upload over the Internet) for security requirements.
Walk-through: Using UML for Security

For a quick impression of what this book is about, we give a short "walkthrough" through part of the UMLsec notation to highlight the UMLsec approach, considering a (fictitious and simplified) model of an Internet-based business application as a running example. For readers who find themselves lacking background on computer security and on the Unified Modeling Language (UML), it is briefly recalled in Chap. 3. The UMLsec extension is then defined and explained in more detail in Chap. 4 (as well as the examples shown in this chapter).

A central idea of the UMLsec extension is to define labels for UML model elements (so-called stereotypes) which, when attached, add security-relevant information to these model elements. This security-relevant information can be of the following kinds:

- Security assumptions on the physical level of the system, such as the «Internet» stereotype shown below.
- Security requirements on the logical structure of the system (such as the «secrecy» stereotype) or on specific data values (such as the «critical» stereotype).
- Security policies that system parts are supposed to obey, such as the «fair exchange», «secure links», «data security», or «no down – flow» stereotypes.

In the first two cases, the stereotypes simply add some additional information to a model. They can be attached to any diagram of the relevant kind. In the third case, there are constraints associated with a stereotype that have to be fulfilled by a diagram so that it can justifiably carry the stereotype. If such a stereotype is attached to a diagram which does not meet this constraints, this results in an incorrect model (as in the case of the «secure links», «data security», and «no down – flow» stereotypes below). This prompts the tool-support available for UMLsec to automatically point out the mistake, which should then be corrected by the developer.
2.1 Security Requirements Capture with Use Case Diagrams

Use case diagrams are commonly used to describe typical interactions between a user and a computer system in requirements elicitation. They may also be used to capture security requirements.

To start with our example, Fig. 2.1 gives a use case diagram describing the following situation: a customer buys a good from a business. The trade should be performed in a way that prevents both parties from cheating. We include this requirement in the diagram by adding a stereotype «fair exchange» to the subsystem containing the use case diagram. A more detailed explanation of what the requirement represented by this stereotype means in this specific situation, and of the activities associated with the use cases, is given in the following subsection.

![Use case diagram](image)

**Fig. 2.1. Use case diagram for business application**

2.2 Secure Business Processes with Activity Diagrams

Activity diagrams can be used to model workflow and to explain use cases in more detail. Similarly, they can be used to make security requirements more precise.

Following our example, Fig. 2.2 explains the use case in more detail by giving the business process realizing the above two use cases. The requirement «fair exchange» is now formulated by referring to the activities in the diagram. Intuitively, the actions listed in the tags {start} and {stop} should be linked in the sense that if one of the former is executed then eventually one of the latter will be (this property can be checked mechanically, as explained in Chap. 6).

This would entail that, once the customer has paid, either the order is delivered to the customer by the due date, or the customer is able to reclaim the payment on that date.
2.3 Physical Security Using Deployment Diagrams

Deployment diagrams are used to describe the physical layer of a system. We use them to check whether the security requirements on the logical level of the system are enforced by the level of physical security, or whether additional security mechanisms (such as encryption) have to be employed.

Fig. 2.2. Purchase activity diagram

Fig. 2.3. Example secure links usage

Continuing with our example, the e-commerce system is supposed to be realized as a web application. The payment transaction involves transmission of data to be kept secret (such as credit card numbers) over Internet links. This information on the physical layer and the security requirement is reflected in the UML model in Fig. 2.3. We then use the stereotype «secure links» to express the demand that security requirements on the communication are
met by the physical layer. More precisely, for each dependency \( d \) stereotyped «secrecy» between subsystems or classes on different nodes \( n, m \), and any communication link \( l \) between \( n \) and \( m \) with some stereotype \( s \), the threat scenario arising from the stereotype \( s \) with regard to an adversary of a given strength should not violate the secrecy requirement on the communicated data (and similarly for «integrity»). In the given diagram, this constraint (associated with the stereotype «secure links») is already violated when considering standard adversaries, because plain Internet connections can be eavesdropped easily, and thus the data that is communicated does not remain secret. For this adversary type, the stereotype is thus applied wrongly to the subsystem (which, again, is pointed out automatically by the UMLsec tool; see Chap. 6).

2.4 Security-Critical Interaction with Sequence Diagrams

Sequence diagrams are used to specify interaction between different parts of a system. Using UMLsec stereotypes, we can extend them with information giving the security requirements relevant to that interaction (for example, to see whether cryptographic session keys exchanged in a key exchange protocol remain confidential from possible adversaries).

With regard to our example, based on the security analysis in the previous subsection we decide to create a secure channel for the sensitive data that has to be sent over the untrusted networks, by making use of encryption. As usual, we first exchange symmetric session keys for this purpose. Let us assume that, for technical reasons, we decide not to use a standard and well-examined protocol such as SSL but instead a 'customized' key exchange protocol such as the (slightly simplified) one in Fig. 2.4. The goal is to exchange a secret session key \( K \), using previously exchanged public keys \( K_C \) and \( K_S \), which is then used to encrypt the secret data \( s \) before transmission. Here \( \{ M \}_K \) is the encryption of the message \( M \) with the key \( K \), \( \text{Sign}_K (M) \) is the signature of the message \( M \) with \( K \), and :: denotes concatenation (a detailed explanation of the figure and the protocol can be found in Sect. 5.2).

Note that the UMLsec model of the protocol given in Fig. 2.4 is similar to the traditional informal notation (for example, used in [NS78]), in which the protocol would be written as follows:

\[
\begin{align*}
C & \rightarrow S : N_i, K_C, \text{Sign}_{K_C^{-1}} (C :: K_C) \\
S & \rightarrow C : \{ \text{Sign}_{K_C^{-1}} (k_j :: N_i) \}_K, \text{Sign}_{K_S^{-1}} (S :: K_S) \\
C & \rightarrow S : \{ \hat{s}_j \}_K.
\end{align*}
\]

We argue in Sect. 5.2 that the traditional notation needs to be interpreted with care and that the UMLsec notation can be seen to be more precise and to lead over more easily to an implementation.
Fig. 2.4. Key exchange protocol

One can now again use stereotypes to include important security requirements on the data that is involved. Here, the stereotype «critical data» labels classes containing sensitive data and has the associated tags {secrecy}, {integrity}, {authenticity}, and {fresh} to denote the respective security requirements on the data. The constraint associated with «data security» then requires that these requirements are met relative to the given adversary model. We assume that the standard adversary is not able to break the encryption used in the protocol, but can exploit any design flaws that may exist in the protocol, for example by attempting so-called “man-in-the-middle” attacks (this is made precise for a universal adversary model in Sect. 3.3.4). Technically, the constraint then enforces that there are no successful attacks of that kind. Note that it is highly non-trivial to see whether the constraint holds for a given protocol. However, using well-established concepts from formal methods applied to computer security in the context of UMLsec, it is possible to verify this automatically.
2.5 Secure States Using Statechart Diagrams

Statechart diagrams, showing the changes in state throughout an object’s life, can be used to specify security requirements on the resulting sequences of states and the interaction with the object’s environment.

As the last station in our quick walk-through, we now assume that for privacy reasons, it should remain secret how much money a customer spends at the website. We thus consider the (again very simplified) specification of the customer account object in Fig. 2.5. The object has a secret attribute money containing the amount of money spent so far by a given customer. It can be read using the operation rm() whose return value is also secret, and increased by placing an order using the operation wm(x). If the amount of money spent already is over 1000 (that is, the object is in the state ExtraService), there is a second special login at the website to a webpage providing the customer with complimentary extra services. There is an associated operation rx() to check whether a customer has the right to log in to that page (which is not assumed to be secret because, by construction of the website, this information is public).

Now we use the stereotype «no down-flow» to indicate that the object should not leak out any information about secret data (such as the money attribute). Unfortunately, the given specification violates this requirement (and thus the model carries the stereotype illegitimately), since partial information about the input of the secret operation wm() is leaked out via the return value of the non-secret operation rx(). Again this can be checked mechanically, and it is another example for a constraint which (given a specification of a realistic size) is infeasible to verify without tool support.

![Customer account data object](image)

**Fig. 2.5.** Customer account data object
3

Background

We briefly present some important concepts used in the course of this book and give references to more comprehensive background reading.

Some previous knowledge of computer security and of UML may be helpful since an in-depth introduction has to be omitted. Below, we give some very short and personal suggestions for background reading; of course, there are many more good introductory references on these topics. We also briefly recall the main concepts needed for our purposes.

3.1 Security Engineering

Good textbooks on computer security engineering include [Gol99, And01], as well as [VM02] with an emphasis on software.¹ [APG95] contains interesting essays on the topic. “Classic” references on security engineering include [SS75, Gas88, AN96]. A good introduction to cryptography can be found, for example, in [MvOV96, GB99]. Applications of formal methods (specifically, the process algebra CSP) to verifying security protocols are explained in [RSG+01] (although this reference is not necessary as background because the concepts we need are defined in Sect. 3.3 and treated in more detail in Sect. 7.5).

Secure communication over untrusted networks requires specific mechanisms such as encryption and cryptographic protocols. A cryptographic protocol is a description of a message exchange, which includes cryptographic data, for establishing a secure relationship between the protocol participants (for example, a secure communication channel). Cryptographic protocols are very difficult to design and prone to very subtle errors. For example, in the famous Needham–Schroeder authentication protocol, a relatively simple protocol invented in 1978, there were problems with the protocol pointed out almost 20 years later [Low95].

¹ For the German-speaking audience, we also recommend [Eck03].
A security policy summarizes the protection requirements of a system. We recall below some important security requirements and concepts which will be considered in the course of this book.

**Fair Exchange**

When trading goods electronically, the *fair exchange* requirement postulates that the trade is performed in a way that prevents the participating parties from cheating. If for example buyer has to make a prepayment, the buyer should be able to prove having made the payment and to reclaim the money if that good is subsequently not delivered.

**Non-repudiation**

One way of providing fair exchange is by using the security requirement of *non-repudiation* of some action, which means that this action cannot subsequently be successfully denied. That is, the action is *provable*, usually with respect to some trusted third party.

**Role-based Access Control**

An important mechanism for controlling access to protected resources is the concept of *role-based access control*. In order to keep permissions manageable, especially in systems with a large or frequently changing user-base, they are not directly assigned to users. Instead, users can have one or more *roles* often related to their function within an organisation, and then permissions are assigned to roles.

**Secure Communication Link**

Sensitive communication between different parts of a system needs to be protected. The relevant requirement of a *secure communication link* is here assumed to preserve secrecy and integrity for the data in transit.

**Secrecy and Integrity**

Two of the main data security requirements are *secrecy* (or *confidentiality*, meaning that some information can be *read* only by legitimate parties) and *integrity* (some information can be *modified* only by legitimate parties).

**Authenticity**

There are different variants of this third main security requirement. Two important ones are message authenticity and entity authenticity. *Message authenticity* (or *data origin authenticity*) means that one can trace back some piece of data to what its original source was (at some point in the past). *Entity authenticity* ensures that one can identify a participant in a protocol (and in particular make sure that the party has actually actively participated in the protocol at the time). The process providing authenticity is called *authentication*.
3.2 Unified Modeling Language

**Freshness**

A message is fresh if it has been created during the current execution round of the system under consideration (for example, during the current protocol iteration) and therefore cannot be a replay of an older message by the adversary. A nonce is a random value that is supposed to be used only once (hence the name), for example to establish that a certain message containing a recently created nonce is itself freshly constructed.\(^1\)

**Secure Information Flow**

A traditional way of ensuring security in computer systems is to design multi-level secure systems [LB73]. In such systems, there are different levels of sensitivity of data. For simplicity, one usually considers two security levels, high (highly sensitive or highly trusted) and low (less sensitive or less trusted). Where trusted parts of a system interact with untrusted parts, one has to ensure that there is no indirect leakage of sensitive information from a trusted to an untrusted part. To ensure this, one enforces the “no down-flow” policy: low data may influence high data, but not vice versa. The opposite of this condition, “no up-flow”, ensures that untrusted parts of a system may not indirectly manipulate high data: high data may influence low data, but not vice versa. These security requirements, called secure information flow or non-interference [GM84], are rather stringent definitions of secrecy and integrity which can detect implicit flows of information that are called covert channels [Lam73].\(^1\)

**Guarded Access**

One of the main security mechanisms is access control, which ensures that only legitimate parties have access to a security-relevant part of the system. Sometimes, access control is enforced by guards: in the case of the Java Security Architecture, guard objects control access to protected objects; similarly for the access decision objects in CORBA.

### 3.2 Unified Modeling Language

UML [RJB99] is the de facto industry standard for specifying object-oriented, or more generally component-oriented, software systems. It is a graphical language that may be used to specify architectural and behavioral aspects of software. Good introductions to UML can be found in [FS97, SP00]. Here, we consider its current version [Obj03].

UML diagrams describe various views on different parts of a system design. There are several kinds of diagrams, describing different aspects of a system at varying degrees of abstraction. In this book, we use the following kinds:
Use case diagrams describe typical interactions between a user and a computer system. They are often used in an informal way for negotiation with a customer before a system is designed.

Class diagrams define the static class structure of the system: classes with attributes, operations, and signals and relationships between classes. On the instance level, the corresponding diagrams are called object diagrams.

Statechart diagrams (or state diagrams) give the dynamic behavior of an individual object or component: events may cause a change in state or an execution of actions. They are an adaptation of Harel's statecharts [HG97].

Sequence diagrams describe interaction between objects or system components via message exchange.

Activity diagrams specify the control flow between several components within the system, usually at a higher degree of abstraction than statecharts and sequence diagrams. They can be used to put objects or components in the context of overall system behavior or to explain use cases in more detail.

Deployment diagrams describe the physical layer on which the system is to be implemented.

Subsystems (a certain kind of packages) integrate the information between the different kinds of diagrams and between different parts of the system specification.

In addition to sequence diagrams, there are collaboration diagrams, which present similar information. Also, there are component diagrams, presenting part of the information contained in deployment diagrams.

For each kind of diagram, we will only need a relatively simple fragment of its various notational elements. In the following few subsections, we will informally explain only those features of the above kinds of diagrams which are needed in this book. There are many other diagram elements. Although they can also be used in the context of our approach, we will not need them in our presentation.

3.2.1 Use Case Diagrams

Use case diagrams can be used to represent interactions between a system and a user in an abstract way. Central elements are use cases and actors. The intention is that instances of use cases and instances of actors interact when the services of the described system are used. One can use other kinds of diagrams (such as activity diagrams or sequence diagrams) to specify this interaction in more detail. An actor has a name and defines a set of roles that users of a system can play when interacting with the system. The interactions are listed as use cases. A use case may be associated with the actors of the use case, meaning that an instance of the use case and a user playing one of the roles of the actor communicate. As with the other diagram kinds, there are many more model elements, and these could also be used with our approach (such as the extends and includes relationships).
An example of a use case is given in Fig. 3.1. There, a Customer actor is supposed to perform a buys good use case and a Business actor is supposed to perform a sells good use case.

![Use case diagram]

**Fig. 3.1.** Use case diagram

### 3.2.2 Class Diagrams

An object is an “entity with a well-defined boundary and identity that encapsulates state and behavior. State is represented by attributes and relationships, behavior is represented by operations, methods, and state machines. An object is an instance of a class” [Obj03, p. Glos.-10]. A class is a “description of a set of objects that share the same attributes, operations, methods, relationships, and semantics. A class may use a set of interfaces to specify collections of operations it provides to its environment” [Obj03, p. Glos.-4].

We use class diagrams to present the classes and their interfaces used in a system, together with their relationships, such as dependencies. A modeling element depends on another modeling element if a change to the latter might affect the former [SP00, p. 7]. In the diagrammatic notation (see Fig. 3.2), a class is represented by a rectangle with three compartments giving its name, its attributes, and its operations. Dependencies between classes are written as broken arrows with an open arrow-head. Interfaces are represented by a rectangle labeled «interface» containing the operations and signals offered by the interface (the interface specification), with a broken arrow with a closed head coming from the class implementing the interface. A label set in «» in a UML diagram is called a stereotype; see Sect. 3.2.8 for an explanation of the concept of UML stereotypes. As shorthand, one may omit the interface specification and instead write a circle attached to the class rectangle. A dependency arrow stereotyped «call» (resp. «send») from a class dep to a class indep indicates that instances of class dep may call operations of (resp. send signals to) instances of class indep. In particular, the instance of class dep knows of the instance of class indep. If the arrow points to an interface of indep, dep may only call the operations or send the signals listed in the corresponding interface specification. For example, in Fig. 3.2, Sender may send the signal transmit with argument d to Receiver, but an object accessing Receiver through the interface receiving would only be able to call the operation receive with no arguments (and get a return value of type Data).
3.2.3 Statechart Diagrams

UML statechart diagrams are used to describe state machines, which specify the sequences of states that an entity (such as an object or component) can go through in response to events, together with its responding actions [Obj03]. They are derived from the statecharts proposed by Harel [HG97].

Statechart diagrams consist of states and transitions between states. A state is “a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event” [Obj03, p. Glos.-14]. A transition with label \( e[q]/a \) indicates that an object in the first state will perform the action \( a \) and enter the target state when the event \( e \) occurs and the condition \( q \) (called guards) is satisfied (that is, the transition fires). The action could be to call an operation or send a signal (call(\( op(args) \)) resp. send(\( sig(args) \)) or to assign a value to an attribute (\( att := val \)). In the case of an operation call or a signal transmission, the keywords call() and send() are usually omitted from the diagram for readability; instead only the operation or signal name and the arguments are given.

Generally, the name of a message sent to an object \( obj \) has the name \( obj \) as a prefix; this may also be omitted if no confusion can arise.

A simple example is given in Fig. 3.3. The statechart consists of three states named “Wait”, “Request”, and “Send” (without substates, actions, or activities) and an initial state. At the start of the execution of the statechart, the “Wait” state is entered and as an entry action the attribute \( i \) incremented. When the message send arrives, its argument is stored in the variable \( d \), the message request is sent out, and the state “Request” is entered. Subsequently, when the return message is received, its two arguments are stored in the variables \( K \) and \( C \) and the state “Send” is entered. Then, if the condition \( \text{Ext}_{K,C}(C) = R : K \) holds (see Sect. 3.3.3 for an explanation of this condition), the message transmit(\( \{d : i\}_K \)) is sent and the state “Wait” is entered again.

States are indicated by boxes which contain the name of the state. They may contain entry (resp. exit) actions that are executed on entry (resp. exit) of the state. Transitions with the same source and target object may be internal.
which means that they are fired without invoking entry or exit actions and internal activities executed as long as the state is active. The intuition behind internal transitions is that they model a response to an event that does not change the state of an object. A state may be divided into sequential (resp. concurrent) substates and is then called a sequential composite state (resp. a concurrent composite state). When a sequential composite state is active, exactly one of its sequential substates is active. When a concurrent state is active, all of its concurrent substates are active. A statechart diagram and its subdiagrams contain each an initial state and may contain one or more final states (denoted by a solid circle and a circle containing a small solid circle, respectively). Final states may not be present in the case of non-terminating behavior (such as in Fig. 3.3).

### 3.2.4 Sequence Diagrams

A sequence diagram "shows object interactions arranged in time sequence. In particular, it shows the objects participating in the interaction and the sequence of messages exchanged. A sequence diagram can exist in a generic form (describes all possible scenarios) and in an instance form (describes one actual scenario)" [Obj03, p. Glos.-13].

Essentially, sequence diagrams give the interaction among a set of objects (or components), the names of which are given in the first line of the diagram. There are vertical lines down from each name, called life lines. When the object is active, this is signified by drawing a box rather than a line for that period of time down the sequence diagram. There are arrows (so-called connections) with attached messages between the life lines that specify that the attached message is supposed to be sent from the object from whose life-line the arrow emerges to the other object. For readability, the prefix obj of the name of a message sent to an object obj may be omitted, since it is implicit in the sequence diagram. For each method msg in the diagram and each number \( n \), \( \text{msg}_n \) represents the \( n \)th argument of the operation call \( \text{msg} \) that was most recently accepted according to the sequence diagram. To increase readability, we allow the definition of syntactic shorthands in the diagram using the operator \( :=. \) For example, \( k \) is written as a shorthand for the cryptographic
expression $\text{fst}(\text{Ext}_{K'}(\text{Dec}_{K'_C^{-1}}(c_k)))$ in Fig. 3.4 (see Sect. 3.3.3 for a definition of such cryptographic expressions). There may also be conditions associated with arrows (written in square brackets []) which have to be fulfilled when the diagram is executed, otherwise the execution does not proceed at the relevant arrow'. Note that the sender or receiver of a message may not be part of the sequence diagram. In that case, the arrows point into or out from the diagram.

An example of a sequence diagram is given in Fig. 3.4.

**Fig. 3.4.** Sequence diagram

#### 3.2.5 Activity Diagrams

An activity diagram (for an example see Fig. 3.5) is, technically (and according to UML 1.x), a special case of a statechart diagram that is used to model processes involving one or more objects or components, whose execution is coordinated by the activity diagram [Obj03, p. 3-156].

Concurrent composite states are written using synchronization bars, such that for each concurrent composite state $S$, transitions from the initial states of the concurrent substates of $S$ (resp. to the final states of the concurrent substates of $S$) are replaced by transitions from a synchronization bar (resp. to a synchronization bar) in the activity diagram, and the lines delineating the concurrent states are omitted. Activity diagrams can be partitioned into swimlanes, each carrying the name of the object (and its class) or component the behavior of which is modeled by the activities in the swimlane. We assume that the partition is well-defined in the sense that an activity in the swimlane labeled with the component $C$ only accesses the data in $C$. For readability,
we may omit the object name prefixes from attribute names, since they are given as the label of the swimlane.

States in activity diagrams can be of the following kinds. A state without internal activity, internal transitions, exit action, or outgoing non-completion transitions, and with at least one outgoing completion transition, is called an action state [Obj03, p. 2-171]. A state whose internal activity models the execution of a non-atomic sequence of steps that has some duration is called a subactivity state [Obj03, p. 2-174]. Action and subactivity states are written as boxes with straight top, and bottom, and convex arcs as sides.

3.2.6 Deployment Diagrams

A deployment diagram is a “diagram that shows the configuration of runtime processing nodes and the components, processes, and objects that live on them” [Obj03, p. Glos.-6].

An example is given in Fig. 3.6. The diagram contains two nodes, “client machine” and “server machine”: “client machine” contains a component “client apps” with interface “get_password”, which in turn contains an object “browser”; “server machine” contains a component “web server”, which
contains an object "access control". The two nodes are connected by a link
stereotyped «Internet», and there is a dependency from ‘web server’ to
"get_password" stereotyped «secrecy» [the latter stereotype is already part
of the UMLsec extension defined in Sect. 4.1.2].

Intuitively, links represent physical communication links between different
nodes in a system, while dependencies describe logical connections between
components. In the above example, the web server is specified to be able to
communicate with the browser (to request the password), and this is made
possible by an Internet connection.

3.2.7 Subsystems

A package is a notational means of simplifying the presentation of UML di-
agrams. One can group together parts of a model, represented by diagrams,
into a package. Then only the package symbol, and not the represented group
of diagrams, has to be shown in the overall diagram.

Here we make use of a specific kind of package called a subsystem (see
Fig. 3.7), which is a “grouping of model elements that represents a behavioral
unit in a physical system” [Obj03, p. Glos.-15]. A subsystem modeling the
complete system under consideration (rather than just a part) is called a
system.

3.2.8 UML Extension Mechanisms

UML offers three main “light-weight” language extension mechanisms: ste-
reotypes, tagged values, and constraints [Obj03] (we do not consider the
“heavy-weight” approach using meta-model extensions here). Stereotypes de-
fine new types of modeling elements extending the semantics of existing types
or classes in the UML meta-model. Their notation consists of the name of the
stereotype written in double angle brackets « », attached to the extended
model element. This model element is then interpreted according to the mean-
ing ascribed to the stereotype. The earlier restriction that at most one stereo-
type can be assigned to any model element has dropped since UML 1.4
[Obj03].

One way of explicitly defining a property is by attaching a tagged value
to a model element. A tagged value is a name-value pair, where the name is
referred to as the tag. The corresponding notation is \{tag = value\} with the
tag name \textit{tag} and a corresponding value to be assigned to the \textit{tag}. Tags can
define either data values (DataTags) or references to other model elements
(ReferenceTags). If the value is of type Boolean, one usually omits \{tag = \textit{false}\}, and writes \{tag\} instead of \{tag = \textit{true}\}.

Another way of adding information to a model element is by attaching
constraints to refine its semantics.
Stereotypes can be used to attach tagged values and constraints as pseudo-attributes of the stereotyped model elements. They are called pseudo-attributes because their semantics is outside the scope of the UML standard. All model elements labeled by a particular stereotype receive the corresponding values and constraints in addition to the attributes, associations, and superclasses that the element has in the standard UML (this usage is new from UML 1.4 [Obj03]).

To construct an extension of the UML one collects the relevant definitions of stereotypes, tagged values, and constraints into a profile [Obj03], which is a stereotyped package (alternatively, [CKM+99] suggests the use of prefaces). A profile:

- identifies a subset of the UML meta-model,
- gives “well-formedness rules”, that is a set of constraints, for this subset,
- gives a semantics in natural language beyond that given by the identified subset, and
- lists common model elements.
Examples for UML extensions include the UML Profile for Software Development Processes [Obi03], the UML Profile for Business Modeling [Obi03], and extensions for real-time [SR98], frameworks [FPR00], and software architectures [KS00].

3.3 Analyzing UML Models

In the definition of the UMLsec profile in Chap. 4, we need to formulate constraints on the UML models that model security requirements that can be rather subtle. To check them mechanically, one needs to refer to an analyzable model of the execution semantics of the used fragment of UML. We define what such a model should look like so we can use it for formalizing the constraints in the UMLsec profile. To demonstrate that this is possible, we present such a semantics for a (restricted and simplified) fragment of UML in Chap. 8. For security analysis, the security-relevant information from the security-oriented stereotypes is then incorporated.

In the following subsection, we define and explain those properties of this semantics that we need to define the UMLsec profile in Chap. 4. The actual formal semantics we use, and more explanation and examples, can be found in Chap. 8, based on the foundation of UML Machines introduced in Chap. 7. Note that at this point, there exists no official adopted formal semantics for UML. However, our approach also works without a formal semantics, by coding up the relevant definitions as part of the tool explained in Chap. 6.

3.3.1 Notation

We assume the usual definitions from elementary set theory and logic (which may be found for example in [AGM00]), including the following definitions.

\( \mathbb{N} \) is the set of non-negative integers, \( \mathbb{N}_0 \), the set of non-negative integers up to and including \( n \) (for any \( n \in \mathbb{N} \)). \( \mathcal{P}(X) \) is the set of subsets of a set \( X \).

Given a sequence (or list) \( l = (l_1, l_2, l_3, \ldots) \), we write \( \text{head}(l) \) for its head \( l_1 \) and \( \text{tail}(l) \) for its tail \( (l_2, l_3, \ldots) \). We write \([\ ]\) for the empty list (in particular for the empty string).

A multi-set (or bag) is a set which may contain multiple copies of an element, with notation \( \{ \} \) instead of the usual brackets. For example, \( \{1,1,1,1,1,1,1,1,1,1\} \) is the multi-set consisting of ten copies of the element 1. For two multi-sets \( M \) and \( N \), \( M \cup N \) denotes their union and \( M \setminus N \) the subtraction of \( N \) from \( M \). For a multi-set \( M \) and a set \( X \), we write \( M \setminus X \) for the multi-set of those elements in \( M \) (preserving their cardinalities) that are also elements of \( X \). We write \( M \subseteq N \) for two multi-sets \( M, N \) if \( M \setminus N = M \). We write \( |M| \) for the set of elements in the multi-set \( M \) and \( \sharp M \) for the number of elements in \( M \).
3.3.2 Outline of Formal Semantics

In UML, objects, and more generally system components, can communicate by exchanging messages from a given set Events. The arrival of such a message is called an event. They consist of the message name from a given set MsgNm, and possibly arguments to the message. Message names may be prefixed with object or subsystem instance names from a given set UMNames. The arguments are assumed to be elements of a given set Exp of expressions (an example for such a set is defined in Sect. 3.3.3). Each object or component may receive messages in an input queue and release messages to an output queue. Thus in our model, every object or subsystem instance \( O \) has associated multi-sets \( \text{inQu}_O \) (input queue) and \( \text{outQu}_O \) (output queue). Our formal semantics models sending a message \( msg = op(exp_1, ..., exp_n) \in \text{Events} \) from an object or subsystem instance \( S \) to an object or subsystem instance \( R \) as follows:

1. \( S \) places the message \( R.msg \) into its multi-set \( \text{outQu}_S \).
2. A scheduler distributes the messages from output queues to the intended input queues (while removing the message head); in particular, \( R.msg \) is removed from \( \text{outQu}_S \) and \( msg \) added to \( \text{inQu}_R \).
3. \( R \) removes \( msg \) from its input queue and processes its content.

In the case of operation calls, we also need to keep track of the sender to allow sending return signals. This way of modeling communication allows for a very flexible treatment; for example, we can modify the behavior of the scheduler to take account of knowledge on the underlying communication layer (for example, regarding security issues, see Sect. 3.3.4).

At the level of single objects, behavior is modeled using statecharts, or (in special cases such as protocols) sequence diagrams. The internal activities contained as states of these statecharts can again be defined each as a statechart, or alternatively, they can be defined directly.

Using subsystems, one can then define the behavior of a system component \( C \) by including the behavior of each of the objects or components directly contained in \( C \), and by including an activity diagram that coordinates the respective activities of the various components and objects.

Thus for each object or component \( C \) of a given system, our semantics defines a so-called UML machine \( [C] \) (which is a state machine that communicates with its environment using messages).

Specifically, the behavioral semantics \( [D] \) of a statechart diagram \( D \) models the run-to-completion semantics of UML statecharts. As a special case, this gives us the semantics for activity diagrams. Any sequence diagram \( S \) gives us the behavior \( [S,C] \) of each contained component \( C \).

Subsystems group together diagrams describing different parts of a system: a system component \( C \) given by a subsystem \( S \) may contain subcomponents \( C_1, ..., C_n \). These subcomponents may communicate through the communication links in the corresponding deployment diagram. On the semantical level,
each link has a corresponding link queue storing the messages that are exchanged along the link while in transit. The behavioral interpretation \([S]\) of \(S\) is defined as follows:

1. It takes a multi-set of input events.
2. The events are distributed from the input multi-set and the link queues connecting the subcomponents and given as arguments to the functions defining the behavior of the intended recipients in \(S\).
3. The output messages from these functions are distributed to the link queues of the links connecting the sender of a message to the receiver, or given as the output from \([S]\) when the receiver is not part of \(S\).

When performing security analysis, after the last step, the adversary model may modify the contents of the link queues in a certain way explained in Sect. 3.3.4.

### 3.3.3 Modeling Cryptography

We introduce some sets to be used in modeling cryptographic data in a UML specification and its security analysis.

We assume an infinite set \(\text{Keys}\) with a partial injective map \((\_)^{-1} : \text{Keys} \to \text{Keys}\). The elements in its domain (which may be public) can be used for encryption and for verifying signatures, those in its range (usually assumed to be secret) for decryption and signing. We assume that every key is either an encryption or decryption key, or both: any key \(k\) satisfying \(k^{-1} = k\) is called symmetric; the others are called asymmetric. We assume that the numbers of symmetric and asymmetric keys are both infinite. We fix infinite sets \(\text{Var}\) of variables and \(\text{Data}\) of data values. We assume that \(\text{Keys}, \text{Var}\), and \(\text{Data}\) are mutually disjoint and that the set \(\text{Data}\) contains the names: \(\text{UMNames}\cup\text{MsgNm}\subseteq\text{Data}\). \(\text{Data}\) may also include nonces (see Sect. 3.1) and other secrets.

The algebra of cryptographic expressions \(\text{Exp}\) is the quotient of the term algebra generated from the set \(\text{Var}\cup\text{Keys}\cup\text{Data}\) with the operations:

- \(\_::\_\) (concatenation)
- \(\text{head}(\_\) and \(\text{tail}(\_\)
- \(\{\_\) (encryption)
- \(\text{Dec}_\_\) (decryption)
- \(\text{Sign}_\_\) (signing)
- \(\text{Ext}_\_\) (extracting from signature)
- \(\text{Hash}_\_\) (hashing)

by factoring out the equations:

- \(\text{Dec}_{K^{-1}}(\{E\}_K) = E\) (for \(K \in \text{Keys}\))
- \(\text{Ext}_K(\text{Sign}_{K^{-1}}(E)) = E\) (for \(K \in \text{Keys}\))
- and the usual laws regarding concatenation, \(\text{head}()\), and \(\text{tail}()\):
- \((E_1 :: E_2) :: E_3 = E_1 :: (E_2 :: E_3)\) for all \(E_1, E_2, E_3 \in \text{Exp}\)
- \(\text{head}(E_1 :: E_2) = E_1\) and \(\text{tail}(E_1 :: E_2) = E_2\) for all expressions 
  \(E_1, E_2 \in \text{Exp}\) such that there exist no \(E, E'\) with \(E_1 = E :: E'\).

We write \(\text{fst}(E) \overset{\text{def}}{=} \text{head}(E)\), \(\text{snd}(E) \overset{\text{def}}{=} \text{head}(\text{tail}(E))\), and \(\text{thd}(E) \overset{\text{def}}{=} \text{head}(\text{tail}(\text{tail}(E)))\) for each \(E \in \text{Exp}\).

This symbolic model for cryptographic operations implies that we assume cryptography to be perfect, in the sense that an adversary cannot "guess" an encrypted value without knowing the decryption key. Also, we assume that one can detect whether an attempted decryption is successful. See for example [AJ01] for a formal discussion of these assumptions.

Note also that our model captures the fact that security-critical data such as keys and nonces are usually assumed to be independent; that is, that no equations should hold between them from which an adversary could derive information (such as \(K = K' + 1\) for two different keys \(K, K' \in \text{Keys}\)). This follows from the fact that the algebra of expressions is the quotient of a free algebra under the equations given above, in particular, only equations that follow from these equations hold in \(\text{Exp}\).

Based on this formalization of cryptographic operations, important conditions on security-critical data (such as freshness, secrecy, integrity, and authenticity) can then be formulated at the level of UML diagrams in a mathematically precise way (see Sect. 4.1).

In the following, we will often consider subalgebras of \(\text{Exp}\). These are subsets of \(\text{Exp}\) which are closed under the operations used to define \(\text{Exp}\) (such as concatenation, encryption, decryption, etc.). For each subset \(E\) of \(\text{Exp}\) there exists a unique smallest (with respect to subset inclusion) \(\text{Exp}\)-subalgebra \(\langle E \rangle\) containing \(E\), which we call \(\text{Exp}\)-subalgebra generated by \(E\). Intuitively, it can be constructed from \(E\) by iteratively adding all elements in \(\text{Exp}\) reachable by applying the operations used to define \(\text{Exp}\) above. It can be seen as the knowledge one can obtain from a given set \(E\) of data by iteratively applying publicly available operations to it (such as concatenation and encryption etc.) and will be used to model the knowledge an attacker may gain from a set \(E\) of data obtained for example by eavesdropping on Internet connections.

### 3.3.4 Security Analysis of UML Diagrams

In this section, we explain the security analysis machinery underlying the UMLsec approach as far as needed in the first half of this book. More details needed for the second half will be given in Sect. 7.5.

Our modular UML semantics allows a rather natural modeling of potential adversary behavior. We can model specific types of adversaries that can attack different parts of the system in a specified way. For example, an attacker of type insider may be able to intercept the communication links in a company-wide local area network. We model the actual behavior of the adversary by
defining a class of UML Machines that can access the communication links of the system in a specified way. To evaluate the security of the system with respect to the given type of adversary, we consider the joint execution of the system with any UML Machine in this class. This way of reasoning allows an intuitive formulation of many security properties. Since the actual verification is rather indirect this way, we also give alternative intrinsic ways of defining security properties below, which are more manageable, and show that they are equivalent to the earlier ones.

Thus for a security analysis of a given UMLsec subsystem specification $S$, we need to model potential adversary behavior. We model specific types of adversaries that can attack different parts of the system in a specified way. For this we assume a function $\text{Threats}_A(s)$ which takes an adversary type $A$ and a stereotype $s$ and returns a subset of \{$\text{delete, read, insert, access}$\} (abstract threats). These functions arise from the specification of the physical layer of the system under consideration using deployment diagrams, as explained in Sect. 4.1. For a link $l$ in a deployment diagram in $S$, we then define the set $\text{threats}_A^S(l)$ of concrete threats to be the smallest set satisfying the following conditions.

If each node $n$ that $l$ is contained in\(^2\) carries a stereotype $s_n$ with access $\in \text{Threats}_A(s_n)$ then:

- If $l$ carries a stereotype $s$ with delete $\in \text{Threats}_A(s)$ then delete $\in \text{threats}_A^S(l)$.
- If $l$ carries a stereotype $s$ with read $\in \text{Threats}_A(s)$ then read $\in \text{threats}_A^S(l)$.
- If $l$ carries a stereotype $s$ with insert $\in \text{Threats}_A(s)$ then insert $\in \text{threats}_A^S(l)$.
- If $l$ is connected to a node that carries a stereotype $t$ with access $\in \text{Threats}_A(t)$ then \{$\text{delete, read, insert}$\} $\subseteq \text{threats}_A^S(l)$.

The idea is that $\text{threats}_A^S(x)$ specifies the threat scenario against a component or link $x$ in the UML Machine System $A$ that is associated with an adversary type $A$. On the one hand, the threat scenario determines, which data the adversary can obtain by accessing components; on the other hand, it determines, which actions the adversary is permitted by the threat scenario to apply to the concerned links. delete means that the adversary may delete the messages in the corresponding link queue, read allows the adversary to read the messages in the link queue, and insert allows the adversary to insert messages in the link queue.

Then we model the actual behavior of an adversary $\bar{ad}$ of type $A$ as a type $A$ adversary machine. Essentially, this is a UML Machine which has the following data (we give a slightly simplified account here which is sufficient to introduce the UMLsec notation; advanced readers can find the complete technical details in Sect. 7.5)\(^2\)

- A set of states State with a control state control $\in$ State.
- A set of current adversary knowledge $K_A \subseteq \text{Exp}$.

\(^2\) Note that nodes and subsystems may be nested one in another.
3.3 Analyzing UML Models

- For each possible control state \( c \in \text{State} \) and set of knowledge \( K \subseteq \text{Exp} \), we have:
  - a set \( \text{Delete}_{c,K} \) which may contain the name of any link \( l \) with \( \text{delete} \in \text{threats}_{A}^{S}(l) \),
  - a set \( \text{Insert}_{c,K} \) which may contain any pair \( (l, E) \) where \( l \) is the name of a link with \( \text{insert} \in \text{threats}_{A}^{S}(l) \), and \( E \in K \), and
  - a set \( \text{newState}_{c,k} \subseteq \text{State} \) of states.

The machine is executed from a specified initial state \( \text{control} := \text{control}^{0} \) with a specified initial adversary knowledge \( K := K_{A}^{0} \) iteratively, where each iteration proceeds according to the following steps:

1. The contents of all link queues belonging to a link \( l \) with \( \text{read} \in \text{threats}_{A}^{S}(l) \) are added to \( K \).
2. The content of any link queue belonging to a link \( l \in \text{Delete}_{\text{control},K} \) is mapped to \( \emptyset \).
3. The content of any link queue belonging to a link \( l \) is enlarged with all expressions \( E \) where \( (l, E) \in \text{Insert}_{\text{control},K} \).
4. The next control state is chosen non-deterministically from the set \( \text{newState}_{\text{control},K} \).

The set \( K_{A}^{0} \) of initial knowledge of an adversary of type \( A \) is defined to contain the sets \( K_{A}^{0} \) and \( K_{A}^{0} \). Here, the set \( K_{A}^{0} \) of accessible knowledge contains all data values \( v \) given in the UML specification under consideration for which each node \( n \) containing \( v \) carries a stereotype \( s_n \) with \( \text{access} \in \text{Threats}_{A}(s_n) \).

In a given situation, the set \( K_{A}^{0} \) of previous knowledge can be used to give the adversary access to additional data supposed to be known before start of the execution of the system (for example, public encryption keys).

Note that an adversary \( A \) able to remove all values sent over the link \( l \) (that is, \( \text{delete} \in \text{threats}_{A}^{S}(l) \)) may not be able to selectively remove a value \( e \) with known meaning from \( l \). For example, the messages sent over the Internet within a virtual private network are encrypted. Thus, an adversary who is unable to break the encryption may be able to delete all messages indiscriminately, but not a single message whose meaning would be known to the adversary.

To evaluate the security of the system with respect to the given type of adversary, we then define the execution of the subsystem \( S \) in the presence of an adversary of type \( A \) to be the UML Machine \( [S]_{A} \) defined from \( [S] \) by applying the modifications from the adversary machine to the link queues as a fourth step in the definition of \( [S] \) as follows:

4. The type \( A \) adversary machine is applied to the link queues as detailed above.

Thus after each iteration of the system execution, the adversary may non-deterministically change the contents of link queues in a way depending on the level of physical security as described in the deployment diagram (see Sect. 4.1).
One possibility to specify security requirements is to define an idealized system model where the required security property evidently holds (for example, because all links and components are guaranteed to be secure by the physical layer specified in the deployment diagram), and to prove that the system model under consideration is behaviorally equivalent to the idealized one, using a notion of behavioral equivalence of UML models. This is explained in detail in Chap. 5.

In the following subsection, we consider particular ways of specifying the important security properties of secrecy, integrity, and authenticity which do not require one to explicitly construct such an idealized system and which are used in the remaining parts of this chapter.

### 3.3.5 Important Security Properties

The formal definitions of the two main security properties of secrecy and integrity considered in this subsection follow the standard approach of [DY83] and are defined in an intuitive way by incorporating the attacker model. We also explain how authenticity can be defined in terms of integrity, define a notion of freshness, and explain an alternative way of specifying secrecy- and integrity-like requirements.

#### Secrecy

The formalization of *secrecy* used in the following relies on the idea that a process specification preserves the secrecy of a piece of data \( d \) if the process never sends out any information from which \( d \) could be derived, even in interaction with an adversary. More precisely, \( d \) is leaked if there is an adversary of the type arising from the given threat scenario that does not initially know \( d \) and an input sequence to the system such that after the execution of the system given the input in presence of the adversary, the adversary knows \( d \) (where “knowledge”, “execution”, etc. have to be formalized). Otherwise, \( d \) is said to be kept secret.

Thus we come to the following definition.

**Definition 3.1.** We say that a subsystem \( S \) preserves the secrecy of an expression \( E \) from adversaries of type \( A \) if \( E \) never appears in the knowledge set \( K \) of \( A \) during execution of \( \llbrace S \rrbrace_A \).

This definition is especially convenient to verify if one can give an upper bound for the set of knowledge \( K \), which is often possible when the security-relevant part of the specification of the system \( S \) is given as a sequence of command schemata of the form *await event e - check condition q - output event e'*( for example, when using UML sequence diagrams or statecharts for specifying security protocols, see Sect. 4.1).

Note that this formalization of secrecy is relatively “coarse” in that it may not prevent implicit information flow, but it is comparatively easy to
verify and seems to be sufficient in practice [Aba00]. Also, it fits well with our formalization of cryptographic operations in Sect. 3.3.3: The encryption operations are modeled as deterministic, given a fixed key and a fixed plaintext. Although the basic algorithms for many cryptographic operations, such as RSA, are in fact deterministic, they are usually randomized when implemented in practice, for example by adding extra random data to the plaintext before encrypting it (so-called “salt” [GB99]). This is done to prevent a guessing attack where an adversary simply encrypts all possible plaintexts with the public encryption key and compares the result to the given ciphertext, which is possible if the set of possible plaintexts is small. In our formalization of cryptographic operations and adversary knowledge, we can abstract from this randomization: Those values that are required to be secret are assumed not to be contained in the adversary knowledge at the start of the system execution, so the adversary cannot use them in the guessing attack mentioned above. For values which are commonly known, but for which it should remain secret whether they are contained in a given encrypted message (such as the Boolean value true), one can define a new symbol in Data which represents this value in this message (for example, PIN\_correct, if this is what true should signify).

Examples

- The system that sends the expression \( \{m\}_K :: K \in \text{Exp} \) over an unprotected Internet link does not preserve the secrecy of \( m \) or \( K \) against attackers eavesdropping on the Internet, but the system that sends \( \{m\}_K \) (and nothing else) does, assuming that it preserves the secrecy of \( K \) against attackers eavesdropping on the Internet.
- A system \( S \) that receives a key \( K \) encrypted with the public key of \( S \) over a dedicated communication link and sends back \( \{m\}_K \) over the link does not preserve the secrecy of \( m \) against attackers eavesdropping on and inserting messages on the link, but does so against attackers that cannot insert messages on the link.

Integrity

The property integrity can be formalized similarly: if during the execution of the considered system, a system variable is assigned a value different from the ones it is supposed to be, then the adversary must have caused this variable to contain the value. In that sense the integrity of the variable is violated. Thus we say that a system preserves the integrity of a variable if there is no adversary such that at some point during the execution of the system in presence of the adversary, the variable has a value different from the ones it should have.

**Definition 3.2.** Given a set \( E \subseteq \text{Exp} \) of acceptable expressions, we say that a subsystem \( S \) preserves the integrity of an attribute \( a \) with respect to \( E \) from
adversaries of type A with initial knowledge $\mathcal{K}_A^0$ if during any execution of $\mathcal{S}_A$, at any point the attribute $a$ is undefined or evaluates to an element of $E$. If $E = \text{Exp} \setminus \mathcal{K}_A^0$, we simply say that $\mathcal{S}$ preserves the integrity of an attribute $a$ from adversaries of type A with initial knowledge $\mathcal{K}_A^0$.\footnote{new}  

Intuitively, this notion is “dual” to that of secrecy, in the sense that secrecy prevents the flow of information from protected sources to untrusted recipients, while integrity prevents the flow of information in the other direction. Again, it is a relatively simple definition, which may, however, not prevent implicit flows of information. For systems or system parts where, at a given point during the development, nothing is known about the values that $a$ should have, one can still use the above definition by setting $E = \text{Exp} \setminus \mathcal{K}_A^0$, where $\mathcal{K}_A^0$ is the initial knowledge of the adversary. Then no adversary can make a take on a value initially known to the adversary, which offers a certain degree of protection, since in many situations, if the adversary can violate the integrity of an attribute at all, he could in fact make it contain an arbitrary value.

**Authenticity**

To formalize message authenticity, we note that a message has its origin at a system part if during any execution of the system, the message appears at first at that part. To provide authenticity then means to secure the information on the message origin.

**Definition 3.3.** Suppose we are given attributes $a$ and $o$ in a subsystem $\mathcal{S}$, where $o$ is supposed to store the origin of the message stored in $a$. We say that $\mathcal{S}$ provides (message) authenticity of the attribute $a$ with respect to its origin $o$ from adversaries of type A with initial knowledge $\mathcal{K}_A^0$ if during any execution of $\mathcal{S}_A$, at any point the value of the attribute $a$ appeared first within the execution in $\text{out}_{\mathcal{Q}_o}$, of all output queues and link queues in $\mathcal{S}$.

Note that message authenticity is closely related to data integrity [MvOV96, p. 359], [Gol03c]. For example, if messages are communicated via a medium under control of an adversary data integrity necessitates message authenticity: if the adversary can remove a message from the communication medium and instead insert a different message successfully purporting to originate with the sender of the earlier message (thus breaking data integrity), message authenticity is violated. Differently expressed, to establish for message authenticity the origin of a specific message, it must actually be the message message that originated at the sender, which is only guaranteed where we have data integrity. Thus, message authenticity implies data integrity. If, however, the identity of the sender of a message is part of the message, integrity of the message implies the possibility to authenticate the sender. In this situation, data integrity implies message authenticity.

This observation can be made more precise.
Fact 3.4. Suppose we are given attributes $x$ and $o$ in a subsystem $S$, where $o$ is supposed to store the origin of the message stored in $x$. If $S$ provides (message) authenticity of the attribute $x$ with respect to its origin $o$ from adversaries of type $A$ with initial knowledge $K^0$, and the origin $o$ is not under control of the adversary, then $S$ preserves the integrity of $x$ from adversaries of type $A$ with initial knowledge $K^0$.

The proof of this fact is immediate from the definitions. Note that, as in the statement of this fact, one may need an additional assumption regarding the integrity of the origin $o$, if this is not a constant within $S$, because checking authenticity with respect to an identity without verifying integrity of that identity may provide little security.

Note also that the converse of the above fact does not hold since the integrity of a message may be provided although the recipient does not know its first origin. In a sense, integrity amounts to authenticity with respect to a non-specified part of the system under consideration.

For more discussions on the relation between message authenticity and data integrity see [Gol03c]. Also, contrary to secrecy, integrity, and message authenticity, the formalization of other kinds of authenticity (such as entity authenticity) seems to be more application-dependent. We therefore do not give a universal definition here (but refer for example to [Gol96, Gol03b] for formalizations of different authenticity properties).

Freshness

Note that freshness of a value may mean the following two properties:

Unpredictability: An attacker cannot guess what its value was.

Newness: The value has never appeared before during the execution of the system.

Both aspects can be considered with our approach: Unpredictability of data is captured by considering a type $A$ of adversary that does not include data in its set of previous knowledge $K^p_A$ (defined in Sect. 3.3.4). Freshness in the sense of newness requires an additional definition.

Definition 3.5. An atomic value data $\in Data \cup Keys$ in a subsystem $S$ is fresh within a subsystem instance or object $D$ contained in $S$ if the value data appears in the specification $S$ only in diagram parts specifying $D$ (the scope of data in $S$).

By the restrictions in Definition 3.5, we only consider freshness of atomic data $d \in Data \cup Keys$, not of compound expressions or variables. Note that, as mentioned in Sect. 3.3.3, different elements of $Data \cup Keys$ are independent. This is why it is sufficient to require of fresh values that they do not appear in the specification outside their scope, as in the above definition.

\[3\] Following a written communication by Gavin Lowe.
This definition implies that a value data that is fresh within a subsystem instance or object $D$ in a subsystem $S$ appears as a subexpression in the trace of messages exchanged within $S$ only after it has been sent out by $D$ as a message argument. See Sect. 7.3.4 for a formal argument supporting the last two observations.

Secure Information Flow

We explain an alternative way of specifying secrecy- and integrity-like requirements, which gives protection also against partial flow of information, but can be more difficult to deal with, especially when handling with encryption.

For this definition, one needs to assign to each piece of system data one of two security levels, high (highly sensitive or highly trusted) and low (less sensitive or less trusted), as explained in Sect. 3.1. The notion is defined by referring to the sequences or input and output values received and generated by the system, using the UML Machine $[S]_A$ defined in Sect. 3.3.4.

Given a set of messages $H$ and a sequence $m$ of event multi-sets, we write:

- $m^H$ for the sequence of event multi-sets derived from those in $m$ by deleting all events the message names of which are not in $H$, and
- $m_H$ for the sequence of event multi-sets derived from those in $m$ by deleting all events the message names of which are in $H$.

Definition 3.6. Given a subsystem $S$ and a set of high messages $H$, we say that:

- A prevents down-flow with respect to $H$ if for any two sequences $i, j$ of event multi-sets and any two output sequences $o \in [S]_A(i)$ and $p \in [S]_A(j)$, $i^H = j^H$ implies $o^H = p^H$ and
- A prevents up-flow with respect to $H$ if for any two sequences $i, j$ of event multi-sets and any two output sequences $o \in [S]_A(i)$ and $p \in [S]_A(j)$, $i^H = j^H$ implies $o^H = p^H$.

Intuitively, to prevent down-flow means that outputting a non-high (or low) message does not depend on high inputs (this can be seen as a secrecy requirement for messages marked as high). Conversely, to prevent up-flow means that outputting a high value does not depend on low inputs (this can be seen as an integrity requirement for messages marked as high).

This notion of secure information flow is a generalization of the original notion of non-interference for deterministic systems in [GM82] to system models that are non-deterministic because of underspecification, see [Jüri02] for a more detailed discussion.
Developing Secure Systems
Secure Systems Development with UML

We present the extension UMLsec of UML which allows one to express security-related information within the diagrams in a UML system specification. The extension is given in form of a UML profile using the standard UML extension mechanisms. Stereotypes are used together with tags to formulate security requirements and assumptions on the system environment; constraints give criteria that determine whether the requirements are met by the system design (by referring to the formal semantics from Chap. 8).

We list requirements on a UML extension for secure systems development and discuss how far our extension meets these requirements. We explain the details of the extension by means of examples.

We indicate with an example how one could use UMLsec in order to apply security patterns. We demonstrate the usefulness of the extension for enforcing established rules of secure systems design.

4.1 UMLsec Profile

For UMLsec, we give validation rules that evaluate a model with respect to listed security requirements. Many security requirements are formulated regarding the behavior of a system in interaction with its environment (in particular, with potential adversaries). To verify these requirements, we use the formal semantics defined in Chap. 8.

4.1.1 Requirements on a UML Extension for Development of Security-Critical Systems

We formulate what we consider the necessary properties of an UML extension for secure systems development. Following the format of the OMG Requests for Proposals (RFPs) we distinguish mandatory and optional requirements.
**Mandatory Requirements**

The following are the main mandatory requirements:

- **Security requirements:** One needs to be able to formulate basic security requirements such as secrecy, integrity, and authenticity of data in a precise way.
- **Threat scenarios:** It should be possible to consider various situations that give rise to different possibilities of attacks.
- **Security concepts:** One should be able to employ important security concepts (for example, that of tamper-resistant hardware).
- **Security mechanisms:** One needs to be able to incorporate security mechanisms such as access control and security protocols.
- **Security primitives:** On a more fine-grained level, one needs to model security primitives such as symmetric and asymmetric encryption.
- **Underlying physical security:** It is necessary to take into account the level of security provided by the underlying physical layer.
- **Security management:** Security management questions (such as secure workflow) need to be addressed.

**Optional Requirements**

It would be very useful to include domain-specific security knowledge (for example, on Java, smart cards, CORBA, etc.).

### 4.1.2 The Extension

We state the profile following the structure in [Obj03].

Note that the goal of the extension is not to include all kinds of security properties as primitives. Instead, we focus on those that have a comparatively intuitive and universally applicable formalization (such as secrecy and integrity). Other properties (such as entity authenticity) have meanings that depend more on the context of their specific use. The idea is that these could be added by more sophisticated users on-the-fly.

**Applicable Subset**

The profile concerns all of UML.

**Stereotypes, Tagged Values, and Constraints**

In Fig. 4.1 we give the list of stereotypes from UMLsec, together with their tags and constraints, following the notation used in [Obj03, p. 3-59]. The stereotypes do not have parents. Fig. 4.2 gives the corresponding tags (which are all DataTags). Again, for simplicity we stay on the instance level in the following; in particular by “subsystem” we mean “subsystem instance” in each case.
<table>
<thead>
<tr>
<th>Stereotype</th>
<th>Base Class</th>
<th>Tags</th>
<th>Constraints</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fair exchange</td>
<td>subsystem</td>
<td>start, stop,</td>
<td>after start eventually reach stop</td>
<td>enforce fair exchange</td>
</tr>
<tr>
<td></td>
<td></td>
<td>adversary</td>
<td></td>
<td></td>
</tr>
<tr>
<td>provable</td>
<td>subsystem</td>
<td>action, cert,</td>
<td>action is non deniable</td>
<td>non repudiation requirement</td>
</tr>
<tr>
<td></td>
<td></td>
<td>adversary</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rbac</td>
<td>subsystem</td>
<td>protected,</td>
<td>only permitted activities executed</td>
<td>enforces role-based access control</td>
</tr>
<tr>
<td></td>
<td></td>
<td>role, right</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Internet</td>
<td>link</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>encrypted</td>
<td>link</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAN</td>
<td>link, node</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wire</td>
<td>link</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>smart card</td>
<td>node</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>POS device</td>
<td>node</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>issuer node</td>
<td>node</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>secure links</td>
<td>subsystem</td>
<td>adversary</td>
<td>dependency security matched by links</td>
<td>enforces secure communication links</td>
</tr>
<tr>
<td>secrecy</td>
<td>dependency</td>
<td></td>
<td></td>
<td>assumes secrecy</td>
</tr>
<tr>
<td>integrity</td>
<td>dependency</td>
<td></td>
<td></td>
<td>assumes integrity</td>
</tr>
<tr>
<td>high</td>
<td>dependency</td>
<td></td>
<td></td>
<td>high sensitivity</td>
</tr>
<tr>
<td>secure dependency</td>
<td>subsystem</td>
<td>secret,</td>
<td>«call», «send» respect data security</td>
<td>structural interaction data security</td>
</tr>
<tr>
<td>critical</td>
<td>object,</td>
<td>secrecy,</td>
<td></td>
<td>critical object</td>
</tr>
<tr>
<td></td>
<td>subsystem</td>
<td>integrity,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>authenticity,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>high, fresh</td>
<td></td>
<td></td>
</tr>
<tr>
<td>no down-flow</td>
<td>subsystem</td>
<td></td>
<td>prevents down-flow</td>
<td>information flow condition</td>
</tr>
<tr>
<td>no up-flow</td>
<td>subsystem</td>
<td></td>
<td>prevents up-flow</td>
<td>information flow condition</td>
</tr>
<tr>
<td>data security</td>
<td>subsystem</td>
<td>adversary, integrity, authenticity, freshness</td>
<td>provides secrecy, integrity, authenticity, freshness</td>
<td>basic data security requirements</td>
</tr>
<tr>
<td>guarded access</td>
<td>subsystem</td>
<td></td>
<td>guarded objects accessed through guards</td>
<td>access control using guard objects</td>
</tr>
<tr>
<td>guarded</td>
<td>object</td>
<td>guard</td>
<td></td>
<td>guarded object</td>
</tr>
</tbody>
</table>

Fig. 4.1. UMLsec stereotypes
Prerequisite Profiles

UMLsec requires no prerequisite profiles.

Well-formedness Rules

We explain the stereotypes and tags given in Figures 4.1 and 4.2. The
constraints use the security-aware interpretation of UML diagrams (formally
defined in Sect. 8.1.7). Some of them are parameterized over the adversary type
with respect to which the security requirements should hold and which may
be given in the diagram.

By their nature, some of the constraints can be enforced at the level of
abstract syntax (such as «secure links»), while others refer to the formal
semantics in Sect. 3.3.2 (such as «no down-flow»). Note that even checking the
latter can be mechanized given appropriate tool support (see Chap. 6).

We give short examples for usage of the stereotypes. To keep the
presentation concise, we sometimes give only those fragments of (instances of)
subsystems that are essential to the stereotype in question. Also, we omit
proving the stated properties formally, since the examples are just for illus-
tration. More substantial examples can be found in Chap. 5.

fair exchange (for use case diagrams)

Intuitively, this stereotype represents the security requirement that any trans-
action should be performed in a way that prevents both parties from cheating.
When applied to a subsystem containing a use case diagram, it requires that this subsystem can be refined by another subsystem only if that is also stereotype «fair exchange». Note that this usage of the «fair exchange» stereotype has only an informal meaning, as opposed to the stereotypes below (in particular, “refinement” is meant here in an informal sense). It just serves as an example, how the security requirements included as stereotypes in the other kinds of diagrams below can also conveniently be included in use case diagrams.

Example For instance, the use case diagram in Fig. 4.3 describes the following situation: a customer buys a good from a business in a way that is supposed to ensure «fair exchange». The diagram can be refined to the activity diagram below, because the latter is also stereotyped «fair exchange», which in that context has a more specific constraint associated, which is also explained below.

![Use case diagram for business application](image.png)

**Fig. 4.3. Use case diagram for business application**

*fair exchange (for activity diagrams)*

This stereotype, when applied to subsystems containing an activity diagram, has associated tags {start} and {stop} taking names of states as values and a tag {adversary} specifying an adversary type relative to which the security requirement should hold. The associated constraint requires that, whenever a {start} state in the contained activity diagram is reached, then eventually a {stop} state will be reached, given the adversary type A that is specified in the tag {adversary} (thus, formally, this security requirement comes down to a liveness property). This is formalized for a given subsystem S as follows. S fulfills the constraint of «fair exchange» if for every adversary adv of type A and every sequence of event multi-sets I_{n+1},...,I_n, there exists an m ∈ N with m ≥ n such that for all I_{n+1},...,I_m and for each (O_1,...,O_m) ∈ Exec([S]_{adv} | state)](I_1,...,I_m), there are at least as many {stop} states in ∪_{i=1,...,m} O_i as there are {start} states.

Note that this requirement cannot be ensured for systems which an attacker can stop completely, and that only one kind of good is concerned.
Example Figure 4.4 gives a subsystem instance describing the following situation: a customer buys a good from a business. Here the adversary type is omitted because it is not relevant, since no communication structure is specified. The semantics of the stereotype «fair exchange» is, intuitively, that the actions listed in the tags {start} and {stop} should be linked in the sense that if one of the former is executed then eventually one of the latter will be.

![Purchase activity diagram](image)

Fig. 4.4. Purchase activity diagram

This would entail that, once the customer has paid, either the order is delivered to the customer by the due date, or the customer is able to reclaim the payment on that date. To avoid illegitimate repayment claims, one could employ the stereotype «provable» with regard to the state Pay, in order to make sure that the Reclaim payment action checks whether the Customer can provide proof of payment (this is omitted here).

provable

A subsystem instance $S$ may be labeled «provable» with associated tags {action}, {cert}, and a tag {adversary} specifying an adversary type relative to which the security requirement should hold. This specifies that $S$ may output the expression $E \in \textbf{Exp}$ given in {cert} (which serves as proof that the action at state {action} was performed) only after the state the name of which is given in {action} is reached, given the adversary type $A$ that is specified in the tag {adversary}. Here the certificate in {cert} is assumed to be unique for each subsystem instance. More formally, $S$ fulfills the constraint if the following holds. Suppose we are given a sequence of event
4.1 UMLsec Profile

multi-sets \( I_1, \ldots, I_k \), an adversary \( adv \) of type \( A \), and a sequence \( (O_1, \ldots, O_k) \) output by \( S \) when executed with an adversary \( adv \) on input of \( (I_1, \ldots, I_k) \), and \( (S_1, \ldots, S_k) \) is the corresponding sequence of executed states (that is, \( ((S_1, O_1), \ldots, (S_k, O_k)) \in \text{Exec}(\mathcal{S}_{adv})(\text{currState}_{outQ_{\text{init}}}) \). Then the following implication is required to hold: if there exists an \( i \) such that the output \( O_i \) equals the expression in \{certificate\}, then we have \( j \leq i \) such that the state multi-set \( S_j \) contains the state specified by \text{action}.

rbac

This stereotype of subsystem instances containing an activity diagram enforces role-based access control in the business process specified in the activity diagram (see Sect. 3.1). It has associated tags \{protected\}, \{role\}, and \{right\}. The tag \{protected\} has as its values the activities in the activity diagram the access to which should be controlled. The \{role\} tag may have as its value a list of pairs \( (actor, role) \) where \( actor \) is an actor in the activity diagram, and \( role \) is a role. The tag \{right\} has as its value a list of pairs \( (role, right) \) where \( role \) is a role and \( right \) represents the right to access a protected resource. The associated constraint requires that the actors in the activity diagram only perform activities for which they have the appropriate rights. For a subsystem \( S \), this is formalized as follows: For every actor \( A \) in \( S \) and every activity \( a \) in the swim-lane of \( A \) in the activity diagram in \( S \), there exists a role \( R \) such that \((A, R)\) is a value of \{role\} and \((R, a)\) is a value of \{right\}.

Example Figure 4.5 gives a subsystem instance for an example of the use of role-based access control. It describes a simplified part of a business process where a credit is being set up for a customer of a bank. Usually, there are bank employees who have the right to set up credits. In the case of large credits (in the example diagram, those exceeding the amount of 10,000), their supervisors have to authorize the credit before the money is transferred. In the example given, the protected resource is thus the authorize credit activity, to which the supervisor in her role of credit approver has the appropriate permission, so the diagram is correctly labeled «rbac» because the associated constraint is respected.

Incidentally, this example is an instance of the security principle of separation of privilege (see also Sect. 4.2). Note that one also needs to make sure that a given employee is not assigned two roles with associated privileges that are supposed to be separated (for example as a vacation substitute), see Sect. 6.3. How to link access control to the level of the technical security architecture is demonstrated using the stereotype «guarded access» introduced below.

Internet, encrypted, LAN, wire, smart card, POS device, issuer node

These stereotypes on links (resp. nodes) in deployment diagrams denote the respective kinds of communication links (resp. system nodes). We require that each link or node carries at most one of these stereotypes. For each adversary type \( A \), we have a function \( \text{Threats}_A(s) \) from each stereotype
Granting a credit <<rbac>> $\Lambda$

- role=(supervisor, credit approver)
- right=(credit approver, authorize credit)
- protected="authorize credit"

![Diagram of role-based access control example]

**Fig. 4.5.** Role-based access control example

$s \in \{\text{"wire"}, \text{"encrypted"}, \text{\ll AN\rr}, \text{\ll smart card\rr},$

\text{\ll POS device\rr}, \text{\ll issuer node\rr}, \text{\ll Internet\rr}\}$

to a set of strings $\text{Threats}_A(s) \subseteq \{\text{delete, read, insert, access}\}$ under the following conditions:

- for a node stereotype $s$, we have $\text{Threats}_A(s) \subseteq \text{\{access\}}$, and
- for a link stereotype $s$, we have $\text{Threats}_A(s) \subseteq \{\text{delete, read, insert}\}$.

Thus $\text{Threats}_A(s)$ specifies which kinds of actions an adversary of type $A$ can apply to node or links stereotyped $s$. The meanings of the actions are explained in Sect. 3.3.4.

Given a UML subsystem (instance) $\mathcal{S}$ with associated UML Machine System $\mathcal{A} = [\mathcal{S}]$ (introduced in Sect. 3.3.2 and formally defined in Sect. 8.1.7), the function $\text{Threats}_A(s)$ gives rise to a function $\text{threats}_A^\mathcal{A}(x)$ of the kind introduced in Sect. 7.5 that takes a UML Machine or link $x \in \text{int}_A \cup \text{lks}_A$ and a type of adversary $A$ and returns a set of strings $\text{threats}_A^\mathcal{A}(x) \subseteq \{\text{delete, read, insert, access}\}$, recursively as follows:

- For a UML Machine $i \in \text{int}_A$ and a type of adversary $A$ we have $\text{threats}_A^\mathcal{A}(i) = \{\text{access}\}$ if the object $O$ contained in $\mathcal{S}$ that gives rise to $i$ lives on a node $n$ such that $n$ and each node in which $n$ is (recursively) contained each carry a (possibly different) stereotype $s$ with $\text{access} \in \text{Threats}_A(s)$. Otherwise, we have $\text{threats}_A^\mathcal{A}(i) = \emptyset$. 


• For a link $l \in \mathit{lks}_A$ for which the corresponding link $l'$ in the UML subsystem (instance) $S$ carries the stereotype $s$, and a type of adversary $A$, we have $\mathit{threats}_A^S(l) = \mathit{threats}_A(l)$ if all the nodes in the UML subsystem that contain the link $l'$ each carry a (possibly different) stereotype $t$ with $\mathit{access} \in \mathit{threats}_A(t)$. Otherwise, we have $\mathit{threats}_A^S(l) = \emptyset$. If for a link $l \in \mathit{lks}_A$ there is no corresponding link in the UML subsystem, which implies that $l$ connects the interpretations of subsystems or objects residing on the same node $n$, then $\mathit{threats}_A^S(l) = \{ \text{delete, read, insert} \}$ if $n$ and each node in which $n$ is (recursively) contained each carry a (possibly different) stereotype $s$ with $\mathit{access} \in \mathit{threats}_A(t)$. Otherwise, we have $\mathit{threats}_A^S(l) = \emptyset$.

This way we can evaluate UML subsystems instances with their formal semantics in Sect. 8.1.7, by referring to the security framework using UML Machine Systems in Sect. 7.5. We make use of this for the constraints of the remaining stereotypes of the profile.

Examples for threat sets associated with some common adversary types are given in Figures 4.6 and 4.7.

Figure 4.6 gives the default attacker, which represents an outsider adversary with modest capability. This kind of attacker is able to read and delete the messages on an Internet link and to insert messages. On an encrypted Internet link (for example, a virtual private network), the attacker can delete the messages (without knowing their encrypted content), but not to read the (plaintext) messages or to insert messages (that are encrypted with the right key). Of course, this assumes that the encryption is set up in a way such that the adversary does not get hold of the secret key (for an example, see Sect. 5.1). The default attacker is assumed not to have direct access to the local area network (LAN) and therefore not to be able to eavesdrop on those connections,\(^1\) nor on wires connecting security-critical devices (for example, a smart card reader and a display in a POS device). Also, smart cards are assumed to be tamper-resistant against default attackers (although they may not be against more sophisticated attackers [AK96]). Also, the default attacker is not able to access POS devices or card issuer systems (such a situation is considered in Sect. 5.3).

Figure 4.7 defines the insider attacker (in the context of the electronic purse system considered in Sect. 5.3). As an insider, the attacker may access the encrypted Internet link (the assumption is that insiders know the corresponding key) and the local system components.

secure links

This stereotype, which may label (instances of) subsystems, is used to ensure that security requirements on the communication are met by the physical layer, given the adversary type $A$ that is specified in the tag \{adversary\} as

\(^1\) With more sophistication, even an external adversary may be able to access local connections, but this is assumed to be beyond “default” capabilities.
Fig. 4.6. Threats from the default attacker

<table>
<thead>
<tr>
<th>Stereotype</th>
<th>Threats\textsubscript{default}()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internet</td>
<td>(delete, read, insert)</td>
</tr>
<tr>
<td>encrypted</td>
<td>(delete)</td>
</tr>
<tr>
<td>LAN</td>
<td>(delete, read, insert)</td>
</tr>
<tr>
<td>wire</td>
<td>(delete, read, insert)</td>
</tr>
<tr>
<td>smart card</td>
<td>(delete, read, insert)</td>
</tr>
<tr>
<td>POS device</td>
<td>(access)</td>
</tr>
<tr>
<td>issuer node</td>
<td>(access)</td>
</tr>
</tbody>
</table>

Fig. 4.7. Threats from the insider attacker card issuer

new

sociated with this stereotype. More precisely, the constraint enforces that for each dependency \( d \) with stereotype \( s \in \{\text{"secrecy"}, \text{"integrity"}, \text{"high"}\} \) between subsystems or objects on different nodes \( n, m \), we have a communication link \( l \) between \( n \) and \( m \) with stereotype \( t \) such that:

- in the case of \( s = \text{"high"} \), we have \( \text{Threats}_A(t) = \emptyset \),
- in the case of \( s = \text{"secrecy"} \), we have \( \text{read} \notin \text{Threats}_A(t) \), and
- in the case of \( s = \text{"integrity"} \), we have \( \text{insert} \notin \text{Threats}_A(t) \).

Example We give an example concerning communication link security in Fig. 4.8. Given the default adversary type, the constraint for the stereotype \text{"secure links"} is violated. The model does not provide communication secrecy against the default adversary, because the Internet communication link between web server and client does not provide the needed security level according to the \text{Threats}_{\text{default}}(\text{Internet}) scenario. Intuitively, the reason is that Internet connections do not provide secrecy against default adversaries. Technically, the constraint is violated because the dependency carries the stereotype \text{"secrecy"}, but for the stereotype \text{"Internet"} of the corresponding link we have \( \text{read} \in \text{Threats}_{\text{default}}(\text{Internet}) \).

secrecy, integrity, high

These stereotypes, which may label dependencies in static structure or component diagrams, denote dependencies that are supposed to provide the respective security requirement for the data that is sent along them as arguments
or return values of operations or signals. These stereotypes are used in the constraint for the stereotype «secure links».

secure dependency

This stereotype, used to label (instances of) subsystems containing static structure diagrams, ensures that the «call» and «send» dependencies between (interfaces of) objects or subsystems respect the security requirements on the data that may be communicated across them, as given by the tags {secrecy}, {integrity}, and {high} of the stereotype «critical». More exactly, the constraint enforced by this stereotype is that if there is a «call» or «send» dependency from an object (or subsystem) $C$ to an interface $I$ of an object (or subsystem) $D$ then the following conditions are fulfilled:

- For any message name $n$ in $I$, $n$ appears in the tag {secrecy} (resp. {integrity} resp. {high}) in $C$ if and only if it does so in $D$.
- If a message name in $I$ appears in the tag {secrecy} (resp. {integrity} resp. {high}) in $C$ then the dependency is stereotyped «secrecy» (resp. «integrity» resp. «high»).

If the dependency goes directly to another object (or subsystem) without involving an interface, the same requirement applies to the trivial interface containing all messages of the server object.

Example Figure 4.9 shows a key generation subsystem instance with the requirement «secure dependency». The given specification violates the constraint for this stereotype, since Random generator and the «call» dependency do not provide the security levels for random() required by Key generator. More precisely, the constraint is violated, because the message random is required to be of high level by Key generator (by the tag {high} in Key generator), but it is not guaranteed to be high level by Random generator (in fact there are no high messages in Random generator and so the tag {high} is missing).
**Fig. 4.9.** Key generation subsystem instance

**critical**

This stereotype labels objects or subsystem instances containing data that is critical in some way, which is specified in more detail using the corresponding tags. These tags are \{secrecy\}, \{integrity\}, \{authenticity\}, \{fresh\}, and \{high\}, representing the corresponding security requirements which were introduced in Sections 3.1 and 3.3. The values of the first two are the names of expressions or variables (that is, attributes or message arguments) of the current object the secrecy (resp. integrity) of which is supposed to be protected. The values of the tag \{authenticity\} are pairs \((a, o)\) of attributes of the «critical» object or subsystem where \(a\) stores the data whose authenticity should be provided and \(o\) stores the origin of that data. The tag \{fresh\} has as its values atomic data (that is, elements of the set \({\text{Data}}\cup{\text{Keys}}\)) that should be freshly generated. These constraints are enforced by the constraint of the stereotype «data security» which labels (instances of) subsystems that contain «critical» objects (see the paragraph below about «data security» for an explanation). The tag \{high\} has the names of messages as values that are supposed to be protected with respect to secure information flow, as enforced by the stereotypes «no down-flow» and «no up-flow». For each of the above tags, one may specify an array \(K\) to be a value of the tag as a shortcut for specifying each field \(K_x\) of the array to be a value of that tag.

**no down-flow, no up-flow**

These stereotypes of (instances of) subsystems enforce secure information flow by making use of the tag \{high\} associated with «critical». More precisely, the constraint for «no down-flow» (resp. «no up-flow») is that the UML machine \(\text{Exec}(S)\) for the subsystem \(S\) prevents down-flow (resp. up-flow) with respect to the messages and their return messages specified in \{high\}, as defined in Sect. 3.3.4 (or more formally in Definition 7.39).
Example The example in Fig. 4.10 shows a bank account data object that allows its secret balance to be read using the operation rb() (whose return value is also secret) and written using wb(x). If the balance is over 10,000, the object is in a state ExtraService, otherwise in NoExtraService. The state of the object can be queried using the operation rx(). The data object is supposed to be prevented from indirectly leaking out any partial information about high data via non-high data, as specified by the stereotype «no down-flow». For example, in a situation where government agencies can request information about the existence of bank accounts of a given person, but not their balance, it may be important that the type of the account allows no conclusion about its balance. The given specification violates the constraint associated with «no down-flow», since partial information about the input of the high operation wb() is leaked out via the return value of the non-high operation rx(). To see how the underlying formalism captures the security flaw using Definition 3.6, it is sufficient to exhibit sequences i, j of event multi-sets and output sequences o ∈ [A]i and p ∈ [A]j of the UML Machine A giving the behavior of the considered statechart, with iH = jH and oH ≠ pH, where H is the set of high messages. Consider the sequences i  def = (wb(0), rx()) and j  def = (wb(10000), rx()). We have iH = ({}, {rx()}) = jH. From the definition of the behavioral semantics of statecharts in Sect. 8.1.3, we can see that o  def = ({}; return(false)) ∈ [A](i) and p  def = ({}; return(true)) ∈ [A](j). But then oH = ({}; return(false)) ≠ ({}; return(true)) = pH, as required.

<table>
<thead>
<tr>
<th>Bank account «no down−flow»</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>rb</strong>: Data</td>
</tr>
<tr>
<td><strong>wb(x)</strong>: Data</td>
</tr>
<tr>
<td><strong>rx</strong>: Boolean</td>
</tr>
<tr>
<td><strong>balance</strong>: Integer</td>
</tr>
</tbody>
</table>

**Fig. 4.10. Bank account data object**

data security

This stereotype labeling (instances of) subsystems has the following constraint. The subsystem behavior respects the data security requirements given
by the stereotypes «critical» and the associated tags contained in the subsystem, with respect to the threat scenario arising from the deployment diagram and given the adversary type $A$ that is specified in the tag {adversary} associated with this stereotype.

More precisely, the constraint is given by the following four conditions (which use the concepts of secrecy, integrity, authenticity, and freshness defined in Sect. 3.3):

- **Secrecy:** The subsystem preserves the secrecy of the data designated by the tag {secr ety} against adversaries of type $A$.
- **Integrity:** The subsystem preserves the integrity of any data $d$ designated by the tag {integrity} of «critical» against adversaries of type $A$, with respect to the sets $E$ of admissible expressions given by a value $(d, E)$ of the {integrity} tag of «data security» (or with respect to the set $\mathbb{Exp} \setminus \mathbb{K}_A^0$, where $\mathbb{K}_A^0$ is the initial knowledge of the adversary, if {integrity} of «data security» has no such value).
- **Authenticity:** For any value $(a, o)$ of the tag {authenticity}, the subsystem provides the authenticity of the attribute $a$ with respect to its origin $o$ against adversaries of type $A$.
- **Freshness:** Within the subsystem $S$ stereotyped «data security», any value $data \in \text{Data} \cup \text{Keys}$ which is tagged {fresh} in the relevant subsystem instance or object $D$ stereotyped «critical» in $S$ should be fresh in $D$.

In each case, the initial knowledge of the adversary (see Sect. 3.3.4) is assumed not to contain the data values that according to the tags of the stereotype «critical» should be guaranteed secrecy, integrity or authenticity, because these requirements cannot be achieved if the adversary already knows them initially. Further assumptions on the initial adversary knowledge can be specified.

Note that it is enough for data to be listed with a security requirement in one of the objects or subsystem instances contained in the subsystem to be required to fulfill the above conditions. Note also that several nested subsystems may each carry the stereotype «data security», such that the above conditions are required to hold with respect to each of them (this is important to note when including one subsystem in another).

**Example** The example in Fig. 4.11 shows the specification of a variant of TLS proposed in [APS99]. The client initiates the protocol by sending a self-signed certificate to the server. The sender returns the encrypted session key together with the certificate $Sign_{K_{CA}}(S :: K_S)$ certifying authenticity of the server public key. The client finally sends the secret encrypted under the session key. As defined in Sect. 3.3.3, $\{M\}_K$ is the encryption of the message $M$ with the key $K$, $Sign_K(M)$ is the signature of the message $M$ with $K$, and :: denotes concatenation. Also, $Dec_K(C)$ is the decryption of the ciphertext $C$ using $K$ and $Ext_K(S)$ is the extraction of the data from the signature using $K$. We recall that for each method $msg$ in the diagram and each number $n$, $msg_n$.
TLS variant «data security»

Client «critical»
- secrecy={s,Kc} 
- freshness={N,c}
- Integrity={s,Nc,Kc,Kc,Ca,i}

S:Server
- secrecy={Kc,i} 
- freshness={i}
- Integrity={s,Kc,Kc,Ca,i}

resp(shrd:Exp, cert:Exp)

Server «critical»
- secrecy={Kc,i} 
- freshness={i}
- Integrity={s,Kc,Kc,Ca,i}

C:Client
- secrecy={s,Kc} 
- freshness={N,c}
- Integrity={s,Nc,Kc,Kc,i}

resd(mstr:Exp)

tls.C tls.S

entry/i:=i+1
entry/j:=j+1

TLS variant «data security»

[1] init(n:Data, k:Key, cert:Exp)

C:Client
- secrecy={s,Kc} 
- freshness={N,c}
- Integrity={s,Nc,Kc,Kc,i}

S:Server
- secrecy={Kc,i} 
- freshness={i}
- Integrity={s,Kc,Kc,Ca,i}

resp(shrd:Exp, cert:Exp)

entry/i:=i+1
entry/j:=j+1

tls.C tls.S

entry/i:=i+1
entry/j:=j+1

Fig. 4.11. TLS protocol variant
represents the n-th argument of the operation call msg that was most recently accepted according to the sequence diagram. Assuming the default adversary type, the specification violates the constraint of its stereotype «data security» that the «critical» values have their «secrecy»; «integrity»; and «authenticity» requirements fulfilled. Recall from Sect. 4.1.2 that the requirements «secrecy», «integrity», and «authenticity» refer to the type of adversary under consideration. In this example, the default adversary has access to the Internet link between the two nodes only. This adversary does not have direct access to any of the components in the specification (this would have to be specified explicitly using the Threats() function). If he did, then secrecy, integrity, and authenticity would fail because the adversary could read and modify the critical values directly as attributes of C and S. More details are given in Sect. 5.2.

In our approach, the properties of secrecy, integrity, and authenticity are taken relative to the considered type of adversary. In case of the default adversary, this is a principal external to the system; one may, however, consider adversaries that are part of the system under consideration, by giving the adversary access to the relevant system components (by defining Threats_A(s) to contain access for the relevant stereotype s). For example, in an e-commerce protocol involving customer, merchant, and bank, one might want to say that the identity of the goods being purchased is a secret known only to the customer and merchant (and not the bank). This can be formulated by marking the relevant data as “secret” and by performing a security analysis relative to the adversary model “bank” (that is, the adversary is given access to the bank component by defining the Threats() function in a suitable way).

Note that the adversary does not necessarily have access to the input queue of the system. Thus it may be sensible, for example, to apply the «secrecy» tag to a value received by the system from the outside. Of course, the condition associated with the «data security» stereotype only ensures that the component marked with this stereotype keeps the values received by the environment secret; additionally, one has to make sure that the environment (for example, the rest of the system apart from the component under consideration) does not make these values available to the adversary either.

The stereotypes «secure links», «secure dependencies», and «data security» describe different conditions for ensuring secure data communication: «secure links» ensures that the security requirements on the communication dependencies between components are supported by the physical situation, relative to the adversary model under consideration. The stereotype «secure dependencies» ensures that the security requirements in different parts of a static structure diagram are consistent. Finally, «data security» ensures that security is enforced on the behavior level. One could for example merge the conditions of «secure links» and «secure dependencies» to give one stereotype; we keep them separate to facilitate understanding and because one might like to use the stereotype «secure dependencies» in situations where no implementation diagram is present.
4.1 UMLsec Profile

guarded access

This stereotype of (instances of) subsystems is supposed to mean that each object in the subsystem that is stereotyped «guarded» can only be accessed through the objects specified by the tag {guard} attached to the «guarded» object. Formally, we assume that we have name $\notin K_A$ for the adversary type $A$ under consideration and each name $\text{name}$ of an instance of a «guarded» object (that is, a reference is not publicly available), and that for each «guarded» object there is a statechart specification of an object whose name is given in the associated tag {guard}. This way, we model the passing of references (see Sect. 5.4).

Example We illustrate this stereotype with the example of a web-based financial application. Two (fictional) institutions offer services over the Internet to local users: an Internet bank, Bankessey, and a financial advisor, Finance. To make use of these services, a local client needs to grant the applets from the respective sites certain privileges. Access to the local financial data is realized using GuardedObjects. The relevant part of the specification is given in Fig. 4.12.

Since the «guarded» objects can only be accessed through their associated guard, the subsystem instance fulfills the condition associated with the stereotype «guarded access» with regard to default adversaries. More details are given in Sect. 5.4.

guarded

This stereotype labels objects (in particular in the scope of the stereotype «guarded access» above) that are supposed to be guarded. It has a tagged value {guard} which defines the name of the corresponding guard object.

The conditions associated with the above stereotypes give a range from more superficial syntactic conditions to relatively deep semantic conditions. This has the advantage that in an analysis of a system one may start out with the simpler syntactic conditions, which are usually easy to check mechanically, and remove mistakes detected by them. Then one may proceed with the more involved conditions, for which the verification of systems of practically realistic size can become rather complex. This approach seems to be more efficient than trying to establish the overall security all at once; in an industrial setting, it also allows a scaling of the necessary costs.

4.1.3 Addressing the Requirements

We go back to the requirements on an extension of UML for the development of security-critical systems in Sect. 4.1.1 and consider UMLsec in turn with respect to them.
Mandatory Requirements

Security requirements: Formalizations of basic security requirements are provided via stereotypes, such as «secrecy», «integrity», and «authenticity».
Threat scenarios: Threat scenarios are incorporated using the formal semantics and depending on the modeled underlying physical layer via the sets $\text{Threats}_{ad}$. Security concepts: We have shown how to incorporate security concepts such as tamper-resistant hardware (using threat scenarios, in this case).

Security mechanisms: As an example, in Sect. 5.4 we demonstrate modeling of the Java Security Architecture access control mechanisms.

Security primitives: Security primitives are either built in (such as encryption) or can be treated (such as security protocols, see Sect. 5.3).

Underlying physical security: This can be addressed as demonstrated by the stereotype «secure link» in deployment diagrams.

Security management: This can be considered in our approach by using activity diagrams (as in Fig. 4.4).

Optional Requirements

Additional domain knowledge has been incorporated regarding Java security and CORBA applications (see Sect. 5.4), as well as smart card security (see Sect. 5.3).

Note that in the presentation of the profile we do not aim for completeness, but explain how the most important concepts can be realized. This should demonstrate how one can add further notions from main stream computer security that have to be omitted here, as well as more application-specific concepts.

We may also consider to what extent our approach addresses the questions raised in [Chu93] regarding a secure systems design process:

1. How can a variety of well-known and lesser-known security techniques be made available to the designer through systematic search? Our approach aims to encapsulate design knowledge to facilitate reuse. One can use a variety of security notions (such as secrecy, integrity, and authenticity) to express security requirements.

2. How can interactions among potentially conflicting or synergistic requirements be managed systematically? One may argue that this can be handled in the context of the formal semantics used; an investigation is beyond the scope of the current work.

3. How can the nature of relationships between design decisions be represented? How can the effect of each design decision be systematically evaluated? One may argue that the effect of design decisions on the security of the system can be systematically evaluated by referring to the formal semantics; an investigation is beyond the scope of the current work.

4. How can security requirements be systematically integrated into the design, together with other types of non-functional requirements? Using UML makes integration into the design and the general development process relatively straightforward, since many developers know UML and can use UMLsec without too much overhead (after learning the security-specific
information given in the UMLsec profile). One can combine our extension of UML with extensions regarding other non-functional requirements, such as performance.

(5) **What drives design actions? What representational structures are appropriate for systematically recording the results of such actions?** As pointed out in [Emm00], a goal-oriented approach to requirements (such as [DvLF93]) may work better with regard to non-functional requirements than use-case-driven approaches (such as [JBR98]). However, [MCY99] shows that goal-oriented analysis and object-oriented analysis complement each other. Thus one can fruitfully employ the known goal tree approach to non-functional requirements [Chu93] in our work using UML. Specifically, in [Jür02] we propose to combine a use-case-driven approach as in [JBR98] for the functional requirements with a goal-driven approach as in [Chu93] for the security requirements. This takes account of the fact that security requirements (such as confidentiality) often apply to specific functions (for example, a certain value) of a system, rather than the system as a whole, because the latter may be infeasible.

Note that when adapting a modeling language to security requirements, one needs to make sure that the features used to express security properties on the design level actually map to system constructs on the implementation level which do provide these properties. Since we assume, for example, that attributes can only be accessed through the operations of an object, and that only the explicitly externally offered operations of a subsystem can be called from outside it, it is generally security critical that this is enforced on the implementation level. We refrain from using UML features such as package visibility to model security functionality because it does not seem to be generally implemented in a security-aware way.

### 4.2 Design Principles for Secure Systems

We demonstrate by examples how one could use our approach to enforce the security design rules stated in [SS75].

**Economy of Mechanism**

Our approach addresses this “meta-property” by providing developers (possibly without background knowledge of security) with guidance on the employment of security mechanisms who might otherwise be tempted to employ more complicated mechanisms since these may seem more secure.

**Fail-safe Defaults**

One may verify that a system is fail-safe by showing that certain security-relevant invariants are maintained throughout the execution of the system;
that is, in particular if the execution is interrupted at some point (possibly
due to malicious intent of one of the parties involved). As an example, secure
log-keeping for audit control is considered with respect to the unlinked load
transaction of the smart-card-based Common Electronic Purse Specifications
(CEPS) [CEP01] in Sect. 5.3.2.

Complete Mediation

This principle concerns a strategy for access control where every access is
checked. As an example, we show in Sect. 5.4 how to use UMLsec to correctly
develop secure Java applications making use of the Java Security Architecture
access control mechanisms. With this approach, one can also enforce complete
mediation.

More feasibly, one can specify a set of sensitive objects and say that a
specification satisfies mediation with respect to these objects if their access
is controlled. One may then give a general policy that defines which access
restrictions should be enforced.

Open Design

Our approach aims to contribute to the development of a system whose secu-
rrity does not rely on the secrecy of its design.

Separation of Privilege

Separation of privilege gives another strategy for granting access to resources.
Again, this can be enforced similarly to the way explained in Sect. 5.4. For
example, one can define guard objects (controlling access to other objects)
that require signatures from two different principals on the applet requesting
access to the guarded object.

In this context, a specification satisfies separation of privilege with respect
to a certain privilege \( p \) if there are two or more principals whose signature is
required to be granted \( p \), at every point of the execution.

More generally, one can formulate such requirements on a more abstract
level and verify UMLsec specifications with respect to these requirements.

Least Privilege

Given functionality requirements on a system, a system specification satisfies
the principle of least privilege if it satisfies these requirements and if every
proper diminishing of privileges of the entities in the system leads to a system
that does not satisfy the requirements. This can be formalized within our
specification framework and the condition can be checked.

Least Common Mechanism

Since we follow an object-oriented approach, this principle is automatically
enforced in so far as data is encapsulated in objects and the sharing of data
between different parts of a system is thus well-defined and can be kept at the
minimum of what is necessary.
Psychological Acceptability

With respect to the development process, this principle is addressed by our approach in so far as it aims for ease of use in the development of security-critical systems, and thus for the psychological acceptability of security issues on the side of the developers.

4.3 Applying Security Patterns

Patterns [GHJV95] encapsulate the design knowledge of software engineers by presenting recurring design problems and standardized solutions. One can use transformations of UMLsec models to introduce patterns within the design process. A goal of this approach is to ensure that the patterns are introduced in a way that has previously been shown to be useful and correct. Also, having a sound way of introducing patterns using transformations can ease formal verification (where desired), since the verification can be performed on the more abstract and simpler level, and one can derive security properties of the more concrete level, provided that the transformation has been shown to preserve the relevant security properties.

In our approach, the application of a pattern \( p \) corresponds to a function \( f_p \) which takes a UML specification \( S \) and returns a UML specification, namely the one obtained when applying \( p \) to \( S \). Technically, such a function (and thus the corresponding pattern) can be presented by defining how it should act on certain subsystem instances, and by extending it to all possible UML specifications in a compositional way. Suppose that we have a set \( S \) of subsystem instances such that none of the subsystem instances in \( S \) is contained in any other subsystem instance in \( S \). Suppose that for every subsystem instance \( S \in S \) we are given a subsystem instance \( f_p(S) \). Then for any UML specification \( U \), we can define \( f_p(U) \) by substituting each occurrence of a subsystem instance \( S \in S \) in \( U \) by \( f_p(S) \).

The challenge then is to define such a function \( f_p \) that is applicable as widely as possible. This should be interesting further research. Here we just demonstrate the idea by an example. Consider the problem of communication over untrusted networks, as exemplified in Fig. 4.13. A well-known solution to this problem is to encrypt the traffic over the untrusted link, as demonstrated in Fig. 4.14 (for a detailed explanation of this pattern see Sect. 5.1). Note that the protocol only serves as a simple example for the use of patterns, not to propose a new protocol of practical value.

One can apply this pattern \( p \) in a formal way by considering the set \( S \) of subsystem instances derived from the subsystem instance in Fig. 4.13 by renaming (that is, by substituting any message, data, state, subsystem instance, node, or component name \( n \) by a name \( m \) at each occurrence, in a way such that name clashes are avoided). Then \( f_p \) sends any subsystem instance \( S \in S \) to the subsystem instance derived from that given in Fig. 4.14 by the same
Fig. 4.13. Security pattern example: sender and receiver

renaming. This gives us a presentation of $f_p$ from which the definition of $f_p$ on any UML specification can be derived as indicated above. Since one can show that the subsystem in Fig. 4.14 is secure in a precise sense (see Sect. 5.1), this gives one a convenient way of reusing security engineering knowledge in a well-defined way within the development context.

4.4 Notes

[Low00] uses UML for requirements capture for security protocols. [Jür01a, Jür01c, Jür01d, Jür02b] gives ideas on how to use UML to develop security-critical systems; [Jür02c] shows how to apply this work to encapsulate prudent rules of security engineering. UMLsec has been introduced in [Jür02b]. There does not seem to be any other work proposing a similarly comprehensive

new↓
new↑
new↓
new↑
Fig. 4.14. Security pattern example: secure channel

extension of UML for the development of security-critical systems at the time of writing.

Compared to research done using formal methods (see Sect. 7.6), less work has been done using software engineering techniques for computer security (examples are [Eck95, EM97, DFS98]). [And94] suggests using software engineering techniques to ensure security. [DS00b] gives an overview of work in software engineering for security and (independently from us) suggests that one should devise a UML extension for secure systems (but does not actually give an extension). See also the work in [FH97] (defining role-based access con-
control rights from object-oriented use cases) and [Fer98] (taking a holistic view on Internet security). Security of object-oriented systems has been considered in [JKS95, SBC97].

Work on security patterns includes [FP01, Sch03a]. Part of the material in Sect. 4.3 has been reported in [Jür01], [JPW02, JB03].

4.5 Discussion

We presented the extension UMLsec of UML for secure systems development, in the form of a UML profile using the standard UML extension mechanisms. Recurring security requirements are written as stereotypes; the associated constraints ensure the security requirements on the level of the formal semantics, by referring to the threat scenario also given as a stereotype. Thus one may evaluate UML specifications to indicate possible vulnerabilities. One can thus verify that the stated security requirements, if fulfilled, enforce a given security policy.

We indicated how one could use UMLsec to model security requirements, threat scenarios, security concepts, security mechanisms, security primitives, underlying physical security, and security management, aspects which were argued to be required for a secure systems extension of UML.

We also considered how UMLsec could be used to encapsulate established rules on prudent security engineering, also by applying security patterns, and thereby to make them available to developers who are not experts in security. While UML was developed to model object-oriented systems, one can also use UMLsec to analyze systems that are not object-oriented (assuming that the underlying assumptions, such as controlled access to data, are ensured).

There does not seem to be a similarly comprehensive secure systems extension of UML published at the time of writing, despite rapidly growing interest, as exemplified by the existence of three European working groups (CORAS [St01], NEPTUNE [nep01], and the newly formed Degas [DEG01]), one German project [arc01], a number of individual researchers in Europe and North America concerned with secure systems development with UML, and a UML02 satellite workshop which includes the topic [JCF02].

For defining UMLsec, we made use, besides the application examples presented here, of experience in the development of security-critical systems using UML (for example, from the co-supervised [Mea01]) and in security consulting (for example, in a security project founded by the German Ministry for Commerce and Technology [Fai01] and industrial projects with a major German bank).
Applications

We demonstrate stepwise development of a security-critical system with UMLsec by the example of a secure channel design, together with a formal verification.

We uncover a flaw in a variant of the handshake protocol of the Internet protocol TLS proposed in [APS99], suggest a correction, and verify the corrected protocol.

We apply UMLsec to a security analysis of the Common Electronic Purse Specifications, a candidate for a globally interoperable electronic purse standard. We discover flaws in the two central parts of the specifications (the purchase and the load protocol), propose corrections, and give a verification.

We show how to use UMLsec to correctly employ advanced Java 2 security concepts such as guarding objects in a way that allows formal verification of the specifications.

The proofs for the statements in this chapter can be found in Appendix C.

5.1 Secure Channels

As an example of the stepwise development of a secure system with UML we give an abstract specification of a secure channel and refine it to a more concrete specification. The abstract specification satisfies secrecy, and by our preservation result the concrete one does as well.

Figure 5.1 gives a high-level system specification in the form of a UML subsystem C for communication from a sender object to a receiver object, including a class diagram with appropriate interfaces. Note that in this simplified example, which should mainly demonstrate the idea of stepwise development, we are only concerned with fixed participants S and R; therefore, authentication is out of scope of our considerations. A more realistic example with a more in-depth security analysis can be found in Sect. 5.2.

In the subsystem, the Sender object is supposed to accept a value in the variable d as an argument of the operation send and send it over the
Fig. 5.1. Example subsystem: sender and receiver

«encrypted» Internet link to the Receiver object, which delivers the value as a return value of the operation receive. Note that the behavior of the sender could also be specified by a statechart consisting of only one state; the version given here is viewed to be more readable. According to the stereotype «critical» and the associated tag {secrecy}, the subsystem is supposed to preserve the secrecy of the variable \( d \).

We show that this is actually the case.

We show a result which is slightly stronger than the one stated above, where the adversary is allowed to have some additional initial knowledge, which will be useful in the following.

**Proposition 5.1.** The subsystem \( C \) preserves the secrecy of the variable \( d \) from adversaries of type \( A = \) default with specified previous knowledge \( K^p_A \), given inputs from \( \text{Data} \setminus K^p_A \).
Note that this statement refers to an idealized model where the adversary is by definition unable to interfere with the protocol. Also, as mentioned above, we consider only fixed participants in this case, so that the intended protocol execution is in fact the only possible one. This is of course not realistic in general, but the aim is to exhibit conditions in the following under which it would be justified to use such an idealized model of a secure channel.

Note also that, taking into account the above discussion, secrecy of \( d' \) follows from that of \( d \) (since in a typical protocol run, they coincide). Integrity is not within the scope of the current considerations (but holds for both \( d \) and \( d' \) since the adversary cannot actively interfere with the protocol).

Now assume that we would like to replace the abstract requirement that the communication should happen over an encrypted link by a more concrete specification of the encryption mechanism. Thus we construct a refinement \( C' \) as in Fig. 5.2.

Since we only want to demonstrate the principle of developing a secure channel, we assume for simplicity that the sender and receiver already know each other's public keys. The protocol then exchanges a symmetric session key using those public keys, since encryption under symmetric keys is more efficient. We assume that the secret keys belonging to the public ones are kept secure. The session keys \( k_x (x \in \mathbb{N}) \) are specified to be created freshly by the receiver before execution of the protocol, as stated by the tag \{fresh\}. As can be seen from the UML specification, the associated constraint is actually fulfilled: The values \( k_x \) belong to the scope of \texttt{Receiver} within the subsystem specification \texttt{SecureChannel} (as defined in Sect. 4.1.2), since expressions of the form \( k_x \) (for any subexpression \( x \)) only appear within the \texttt{Receiver} object and the associated statechart. For readability, in this chapter we just write \( k_x \) \texttt{: Data} to denote an array whose fields \( k_x \) have the type \texttt{Data}.

Recall that we leave out the explicit assignment of initial values to constant attributes and instead take these constants as attribute names (such as the keys in this example). Note that by definition of the algebra of expressions (see Sect. 3.3.3), the keys and nonces (as different constant symbols in \texttt{Keys} \union \texttt{Data}) are mutually distinct and independent, and also independent of the other expressions in the diagram (this observation will be proven in Chap. 7).

Also, to increase readability, we again use the notation \texttt{var := exp} (where \texttt{var} is a local variable not used for any other purpose and \texttt{exp} may not contain \texttt{var}) as a syntactic shortcut. Before assigning a semantics to the diagram, the variable \texttt{var} should be replaced by the expression \texttt{exp} at each occurrence

The behavior of the sender thus includes retrieving the signed and encrypted symmetric session key \( k_j \) from the receiver, checking the signature, and encrypting the data under the symmetric key, together with a sequence number \( i \) (to avoid replay; we assume the natural numbers \( i, j \) to be in \texttt{Data} here and in the following; \( \mathbb{N} \subseteq \texttt{Data} \)). The receiver first gives out the key \( k_j \) with a signature and also with a sequence number \( j \), and later decrypts the received data, checking the sequence number.

The core message exchange between sender and receiver is thus as follows:
\[ R \rightarrow S : \{\text{Sign}_{K_\text{\text{\$}}} (k_j :: j)\}_{k_j} \]
\[ S \rightarrow R : \{d :: i\}_{k_i}. \]

![Diagram of SecureChannel](image)

**Fig. 5.2.** Example subsystem: secure channel

We show that \( C' \) is a refinement of \( C \) in the sense of Definition 8.4.

**Proposition 5.2.** The subsystem \( C' \) is a delayed black-box refinement of \( C \) given adversaries of type \( A = \text{default} \) with
5.1 Secure Channels

\[ K^p_A \cap \left( \{ K_S^{-1}, K_R^{-1} \} \cup \{ K_n, \{ x :: n \} \} \right) = \emptyset \]

and for which \( \text{Sign}_{K_R^{-1}}(K' :: n) \in K^p_A \) implies \( K' = K_n \).

The condition in the statement of the above proposition means that the previous adversary knowledge \( K^p_A \) may not contain the secret keys \( K_S^{-1}, K_R^{-1} \) of the sender and the receiver, the secret session keys \( K_n \), any encryptions of the form \( \{ x :: n \} \), and any signatures \( \text{Sign}_{K_R^{-1}}(K' :: n) \) except for \( K' = K_n \).

Recall from Sect. 3.3.4 that \( K^p_A \) denotes the knowledge of the adversary before the start of the execution of the system, that is in this case, before the first iteration of the protocol. Thus the condition does not prevent the adversary from remembering information gained from early iterations of the protocol and use it in later iterations. The condition is thus not unrealistic, and it is in fact necessary because if the adversary knows the expression \( \{ x :: n \} \) before the execution of the protocol which may be different from the expression \( \{ y :: n \} \) which will be sent out by \( S \) in the \( n \)th round of the protocol, the adversary could substitute \( \{ y :: n \} \) with \( \{ x :: n \} \) without being noticed which would destroy the integrity of the communication channel and then \( C' \) would not be a refinement of \( C \). Note that the sequence number \( n \) is necessary to enable the receiver to check that the right session key is used for decryption in the condition \( \text{tail}(\text{Dec}_k(E)) = j \), to prevent replay.

The analysis in the above proposition also covers the possibility that there may be parallel executions of other instances of \( \text{Client} \) and \( \text{Server} \), because these can be simulated by the adversary. The result can be refined to establish a property of “forward security” in the sense that the compromise of a current session key does not necessarily expose future traffic (as defined in [And02]). We omit this here because we only want to demonstrate the basic technique using this example, but refer to Theorem 5.5 for such a result for a different protocol.

Note that \( C' \) is not an undelayed refinement of \( C \) (because of the delay caused by the key exchange and possible additional delay caused by the adversary). While in \( C \) the shortest output sequence containing a \( \text{return} \) (after some input \( \text{send} \)) is \( (\emptyset, \emptyset, \emptyset, \{ \text{return} \}) \) (in case the adversary does not delete any messages), and in \( C' \), it is \( (\emptyset, \emptyset, \emptyset, \emptyset, \{ \text{return} \}) \) (because of the key exchange).

**Proposition 5.3.** The subsystem \( C' \) preserves the secrecy of the variable \( d \) from adversaries of type \( A = \text{default} \) with

\[ K^p_A \cap \left( \{ K_S^{-1}, K_R^{-1}, K \} \cup \{ x :: n \} \right) = \emptyset \]

and for which \( \text{Sign}_{K_R^{-1}}(K') \in K^p_A \) implies \( K = K' \), given inputs from \( \text{Data} \) \( \setminus K^p_A \).

Thus the specification fulfills the constraints of the stereotype « data security » with respect to the adversary type stated above.“

As for Proposition 5.2, this result can be refined to establish a version of “forward security”:

new↓

new↑

new↓

new↑

new↓
5.2 A Variant of the Internet Protocol TLS

We analyze a variant of the handshake protocol of TLS\(^1\) proposed in [APS99]. We uncover a flaw (first published in [Jiř01g]), suggest a correction, and verify it.

The Handshake Protocol

new↓

The goal of the protocol whose core is given as a UML subsystem \(\mathcal{T}\) in Fig. 5.3 is to establish a secure channel over an untrusted communication link between a client and a server. This channel is supposed to provide secrecy and server authenticity, as specified by the \{secrecy\} and \{authenticity\} tags.

new↑

We assume that the set of data values Data includes names \(C\) and \(S\) for each instance \(C: \text{Client}\) and \(S: \text{Server}\). The protocol assumes that each client \(C\) is given the server name \(S_i\) before the \(i\)th execution round of the protocol part under consideration (but not conversely, since no client authenticity is to be provided by the protocol). In our model we restrict ourselves to considering the first \(l\) executions of the protocol, where \(l\) is an arbitrary but fixed natural number. Note that each \(C\) may be given a different sequence of server names. More precisely, these would have to be referred to as \(C.S_i\); we omit the instance prefixing for readability where no confusion can arise. \(C\) and \(S\) are variables representing arbitrary names; in particular, both client and server can run the protocol with arbitrary servers and clients. Note also that the adversary controls the communication link between client and server (which in our model is captured by enabling the adversary to read, delete, and insert messages at the corresponding link queue). Therefore, the adversary is able to insert any message communicated over the link in the adversary’s current knowledge. In particular, the adversary may also perform the protocol with either the client or the server.

We assume that each \(C\) (resp. each \(S\)) has a public key \(K_C\) (resp. \(K_S\)) with associated private key \(K_C^{-1}\) (resp. \(K_S^{-1}\)). We assume that there is a secure (with respect to integrity) way for \(C\) to obtain the public key \(K_{CA}\) of the certification authority, and for \(S\) to obtain a certificate \(\text{Sign}_{K_{CA}^{-1}}(S :: K_S)\) signed by the certification authority that contains its name and public key. Also, each client is given the sequence of secrets \(s_1,\ldots,s_l \in \text{Data}^x\) to be transmitted and the nonces \(N_1,\ldots,N_l \in \text{Data}\) (again we write \(s_\cdot \in \text{Data}^x\) to denote an array with fields \(s_i\) in Data). The nonces are specified to be created freshly by the receiver before execution of the protocol, as stated by the tag \{fresh\}. As can be seen from the UML specification, the associated constraint is actually fulfilled: The values \(N_i\) belong to the scope of Client within the subsystem specification TLS variant (as defined in Sect. 4.1.2), since expressions of the form \(N_x\) (for any subexpression \(x\)) only appear within the

\(^1\) TLS (transport layer security) is the successor of the Internet security protocol SSL (secure sockets layer).
### Fig. 5.3. Variant of the TLS handshake protocol
Client object and the associated view of the sequence diagram. Similarly, the sequence of session keys $k_1, \ldots, k_i \in \text{Keys}^s$ given to each server is specified to be fresh. Again, for readability, we leave out the explicit assignment of initial values to constant attributes (such as the keys, the nonces, and the values $s_x$ and $S_x$ here) and instead take these constants as attribute names. Note that by definition of the algebra of expressions, the keys and nonces (as different constant symbols in \textbf{Keys} and \textbf{Data}) are mutually distinct and distinct from other values (and thus independent of them by Fact 7.36). The subsystem specification given here could also be instantiated with other initial values for the keys, the nonces, and the values $s_x$ and $S_x$, as long as the keys and nonces remain distinct from other values. In that sense, the specification can be viewed as parameterized over these values.

For readability we leave out a time-stamp, a session id, the choice of cipher suite and compression method, and the use of a temporary key by $S$ since these are not relevant for the security requirements under consideration.

We recall that for each method $\text{msg}$ in the diagram and each number $n$, $\text{msg}_n$, represents the $n$th argument of the operation call $\text{msg}$ that was most recently accepted according to the sequence diagram. Also, to be more explicit, we add the names of the objects appearing in a sequence diagram as arguments to the name of the sequence diagram, and let the object names themselves be parameterized. Thus we consider the sequence diagram $\text{tls}[C, S]$ involving the object $C$ and the object $S_i$. The sequence diagram is then used to specify the activities $\text{tls} \cdot C$ and $\text{tls} \cdot S$ for objects $C$ and $S$ (which is well-defined because the object $S_i$ in the sequence diagram does not use the parameter $i$).

The protocol proceeds as follows. We consider the $i$th execution round $C(i)$ of the client $C$ and the $j$th execution round $S_i(j)$ of the server $S_i$ and assume that $S_i(j) = S$ (that is, in the current execution round $i$, the instance $C$ aims to communicate with the instance $S_i$, which is in its $j$th execution round). The client $C$ initiates the protocol by sending the message $\text{init}(N_i, K_C, \text{Sign}_{K_C}(C :: K_C))$ to the server $S$. If the condition $[\text{snd}(\text{Ext}_{K_C}(c_C)) = K']$ holds, where $K' := \arg_{S, t, 1, 2}$ and $c_C := \arg_{S, t, 1, 3}$ (that is, the key $K_C$ contained in the signature matches the one transmitted in the clear), $S$ sends the message $\text{resp}\left(\{\text{Sign}_{K_S}(k_j :: N')\}^{K'}, \text{Sign}_{K_C}(S :: K_S)\right)$ back to $C$ (where $N' := \arg_{S, t, 1, 2}$). Then if the condition

$$[\text{fst}(\text{Ext}_{K_C}(c_S)) = S \land \text{snd}(\text{Ext}_{K'}(\text{Dec}_{K_C}(c_S))) = N]$$

holds, where $c_S := \arg_{C, t, 1, 1}$, $c_k := \arg_{C, t, 1, 2}$, and $K' := \text{snd}(\text{Ext}_{K_C}(c_S))$ (that is, the certificate is actually for $S$ and the correct nonce is returned), $C$ sends $xchd(\{S\}_k)$ to $S$, where $k := \text{fst}(\text{Ext}_{K_C}(\text{Dec}_{K_C}(c_S)))$. If any of the checks fail, the respective protocol participant stops the execution of the protocol.

In the traditional informal notation (for example, used in [NS78]), the protocol would be written as follows:

$$C \rightarrow S : N_i, K_C, \text{Sign}_{K_C}(C :: K_C)$$
\[ S \rightarrow C : \{ \text{Sign}_{K^{-1}}(k_j \oplus N_i) \}_{k_i}, \text{Sign}_{K^{-1}}(S \oplus K_S) \]
\[ C \rightarrow S : \{ s_i \}_{k_i}. \]

This notation needs to be interpreted with care [Aba00]. For example, from the first line, we can conclude that \( C \) sends \( N_i, K_C, \text{Sign}_{K^{-1}}(C \oplus K_C) \) to the network, with intended recipient \( S \), and that \( S \) expects a message of the form \( N_i, K_1, \text{Sign}_{K^{-1}}(X \oplus K_2) \), seemingly coming from \( C \). If the message is sent over an untrusted network, we cannot conclude for example that \( K_1 = K_C \). Therefore, one needs to make (standard) assumptions such as that \( S \) checks that the three occurrences of \( K_C \) do indeed coincide (that is, that \( K_1 = K_2 = K_3 \) holds). Unfortunately, misinterpretation of protocol specifications is a major source of security weaknesses in practice. Therefore, when using this notation, one should make sure that the above-mentioned assumptions are understood by the implementor of a protocol.

Our aim here is to use the UML as a notation that is widely used among software developers beyond the community of security experts, without deviating from its standard definition any more than what may seem necessary. Since the UML sequence diagram semantics does not support the above-mentioned assumptions, we include them explicitly by referring to sent and received values in different ways and including checks in the sequence diagram to ensure that they actually coincide.

Note that there are formal approaches like Casper [Low98] closer to the traditional notation which take input in essentially the form described above (possibly with some additional information) and automatically generate the standard checks. The motivation there is to free the specification writer from the obligation to manually include the correct checks, and to have a more compact notation that is more easily accessible. It would be interesting further work to see whether one can make use of these ideas in the context of UMLsec.

The Flaw

When analyzing the specified protocol for the relevant security requirements using the automated analysis tools presented in Sect. 6.2.1', we observed the following attack.

**Theorem 5.4.** For given \( C \) and \( i \), the UML subsystem \( T \) given in Fig. 5.3 does not preserve the secrecy of \( C.s_i \) from adversaries of type \( A = \text{default} \) with \( K_A, K^{-1}_A \in K_A^h \).

This means that the protocol does not provide its intended security requirement, secrecy of \( s_i \), against a realistic adversary.

The message flow diagram corresponding to this man-in-the-middle attack follows.
The authors of [APS99] have been informed about the problem.

The Fix

We propose to change the protocol to get a specification $T'$ by substituting $k_j :: N_i$ in the message resp by $k_j :: N_i :: K_C$ as in Fig. 5.4, and by including a check regarding this new message part at the client. Here, the public key $K_C$ of $C$ is included representatively for the identity of $C$ (in fact, one could also use $k_j :: N_i :: C$ instead).

Again, in traditional informal notation, the modified protocol would be written as follows:

\[
C \rightarrow S : N_i, K_C, \text{Sign}_{K_C^{-1}}(C :: K_C)
\]
\[
S \rightarrow C : \{\text{Sign}_{K_C^{-1}}(k_i :: N_i :: K_C)\}_{K_i}, \text{Sign}_{K_C^{-1}}(S :: K_S)
\]
\[
C \rightarrow S : \{S_i\}_{k_i}.
\]

We explain informally why this modification prevents the attack described in Theorem 5.4. Since the certificate sent in the first message of the protocol is only a self-signed certificate (which does not provide full client authenticity), the adversary can still send a certificate to the server claiming that the public key of the adversary in fact belongs to the client, as in the attack described above. However, when the adversary then forwards the response from the server to the client, the server signed certificate contains the public key received by the server in the first message of the protocol. If the adversary again forwards this certificate to the client, the client will notice that a false public key has been submitted on the client’s behalf and will stop execution of the protocol because the check that has been newly introduced fails. Conversely, the client will only send the secret under the session key received if it is signed by the server concatenated with the public key of the client. This certificate, in turn, the server only sends out encrypted under the same public key, which the adversary cannot decrypt. Here it is essential that the session keys differ for different iterations of the protocol.

Of course, the above arguments may convince the reader that the particular attack exhibited in Theorem 5.4 is prevented by the modification proposed here, but they give little confidence that the modified protocol is immune against all other attacks that may be possible. We therefore prove formally
Fig. 5.4. Repaired variant of the TLS handshake protocol
that the protocol specification is secure with regards to our adversary model. More specifically, we show that the protocol specification in fact fulfills the constraints associated with the `data security` stereotype with respect to the adversary given below. We restrict ourselves to proving this for the `{secrecy}` property; `{integrity}` and `{authenticity}` can be established similarly.¹

**Theorem 5.5.** Suppose we are given a particular execution of the repaired TLS variant subsystem \( T' \) (including all clients and servers), a client \( C \), and a number \( I \) with \( S = S_I \), and suppose that the server \( S \) is in its \( I \)th execution round in the current execution when \( C \) is in its \( I \)th execution round initiates the protocol (that is, \( C.i = I \) and \( S.j = J \)). Then this execution of \( T' \) preserves the secrecy of \( C.s_I \) against adversaries of type \( A = \text{default} \) whose previous knowledge \( K_A^p \) fulfills the following conditions:

- we have \[
\left\{ C.s_I, K_C^{-1}, K_S^{-1} \right\} \cup \left\{ S.k_j : j \geq J \right\} \\
\cup \left\{ \text{Sign}_{K_S^{-1}}(X :: C.N_I :: K_C) : X \in \text{Keys} \right\} \cap K_A^p = \emptyset
\]
- for any \( X \in \text{Exp}, \text{Sign}_{K_S^{-1}}(C :: X) \in K_A^p \) implies \( X = K_C \), and
- for any \( X \in \text{Exp}, \text{Sign}_{K_C^{-1}}(S :: X) \in K_A^p \) implies \( X = K_S \).

¹The condition in the statement of this theorem requires that the previous adversary knowledge \( K_A^p \) may not contain the current secret \( C.s_I \), the secret keys \( K_C^{-1}, K_S^{-1} \) of the sender and the receiver, the current and future session keys \( S.k_j \), any encryptions of the form \( \text{Sign}_{K_S^{-1}}(X :: C.N_I :: K_C) \) and any signatures \( \text{Sign}_{K_S^{-1}}(C :: X) \) (except for \( X = K_C \)) and \( \text{Sign}_{K_C^{-1}}(S :: X) \) (except for \( X = K_S \)). This result covers the possibility that the adversary may gain information from previous or parallel executions of the protocol, possibly with other instances of `Client` or `Server`. With respect to parallel executions of other instances, the restrictions on the adversary knowledge are loose enough to allow the adversary to simulate other instances of the two classes (by giving the adversary access to their private keys and certificates). With respect to previous executions, one should note that the previous adversary knowledge \( K_A^p \) refers to the knowledge of the adversary before the overall execution of the system, not at the point of the system execution where \( C.i = I \) and \( S.j = J \) (see the definition in Sect. 3.3.4 and also the discussion and corollary below). In particular, the condition in the statement of the above theorem does not prevent the adversary from remembering information gained from earlier iterations of the current iteration of the protocol and use it in later iterations. It does, however, assume that the adversary does not know the message \( \{ \text{Sign}_{K_S^{-1}}(k_j :: N_i :: K_C) \} \) of the server in the current protocol run before the current protocol. This assumption is in fact necessary, because otherwise the attack described in Theorem 5.4 would still work.
(because the adversary would already have the certificate the client expects which includes the client’s key $K_C$, and can in addition still get the current session key from the server as in the earlier attack by sending the message $N_i :: K_A :: \text{Sign}_{K_A}(C :: K_A)$ containing the adversary’s key $K_A$ to the server).

Note also that since in the statement of the theorem we allow the keys $S.k_j$ for $j < J$ to be included in the previous adversary knowledge $K_A^p$, the theorem establishes a form of “forward security” in the sense that the compromise of a current key does not necessarily expose future traffic (following the terminology in [And02]). It is, however, not sufficient to only require that $S.k_j \notin K_A^p$, because the adversary may initiate an intermediate interaction with $S$ to increase its counter $j$.

The statement of the theorem concerns particular instances of Client and Server and particular execution rounds. It is formulated in a “rely-guarantee” way (stating that if the knowledge previously acquired by the adversary satisfies the conditions of the theorem, then this execution preserves secrecy) because this allows one to consider security mechanisms (like security protocols) in the system context. To do this, one needs to specify explicitly which values the remaining part of the system has to keep secret from the adversary for the protocol to function securely. For example, the theorem needs to assume that the certification authority does not issue any false certificates (the third condition of the theorem).

Although the conditions in the statement of the theorem only concern the previous knowledge of the adversary before the overall execution of the system, it follows from the theorem that the adversary knowledge before each iteration of the system satisfies these conditions as well; that is, that each iteration of the execution of the system preserves the conditions on the adversary knowledge: if the condition on the adversary knowledge were to be violated in the course of the iterations before the one currently under consideration, the result of the theorem would not be valid (and this statement holds for each “current” iteration). Since the theorem was proved above, this cannot be the case.

We have the following corollary to the above theorem (where we assume that the sets Client and Server of clients and servers are finite):

**Corollary 5.6.** Any execution of $T^+$ over all clients and servers and all execution rounds preserves the secrecy of each $C.s_i$ (for $C : \text{Client}$ and $1 \leq i \leq l$) against adversaries of type $A = \text{default}$ whose previous knowledge $K_A^p$ before the overall execution of $T^+$ fulfills the following conditions:

- we have
  \[
  \left\{ (K_c^{-1}, K_s^{-1}, c.s_i, s.k_j, \{ \text{Sign}_{K_A^p}(C :: c.N_i :: K_c) \})_{K_c} : \\
  c : \text{Client} \land s : \text{Server} \land 1 \leq i \leq l \land 1 \leq j \land X \in \text{Keys} \right\} \cap K_A^p = \emptyset,
  \]

- for any $X \in \text{Exp}$ and any $c : \text{Client}$, $\text{Sign}_{K_c^{-1}}(c :: X) \in K_A^p$ implies $X = K_c$, and
• for any $X \in \text{Exp}$ and any $s : \text{Server}$, $\text{Sign}_{K_{CA}}(s : X) \in \mathcal{K}_A^p$ implies $X = K_s$.

The condition in the statement of this corollary generalizes that of Theorem 5.5 to arbitrary clients and servers. Note that the protocol rounds of each client and server do not have to correspond in any particular way. This means that any combination of clients $c$, servers $s$, secrets $c.s_i$, and session keys $s.k_j$ may occur (we only know that the same session key is not to be used repeatedly). In particular, the $C.s_j$ under consideration could be transmitted encrypted under $s.k_j$ for any server $s$ and server round $j$. Thus we need to assume $s.k_j \notin \mathcal{K}_A^p$ for any $s$ and $j$. Again note that $\mathcal{K}_A^p$ denotes the knowledge of the adversary prior to even the first execution round of the protocol.

5.3 Common Electronic Purse Specifications

In this section, we apply UMLsec to a security analysis of the Common Electronic Purse Specifications (CEPS) [CEP01]. CEPS are a candidate for a globally interoperable electronic purse standard supported by organizations (including Visa International) representing 90% of the world’s electronic purse cards and likely to become an accepted standard [AJSW00], making its security an important goal.

Stored value smart cards (“electronic purses”) have been proposed to allow cash-free point-of-sale (POS) transactions offering more fraud protection than credit cards: Their built-in chip can perform cryptographic operations, which allow transaction-bound authentication (whereas credit card numbers are valid until the card is stopped, enabling misuse). The card contains an account balance that is adjusted when loading the card or purchasing goods.

The scheme participants are the card issuer (issuing the cards), the funds issuer (processing the funds needed for a card load), the load acquirer operating a load device (where a card can be loaded), the merchant operating a POS device (where a card can be used to purchase a good), the card running a card application, and system operators for the processing of the transaction data. Possible transactions are Purchase, Purchase Reversal, Incremental Purchase, Cancel Last Purchase, Currency Exchange, Load, and Unload.

Here we consider two central parts of the CEPS: the purchase transaction, an off-line protocol which allows the cardholder to use the electronic value on a card to pay for goods, and the load transaction, an on-line protocol which allows the cardholder to load electronic value on a card. In each case, we give a simplified account to keep the presentation feasible. For example, we do not consider exception processing: if, for instance, a certificate verification fails, our model simply stops further processing. Also, for simplicity, we omit the request messages to the smart card that are only included in the protocol because current smart cards communicate only by answering requests.
5.3.1 Purchase Transaction

The participants involved in the off-line purchase transaction protocol are the customer’s card and the merchant’s POS device. The POS device contains a Purchase Security Application Module (PSAM) (for example, a smart card) that is used to store and process data (and assumed to be tamper-resistant). After the protocol, the account balance in the customer’s card is decremented, and the balance in the PSAM is incremented by the corresponding amount. The card issuer later receives transaction logs.

In addition to transactions using public terminals it is also intended to use CEPS cards for transactions over the Internet [CEP01, Bus. Req. ch. X].

**Specification**

Figure 5.5 gives an overview of a POS device (from [CEP01, Tech. Spec. p. 77]).

![POS Device Functional Components Diagram]

**Fig. 5.5.** POS device overview
In Fig. 5.6 we give a specification of the (simplified) purchase transaction as a UML subsystem \( P \).

We recall that for each method \( \text{msg} \) in the diagram and each number \( n \), \( \text{msg}_n \) represents the \( n \)th argument of the operation call \( \text{msg} \) that was most recently accepted according to the sequence diagram. Again, we use the notation \( \text{var} := \text{exp} \) (where \( \text{var} \) is a local variable not used for any other purpose and \( \text{exp} \) may not contain \( \text{var} \)) as a syntactic short-cut. Before assigning a semantics to the diagram, the variable \( \text{var} \) should be replaced by the expression \( \text{exp} \) at each occurrence.

Apart from incremental transactions (not considered here), security functionality is provided only by the PSAM (and not the rest of the POS device). Thus our protocol participants are the CEP (Common Electronic Purse) card \( C \) (with identity \( ID_C \) and public (resp. private) keys \( K_C \) (resp. \( K_C^{-1} \))) and the PSAM \( P \) (with identity \( ID_P \) and public (resp. private) keys \( K_P \) (resp. \( K_P^{-1} \))). Both also have stored the public key \( K_{CA} \) of the certification authority before the transaction.

Note that of course the protocol will be used with different cards during the lifetime of a PSAM; for simplicity, we omit this aspect. Card revocation is not considered here. Also, we assume that the sequence of transaction amounts \( M_{NT} \) (indexed by the transaction number \( NT \)) is given, as well as the sequence of session keys \( SK_{NT} \). These keys are required to be fresh at the PSAM object (as indicated by the tag \{fresh\}; see Sect. 4.1.2), and in fact one can see from the specification that expressions of the form \( SK_x \) (for any subexpression \( x \)) appear only at the PSAM object and the associated view of the sequence diagram. Also, again, by definition of the algebra of expressions (see Sect. 3.3.3), the keys (as different constant symbols in \textbf{Keys}) are mutually distinct, and therefore mutually independent in the sense of Definition 7.34, by Fact 7.36 (and also independent of the other expressions in the diagram).

Again, we write \( M_x \) to denote an array whose fields \( M_x \) have the type \textbf{Data}.

We leave as implicit the actual adjustment of the balance on the card (which includes checking that the balance is greater than the charged amount).

At the beginning of its execution in the POS device, the PSAM creates a transaction number \( NT \) with value 0. Before each protocol run, \( NT \) is incremented. If a certain limit is exceeded, the PSAM stops functioning (to avoid rolling over of \( NT \) to 0). Note that here we assume an additional operation, the \( + \), to build up expressions.

The protocol between the card \( C \), the PSAM \( P \), and the display \( D \) is supposed to start after the card \( C \) is inserted into a POS device containing \( P \) and \( D \) and after the amount \( M \) is communicated to the PSAM (by typing it into a terminal assumed to be secure).

Each protocol run consists of the parallel execution of the card’s and the PSAM’s part of the protocol. The card and PSAM begin the protocol by exchanging certificates \( ID_C, K_C, \text{Sign}_{K_{CA}}(ID_C::K_C) \) (resp. \( ID_P, K_P, \text{Sign}_{K_{CA}}(ID_P::K_P) \)) containing their identifier \( ID_C \) (resp. \( ID_P \) and their
5.3 Common Electronic Purse Specifications

Fig. 5.6. Specification for the CEPS purchase transaction
public key $K_C$ (resp. $K_P$), together with the same information signed with the private key $K_{CA}^{-1}$ of the certification authority. Both check the validity of the received certificate (that is, they check that the signature consists of the received identifier and public key, signed with the private key $K_{CA}^{-1}$ of the certification authority, by verifying the signature with the public key $K_{CA}$).

Note that the card $C$ “knows” that it has received a valid certificate, but does not know whether it has received the certificate for the PSAM $P$ at the present physical location, because it has no information regarding the identity of $P$ that $ID_P$ itself could be verified against.

The PSAM then proceeds by sending the Debit-for-Purchase message containing the transaction number $NT$, and an encryption of the following data under the public key $k_C$ received in the card’s certificate: The concatenation of the price $M_{NT}$ of the good to be purchased, a (symmetric) session key $SK_{NT}$, and the following data signed with the private key $K_{CA}^{-1}$: the amount $M_{NT}$, the key $SK_{NT}$, $P$’s identifier $ID_P$, the data $id_C$ earlier received as $C$’s identifier, and the transaction number $NT$. The card then checks the validity of the signature with the earlier received public key $k_P$ against the received data amount $m$, the received key $sk$, the received identifier $id_P$, the own identifier $ID_C$, and the received transaction number $nt$. The card then returns, firstly, $E$, which consists of the values $ID_C$, $id_P$, $m$, and $nt$, signed with the private key $K_{CA}^{-1}$ and encrypted under the key $sk$, and, secondly, the values $m$ and $E$ signed with the key $sk$. The PSAM verifies that the second part of the received message is the concatenation of the amount $M_{NT}$ sent out previously and the first part of the message, signed with the key $SK_{NT}$ sent out earlier, and verifies that the first part of the message, after decryption with the key $SK_{NT}$, gives the signature of the concatenation of the values $id_C$, $ID_P$, $M_{NT}$, and $NT$. If all the verifications succeed, the protocol finishes, otherwise the execution of the protocol stops at the failed verification.

Security Threat Model

The CEPS require the smart card and the PSAM (but not the POS device [CEP01, Bus.req, p. 13, Funct. req, p. 20]) to be tamper-proof. The purchase transaction is supposed to provide mutual authentication between the terminal and the card using a certificate issued by a certification authority and containing the card’s or PSAM’s public key.

The smart card is inserted into a POS device and can thus communicate with the PSAM. Since there is no direct communication between the cardholder and the card, the information displayed by the POS device regarding the transaction has to be trusted at the point of transaction. Security for the customer against fraud by the merchant is supposed to be provided by checking the card balance after the transaction and complaining to the merchant, and if necessary to the card issuer, in the case of incorrect processing. Similarly, security for the merchant against the customer is supposed to be provided by exchanging the purchased good only for a
signed message from the card containing the transaction details, for which
the merchant will receive the corresponding monetary amount from the is-
suer in the settlement process afterwards. More precisely, the merchant pos-
sessing the PSAM with identifier \( ID_P \) will, when presenting the signature
\( E = \text{Sign}_{K_C^{-1}}(ID_C :: ID_P :: M_{NT} :: NT) \), receive the monetary amount \( M_{NT} \)
from the account of the cardholder of the card with identifier \( ID_C \), once for each \( NT \), provided \( K_C \) is the key for \( ID_C \).

The idea is that risk of fraud is kept small since fraud should be either
prevented or at least later detected in the settlement, and certificates of cards
or PSAMs actively involved in fraud can be revoked (using revocation lists
whose treatment is omitted here). Note that some kinds of fraud can only
be detected after a transaction. For example, the cardholder is unable to
communicate with the card directly to authorize the transaction. Therefore,
the POS device could simply charge a higher amount to the card than shown
in its display.

Thus we have the following three security goals:

Cardholder security: The merchant can only claim the amount which is reg-
istered on the card after the transaction (and thus can be checked with
the cardholder’s cardreader).

Merchant security: The merchant receives a valid signature in exchange for
the sold good.

Card issuer security: The sum of the balances of all valid cards and all valid
PSAMs remains unchanged by the transaction.

When investigating the threats, one needs to take into account that the
protocol is also expected to be used over the Internet, and that the POS
device in which the PSAM resides and which provides the communication link
between the card and the PSAM is not considered to be within the security
perimeter, as mentioned above.

The above discussion leads us to the following formalized security goals.

We call a key \( K_X \) valid for a card or PSAM with identifier \( ID_X \) if there exists
\( \text{Sign}_{K_{CA}^{-1}}(ID_X :: K_X) \) in a participant’s knowledge.

Cardholder security: For all \( ID_C, ID_P, M_{NT}, NT, K_C^{-1} \) such that \( K_C \) is valid
for \( ID_C \), if \( P \) is in possession of \( \text{Sign}_{K_C^{-1}}(ID_C :: ID_P :: M_{NT} :: NT) \) then
\( C \) is in possession of \( \text{Sign}_{K_P^{-1}}(M_{NT} :: SK_{NT} :: ID_P :: ID_C :: NT) \) (for some
\( SK_{NT} \) and \( K_P^{-1} \)) such that the corresponding key \( K_P \) is valid for \( ID_P \).

Merchant security: Each time \( D \) receives the value \( M_{NT} \), \( P \) is in possession
of \( \text{Sign}_{K_{CA}^{-1}}(ID_C :: K_C) \) and \( \text{Sign}_{K_{C}^{-1}}(ID_C :: ID_P :: M_{NT} :: NT) \) for some
\( ID_C, K_C^{-1} \), and a new value \( NT \).

Card issuer security: After each completed purchase transaction, let \( S \) be the
sum of all \( M_{NT} \) in the sequence consisting of the processed elements of the
form \( \text{Sign}_{K_{CA}^{-1}}(ID_C :: ID_P :: M_{NT} :: NT) \) (with possibly varying \( ID_C, ID_P, \)
and \( K_C^{-1} \)), such that the corresponding key \( K_C \) is valid for \( ID_C \) and where
the $NT$ are mutually distinct for fixed $C$). Also, let $S'$ be the sum of all $M'_{NT'}$ in the sequence of processed $Sign_{K^{-1}_p}(M'_{NT'} :: SK'_{NT'} :: ID_C :: ID_p :: NT')$ (with possibly varying $ID_C$, $ID_p$, and $K^{-1}_p$, such that the corresponding key $K_p$ is valid for $ID_p$, and where the $NT'$ are mutually distinct for fixed $C'$). Then $S$ is no greater than $S'$.

Results

According to the assumptions of the CEPS, we consider a threat scenario where the attacker is able to access the POS device links, and can access other PSAMs over the Internet, but is not able to tamper with the smart cards (that is, the insider attacker from Fig. 4.7).

Vulnerability

Under the current threat scenario, we find the following weakness with regards to the above goal of merchant security, arising from the fact that the POS device is not secured against a potential attacker that may try to betray the merchant, and that the CEPS are also to be used over the Internet. The attacker could for example be an employee, which is a realistic scenario (for examples of such a situation see [And01]). We first sketch the idea of the attack informally and then exhibit a corresponding attacker within our formal model.

The idea of the attack is simply that the attacker redirects the messages between the card $C$ and the PSAM $P$ to another PSAM $P'$, for example with the goal of buying electronic content, and to let the cardholder pay for it. We assume that the attacker manages to have the amount payable to $P'$ equal the amount payable to $P$. The attacker also sends the required message to the display which will then reassure the merchant that the required amount has been received. The attack has a good chance of going undetected: the cardholder will not notice anything suspicious, because the deducted amount is correct. Also, the card registers the identifier $id_{P'}$ rather than $id_P$, but the identifiers are non-self-explanatory data that the cardholder cannot be assumed to verify (and the card has no information about what the identity of $P$ should be). Furthermore, the identifier $id_C$ in the Deb message is as expected, since $P'$ correctly assumes to be in a transaction with $C$. The merchant who owns $P$ will notice only later a lacking amount of $M_{NT}$. Note that the PSAM $P$ is not in any way involved in this attack.

The message flow diagram corresponding to this attack follows (using the notation of Fig. 5.6 and where $E := \{Sign_{K^{-1}_C}(ID_C :: ID_{P'} :: M_{NT} :: NT)\}_{sk}$).
5.3 Common Electronic Purse Specifications

We now show that this attack is actually detected in our formal model, by exhibiting a suitable attacker.

**Theorem 5.7.** \( P \) does not provide merchant security against insider adversaries with \( \{\text{Sign}_{K_{CA}^{-1}}(ID_{C'}::K_{C'}), K_{C'}^{-1}\} \subseteq K_A^P \).

This vulnerability has been reported in [JW01b] (in a simplified protocol model) and the CEPS security working group has been informed (whose chairman acknowledged the weakness [Hit01]).

Note also that the attack is simplified if we assume that the attacker can also eavesdrop on the connection between the terminal where the amount \( M_{NT} \) is entered and the PSAM \( P \) (then the attacker only has to intercept \( M_{NT} \), redirect all messages from \( C \) to \( P' \) and back, and finally send \( \text{Disp}(M_{NT}) \) to the display). If in addition to this we assume that the cardholder coincides or collaborates with the attacker, the attacker could simply intercept and remove \( M_{NT} \) and send \( \text{Disp}(M_{NT}) \) to the display, because then the cardholder receives the good without having to pay for it.

**Proposed Solution**

The problem can be solved by securing the communication link between the PSAM and display, for example by using a smart card with integrated display as the PSAM (and by making sure that this PSAM cannot be replaced without being noticed). This modification leads to the specification \( P' \) with the modified deployment diagram given in Fig. 5.7 (and an otherwise unchanged protocol specification).

We now discuss the security of the improved version of the protocol.
Fig. 5.7. Modified part of CEPS purchase specification

Firstly, we argue that the specification provides the security properties against insider adversaries ascribed to it according to its stereotypes following Sect. 4.1.2.

**Proposition 5.8.** $P'$ provides secrecy of $K^{-1}_C, K^{-1}_P$ and integrity of $K_1^{-1}, K_C, K_{CA}, ID_C, K_{P^{-1}}, K_P, M_{NT}, SK_{NT}, NT$ against insider adversaries with $K_A^p \cap \{K_C^{-1}, K_P^{-1}\} = \emptyset$.

Note that the proposition does not imply that $C$ and $P$ terminate the protocol with the same value for $M_{NT}$. In fact, this cannot be guaranteed, since a “redirection attack” similar to the above still applies, only that the display can no longer be manipulated, which means that it would be noticed immediately if the PSAM received less money than expected (but the money could in principle still come from a different card than the one inserted into the POS device). The kinds of integrity property relevant here are considered below as “cardholder security” and “merchant security”.

Note also that the secure definition of $M_{NT}$ (which is outside the current specification) relies on a secure connection between the terminal where the amount is entered and the PSAM. Also, the creation of the session keys $SK_{NT}$ is outside current scope (the values are simply assumed to be given).

We consider the formalized security goals from the above.

**Theorem 5.9.** Consider adversaries of type $A = \text{insider}$ with

$$K_A^p \cap \left( \{K_C^{-1}, K_P^{-1}, K_{CA}^{-1}\} \cup \{SK_{NT} : NT \in \mathbb{N}\} \right)$$

$$\cup \{Sign_{K_C}(E) : E \in \text{Exp}\} \cup \{Sign_{K_P}(E) : E \in \text{Exp}\}$$

$$\cup \{Sign_{SK_{NT}}(E) : E \in \text{Exp} \land NT \in \mathbb{N}\} = \emptyset$$
and such that for each \( X \in \mathbf{Exp} \) with \( \text{Sign}_{K_C^{-1}}(X :: K) \in K_A^\mathcal{P} \), \( X = ID_C \) implies \( K = K_C \) and \( X = ID_P \) implies \( K = K_P \). The following security guarantees are provided by \( \mathcal{P}^\mathcal{A} \) in the presence of adversaries of type \( \mathcal{A} \):

**Cardholder security:** For all \( ID_C, ID_P, M_{NT}, NT, K_C^{-1} \) such that \( K_C \) is valid for \( ID_C \), if \( P \) is in possession of \( \text{Sign}_{K_C^{-1}}(ID_C :: ID_P :: M_{NT} :: NT) \) then \( C \) is in possession of \( \text{Sign}_{K_C^{-1}}(M_{NT} :: SK_{NT} :: ID_P :: ID_C :: NT) \) (for some \( SK_{NT} \) and \( K_P^{-1} \) such that the corresponding key \( K_P \) is valid for \( ID_P \)).

**Merchant security:** Each time \( D \) receives the value \( M_{NT} \), \( P \) is in possession of \( \text{Sign}_{K_C^{-1}}(ID_C :: K_C) \) and \( \text{Sign}_{K_C^{-1}}(ID_C :: ID_P :: M_{NT} :: NT) \) for some \( ID_C, K_C^{-1} \), and a new value \( NT \).

**Card issuer security:** After each completed purchase transaction, let \( S \) be the sum of all \( M_{NT} \) in the sequence consisting of the processed elements of the form \( \text{Sign}_{K_C^{-1}}(ID_C :: ID_P :: M_{NT} :: NT) \) (with possibly varying \( ID_C, ID_P, \) and \( K_C^{-1} \), such that the corresponding key \( K_C \) is valid for \( ID_C \) and where the \( NT \) are mutually distinct for fixed \( C \)). Also, let \( S' \) be the sum of all \( M_{NT}' \) in the sequence of processed \( \text{Sign}_{K_P^{-1}}(M_{NT}' :: SK_{NT}' :: ID_C :: ID_P :: NT') \) (with possibly varying \( ID_C', ID_P', \) and \( K_P^{-1} \), such that the corresponding key \( K_P' \) is valid for \( ID_P' \), and where the \( NT' \) are mutually distinct for fixed \( C' \)). Then \( S \) is no greater than \( S' \).

Note that the card cannot verify that the identity \( ID_P \) corresponds to the actual PSAM with which it communicates; the certificate only proves that \( K_P \) is a valid public key that is linked to some identity \( ID_P \). There is no information in \( ID_P \) that links it to the physical POS device containing the PSAM owning \( ID_P \) (such as the name of the shop, or its location); this information exists only at the card issuer and is not obtained during the transaction. Thus, the card "knows" it owes money to the PSAM \( P \) with which it communicates, but does not know whether \( P \) is registered as being in the physical location where the card currently is (and the card does not know what this physical location is). Including this information would probably improve the security of the protocol (for example, the attack described above could be detected by the cardholder immediately after the transaction with a portable cardreader, even if the POS device display is not within the security perimeter), but would probably also incur higher organizational expenses. Even the validity of \( ID_P \) is not relevant to the cardholder in the case of a successful purchase; if \( ID_P \) is not a valid identity, the cardholder will have the purchased good anyway, but may not have to pay for it because in the settlement process there will not be a legitimate claimer of the money. However, the validity of \( ID_P \) gives the cardholder a better prospect of claiming back an amount which has been illegitimately charged to the card by a POS device, and therefore the certificate for the POS is not redundant.
Fig. 5.8. Specification for load transaction
5.3.2 Load Transaction

Load transactions in CEPS are on-line transactions using symmetric cryptography for authentication. We only consider unlinked load (where the card-holder pays cash into a (possibly unattended) loading machine and receives a corresponding credit on the card) since linked load (where funds are transferred for example from a bank account) offers fewer possibilities for fraud according to the CEPS, because funds are moved only within one financial institution [CEP01, Funct, Req, p. 12] (a verification of this judgment is outside the scope of the current work).

A load secure application module (LSAM) is used by load acquirers for unlinked loads. It provides the necessary cryptographic and control processing. The LSAM may reside within the load device or at the load acquirer host. The load acquirer keeps a log of all transactions processed.

Specification

We give a specification of the CEPS load transaction (slightly simplified by leaving out security-irrelevant details, but including exception processing).

The specification is given in form of the UML subsystem $L$ in Fig. 5.8 (and, for readability, the enlarged class and statechart diagrams are given in Fig. 5.9 to 5.12; also, the values exchanged in the protocol are listed in Fig. 5.14). For illustration, we also give a sequence diagram for one scenario of the system behavior (namely, the case where no exception occurs) in Fig. 5.13.

Again, we use the notation $\text{var} ::= \text{exp}$ (where $\text{var}$ is a local variable not used for any other purpose and $\text{exp}$ may not contain $\text{var}$) as a syntactic shortcut. Before assigning a semantics to the diagram, the variable $\text{var}$ should be replaced by the expression $\text{exp}$ at each occurrence. Also, for increased readability, we use pattern matching: for example, $(lda', m') ::= \text{args}_{C,1}$ means that when deriving the formal semantics of the sequence diagram, one would have to replace $lda'$ with $\text{args}_{C,1,1}$ and $m'$ with $\text{args}_{C,1,2}$ in each case.

As with the purchase protocol, the link between the LSAM and the loading device, and the loading device itself, need to be secured (otherwise an attacker could initiate the protocol without having inserted cash into the machine). For simplicity, we leave out the communication between the LSAM and loading device to determine the amount to be loaded, but assume that the amount is communicated to the LSAM in a secure way. Here, a CEP card name $\text{exp}$ is called valid if the name is registered at the card issuer and not on the list of revoked cards.

For the participants of the protocol, we have the classes Card, LSAM, and Issuer. Also, each of the three classes has an associated class used for logging transaction data (named CLog, LLog, and ILog, respectively). The logging objects simply take the arguments of their operations and update their attributes accordingly; their behavior is for readability omitted in Fig. 5.8.
We assume a sequence of random values $r_{nt}$ to be given that is shared between the card $C$ and its card issuer $I$. These random values are required to be fresh within the Load subsystem (as indicated by the tag $\{\text{fresh}\}$ attached to Load; see Sect. 4.1.2). Note that when viewing the Load subsystem in isolation, the associated condition is vacuous (requiring that any appearance of an expression $r_{nt}$ in Load must be in Load). Using the $\{\text{fresh}\}$ tag at a top-level subsystem is still meaningful, because one may want to include the subsystem in another subsystem also stereotyped $\{\text{data security}\}$, which would extend the scope of the freshness constraint to the larger subsystem. In this example, it would not make sense to attach the $\{\text{fresh}\}$ tag with value $r_{nt}$ to any of the objects in Load, because the random values are supposed to be shared among Card and Issuer. As usual, we write $r_{nt}$ : $\text{Data}$ to denote an array with fields in

Fig. 5.9. Load transaction class diagram

Fig. 5.10. Load transaction: load acquire
Fig. 5.11. Load transaction: card

Fig. 5.12. Load transaction: card issuer
Fig. 5.13. Sequence diagram for load transaction
Data. Also given are the random numbers $rl_n, r2l_n$ and the symmetric keys $r_n$ of the LSAM. These values are supposed to be generated freshly by the LSAM (as indicated by the tag \{fresh\} attached to LSAM). In fact, one can see that expressions of the form $rl_x, r2l_x, r_x$ (for any subexpression $x$) only appear in the object and the statechart associated with LSAM. Again, by definition of the algebra of expressions (see Sect. 3.3.3), the keys and random values (as different constant symbols in \textbf{Keys} \cup \textbf{Data}) are mutually distinct, and therefore mutually independent in the sense of Definition 7.34, by Fact 7.36 (and also independent of the other expressions in the diagram). Finally, we are given the transaction amounts $m_n$. Before the first protocol run, the card and LSAM initialize the card transaction number $nt$ and the acquirer-generated identification number $n$, respectively. Also, before each protocol run, the card and LSAM increment the card transaction number $nt$ and the acquirer-generated identification number $n$, respectively, as long as a given limit is not reached (to avoid the rolling over of the numbers).

<table>
<thead>
<tr>
<th>Variable</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C$</td>
<td>card</td>
</tr>
<tr>
<td>$L$</td>
<td>LSAM</td>
</tr>
<tr>
<td>$I$</td>
<td>card issuer</td>
</tr>
<tr>
<td>$r_\text{c,nt}$</td>
<td>secret random values shared between card and issuer</td>
</tr>
<tr>
<td>$rl_n, r2l_n$</td>
<td>random numbers of LSAM</td>
</tr>
<tr>
<td>$r_n$</td>
<td>symmetric keys of LSAM</td>
</tr>
<tr>
<td>$m_n$</td>
<td>transaction amounts</td>
</tr>
<tr>
<td>$m, rl, hl$</td>
<td>$m_n, rl_n, hl_n$ as received at card issuer</td>
</tr>
<tr>
<td>$nt$</td>
<td>card transaction number</td>
</tr>
<tr>
<td>$n$</td>
<td>acquirer-generated identification number</td>
</tr>
<tr>
<td>$ld_i$</td>
<td>load device identifier</td>
</tr>
<tr>
<td>$cep$</td>
<td>card identifier</td>
</tr>
<tr>
<td>$s1$</td>
<td>card signature: $\text{Sign}<em>{K</em>{CI}}(cep::lda::m::nt)$</td>
</tr>
<tr>
<td>$h_{\text{c,nt}}$</td>
<td>card hash value: $\text{Hash}(lda::cep::nt::r_{c,nt})$</td>
</tr>
<tr>
<td>$h_{\text{c,nt}}$</td>
<td>$h_{\text{c,nt}}$ as created at issuer</td>
</tr>
<tr>
<td>$r_{c,he}$</td>
<td>$r_{c,nt}, h_{\text{c,nt}}$ as received at load acquirer</td>
</tr>
<tr>
<td>$K_{CI}$</td>
<td>key shared between card and issue</td>
</tr>
<tr>
<td>$K_{EL}$</td>
<td>key shared between LSAM and issuer</td>
</tr>
<tr>
<td>$md_n$</td>
<td>$\text{Sign}_{r_n}(cep::nt::lda::m_n::s1::hc::hl_n::h2l_n)$ (signed by LSAM)</td>
</tr>
<tr>
<td>$h_{ld}$</td>
<td>hash of transaction data: $\text{Hash}(lda::cep::nt::rl)$</td>
</tr>
<tr>
<td>$h2l_n$</td>
<td>hash of transaction data: $\text{Hash}(lda::cep::nt::r2l)$</td>
</tr>
<tr>
<td>$s2$</td>
<td>issuer signature: $\text{Sign}<em>{K</em>{CI}}(cep::nt::s1::hl)$</td>
</tr>
<tr>
<td>$s3$</td>
<td>card signature of the form $\text{Sign}<em>{K</em>{CI}}(cep::lda::m::nt)$</td>
</tr>
</tbody>
</table>

Fig. 5.14. Values exchanged in the load specification

We give a textual explanation of the UML specification. We use the following informal convention: whenever a value $x$ is supposed to be sent from a
protocol participant $P_1$ to a participant $P_2$, the received value is written as $x'$ (the two values are distinguished since an adversary may modify the transmitted value; in the UML specification this is realized automatically, since the different statechart diagrams have separate namespaces).

The protocol between a card $C$, an LSAM $L$, and a card issuer $I$ is supposed to start after the card $C$ (issued by $I$) is inserted into a loading device containing $L$ and the cardholder inserts the amount $m_n$ of cash into the loading device.

The LSAM initiates the transaction after the CEP card is inserted into the load device, by sending the “Initialize for load” message Init with arguments the load device identifier $lda$ and the transaction amount $m_n$ (which is the amount of cash paid into the load device by the cardholder that is supposed to be loaded onto the card). Whenever the card receives this message after being inserted into the load device, it sends back the “Initialize for load response” message RespL to the LSAM, with arguments the card identifier $cep$, the card’s transaction number $nt$, the card signature $s_1$, and the hash value $hc_{nt}$. $s_1$ consists of the values $cep$, the received load acquirer identifier $ld_{a}'$ and amount $m'$, and $nt$, all of which are signed with the key $K_{CI}$ shared between the card $C$ and the corresponding card issuer $I$. $hc_{nt}$ is the hash of the values $lda$, $cep$, $nt$, and $rc_{nt}$. $rc_{nt}$ is a secret shared between the card and the issuer. The LSAM then sends to the issuer the “load request” message Load with arguments the received card identifier $cep'$, $lda$, $m_n$, the received transaction number $nt'$ and card signature $s_{1}'$, and the values $Enc(K_{LL}, r_n)$, $ml_n$, $hl_n$, and $h2l_n$. $Enc(K_{LL}, r_n)$ is the encryption of the key $r_n$ under the key $K_{LL}$ shared between the LSAM and the issuer. $ml_n = Sign_{\rho} (cep': nt': lda :: m_n :: s_{1}' :: hc': hl_n :: h2l_n)$ is the signature of the data $cep'$, $nt'$, $lda$, $m_n$, $s_{1}'$, $hc'$ (which is $hc_{nt}$ as received by the LSAM), $hl_n$, and $h2l_n$ using the key $r_n$, $hl_n$ is the hash of the values $lda$, $cep'$, $nt'$, and $rl_n$ and $h2l_n$ the hash of the values $lda$, $cep'$, $nt'$, and $r2l_n$.

The issuer checks if the received card identifier $cep''$ is valid and verifies if the received signature $s_{1}''$ is a valid signature generated from the values $cep''$, the received load device identifier $lda''$, the received amount $m''$, and the received transaction number $nt''$ with the key $K_{CI}$ (that is, if $Ext_{K_{CI}} (s_{1}'') = cep'' :: lda'' :: m'' :: nt''$ holds). The issuer retrieves $r'$ from the received ciphertext $R$ (presumably $Enc(K_{LL}, r)$) (using the key $K_{LL}$ shared between the LSAM and the issuer, that is $r' := Dec_{K_{LL}} (R)$) and checks if the received signature $ml'$ is a valid signature of the values $cep'$, $nt'$, $lda'$, $m'$, $s_{1}''$, $hc_{nt}$, hl, and $h2l$ using the key $r'$, that is if $Ext_{1} (ml') = cep :: nt :: lda :: m :: s_{1} :: hc_{nt} :: hl :: h2l$. Here $hc_{nt}$ is the hash of the values $lda''$, $cep''$, $nt''$, and $rc_{nt}$.

If all these checks succeed, the issuer sends the “respond to load” message RespL with argument $s_2$ to the LSAM. $s_2$ consists of the values $cep''$, $nt''$, $s_{1}''$, and $hl'$, signed with the key $K_{CI}$.
Otherwise, the issuer sends $\text{RespL}$ with argument 0 to the LSAM, sends
the message $\text{llog}$ with arguments $cep", lda"$, the amount 0 (since the load was
unsuccessful), $nt", r", ml'$, and 0 (no $r2l$ received from LSAM) to its logging
object and finishes the protocol run.

If the LSAM receives an $s2' \neq 0$ as the argument of $\text{RespL}$, it sends the
“credit for load” message $\text{Credit}$ with arguments the received signature $s2'$
and the value $rl$ to the card.

If the LSAM receives a zero as the argument of $\text{RespL}$, it sends the “credit
for load” message $\text{Credit}$ with arguments 0,0 to the card and finishes the
protocol run by returning the cash to the cardholder.

If the card receives the message $\text{Credit}$, it checks whether its first argument
$s2'$ is the signature of the values $cep$, $nt$, $s1$, and $hl"$, which is defined to be the
hash of the values $lda'$, $cep$, $nt$, and the second argument $rl'$ of $\text{Credit}$. Also, it
checks whether $rl' \neq 0$.

If either of the two checks fail, the card sends the “response to credit for
load” message $\text{RespC}$ with arguments $s3$ and $rcnt$ to the LSAM, where $s3$
consists of the values $cep$, $lda'$, the amount 0, and $nt$, signed with the key
$K_{CI}$. The card also sends the logging message $\text{Clog}$ to the object $\text{CLog}$, with
arguments $lda'$, the amount 0, $nt$, $s2'$, and $rk'$.

If both checks succeed, the card attempts to load itself with the amount
$m'$. If it succeeds, it sends the message $\text{RespC}$ with arguments $s3$ and 0, where
$s3$ is defined to be the signature of the values $cep$, $lda'$, $m'$, and $nt$ using the
key $K_{CI}$. If it fails, it sends the message $\text{RespC}$ with arguments $s3$ and $rcnt$,
where $s3$ is defined to be the signature of the values $cep$, $lda'$, the amount 0,
and $nt$ using the key $K_{CI}$.

If the LSAM receives a message $\text{RespC}$ with arguments $s3'$ and $rc'$ (assuming
it has not finished already), it checks whether $rc' \neq 0$ and the $hc'$ received
in the first message from the card is the hash of the values $lda'$, $cep'$, $nt'$, and
$rc'$.

If yes (that is, the load was unsuccessful), the LSAM sends the “transaction
completion message” $\text{Comp}$ with arguments $cep'$, $lda$, the amount 0, $nt'$, $r2l$,
and $s3'$ to the issuer, Also, it sends the logging message $\text{llog}$ with arguments
$cep'$, the amount 0, $nt'$, and $rc$ to its logging object $\text{lLog}$. Then it finishes by
returning the cash to the cardholder.

If no, the LSAM sends the message $\text{Comp}$ with arguments $cep'$, $lda$, $m_n$, $nt'$, 0 (no $r2l$), and $s3'$ to the issuer. Also, it sends $\text{llog}$ with arguments
$cep', m, nt'$, and 0 to $\text{lLog}$. Then it finishes without returning the cash to the
cardholder.

If the issuer device receives the message $\text{Comp}$ with arguments $cep"$, $lda"$, $m", nt", r2l$, and $s3'$ from the LSAM (assuming it has not finished already),
it sends the message $\text{llog}$ with arguments $cep"$, $lda", m", nt", r", ml'$, and $r2l$
to the object $\text{lLog}$ and finishes. In this case, either $m"$ is supposed to be the
transaction amount and $r2l = 0$, or $m" = 0$ and $r2l \neq 0$. 

Security Threat Model

We consider the threat scenario for the load transaction. Again, the assumption is that the card, the LSAM, and the security module of the card issuer are tamper-resistant (in particular that the contained secret keys cannot be retrieved). The protocol can, for example, be attacked by attacking the communication links between the protocol participants. Also, one of the participants (cardholder, load acquirer, or card issuer) could exchange their respective device with one exhibiting different behavior. Again, since there is no direct communication between the cardholder and the card, security for the customer against fraud by the load acquirer is supposed to be provided by checking the card balance after the transaction and complaining to the load acquirer, and if necessary to the card issuer, in the case of incorrect processing.

Security for the load acquirer against the customer partly relies on the fact that the signed message from the load acquirer acknowledging receipt of the payment is sent to the card only after the cash is inserted into the loading device. However, since the load acquirer is obliged to return the cash in the case of a failure in the loading process, one needs to make sure in turn that the cash is returned only in exchange for a valid certificate from the card stating that the loading process has been aborted (otherwise the cardholder could later claim not to have received the cash-back).

More precisely, the value \( ml_n \) "provides a guarantee that the load acquirer owes the transaction amount to the card issuer", as required in [CEP01, Tech. Spec. 6.6.1.6] (for each new \( n \)). This guarantee is negated if the load acquirer is in possession of the value \( rc_{nt} \) (that is, sent from the card to the LSAM in the case the card wants to abort the loading protocol after the LSAM has released \( ml_n \)). A failed load is signaled by the LSAM to the issuer by sending the value \( r2l_n \), which can be verified by the card issuer by computing the hash of \( lda :: cep :: nt :: r2l_n \) and comparing it to the value \( h2l_n \) received earlier from the LSAM. The load acquirer can verify that \( rc_{nt} \) is genuine by comparing the hash of \( lda :: cep :: nt :: rc_{nt} \) with the value \( hc_{nt} \) received in the first message from the card, which is checked to be genuine by the card issuer, who receives it in the value \( ml_n \). The value \( rl_n \) gives a guarantee by the LSAM to the card that the load can be completed and that the load acquirer will pay the transaction amount to the card issuer. The card can verify the validity of \( rl_n \) by computing the hash \( h_{ld} \) of \( lda :: cep :: nt :: rl_n \) and verifying that the signature \( s2 \) forwarded by the LSAM from the card issuer was constructed from \( cep :: nt :: s1 :: h_{ld} \). The signatures \( s1 \) and \( s3 \) from the card indicate, respectively, the card's intention to load the contained amount and the card's notification to have loaded the contained amount.

While it may seem reasonable that the cardholder trusts the card issuer, it may not be reasonable to expect that the load acquirer trusts the card issuer. The aim of the CEPS is to provide a globally interoperable system. Since many card issuers will also operate load acquirers within their regional boundaries, this means that cardholders must be able to load their cards at load acquirers
(outside these boundaries) that are operated by competing card issuers. Competing card issuers may not trust each other, especially when jointly operating a relatively complex system that may provide temptation for fraud even at corporate level (that this temptation exists in practice can be deduced from an example in [And01] about the urban train operators in a major English metropolis that attempted to cheat each other about passenger numbers on their respective parts of the urban train system to increase their own revenue at the expense of their competitors). The CEPS plainly contend that “electronic purse system participants must be assured that load/unload devices must not link to the system without security that protects all participants from fraud” [CEP01, Bus. req. p. 19]. However, the cardholder and the load acquirer may not trust each other, and the card issuer may not trust either the cardholder or the load acquirer. In particular, the issuer needs to have valid proof in case the cardholder or the load acquirer disputes a transaction in the post-transaction settlement process. Thus the security of the system relies crucially on the validity of the audit data.

Following the above discussion, we derive the following security conditions:

Cardholder security: If the card appears to have been loaded with a certain amount according to its logs, the cardholder can prove to the card issuer that there is a load acquirer who owes the amount to the card issuer.

Load acquirer security: A load acquirer has to pay an amount to the card issuer only if the load acquirer has received the amount in cash from the cardholder.

Card issuer security: The sum of the balances of the cardholder and the load acquirer remains unchanged by the transaction.

Note that the protocol does not ensure that if the cardholder inserts cash into the loading device, the card will be loaded – there is the usual risk that the machine simply retains the money without further action, or loads the card with a smaller amount than was inserted. In this case the cardholder can only make a complaint, if necessary through the card issuer in the post-transaction settlement scheme. The correct functioning of the settlement scheme relies on the fact that the cardholder should only be led to believe (for example, when checking the card with a portable cardreader) that a certain amount has been correctly loaded if the cardholder is later able to prove this using the card – otherwise the load acquirer could first credit the card with the correct amount, but later in the settlement process claim that the cardholder tried to fake the transaction.

Results

We turn to the formalizations of the above security conditions.

We start with the condition providing security for the load acquirer. According to the CEPS, the value $ml_n$, together with the value $rl_n$, sent in the
Credit for Load message to the card, is taken as a guarantee that the amount $m$ specified in $ml_n$ has to be paid by the specified load acquirer to the issuer of the specified card, unless it is negated with the value $rc_{nt}$ [CEP01, Tech. Spec. 6.6.1], The security condition is thus formalized as follows:

Load acquirer security: Suppose that the card issuer $I$ possesses the value $ml_n = \text{Sign}_{r_n}(cep :: nt :: lda :: m_n :: s1 :: hc_{nt} :: hl_n :: h2l_n)$ and that the card issuer $C$ possesses $nl_n$, where $h_n = \text{Hash}(lda :: cep :: nt :: nl_n)$. Then after execution of the protocol either of the following two conditions hold:

- a message $L \text{Log}(cep, lda, m_n, nt)$ has been sent to $L : L \text{Log}$ (which implies that $L$ has received and retains $m_n$ in cash) or
- a message $L \text{Log}(cep, lda, 0, nt)$ has been sent to $L : L \text{Log}$ (that is, the load acquirer assumes that the load failed and returns the amount $m_n$ to the cardholder) and the load acquirer $L$ has received $rc_{nt}$ with $hc_{nt} = \text{Hash}(lda :: cep :: nt :: rc_{nt})$ (thus negating $ml_n$).

Vulnerabilities

When trying to prove the above condition, one comes across the following weaknesses which break both conditions required to hold for load acquirer security. We first explain the problem intuitively before we prove the corresponding result.

Firstly, the value $ml_n$ is only protected with the key $r_n$ which in turn is only protected with the key $K_{LL}$ shared between the load acquirer and the card issuer. Further, the hash value $hl_n$ does not depend on the amount $m$. Thus the card issuer can modify the amount $m_n$ contained in $ml_n$ to a greater amount $\tilde{m}$. In more detail, having received $\{r_n\}_{K_{LL}}$ from the load acquirer, the issuer can replace the value $ml_n = \text{Sign}_{r_n}(cep :: nt :: lda :: m_n :: s1 :: hc_{nt} :: hl_n :: h2l_n)$ received from the load acquirer by the value $\tilde{ml} = \text{Sign}_{r_n}(cep :: nt :: lda :: \tilde{m} :: s1 :: hc_{nt} :: hl_n :: h2l_n)$. Consequently, the load acquirer only receives $m_n$ in cash, but has to pay $\tilde{m}$ to the card issuer.

Here we assume that the card issuer is in the judicially stronger position (for example, the load acquirer may have signed a contract to pay whichever amount $m$ contained in such an $ml_n$). In a different judicial situation, the load acquirer might instead betray the card issuer, by claiming that the card issuer modified $ml_n$ to contain a greater amount $m$, and thus pay only the (allegedly correct) smaller amount $m'$. This is an example of the observation (put forward for instance in [And01]) that security analysis of practical systems has to take into account the legislative situation.

Secondly, there is a vulnerability against the load acquirer arising when the card sends an $rc_{nt}$ to the load acquirer in the RespC message. The only way in which the load acquirer can verify the validity of this value is against the hash $hc_{nt}$ sent from the card to the load acquirer in the RespC message. Since neither the secret $rc_{nt}$ shared between the card and the issuer nor the hash $hc_{nt}$ is protected by any signature, the load acquirer has no way to prove in the post-transaction settlement process that $rc_{nt}$ is genuine, and that thus
Fig. 5.15. Specification for modified load transaction
the cash has been returned to the cardholder: The card issuer can simply claim that the card did not send a value \( r_{\text{nt}} \) to the load acquirer, but that the load acquirer invented \( r_{\text{nt}} \) (and computed \( h_{\text{nt}} \) from it). Since the card issuer controls the settlement process, the load acquirer would have to pay (or go to court, with unclear prospects of success).

**Theorem 5.10.** \( \mathcal{L} \) does not provide load acquirer security against adversaries of type insider with \( \{ \text{cep}, \text{lda}, m_n \} \subseteq \mathcal{K}_{\text{nt}}^\alpha \).

Again, the CEPS security working group has been informed.

Note that even if the signatures \( s_1 \) and \( s_3 \) are considered part of the guarantee that the load acquirer has to pay the contained amount, this does not remove the weakness entirely, but only requires the card issuer to also modify the issued cards. The load acquirer is not able to verify that the signatures \( s_1 \) and \( s_3 \) created with the key \( K_{\text{CI}} \) shared between the card and the issuer contain the correct amount \( m \).

**Proposed Solution**

We propose the following modifications to the protocol:

- \( ml_n \) should be protected by an asymmetric key: \( ml_n := \text{Sign}_{K_{\text{nt}}^{-1} \cdot (\text{cep} \cdot \text{nt} \cdot \text{lda} \cdot m \cdot sl \cdot hc \cdot hl_n \cdot h2l_n)} \) for a private key \( K_{\text{nt}}^{-1} \) of the load acquirer with associated public key \( K_{\text{nt}} \), and
- in the message RespL, the issuer should also send a signature certifying the validity of \( h_{\text{nt}} \): \( \text{RespL}(s2, \text{Sign}_{K_{\text{nt}}^{-1}}(h_{\text{nt}})) \) for a private key \( K_{\text{nt}}^{-1} \) of the card issuer with associated public key \( K_{\text{nt}} \).

The modified UML subsystem specification \( \mathcal{L}' \) is given in Fig. 5.15 (and for readability, the enlarged class and the modified statechart diagrams are given in Fig. 5.16 to 5.18), with the corresponding exemplary sequence diagram in Fig. 5.19.

We now discuss the security of the improved version of the protocol.

Firstly, we argue that the specification is a well-defined UMLsec specification in the sense of Sect. 4.1.2.

**Proposition 5.11.** \( \mathcal{L}' \) provides secrecy of \( K_{\text{CI}}, K_{\text{nt}}^{-1}, K_{\text{nt}}^{-1} \) and integrity of \( K_{\text{CI}}, K_{\text{nt}}^{-1}, K_{\text{nt}}^{-1}, \text{cep}, \text{nt}, r_{\text{nt}}, \text{lda}, n, nl_n, r2l_n, m_n \) against insider adversaries with \( \mathcal{K}_{\text{nt}}^\times \cap \{ K_{\text{CI}}, K_{\text{nt}}^{-1}, K_{\text{nt}}^{-1} \} = \emptyset \).

We now consider the following formalizations of the above security goals with respect to the modified specification.

We consider:

- the joint knowledge set \( \mathcal{K} \) of all participants except \( L \) (that is, any object in the classes Card or Issuer, any adversary – that is, according to the threat scenario, not able to penetrate the smart card on which \( L \) resides), and any object in LSAM except \( L \) – and
5.3 Common Electronic Purse Specifications

Fig. 5.16. Modified load transaction class diagram

Fig. 5.17. Modified load transaction: load acquirer

- the knowledge set $K_L$ of $L$.

**Theorem 5.12.** In the presence of adversaries of type $A = \text{insider}$ with

$$K_A^p \cap \{ K_{CI}, K_L^{-1}, K^{-1}_L \} \cup \{ rc_{nt} : nt \in \mathbb{N} \} \cup \{ rl_m, rl_{m+1} : m \in \mathbb{N} \} = \emptyset$$

the following security guarantees are provided by $L'$:

**Cardholder security:** For any message $Clog(\text{lda}, m, nt, s2, r)$ sent to $c : CLog,

if $m \neq 0$ (that is, the card seems to have been loaded with $m$) then $rl \neq 0$

and

$$\mathcal{E}_{K_{CI}}(s2) = \text{cep} :: nt :: \text{Sign}_{K_{CI}}(\text{cep} :: \text{lda} :: m :: nt) ::
\text{Hash}(\text{lda} :: \text{cep} :: nt :: rl)$$
Fig. 5.18. Modified load transaction: card issuer

holds (that is, the card issuer certifies rl to be a valid proof for the trans-
action). For any two messages Clog(ida, m, nt, s2, rl) and Clog(ida', m',
nt', s2', rl') sent to c : CLog, we have nt ≠ nt'.

Load acquirer security: Suppose that we have mln ∈ K and rln ∈ K where
mln = SignK−1(cep :: nt :: lda :: m :: nt :: y :: hl :: h2ln) with hl = Hash
(lda :: cep :: nt :: rl), and h2ln = Hash(lda :: cep :: nt :: r2ln), for some
c, ep, nt, s1, and y. Then at the end of an execution of L either of the
following two conditions hold:

- a message Llog(cep, ida, m, nt, x) has been sent to l : LLog (which
implies that L has received and retains m, in cash) or
- a message Llog(cep, ida, 0, nt, x) has been sent to l : LLog, for some
x (that is, the load acquirer assumes that the load failed and re-
turns the amount m, to the cardholder), and we have x' ∈ K and
z ∈ K with z = SignK−1(cep :: ida :: m :: nt :: y') where y' =
Hash(lda :: cep :: nt :: x') = y (that is, the load acquirer can prove that
the load was aborted).

Card issuer security: For each message Clog(ida, m, nt, s2, rl) sent to c : CLog,
if m ≠ 0 and

\[ \text{Ext}_{KCI}(s2) = \text{cep} :: nt :: \text{Sign}_{KCI}(cep :: ida :: m :: nt) \]
Fig. 5.19. Sequence diagram for modified load transaction
We had to change the condition of \textit{load acquire security} slightly to accommodate the changes in the protocol. To see that it is formalized in an adequate way, note that a value \( m_{l_n} = \text{Sign}_{K^{-1}}(cep :: nt :: lda :: m_n :: s1 :: hc :: h2l_n) \) is known outside \( L \) only after the load acquirer has received the amount \( m_n \) in cash. This follows from the facts that a protocol at \( L \) is started only after the cash is inserted, that \( m_{l_n} \) is signed with the key \( K^{-1} \), and that this key is only accessible to \( L \) by Proposition 5.11. Thus the critical question is whether the cash is returned to the cardholder after \( rl_n \) becomes known outside \( L \). According to the specification of \( L \) this may happen only after a message of the form \( \text{Ilog}(cep, 0, nt, \text{rc}) \) is sent to \( L : \text{ILog} \).

### 5.4 Developing Secure Java Programs

Dynamic access control mechanisms such as those provided by Java since the JDK 1.2 Security Architecture [Gon99, Kar00] in the form of GuardedObjects can be difficult to administer since it is easy to forget an access check [Gon98, BV99]. If the appropriate access controls are not performed, the security of the entire system may be compromised. Additionally, access control may be granted indirectly and unintentionally by granting access to an object containing the signature key that enables access to another object. We show how to use UMLsec to address these problems by providing means of reasoning about the correct deployment of security mechanisms such as guarded objects.

After presenting some background on access control in Java in the following subsection, we outline the part of a design process relevant to enforcing access control in Java and give some results on verifying access control requirements. In Sect. 5.4.3 we illustrate our approach with the example of the development of a web-based financial application from formal specifications.

#### 5.4.1 Access Control in Java

Authorization or \textit{access control} is one of the cornerstones of computer security [SS94]. The objective is to determine whether the source of a request is \textit{authorized} to be granted the request. Distributed systems offer additional challenges. The trusted computing bases (TCBs) may be in various locations and under different controls. Communication is in the presence of possible adversaries. Mobile code is employed that is possibly malicious. Further complications arise from the need for delegation (that is, entities acting on behalf
of other entities) and the fact that many security requirements are location-
dependent (for example, a user may have more rights at the office terminal
than when logging on from home).

In the JDK 1.0 Security Architecture, the challenges posed by mobile code
were addressed by letting code from remote locations execute within a sandbox
offering strong limitations on its execution. However, this model turned out
to be too simplistic and restrictive.

From JDK 1.2, a more fine-grained security architecture is employed which
offers a user-definable access control, and the sophisticated concept of guarded
objects [Gon99, Kar00]. Permissions are granted to protection domains. A
protection domain [SS75] is a set of entities accessible by a principal. In the
JDK 1.2, protection domains consist of classes and objects. They are specified
depending on the origin of the code (as given by a URL) and on the key with
which the code may be signed. The system security policy set by the user (or
a system administrator) is represented by a policy object instantiated from
the class java.security.Policy. The security policy maps protection domains to
sets of access permissions given to the code.

There is a hierarchy of typed and parameterized access permissions, of
which the root class is java.security.Permission and other permissions are subclassed
either from the root class or one of its subclasses. Permissions consist of
a target and an action. For file access permissions in the class FilePermission,
the targets can be directories or files, and the actions include read, write,
execute, and delete.

An access permission is granted if all callers in the current thread history
belong to domains that have been granted the said permission. The history of
a thread includes all classes on the current stack and also transitively inherits
all classes in its parent thread when the current thread is created.

The sophisticated JDK 1.2 access control mechanisms are not so easy to
use. The granting of permissions depends on the execution context. Some-
times, access control decisions rely on multiple threads, A thread may involve
several protection domains. Thus it is not always easy to see if a given class
will be granted a certain permission.

This complexity is increased by the mentioned guarded objects [Gon99].

If the supplier of a resource is not in the same thread as the consumer, and
the consumer thread cannot provide the access control context information,
one can use a GuardedObject to protect access to the resource. The supplier of
the resource creates an object representing the resource and a GuardedObject
containing the resource object, and then hands the GuardedObject to the con-
sumer, A specified Guard object incorporates checks that need to be met so
that the resource object can be obtained. For this, the Guard interface con-
tains the method checkGuard, taking an Object argument and performing the
checks. To grant access the Guard objects simply returns, to deny access it
throws a SecurityException. GuardedObjects are a quite powerful access con-
trol mechanism. However, their use can be difficult to administer [Gon98].
For example, guard objects may check the signature on a class file. This way,
access to an object may be granted indirectly (and possibly unintentionally) by giving access to another object containing the signature key for which the corresponding signature provides access to the first object.

5.4.2 Design Process

We sketch the part of a design process for secure systems using UML that is concerned with access control enforcement using guarded objects:

1. Formulate the permission sets for access control of sensitive objects,
2. Use statecharts to specify Guard objects that enforce appropriate access control checks,
3. Make sure that the Guard objects protect the sensitive objects sufficiently in that they only grant access implied by the security requirements (by making use of the precise semantics given in Sect. 8.1),
4. Ensure that the access control mechanisms are consistent with the functionality required by the system in that the objects that depend on guarded objects may perform their intended behavior (again by making use of the precise semantics).

Here the access control requirements in step (1) can be of the following form:

- origin of requesting object (based on URL)
- signature of requesting object
- external variables (such as time of day etc.)

In Sect. 5.4.3 we discuss a specification following these steps. They enforce the following two requirements:

Security requirement: The access control requirements are strong enough to prevent unauthorized influence, given the threat scenario arising from the physical layer.

Functionality requirement: The access control requirements formulated are not overly restrictive, denying legitimate access from other components of the specification.

The functionality requirement is important since it is not always easy to see if stated security requirements are at all implementable. If their inconsistency is only noticed during implementation then, firstly, resources are wasted since work has to be redone. Secondly, most likely security will be degraded in order to reduce this extra work.

Before coming to the main example in the next subsection, we give a short example to point out that the kind of weaknesses in using the Java security access control mechanisms can be quite subtle (rather than just mistakenly sending out secret keys or forgetting to set access rules):
Example

The statechart in Fig. 5.20 describes the behavior of a guard object \texttt{grd} enforcing a slightly more complicated access control policy.

To facilitate understanding, we give a typical message exchange of this access control mechanism to establish \( K_S \) in Fig. 5.21. In the envisioned situation, there is an object \texttt{req} used to grant to other objects the right to access a particular guarded object by signing the class files with a key \( K_S \). There should be a possibility to update the key \( K_S \); by substituting \( K_S \) with a different key \( K'_S \) it can be achieved that an object the class file of which is signed by \( K_S \) is no longer allowed access to the guarded object. Thus the object \texttt{req} needs to be able to submit the current signing key \( K_S \) to the guard object. For this, first a shared key \( K_M \) is established using the public key \( K \) of the guard object, which is used to encrypt the submitted key \( K_S \) (since this is more secure and more efficient than directly using \( K \), if \( K_S \) is updated rather frequently). The identity of \texttt{req} is taken as given and is bound to a public key in the certificate \texttt{cert} signed with the key \( K_{CA} \) of a certification authority. On request \texttt{cert()}, the guard object sends out a self-signed certificate certifying its public key \( K \). The object \texttt{req} sends back the symmetric key \( K_M \) signed with its private key (corresponding to the public key in \texttt{cert}) and encrypted under \( K \), together with the certificate \texttt{cert} (the functions \texttt{fst} and \texttt{snd} applied to a pair return their first and second components, respectively). The guard object can receive the signature key \( K_S \) encrypted under \( K_M \) and will then grant access to those objects in class files signed by \( K_S \). We assume that the guard object is given the signature of the requesting object using the method \texttt{checkGuard()}. Note that here we do not focus on the exception processing mechanism; thus a guard object that does not grant access simply does not return. Also, for simplicity we assume that the guard object receives the key \texttt{sig} with which the requesting applet was signed as the argument of the operation \texttt{checkGuard}, and that the execution context of the applet checks that it was actually signed with this key.
This access control mechanism, which for the sake of the example is derived from the protocol in Sect. 5.2, contains a flaw analogous to the one pointed out there: an adversary $A$ intercepting the communication between req and grd (and modifying the exchanged values) can find out $K_M$ and thus make grd accept a key $K_S$ chosen by $A$. The critical part of the message exchange corresponding to this attack is given in Fig. 5.22. Here the intended access control policy is not enforced since the preservation of secrecy of the signing key $K_S$ is violated (in a subtle way). With our approach one can exhibit subtle flaws like this (in this case, one would notice the flaw when trying to show formally that the secrecy of $K_S$ is preserved, as in Sect. 5.2).

**Fig. 5.22.** Guard object security flaw

### 5.4.3 Example: Financial Application

We illustrate our approach with the example of a web-based financial application. Although much simplified, the example points out some typical issues when considering access control for web-based e-commerce applications (namely, to have several entities -- service providers and customers -- interacting with each other while granting the other parties a limited amount of
trust and by enforcing this using credentials). Since we would only like to illustrate the general idea, we only give parts of a system specification, rather than a complete UML subsystem. We show in UML diagrams how to employ GuardedObjects to enforce these security requirements. We argue that the specification given by the UML diagrams is secure in that it does not grant any access not implied by the security requirements (again, a formal treatment is omitted since we would only like to illustrate the idea).

Two (fictional) institutions offer services over the Internet to local users: an Internet bank, Bankeasy, and a financial advisor, Finance. To make use of these services, a local client needs to grant the applets from the respective sites certain privileges:

1. Applets that are signed by the bank can read and write the financial data stored in the local database, but only between 1 pm and 2 pm (when the user usually manages his or her bank account).
2. Applets (for example, from the financial advisor) can access an excerpt of the local financial data to give the user advice on stock purchases. Since these applets also need access to the Internet to obtain stock information, but the financial information is not supposed to leave the local system, they have to be signed by a certification company, CertiFlow, certifying that they do not leak out information.
3. Applets signed by the financial advisor may use the micropayment signature key of the local user (to purchase stock rate information on behalf of the user), but this access should only be granted five times a week.

Financial data sent over the Internet is encrypted to ensure integrity and confidentiality. Access to the local financial data is realized using GuardedObjects. Thus, the relevant part of the specification is given as in Fig. 5.23.

We only give a partial specification, containing the simplified relevant part of the Java Security Architecture which receives requests for object references and forwards them to the guard objects of the three guarded objects. We omit the behavior of the guarded objects (and also the activity diagram which would include their behavior, and the deployment diagram). The access controls are realized by the Guard objects FinGd, ExpGd, and MicGd, whose behavior is specified (we assume that the condition slot is fulfilled if and only if the time is between 1 pm and 2 pm, and that the condition limit is fulfilled if and only if the access to the micropayment key has been granted less than five times in the current calendar week; we omit how to implement this). Here we assume that the execution context of an applet checks that the applet was actually signed by the authority sig whose name is given as the second argument to getObj, and again we do not model exception processing. In accordance with the UMLsec profile in Chap. 4, we assume that the names of the objects that are stereotyped «guarded» are not in $K^P_A$ and thus not initially known to the adversary (possibly contrary to the names of objects not
Fig. 5.23. Financial application specification

stereotyped «guarded»). In this way we model the passing of references in the Java 2 Security Architecture.

Now according to step (3) in Sect. 5.4.2, one may convince oneself that the guard objects sufficiently protect the guarded objects, as required by the access control requirements stated above. We omit the formal treatment. Note
that one might also formalize these requirements using an access control logic (such as [ABLP93]) and then proving that any access granted by one of the guard objects is legitimate in the sense that it may be derived from the original formalization in the logic. In this way our approach helps to bridge the gap between formal security policy models and system specifications.

Regarding step (4) in Sect. 5.4.2, one may also convince oneself that any legitimate access according to the above requirements is granted (again, a more formal statement is omitted).

5.5 Further Applications

We give a short overview of a further application of the model-based security approach we have performed. Other applications currently in progress include the modeling and analysis of a smart-card-based biometric authentication system together with a major German telecommunications and IT company, and of an automotive software-download system with a major German car manufacturer.

5.5.1 Modeling and Verification of a Bank Application

In a joint work [GHJW03] with Johannes Grünbauer, Guido Wimmel, and Helia Hollmann, in a project with a major German bank, we have applied our ideas about model-based development of security-critical systems to a Web-based banking application, by making use of the CASE tool AUTOFOCUS [SH99], which has a UML-like notation.

The application can be used by clients to fill out and sign digital order forms. The main security requirements of this application are that the personal data in the forms must be kept confidential, and that orders cannot be submitted in the name of others.

For this purpose, when the user logs in, first an authentication protocol is run and an encrypted connection is established. The second part of the transaction (filling out and digitally signing the order form) is carried out over this connection.

The authentication protocol is based on an underlying SSL connection layer which is initially established and which is supposed to provide a secure connection with regard to confidentiality and server authentication. The session key generated during the SSL handshake is used to encrypt the messages of the authentication protocol on the second layer. The protocol authenticates the client by making use of a cardreader and a smart card to compute digital signatures on the client's side. There is a need for a layered protocol here because the SSL client authentication feature cannot be used due to technical restrictions imposed by the architecture of the bank system (the web server does not support the forwarding of client certificates).
The complete protocol run is shown in Fig. 5.24. After the ClientHello message, a nonce (a randomly generated number) is sent by the web server. The client signs this nonce with its or her own private key and sends it together with its or her certificate back to the web server. The certificate contains the client’s identity, a global identification number which references the client’s data on the backend, and the client’s public key. The web server checks the signature of the nonce and compares the received nonce with the one sent before. Furthermore a plausibility check of the global ID will be done and it will be saved for later purposes. The authentication is finished after the checks have been successful. The web server now sends the global ID and an empty form to the backend system, where it is filled with the client’s data and sent back to the client. The global ID is also stored on the backend. The client signs his or her data with his or her private key, thus creating an electronic signature. The backend checks the signature of the received data object and the certificate. The received global ID and the signed data object are compared with the ones stored. On success an order is generated and an acknowledgment is sent to the client. The end of connection signal can be caused by a timeout or a logout event.

In [GHJW03], the system architecture and the protocol are specified using the tool AUTOFOCUS in a notation which is very similar to UML deployment diagrams and UML statecharts (see there for the details). This model is then verified with regard to the relevant security requirements. For this purpose, the tool AUTOFOCUS generates an input file for the symbolic model checker SMV [McM93] which carries out the actual model checking process. Also, the
required security properties are translated into the SMV language as well, and during the model checking process, SMV checks whether they are true with respect to the model (including both the modeled protocol and the adversary model). If SMV finds any flaw in the protocol, this counter-example is translated by the tool AutoFocus into a notation similar to UML sequence diagrams, which helps to understand the way the protocol can be attacked. More details can be found in [GHJW03].

5.6 Notes

Information on smart cards is gathered in [RE00]. An overview of electronic payment systems is given in [AJSW00]. Smart card protocols have been investigated using formal logic in [ABKL93]. Smart card payment systems are analyzed using formal methods in [And99, SCW00]. Parts of the results in Sect. 5.3.1, Sect. 5.3.2 resp. Sect. 5.4 have been presented in [JW01b, Ji04a], [Ji01c] resp. [Ji01h].

Java 2 security and in particular the advanced topics of signed, sealed, and guarded objects is explained in [Gon99]. There has also been some work giving formal reference models for Java 2 access control mechanisms, thus clarifying possible ambiguities in the informal accounts and enabling proof of compiler conformance to the specification [KG98, WF98, Kar00] (but without considering signed, sealed, or guarded objects). To our knowledge, the use of signed, sealed, or guarded objects in JDK 1.2 has not previously been considered in a formal model.

5.7 Discussion

We gave examples of secure systems development using UMLsec. We exemplified stepwise formal development of a security-critical system by considering a secure channel design. We uncovered a flaw in a variant of the handshake protocol of the Internet protocol TLS proposed in [APS99], suggested a correction, and verified the corrected protocol. We examined the Common Electronic Purse Specifications, discovered flaws in the two central parts of the specifications, proposed corrections, and gave a verification. We demonstrated how to use UMLsec for formal development of security-critical Java systems.

These case studies aim to demonstrate the adequacy of the UMLsec definition for modeling and verifying secure systems. We chose examples that go beyond the type of concise core protocol specifications taken from academic publications which are often used as examples for the applications of formal tools, but instead considered also some specifications that are currently put to industrial use (such as the CEPS). Necessarily, these specifications, and in particular the data types which are used, are more complex. For example, this causes some of the diagrams to have relatively complex labels for data
and messages. We should point out that this is due to the nature of the specifications themselves, and not due to that fact that UML has been used as a notation. One could argue that considering such relatively complex specifications is facilitated by the use of a graphical specification language such as UML. We believe that the case studies that were considered indicate that the fragment of UML chosen is sufficient for our needs. The actual messages that are exchanged (which in UML are written as labels on transitions) may have to be included in a specification regardless of the notation in use; using a graphical specification language has the advantage of providing a quick overview of the physical and logical structure of the system that is modeled.

Although the central message exchanges of the two CEPS protocols are sketched in [CEP01] in a sequence diagram like notation, that sketch is still far from the level of preciseness we provide here. It should be hoped, however, that the level of formality can be raised by providing a general framework which provides a payoff for the increased effort (eventually, by also providing tool support).
Part III

Tool Support
Tool support for UMLsec

For the ideas that were presented in the previous chapters to be of benefit in practice, it is important to have advanced tool support to assist in using them. In this chapter, we present the necessary background and some results achieved so far toward developing tool support for UMLsec. The developed tools can be used to check the constraints associated with UMLsec stereotypes mechanically, based on XMI output of the diagrams from the UML drawing tool in use. We also explain a framework for implementing verification routines for the constraints associated with the UMLsec stereotypes. The goal is that advanced users of the UMLsec approach should be able to use this framework to implement verification routines for the constraints of self-defined stereotypes.

Furthermore, we present research on linking the UMLsec approach with the automated analysis of security-critical data arising at runtime. Specifically, we present research on the construction of a tool which automatically checks the SAP R/3 configuration for security policy rules formulated as UML specifications. Because of its modular architecture and its standardized interfaces, the tool can be adapted to check security constraints in other kinds of application software (such as firewalls or other access control configurations).

Finally, we present some approaches for linking UML models to implementations. The aim is to ensure that the benefits gained from the model-based approach on the level of the system model (such as increased confidence in satisfaction of critical requirements) actually carry over to the implemented system, as one would hope.

6.1 Extending UML CASE Tools with Analysis Tools

We present some background useful for constructing tool support for XMI-based analysis of UML models, based on [JS03b]. In the first subsection, we explain how the syntax of UML diagrams is defined on a technical level using the Meta-Object Facility (MOF) and how the data contained in UML
diagrams can be saved using the XML Metadata Interchange (XMI) format. In the second subsection, we explain how one can conveniently access the information stored in an XMI file.

6.1.1 Meta-Object Facility (MOF)

Early tool support for processing UML models had to rely on storage formats of the various UML tools which made exchange and reuse of the models and tools impossible. Having chosen a UML tool, the developer was tied to using it through the whole project. Applying emerging technologies to the UML modeling on the industrial level was virtually impossible. To suggest any custom UML processing, one would have to develop a complete UML editor and persuade the developers to use it.

The development of XML as a universal data storage format changed this situation. In the year 2000, the Object Management Group (OMG) issued the first specification for the XML Metadata Interchange (XMI) language [Obj02b] which became a standard for exchanging UML models between tools.

The XMI language is compliant with MOF [Obj02a], which is a framework for specifying meta-information (also called meta-models). It allows software systems to be modeled particularly flexibly in an approach based on several layers of information (see Fig. 6.1). MOF is a standard defining an abstract language and a framework for specifying, constructing and managing modeling languages (also called meta-models), such as UML and CWM (Common Warehouse Model). Initially it was developed to define CORBA-based ser-

![Fig. 6.1. MOF framework: meta-levels](image)

vices for managing meta-information. Currently, its applications include the definition of modeling languages such as UML and CWM.

We explain the different MOF layers at the hand of an example in Fig. 6.2. The lowest level M0 deals with the data instances, for example “Bob Marley”, “Kingston”. The level M1 describes data models, in software development this corresponds to the UML model of the application. An example for this layer is a Person with attribute City. The next abstraction level M2 is the modeling
language itself. There exist different modeling languages for different application domains, and the last abstraction level M3 is the common environment for defining these modeling languages, standardized by the MOF. The MOF makes use of the following three concepts:

MOF objects define object types for the target model. The information associated with an MOF object includes a name, a set of attributes (which may be predefined or customized), a set of operations, a set of association references, and a set of supertypes it inherits. The MOF object is a container for its component features (its attributes, operations, and association references). It may also contain MOF definitions of data types and exceptions.

MOF associations define links between two MOF objects. These links are always binary and directed. A link is a container for two association ends, each representing one object which the link is connected to.

MOF packages group related MOF elements for reuse and modularization. An MOF package is defined by a name, a list of imports (which defines a set of other MOF packages whose components may be reused by components defined within the package), a list of supertypes which defines a set of other MOF packages whose components form a part of the package, and a set of contained elements including other objects, associations, and packages.

The MOF also defines the following secondary elements:

- Data types can be used to define constructed and reference data types.
- Constants define compile-time constant expressions.
- Exceptions can be raised by object operations.
- Constraints can be attached to other MOF elements. Constraint semantics and verification are not part of the MOF specification, and therefore they can be defined with any language.

The MOF is related to two other standards:

XML Metadata Interchange (XMI) is a mapping from MOF to XML. It can be used to automatically produce an XML interchange format for any language described with MOF. For example, to produce a standardized UML
interchange format, we need to define the UML language using MOF, and use the XMI mapping rules to derive DTDs and XML Schemas for UML serialization. The MOF itself is defined using MOF, and therefore XMI can be applied not only for meta-model instances, but for meta-models themselves (as they are also instances of a meta-model, which is an MOF). The Java Metadata Interface (JMI) standard defines a MOF-to-Java mapping (similarly to the MOF-to-XML mapping provided by XMI). It is used to derive Java interfaces tailored for accessing instances of a particular meta-model. As MOF itself is MOF-compliant, it can be used to access meta-models as well. The standard also defines a set of reflective interfaces that can be used similarly to the meta-model-specific API without prior knowledge of the meta-model.

Today, many UML editors support model interchange in the XMI format. Together with the wide support for the XML language, including a broad range of libraries, editors, and accompanying technologies, this enables development of lightweight UML processing tools.

6.1.2 XML-Based Data-Binding with MDR

There exist at least three technologies for processing XMI files:

- Common high-level languages with appropriate libraries for parsing XML files (such as Java, C++, and Perl).
- Specialized XML parsing and transformation languages (such as XPath and XSLT).
- XMI data-binding, where a framework extracts the data from an XMI file, which can then be accessed for example through a Java method.

The first two methods, although flexible, require some effort related to parsing the XMI file. This suggests trying to use XMI data-binding for our purposes. There exist libraries supporting data-binding for the more general case of XML, such as the widely used Castor library [Cas03]. However, there exist XMI-specific data-binding libraries which directly provide a representation of an XMI file on the abstraction level of a UML model. This allows the developer to operate directly with UML concepts (such as classes, statecharts, stereotypes, etc.). For UMLsec tool support, we use the MDR (Meta-Data Repository) library which is part of the Netbeans project [Net03] and also used by the freely available UML modeling tool Poseidon 1.6 Community Edition [Gen03]. Another such library is the Novosoft NSUML project [NSU03].

The MDR library implements an MOF repository with support for XMI and JMI standards. Figure 6.3 illustrates how the repository is used for working with UML models.

The XMI description of the modeling language is used to customize the MDR for working with a particular model type, UML in this case (step 1). The
XMI description of UML 1.4 is published by the Object Management Group (OMG). A storage customized for the given model type is created (step 2). Additionally, based on the XMI specification of the modeling language, the MDR library creates the JMI implementation for accessing the model (step 3). This allows the application to manipulate the model directly on the conceptual level of UML. The UML model is loaded into the repository (step 4). Now it can be accessed through the supplied JMI interfaces from a Java application. The model can be read, modified, and later saved in an XMI file again.

Because of the additional abstraction level implemented by the MDR library, using it in the UMLsec tool is hoped to facilitate upgrading to upcoming UML versions, and promises a high-standard compatibility.

6.2 Automated Tools for UMLsec

This section presents joint work with Pasha Shabalina, Ewgeny Alter, Shasha Meng, Marilfy Schwaiger, Gergely Kokavecz, Stefan Schwarzmueller, and Shunwei Shen on tool support for the automated analysis of UMLsec models with regard to security requirements, which is currently under development at TU Munich (see [JS03, JS03a]).
6.2.1 Tool Functionality

There are several possible degrees of functionality in the verification of UMLsec models, depending on which of the stereotypes in Sect. 4.1.2 should be verified:

Static features: The tool should be able to verify security properties included as stereotypes in the structure and deployment diagrams, such as «secure links» and «secure dependency».

Simple dynamic features: The model behavior, described by the statechart and sequence diagrams, is analyzed to verify basic security requirement, defined on the behavioral level (such as «fair exchange»).

Complex dynamic features: The UMLsec model describing dynamic behavior is translated into the input language of an analysis tool (such as a temporal logic formula in the case of a model-checker), and can thus be verified against even subtle dynamic properties, such as «data security».

The following aspects have to be considered when trying to construct tool-support for secure systems development with UML following the UMLsec approach.

To be able to apply verification tools (such as model-checkers), one needs a front-end which automatically produces a semantic model and includes the relevant formalized security requirements, when given a UMLsec model. This avoids requiring the software developers themselves to perform this formalization, which usually needs a high level of specialized training in formal methods. UMLsec supports this approach by offering predefined security primitives (such as security requirements or mechanisms) with a strictly defined semantics, which can be applied by a developer who may not be expert in security by including the relevant stereotypes in the UML model. These primitives are translated into the targeted formal language, protecting from potential errors in manual formalization of the security properties (see Chap. 7 for a definition of the formal language used for UMLsec and the formalization of the security primitives and Chap. 8 for the formal semantics of the (restricted and simplified) fragment of UML used). Since security requirements are usually defined relative to an adversary, to analyze whether the UML specification fulfills a security requirement, the tool support has to automatically include the adversary model arising from the physical view contained in the UML specification. As usual, to keep the mechanical analysis feasible, one has to use a high-level model of cryptographic algorithms which abstracts away the details on the level of bit sequences (see Chap. 7).

The architecture and basic functionality of the UMLsec analysis suite are illustrated in Fig. 6.4. The overall architecture is divided between the UML drawing tool in use and the analysis suite. This way the analysis suite can be offered as a web application (where the users use their drawing tools to construct the UML model which is then uploaded to the analysis suite), which facilitates maintenance. Additionally, a locally installable version is under
development at the time of writing. Plugins for various UML drawing tools are also planned.

The usage of the analysis suite as illustrated in Fig. 6.4 proceeds as follows. The developer creates a model and stores it in the UML 1.4/XMI 1.2 file format. The file is imported by the UMLsec tool into the internal MDR repository. The tool accesses the model through the JMI interfaces generated by the MDR library. The static checker parses the model, verifies its static features, and delivers the results to the error analyzer. The dynamic checker translates the relevant fragments of the UML model into the model-checker input language. The model-checker is spawned by the UML suite as an external process; its results (a counter-example in case a problem was found) are delivered back to the error analyzer. The error analyzer uses the information received from both the static checker and dynamic checker to produce a text
report for the developer describing the problems found, and a modified UML model, where the found errors are visualized and (as far as possible) corrected.

6.2.2 Implementation Details

We now explain a framework for implementing verification routines for the constraints associated with the UMLsec stereotypes. The goal is that advanced users of the UMLsec approach should be able to use this framework to implement verification routines for the constraints of self-defined stereotypes. In particular, the framework includes the UMLsec tool web interface, so that new routines are also accessible over this interface.

The idea behind the framework is thus to provide a common programming framework for the developers of different verification modules (tools). Thus a tool developer should be able to concentrate on the verification logic and not be required to become involved with the input/output interface. Different tools, implementing verification logic modules (static checkers or dynamic checkers in Fig. 6.4), can be independently developed and integrated. At the time of writing, there exist verification modules for most UMLsec stereotypes.

An added tool implementation needs to obey the following assumptions:

- It is given a default UML model to operate on. It may load further models if necessary.
- The tool exposes a set of commands which it can execute.
- Every single command is not interactive. They receive parameters, execute, and deliver feedback.
- The tool can have an internal state which is preserved between commands.
- Each time the tool is called with a UML model, it may give back a text report and also a UML model.

These assumptions were made in order for the framework to cover as much common functionality as possible while not becoming overly complicated. Experience indicates that the assumptions are not too restrictive, given the architecture at hand (see Fig. 6.4).

The tool architecture in Fig. 6.5 then allows the development of the verification logic independently of the input and output media with minimum effort. Each tool is required to implement the ITextMode interface which exposes tool functionality in text mode, with a string array as input and text as output. The framework provides default wrappers for the graphical user interface (GUI) GuiWrapper and the web mode WebWrapper. These wrappers enable use of the tool without modifications in the GUI application (which is part of the framework) or through a web interface by rendering the output text on the respective media. However, each tool may itself implement the IGuiMode and/or IWebMode to fully exploit the functionality of the corresponding media, for example to fully use GUI mode capabilities to display graphical information.
At the time of writing, the UMLsec tools are working with the UML 1.4 version, which can be stored in an XMI 1.2 format [Obj02b] by a number of existing UML design tools (including Poseidon 1.6 Community Edition [Gen03]; see [Sha02] for an overview).

6.3 UMLsec Versus Runtime Data: SAP R/3 Permissions

This section presents research on linking the UMLsec approach with the automated analysis of security-critical data arising at runtime.

Specifically, it presents joint research with Sebastian Höhn on the construction of a tool which automatically checks the SAP R/3 configuration for security policy rules (such as separation of duty) [HJ03a]. The permissions are given as input in an XML format through an interface from the SAP R/3 system, the rules are formulated as UML specifications in a standard UML CASE tool and output as XMI (as part of the UMLsec framework explained in the previous section), and the tool checks the permissions against the rules using an analyzer written in Prolog. Because of its modular architecture and its standardized interfaces, the tool can be adapted to check security constraints in other kinds of application software (such as firewall, or other access control configurations).

Configuring user security permissions in standard business applications (such as SAP R/3 systems) is difficult and error-prone. There are many examples of wrongly configured systems that are open to misuse by unauthorized parties. To manually heck permission files of realistic size in a medium to

---

1 An upgrade to UML 2.0 is in development at the time of writing.
large organization – which can consist of up to 60,000 entries – can be a daunting task.

The management and configuration of security-related resources in standard business applications are important tasks. Potential threats include public disclosure of confidential information but also direct financial loss. In up to half the number of overall cases, the incidents are caused from within the company [Ric03]. This demonstrates the importance of properly configuring security permissions in business applications. It is important to realize that the existence of security mechanisms itself does not provide any level of security if they are not properly configured.

That this is actually the case is often clear to see. This applies especially to the financial sector, where user permissions have to satisfy more complex correctness conditions. One example is the rule of “separation of duty”, meaning that a certain transaction should only be performed jointly among two distinct employees (for example, granting a large loan). Difficulties arise firstly from the inherent dynamics of permission assignment in real-life applications, for example due to temporary delegation of permissions (for example, to vacation substitutes). Secondly, they arise from the sheer size of data that has to be analyzed (in the situation of a large German bank, which the current work had motivated, about 60,000 data entries). A manual analysis of the security-critical configurations through system administrators on a daily basis is thus practically impossible, and might result in security weaknesses in practice.

This observation motivated the current research which has been initialized in a discussion with the above bank and its security consulting partner. This resulted in a tool which automatically checks SAP R/3 configurations for security policy rules (such as separation of duty). This allows the user of the tool to construct a link between general information of the system (such as business processes specified in UML diagrams) to security-critical run time information. This is very useful from a security viewpoint, since security is an overall requirement which needs to take into consideration all aspects of a system in an integrated way to avoid potential weaknesses at the interfaces between different system parts.

The tool is part of the UMLsec framework explained in Sect. 6.2. The permissions are given as input in an XML format through an interface from the SAP R/3 system, the rules are formulated as UML specifications in a standard UML CASE tool and output as XML, and the tool checks the permissions against the rules using an analyzer written in Prolog. Because of its modular architecture and its standardized interfaces, the tool can be adapted to check security constraints in other kinds of application software (such as firewalls or other access control configurations). In this section, we explain the design of this tool.
6.3.1 Automated Analysis of Security Rules

The Goals

The tool is supposed to take a detailed description of the relevant data structure of the business application, the business data, and some rules written by the administrator. Using this information, the tool checks whether the rules hold for the given configuration. Violation of rules is included in the generated security report. The tool should accomplish the following specific tasks:

- It should read the configuration from the business application.
- It should automatically generate a report of possible weaknesses.
- It should provide a flexible configuration of the report’s data.
- It should be easily configurable for different business applications.
- It should be able to check large-scale databases.
- The checking should be based on freely configurable rules.

Two other goals are particularly important to enable use of the tool beyond the specific task of checking SAP permissions for the SAP installation at hand: it has to be easy to integrate the tool with different business applications, and the rules that have to be checked need to be very flexible. To make the tool as flexible as possible and also as easy to use as one could, a modular design is of great importance.

Architecture

The tool mainly consists of three parts (Fig. 6.6). They store the information describing the relevant data structure of the business application, define the rules, and evaluate the rules. An additional part is needed to import the data from the business application (such as the SAP system). This can be the user data and some structural information about transactions.

The complete separation of the tool and the business application provides additional security and privacy. Firstly, by separating the tool from the business application, there is no way the tool could add any weaknesses to this security-critical part of the company’s IT system. Secondly, in this way it can be made sure that only the information needed for the analysis is exported to the tool, which prevents any unnecessary exposure of confidential data. The information itself is completely stored in XML. The business application’s data has to be exported to XML files. The data structure of the business application is defined by UML class diagrams. Any case tool capable of saving XMI data according to the tool’s schema files can thus be used to do the modeling. Rules are stored in XML as well. With all this information, the tool can check the rules and create the report.

As an option, the report can use templates to generate the layout that the user wants. To adapt the level of information to the given needs, every rule has a “level of verbosity”. Then the rule is only evaluated if the report’s desired “level of verbosity” is higher than the rule’s level.
The Business Application as a Model

Throughout the description of the analyzer there will be several types of information that fit into different layers on OMG's meta-model framework (see Sect. 6.1). In this framework there are UML models on layer 1 (M1) and application data on layer 0 (M0) (see [Obj02a], pp. 2-2 to 2-3).

According to this separation of “model” and “information” the analyzer needs two distinct types of data. First it needs “metadata” which is the description of the data structure of the business application itself and is given as a UML model of the application. This is what is sometimes called the “structure of the business” application and it is on level M1. On the other hand, the analyzer needs to know about the data itself; this is what is called “instance data” and it is information on level M0.

To illustrate the separation of data on layer M1 and data on layer M0 we consider an example. Assume there is “some” user data in the business application. Every user has a name and a password. To formally describe the meaning of “some” in the expression “some user data” there is a “model” that tells the tool about the class user and its attribute name and password. This is done with a UML model and is data on level M1. When the tool checks the rules and needs to evaluate information of some special user, for example
“John”, it needs what is called “information” in the “meta-model framework”. This information is called “instance data” and it is given as XML documents (this is, as the analyzer uses it, placed on layer M0) [Obj02a].

**Permissions**

To associate permissions for transactions via roles to users in role-based access control (RBAC), the tool uses UML class diagrams. These diagrams can be directly used to give this information, and we do not need to introduce any additional features. The tool reads the class diagram and evaluates classes and associations.

In general, the analyzer is not restricted to such an RBAC model or to any specific model at all. It is capable of evaluating rules on any class diagram that has the connection attributes assigned as names of the associations and the direction of associations defined by the navigable flag. The analyzer evaluates the model as a graph with classes as nodes and associations as edges, where edges are directed. As we will see later, for the evaluation of rules, we need to require that there must be a path between the two classes involved in that rule, and there must be instance data so that the connecting attributes of each class match.

To explain this in more detail, we consider the example in Fig. 6.7: the class diagram assigning permissions to users consists of the classes user, role, transaction, and permission, with attributes as in Fig. 6.7. There is an association role_id between user and role, an association role_id between role and transaction, and an association transaction_id between transaction and permission. The analyzer uses this model to automatically find a user’s permissions.

Note that when assigning a permission \( p \) to a user \( u \) via a role \( r \), and the user \( u \) also happens to have another role \( r' \), then (of course) it is not admissible to conclude that any user \( u' \) with the role \( r' \) should also be granted the permission \( p \). In that sense, assigning permissions to users via roles is “unidirectional.” In the class diagrams defining permissions, this is specified by using the “navigable” flag of UML class diagrams. This flag is an attribute of an association’s endpoint. If this flag is set to “true” at the endpoint of a class \( c \) (signified by an arrow at that side of the association), our rule-analyzer may associate information from the other end of the association with \( c \). If it is set to “false”, this information may not be evaluated. In this way our tool may gather the permissions with respect to transactions granted to a given user by traversing the class diagram along the associations in the navigable directions permitting a “flow of information”. Thus the tool “collects” all users that have a given role, but does not recursively collect all users that have any of the roles that a given user has (as explained above).

To know how the elements in the application are connected there must be some kind of ID that can be evaluated at both sides of the connection. As in the short example above, the user would have some kind of “role-id” in his or her user data, and a role would have the same id. The application retrieves
the user’s “role-id” and finds the role with the same id. To express that mechanism in our static UML class diagram, there is an association between the classes that exchange information. The user class would be associated with the role class, so the tool knows there is some kind of interaction (that is, the application is able to find a user’s role). To enable the search of information the analyzer implicitly adds an additional attribute to either class at the association’s endpoints. These new attributes are assigned the association’s name.

As with every programming environment, class names have to be unique throughout the data structure of the business application. That should not cause a restriction in any environment.

6.3.2 Instance Data

Besides the structural data elements explained above, we need so-called “instance data”. Here an instance may, for example, be a real user of the system. This information is very important for most of the rules one would like to evaluate. There are, of course, rules that do not need instance data (if one is checking the UML data structure model itself for some constraints, for example), but in general there will be instance data. It is read by the analyzer from additional XML files (for an example see Fig. 6.8), containing a tag for every class, and within that tag another tag for each attribute. The analyzer is able to generate the XML Schema file for a UML model specified by the user, because the contents of the instance file depend on the model of the
business application. With the generated XML Schema file the analyzer is able to validate the input file.

Rules

As defined in the previous section, the business application data structure is represented by a class diagram, that is a directed graph together with the data from the business application. These two pieces make up a rather complex graph whose structure can be seen in Fig. 6.9 as an example. One can see that for every user in the business application data structure, a node is added. The model gives the tool the information that there is a connection between “user” and “role”, but in the graph in Fig. 6.9 there are only edges between certain users and certain roles. It shows that there is an edge between user “john” and role “users”, because there is the attribute “role” that instantiates it. There is no edge between user “john” and role “admins”, because “john” does not have “admins” in his roles. This is the graph that the analyzer uses to analyze the rules.

Rules in this instance consist of the following elements:

- a name (used as a reference in the security report)
the type of the rule, which can be either of PROHIBITION or PRECONDITION (meaning that the condition given in the sub-rule defined below should either not be fulfilled, or be fulfilled)
• a message (printed in the report if the rule fails)
• a priority level (to build a hierarchy of importance, so that less important rules can be turned off easily – typical values may include DEBUG, INFO, WARNING, ERROR, FATAL, or a numeric value)
• a sub-rule, which defines a path in the analyzer’s graph and a set of constraints, as defined below

A sub-rule has the following elements:

• the head, which is the starting point of the path in the analyzer’s graph defined by the sub-rule
• the target, which is the target of that path
• a list of constraints, which defines conditions that the path has to satisfy

Here a constraint consists of the following elements:

• element, the node that has to be checked
• condition, to be checked on that node

We consider the following example. If it has to be ensured that a certain user, say “john”, does not have the role “admins” assigned, the following parameters would be set for the rule:

name: check user roles
type: PROHIBITION
message: check user for given roles
priority: ERROR=4

In this example, we have a single sub-rule:

head: user
target: role
constraint: head.user.name == param.user.name
constraint: target.role.name == param.role.name

This rule has two parameters that the user has to provide when generating the report, indicated by the keyword param: the user-name “john” and the role “admins”. A suitable XML document that provides these parameters for every rule is expected as input.

The evaluation of this example rule is as follows. The analyzer attempts to find the head of the rule (that is, “user: john”) in the analyzer’s graph. Afterwards, it tries to find a path to the target (that is, “role: admins”). If that succeeds it prints the given message in the security report, if the user wants messages with priority ERROR printed in his report. The separation between the rule itself and the two parameters (“param.user.name” and
“param.role.name”) is introduced to make editing more comfortable. One does not need to edit a rule for every user and every role that have to be checked.

With the help of these elements rather powerful rules can be defined. To the analyzer the model is a graph representing the business application data structure. The head and the target represent nodes within that graph. For example, head could be “user” and target could be “role”. With that definition there should exist a path between head and target. If it does not, the rule fails. If that path exists, the analyzer will try to fill that path with valid data from the given instance data. This means that for a valid connection from head to target, every association along that path is instantiated with a discrete entry from the business application’s data. If there is no valid instantiation, the rule fails. If there is one, the constraints are checked. Every instantiated element will be examined, and if one of the conditions fails, the rule fails. Otherwise, it succeeds.

To make the rules more expressive, a rule can consist of several sub-rules, where a sub-rule does not have the additional name, type, message, and level attributes. In this way the analyzer is powerful enough to check rules such as separation of duty, for example by using the sub-rules:

- check for distinct role A,
- check for distinct role B, and
- ensure that no user has both of them.

For a rule to succeed, each of the sub-rules has to succeed.

The additional information is needed to configure the analyzer properly, and to customize the report. The name of the rule is used to output which rules failed. The type is given to distinguish between preconditions and prohibitions, meaning that either the success or failure of that rule is reported. So it is conveniently possible to define states that must be fulfilled for every configuration and to define states that may not appear within a configuration. For example, it may be vital for a system to have the password set for the super-user account. Conversely, for separation of duty, it would be forbidden for the same user to have two exclusive roles. A message is printed if a precondition fails or if a prohibition succeeds. The message attribute simply gives the text that is written to the report if a message is printed. A template system prints out the messages with any of the instance’s attributes in a freely configurable manner. So it is possible to insert values from the violating instance into the message, for example as “there is no password for user Joe”. Only with such a feature do the messages become readable and thus the tool easily usable by a human user.

The level attribute gives a “level of verbosity” to a rule. So the user can have the tool evaluate some rules only. Level 1 means that, the rule is relatively unimportant. An increasing number will show increasing importance for the rule. The analyzer evaluates only the rules with a level higher than that given as “level of verbosity” to that report.
6.3.3 Evaluating Rules

We use Prolog for the evaluation of the rules, which allows a rather elegant treatment as it is designed for evaluating logical statements. In our experience, it is also sufficiently efficient for a real-life application (although it may not be time-critical considering the possibility of overnight batch processing).

If one translates structural elements to “Atomic Prolog Terms” and the analyzer rules to “Non-Atomic Terms”, one can ask the Prolog interpreter for the instances of the Prolog rules. The advantage of using Prolog is that we can concentrate on the essential problems specific to the analyzer without having to solve the hard problems of finding the instances along the paths. How the analyzer rules defined above can be translated into Prolog will be explained in the following.

Translating Rules to Prolog Clauses

First of all, the data structure of the business application is defined in Prolog. For this each class from the model describing the business application is converted to a predicate with an argument for each attribute. A class user (U) with two attributes (name (n), role-id (r)) gives the following expression:

\[ U(n, r). \]

To evaluate an expression like “the user’s role”, we need an additional predicate for each association. The “connecting” predicates have the following
form. Assume there is a predicate \( U(n, r) \) and a predicate \( R(m, r) \). Then the connection \( C(n, m, r) \) is given by the following term:

\[
C(n, m, r) \rightarrow U(n, r) \land R(m, r).
\]

This means that there is a user \( n \) in role \( m \) if there is a user \( n \) and a role \( m \) such that the role-id \( r \) is the same. These predicates can be extended to paths with any number of intermediate nodes, because Prolog evaluates all predicates to true, and the provided connecting attributes match, as in the following example:

\[
A(u, v, w, x, y, z) \rightarrow B(u, v) \land C(v, w) \land D(w, x, y) \land E(y, z).
\]

Note that the tool could be modified to eliminate the arguments not needed to determine the existence of a path. However, it is convenient to be able to include this additional information in the report.

After the structure is added, the instance will be added, too. For every class several predicates are created. In Prolog syntax this is what it looks like:

\[
\text{user} (\text{john}, 500).
\]

With the above rules in place, the analyzer can ask for instantiations of the discrete rules (for example, if user john has exclusive roles “start transaction” and “commit transaction”, separation of duty is violated for this transaction).

A short remark regarding the efficiency of the analysis is in order. The “connecting” predicates are added only when a rule needs them. If one were to insert every possible connection from every imaginable head to every target, there were up to \( n(n-1) \) of these connections. But to evaluate \( m \) sub-rules one would need at most \( m \) of these connections. Thus a connection is only added when a sub-rule implies it, significantly reducing the number of connections in general.

**Evaluating Separation of Duty in SAP Systems**

We use an example configuration from [Sch03b] to explain how separation of duty in SAP systems can be evaluated by the analyzer. First of all, the structure of the business application needs to be defined. For simplicity it will be assumed that the structure looks like the one presented in Figure 6.7. It certainly is just a very small part of the SAP security concept but as an example, it will be sufficient. There are three employees: Karen, Susan and John. Karen and Susan are just employees in any department, and John is a purchasing agent at the company. To have separation of duty, Karen may create a purchase and Susan may release that purchase to John. John may order the desired goods from some supplier firm. With this in place the Prolog rules would be very straightforward:
<table>
<thead>
<tr>
<th>User</th>
<th>Role</th>
<th>Transaction</th>
<th>Permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>Karen</td>
<td>employee (in charge of service)</td>
<td>Create purchase</td>
<td>Is allowed to create some purchase in SAP.</td>
</tr>
<tr>
<td>Susan</td>
<td>employee (in charge of service, senior in rank to Karen)</td>
<td>Commit purchase</td>
<td>Is allowed to release purchase created by Karen.</td>
</tr>
<tr>
<td>John</td>
<td>employee purchasing agent</td>
<td>Place orders</td>
<td>Is allowed to place orders with some delivery agent.</td>
</tr>
</tbody>
</table>

Fig. 6.11. Small separation of duty example

user(Karen, 1)
role(create-purchase, 1)
...

To have separation of duty in place there are two exclusive roles, which may not be assigned to the same user: “create-purchase” and “release-purchase”. John just places the orders, he does not do any supervision here. The first sub-rule must have the head “user” and the target “role”. The second sub-rule must have the same head and target but it needs a condition:

```
rule1.user.name = rule2.user.name
```

The type of this rule is PROHIBITION, the other attributes do not matter for this example. What does the tool do now? It has created the predicates and inserted the users and the role from the instance files. Afterwards it searches the paths for the rule. The path from user to rule is quite obvious, so the “connecting” predicate is created: at

```
user_role(name, role_id, rname)
   :- user(name, role_id),
      role(rname, role_id).
```

With that predicate the rule can be evaluated to:

```
user_role_rule(name, role_id1, role_id2)
   :- user_role(name1, role_id1, X),
      user_role(name2, role_id2, Y),
      name1 = name2.
```

Now Prolog can be asked for:

```
user_role_rule(X, ‘create-purchase’, ‘release-purchase’).
```
and it calculates the correct answer. In the example in Fig. 6.11 there is no solution to the predicate, because there is only Karen for role “create-purchase” and Susan for role “release-purchase”, and user Karen is not equal to user Susan.

Although this example is very simple, it serves as a demonstration of how the analyzer can be used. In a real application, the path from user to role might contain several nodes or one might not know the roles that have to be exclusive, just the permissions, so one could exclude permissions contained in roles with several hundreds of entries each. In cases were a role contains several hundreds of permissions, it is not obvious whether separation of duty is in place.

Note that we do not currently aim to treat object-based permissions, but remain at the class level. While it should be possible to extend our approach in that direction, it is beyond the scope of the current investigation. In particular, this applies to a special kind of separation of duty specific to SAP systems. The system can be configured to require more than one user with a certain role to start a transaction. Since the checks needed to enforce this requirement are performed within the SAP system, it would not make sense to repeat them at the analysis level as well. But one should note that this “internal” separation of duty differs from what is presented in our examples. It is often useful to have separation of duty throughout different departments, so that the internal one is not sufficient (that is, if there is some kind of revision after the transaction was performed).

**SAP Transactions**

Another example of the use of the analyzer to improve security is when the transactions are also part of the data structure. Because of the design of the SAP system, there are no security checks performed when a transaction calls another one (maybe somewhat surprisingly so). By this transitivity, it is very difficult in large systems to see who can execute a transaction. The permission to execute a transaction includes the permission to execute every transaction called by the first one and there does not seem to be a possibility to disable this feature. Thus creating a transaction in SAP is a permission that gives access to everything. One should notice that an employee who is allowed to create a transaction and execute it can execute any transaction by calling it from his or her self-created one.

If access needs to be restricted to some transactions, it is therefore not sufficient to ensure that the permission is given only in the roles associated with that transaction, and that only the users allowed to execute that transaction are assigned those roles. It has to be ensured furthermore that there is no transaction calling the restricted one, because SAP would not perform security checks there and one would not prevent execution of the restricted transaction.
To do so, one may model the transactions with their sub-transactions as part of the analyzer’s model. Then the tool creates rules to check whether permissions grant any user additional rights that are not part of his or her role. It is usually not advisable to report every transaction that can be executed without explicit permission. Because of the error-prone design, there will be a lot of transactions that are meant to be called implicitly. But the possibility to check for some very “dangerous” transactions (in particular the ones for changing permissions and roles) is a great enhancement of security.

**Use Case for Checking SAP Permissions**

Figure 6.12 presents a sample “use case” for checking the permissions on a running SAP system. The SAP database is used to generate the information necessary for the analyzer. An employee creates a UML model describing the SAP system. We use the CASE tool Poseidon for UML to do so. These two documents describe the business application. With these documents in place one can create the rules. For creating the rules there is a GUI but the XML files necessary can be edited manually, too.

When all the documents are prepared, the analyzer can check the rules automatically. After the analyzer has finished the checks, the user can read the security report and start reconfiguring the business application in order to fulfill all the conditions contained in his rule set.

The security report is formatted as defined by the templates that are part of the analyzer. The analyzer writes a freely configurable HTML file for review with a web browser.

**Further Applications**

The analyzer can be used not only to check SAP systems, but also to check most configurations of large-scale applications. The modular architecture makes it easy to adapt to a new application. One needs to define the application’s structure in UML, then the instance data must be converted to proper XML files, corresponding to the XML Schema provided by the tool’s schema generator. Afterwards, the rules have to be defined. Then the GUI can be used, or the XML files can be written manually or generated by any tool fitting the needs of the application. Finally the report can be generated by the analyzer.

Although the main focus here is on applications in security, given an open architecture, one should be able to use the tool for a wide range of rule-checking tasks of configuration files not necessarily related to security.

**6.4 Linking Models to Code**

As noted for example in [FS97], of ultimate benefit in software development are not “pretty pictures”, but the running implementation of a system. In this
section, we present some approaches for linking UML models to implementations. The aim is to ensure that the benefits gained from the model-based approach on the level of the system model (such as increased confidence in satisfaction of critical requirements) actually carries over to the implemented system, as one would hope.

### 6.4.1 Test-Sequence Generation

In this section we briefly refer to joint work with Guido Wimmel on model-based testing of security-critical systems using UML-like notations. Details have to be omitted for space reasons but can be found in [JW01a, JW02, WJ02].

In specification-based testing (see for example [DBG01] as a more recent example of the many approaches documented in the literature), test sequences are generated from an abstract system specification to provide confidence in
the correctness of an implementation. The traditional approach in this direction, namely that of conformance testing, establishes that an implementation conforms to its specification. However, a complete test coverage is often infeasible, resulting in a need for test-case selection. For security-critical systems, finding tests likely to detect possible vulnerabilities is particularly difficult, as they usually involve subtle and complex execution scenarios and the consideration of domain-specific concepts such as cryptography and random numbers.

[JW01a, JW02, WJ02] present research aiming to generate test sequences for transaction systems from a formal security model supported by the CASE tool AutoFocus which has a UML-like notation. To test an implementation for vulnerabilities, we compute test sequences from the security model covering possible violations of the security requirements. The test sequences are determined with respect to the system’s required security properties, using mutations of the system specification and attack scenarios. To be able to apply them to an existing implementation, the abstract test sequences are concretized.

The motivation for specification-based testing is that, in general, the implementation of a system is very complex. To allow proofs of security properties, abstraction techniques are used: in models of cryptographic transactions, messages, keys, and random numbers are usually represented by abstract data entities which can be arguments to abstract operations such as encryption or hashing, and part of the actual messages exchanged may have been left out. Besides, as the security model is usually developed independently of the implementation (mostly after the implementation, though this is not desirable), it cannot be concluded from the correctness of a security model that the implementation is secure.

Confidence in the correctness of an implementation can be gained by extensive testing. Testing for security holes is usually restricted to penetration testing (a so-called “tiger team” of experts manually tries to break the system or tools such as SATAN are used to search for known vulnerabilities). This approach is not satisfactory as it depends largely on the skill of the employed tiger team or the knowledge encoded into the tool, which does not consider application-specific security requirements.

[JW01a, JW02, WJ02] show how to complement this approach by generating test sequences from a security specification. The aim is to find those test sequences that are most likely to detect possible vulnerabilities. For this purpose, one adapts methods from classical specification-based testing to the application domain of security-critical systems. Specifically, domain-specific concepts such as cryptography, knowledge of or access to secrets, and threat scenarios are included. Test sequences likely to detect vulnerabilities are computed using mutations of the specification that lead to violation of the security requirements. Further, it is shown how to translate the abstract test sequences derived from the security model to concrete test sequences that can be applied to an existing implementation.
Vulnerability Coverage Using Mutations

As it is not feasible to exhaustively test every behavior of a security-critical system, first appropriate test-case specifications have to be selected. For security testing, the aim is to cover a large number of possible vulnerabilities.

One can use structural coverage criteria such as state or transition coverage on the models [OXL99] and restrict them to those that are marked “critical”, but this has the drawback that it does not take into account the security requirements.

The difficulty with defining coverage criteria related to the security requirements is that they are mostly universal properties. Therefore, a security requirement $\Phi_i$ can only be used to verify the model, not the implementation. If a trace fulfilling $\neg\Phi_i$ is found, the model violates the security requirement and must be corrected. Otherwise, $\Phi_i$ by itself cannot be used to select relevant traces, as all traces satisfy $\Phi_i$.

In this case, mutation testing and fault injection techniques [Off95, VM98] prove to be promising approaches. In mutation testing, errors are introduced into a program (leading to a set of mutants), and the quality of a test suite is measured by its ability to distinguish the mutants from the original program (to “kill” the mutants). Fault injection works in a similar way, but is often also used for reliability evaluation (determining if a program tolerates a perturbation of the code or data states).

We introduce errors into the specification of the security-related behavior, generate the threat scenarios, and determine if and how the introduced errors can lead to security violations. The introduced errors can correspond to errors in the implementation or to attacks leading to such errors, for example subjecting a smart card to environmental stress.

A mutation function $\varepsilon$ can be based on general possible mutations for expressions and operands (for example, operator or operand replacement) proposed for Ada by Offutt et al [OVP96]. For security testing, $\varepsilon$ should be based on common programming errors likely to lead to vulnerabilities, such as missing plausibility checks or wrong use of identities [AKS96]. In addition, in our model cryptography must be taken into account, leading to mutations corresponding to confusion of keys or secrets, or to missing or wrongly implemented verification of authentication codes.

Details can be found in [JW01a, JW02, WJ02].

Concretization of Abstract Tests

The abstract test sequences computed from the formal security specification still have to be translated to concrete test data (that is, byte sequences) that can be used to test the actual implementation.

In many cases, concretization can be achieved using straightforward mappings between abstract and concrete test data [DBG01], and executing the test using a test driver that passes the inputs to the component to be tested and
verifies if the outputs are as expected. However, testing security-critical systems involves additional complications, mainly because of non-determinism, for example arising from randomly generated keys and nonces, and the use of cryptographic primitives:

- In formal specifications, cryptographic primitives are usually modeled symbolically, rather than as sequences of bytes, to make verification feasible (see [AJ01] for a justification of this general approach). The test driver has to map these symbols to sequences of bytes in a consistent way. Conversely, sequences of bytes created and output by the tested component (for example, random values such as nonces or session keys) must be stored by the test driver and used in place of the relevant symbols in the test data of the remainder of the execution.
- Sometimes, values (such as transaction numbers or time-stamps) are abstracted away in formal specifications to simplify verification (and because they are seen to be independent of a security property at hand). These have to be included in the concrete test data in a consistent way.
- If encryption is used, the test driver must know the corresponding keys and encryption algorithms to be able to compute the encrypted input data and verify encrypted output data.
- Hash values or message authentication codes contained in the output data can only be verified when the complete data that was hashed is available to the test driver.

Thus, [JW01a, JW02, WJ02] define a concretization of abstract messages.

6.4.2 Code Generation and Code Analysis

There are other ways to link models to code besides test-sequence generation, as we briefly point out.

*Code generation* can be used to directly generate code from a UML model. Where this is possible, there is usually no need for conformance testing. Testing for security properties may, however, still be useful to detect weaknesses not apparent on the design level.

So far, code generation is mainly used in a fragmentary way (for example, to generate class definitions from class diagrams). More extensive support for code generation (for example, code generation from statecharts) is being developed. However, it will remain to be seen to what extent the use of UML as a visual programming language (as opposed to an executable specification language) will be established, because when writing programs for complete systems visually questions of scalability may become more apparent than when writing abstract specifications of parts of a system. For this reason, test-sequence generation may still be useful in the foreseeable future.
**Code analysis:** The same concepts and definitions used in the previous chapters to analyze UML models for security requirements can also be used to analyze (suitable abstractions of) programs. In particular, there exist tools which extract a state machine model from source code and allow our algorithms to be used after a suitable adaptation (see for example [klo03]).

### 6.5 Notes

The literature on how to use XMI to provide tool support for UML includes [Ste01b, Ste03a] (including an example using the Edinburgh Concurrency Workbench for analyzing UML models). There are several existing tools for automatic verification of UML models described in the literature. The HUGO Project [SKM01] checks the behavior described by a UML Collaboration diagram against a transitional system comprising several communicating objects; the functionality of each object is specified by a UML Statechart diagram. The vUML Tool [LP99] analyzes the behavior of a set of interacting objects, defined in a similar way. The tool can verify various properties of the system, including deadlock freeness and liveliness, and find problems like entering a forbidden state or sending a message to a terminated object. Both tools do not have any special features for describing the security features of the system being modeled. [CCR01] presents a tool which extracts execution sequences from UML statecharts (however, without connection to a UML drawing tool). [BD00a, DC02] present results directed toward tool support for UML using the model-checker FDR associated with the process algebra CSP.

One approach to analyzing security configurations is called the “Configuration Review Test” [Po92]. There seems to be no implementation of these tests that uses rules for this purpose. Existing tools for this approach check some conditions of specific applications, mostly operating systems. These tools are designed to check for certain security weaknesses, common to a number of systems. Compared to these specific tools, the open architecture presented in Sect. 6.3 is new for configuration review tests.

Penetration tests are commonly used to assess the security of a system [Wei95]. In our view, they are complementary to our approach explained in Sect. 6.3. On the one hand, penetration tests would profit from the information gathered by the analyzer’s report. On the other hand, the analyzer presented here does not warn about weaknesses in the software itself (such as programming errors or buffer overflows), but it reports configuration errors. To make a penetration test reveal the errors that the analyzer is designed to check for, one would have to try out every possible transaction. This is usually impractical because of their high number. Also, when performed on a live system, penetration tests can be rather invasive.

There are several recent approaches using UML for security analysis, including [LBD02]. More generally, there has been a lot of work on formulating security requirements in object-oriented data models (see for example [JKS95]).
and the references therein). Other approaches using logic programming for
access control analysis include [BdVS02]. [RS01] uses SQL to administer per-
misions for distributed data. Compared to that approach, our tool can be
applied not only to databases, but more generally to security configurations.
[GHR03] uses a model-checker to analyze Linux configurations.

There has been extensive research into specification-based testing, including
[DF93, PS97, HNS97]; a complete overview has to be omitted. Dushina et al explain
concretization in their Genevieve framework [DBG01]. In intrusion
detection (see for example [US01] for a model-based approach), a running
system is monitored for attacks. The AVA approach [VM98] focuses on identify-
ing critical statements rather than finding test sequences (for which random
distributions are used).

6.6 Discussion

After explaining the necessary background, we presented tool support for the
automated analysis of UMLsec models with regard to security requirements,
which is currently under development at TU Munich. The UMLsec analysis
suite includes a framework for implementing verification routines for the con-
straints associated with the UMLsec stereotypes. The goal is that advanced
users of the UMLsec approach should be able to use this framework to imple-
ment verification routines for the constraints of self-defined stereotypes (and
link it with the UMLsec web interface).

Although there is an increasing amount of research on advanced tool
support for UML, it seems that little work has done to provide advanced
tool-support (such as model-checkers) for verifying specific properties in
application-specific UML extensions. Although, in principle, one may use the
more general tools for this purpose, one would probably have to formalize
the specific requirements oneself, for which most software developers may not
have the appropriate training.

Hence the goal in our approach here is to provide tool support which can
specifically check the constraints associated with UMLsec stereotypes,
without requiring the user to formalize these requirements first. Additionally,
advanced users can still use the general functionality of the included analysis
engines. Further, they can actually write their own verification routine for a
newly defined stereotype and include it in the UMLsec tool by making use of
the framework available for this.

We also presented research on linking the UMLsec approach with the au-
tomated analysis of security-critical data arising at runtime. The example
presented here is a tool which automatically checks the SAP R/3 configu-
ration for security policy rules. Because of its modular architecture and its
standardized interfaces, it should be possible to adapt it to check security
constraints in other kinds of application software (such as firewalls or other
access control configurations).
Although there already exist commercial tools for analyzing SAP data, the work presented here offers a greater range of properties to be checked, and also offers the new possibility to enhance the security analysis by linking it with other information, such as security-critical business process specifications formulated in UML diagrams, in an integrated setting within the UMLsec framework.

Finally, we briefly referred to some approaches for linking UML models to implementations to make sure that the implementation is actually secure, and not just the model. There are three approaches to achieve this, namely model-based test-sequence generation, code generation, and code analysis.
A Formal Foundation

In this chapter, we present some background used in Chap. 8 to define a formal model for a part of UML to enable advanced tool support.

We introduce the notion of UML Machines that is inspired by Abstract State Machines (ASMs, defined in [Gur95]). They give a mathematically rigorous framework for the approach to secure software engineering explained in the previous chapters. While having a sound mathematical foundation, their notation (a more formal kind of "pseudo-code") is rather flexible and allows capture of complex concepts relatively straightforwardly. In particular, they let us model interaction with the environment of a system. We also define UML Machine Systems (UMSs) that allow one to build up UML Machine specifications in a modular way and to treat external influences on the system beyond the planned interaction (such as attacks on insecure communication links). We define notions of refinement and rely-guarantee specifications for UML Machines and prove that rely-guarantee specifications are preserved under refinement. Finally, we explain how we use UML Machine Systems to specify security-critical systems (that may employ cryptographic operations). We also give definitions for secrecy, integrity, and secure information flow. We give equivalent internal characterizations of secrecy and integrity which allow easier verification. We show secrecy, integrity, and secure information flow to be preserved under refinement, avoiding the so-called refinement problem.

The proofs for statements in this chapter are given in Appendix C.

7.1 UML Machines

Our choice of the formalism of UML Machines (inspired by Abstract State Machines [Gur95, BS03a]) is motivated by our goal to use them to formulate our ideas on security modeling (in particular, security modeling with UML) in a mathematically precise way, which is facilitated by the expressiveness and flexibility of UML Machines. We will also explain how this formalism can be used as a foundation for advanced tool support. UML Machines are transition
systems the states of which are algebraic structures, and which have built-in communication mechanisms inspired by the corresponding mechanisms in UML.

We will use UML Machines to specify components of a system that interact by exchanging messages from a given set Events which are dispatched from (resp. received in) multi-set buffers called output queues (resp. input queues)\(^1\). The idea is that a UML Machine may interact with its environment by adding values to its output queue and by retrieving the values from its input queue.

We assume a set of variables. Terms are defined as usual by starting with variable names and applying function names recursively:

- A variable is a term.
- If \( f \) is a function name of arity \( r \geq 0 \) and \( t_1, \ldots, t_r \) are terms, then \( f(t_1, \ldots, t_r) \) is a term.

**Definition 7.1.** Given a set \( F \) of function names containing at least the nullary function names true, false, and undef, a state \( A \) consists of:

- a set \( X \) (its base set) and
- interpretations of the function names in \( F \) on \( X \): an \( r \)-ary function name \( f \) is interpreted as a function \( f : X^r \rightarrow X \).

\( F \) is called the vocabulary of \( A \) and is denoted as \( \text{Voc} A \).

As usual, a set can be interpreted as a function taking values in \( \{ \text{true}, \text{false} \} \). Also, one often notationally identifies an algebra with its base set, for example, by writing \( x \in A \) instead of \( x \in X \) in the above situation. Similarly, we only distinguish between a function name and its interpretation when necessary to avoid misunderstanding. We write \( \text{Bool} \) for the set \( \{ \text{true}, \text{false} \} \). A variable assignment over a state \( S \) is a function from a set of variables to the base set of \( S \). It is extended to evaluations of terms in the usual way using the interpretation of function names.

**Definition 7.2.** A UML Machine \( A \) consists of

- a set \( \text{Voc} A \) of function names that contain at least the set names \( \text{inQu}_A \) and \( \text{outQu}_A \),
- an initial state \( \text{Init} A \) of vocabulary \( \text{Voc} A \), and
- a transition rule \( \text{Rule} A \) which can be of the form as defined inductively in Fig. 7.1.

The set names \( \text{inQu}_A, \text{outQu}_A \) model the input buffer and the output buffer of the UML Machine \( A \) that may change them only by deleting elements from \( \text{inQu}_A \) and by adding elements to \( \text{outQu}_A \). We assume that at the initial state \( \text{Init} A \) of the UML Machine, they always have the value \( \emptyset \).

Below we give an informal semantics for the transition rules; the formal semantics can be found in Appendix B. A UML Machine \( A \) is executed by... \(^1\) Here we follow the UML terminology which is confusing in so far as input/output queues are not queues, but multi-sets.
\[ R := \text{skip} \]
\[
f(s) := t
\]
\[
\text{if } g \text{ then } R \text{ else } S
\]
\[
do - \text{in - parallel } R_1 \ldots R_n \text{ enddo}
\]
\[
\text{choose } v \text{ with } g(v) \text{ do } R(v)
\]
\[
\text{seq } R_1 \ldots R_n \text{ endseq}
\]
\[
\text{forall } v \text{ with } g(v) \text{ do } R(v)
\]
\[
\text{iterate}(R)
\]
\[
\text{loop } v \text{ through list } X \ R(v)
\]
\[
\text{loop } v \text{ through set } X \ R(v)
\]
\[
\text{while } g \text{ do } R
\]
\[
\text{case } v \text{ of}
\]
\[
v \in X_1 : \text{ do } R_1
\]
\[
\ldots
\]
\[
v \in X_n : \text{ do } R_n
\]
\[
\text{else } S
\]

Fig. 7.1. UML Machine rules

iteratively firing the transition rule Rule \( A \), starting from the initial state Init \( A \). Thereby, its current state is updated; that is, the interpretations of its functions are redefined in terms of the previous interpretations. This way, the UML Machine changes between different states. By Definitions 7.1 and 7.2, each state consists of a base set and interpretations of the function names in \( F \), which includes the names in \( Qu_A \) and \( Qu_A^\uparrow \).

Skip rule: \text{skip} is a rule. It causes no change.

Update: Given terms \( s_1, \ldots, s_r \) and \( t \), we have an update rule \( f(s_1, \ldots, s_r) := t \). Suppose that at the point of execution of a given instance of this rule, the terms \( s_1, \ldots, s_r, t \) evaluate to the values \( \bar{s}_1, \ldots, \bar{s}_r, \bar{t} \) in the base set of a UML Machine \( A \), respectively. Then the execution of this rule updates the interpretation of the \( r \)-ary function name \( f \) at the \( r \)-tuple \( (\bar{s}_1, \ldots, \bar{s}_r) \) of values of \( A \) to map to \( \bar{t} \). Thus an update rule updates the function at a single position; all other interpretations are left unchanged.

Conditional: If \( g \) is a closed logical formula (of first-order logic) and \( R, S \) are rules then
\[
\text{if } g \text{ then } R \text{ else } S
\]
is a rule. If $g$ holds, the rule $R$ is executed, otherwise $S$. If $S$ is equal to `skip` then else `skip` can be omitted, provided indentation is used to prevent confusion.

Blocks: If $R_1, \ldots, R_k$ are rules, then

\[
\text{do} \quad \text{in} \quad \text{parallel} \\
R_1 \\
\ldots \\
R_k \\
\text{enddo}
\]

is a rule. To fire this rule, $R_1, \ldots, R_k$ are executed simultaneously, if they are mutually consistent (that is, updates concerning the same function name define the same value: for any two update rules $f(\overline{s}) := t$ and $f(\overline{s}) := t'$, we have $t = t'$). Otherwise the execution of the UML Machine stops.

Note that this `parallel composition` is truly parallel, rather than a nondeterministic interleaving. For example,

\[
\text{do} \quad \text{in} \quad \text{parallel} \quad x := y \quad y := x \quad \text{enddo}
\]

swaps $x$ and $y$.

Choose: If $v$ is a variable, $g(v)$ is a logical formula with one free variable, $v$, and $R(v)$ is a rule, then

\[
\text{choose } v \text{ with } g(v) \text{ do} \\
R(v)
\]

is a rule that chooses an element $a$ of the base set of $A$ such that $g(a)$ holds and executes $R(a)$ (in case such an $a$ does not exist, the rule is interpreted as `skip`).

Sequential composition: If $R_1, \ldots, R_n$ are rules, then

\[
\text{seq} \\
R_1 \\
\ldots \\
R_n \\
\text{endseq}
\]

is a rule, meaning that $R_1, \ldots, R_n$ are executed sequentially. If no confusion can arise, the shorter notation $R_1; \ldots; R_n$ may be used.

Forall: If $v$ is a variable, $g(v)$ is a first-order formula with one free variable, $v$, and $R(v)$ is a rule, then

\[
\text{forall } v \text{ with } g(v) \text{ do} \\
R(v)
\]

is a rule, which is fired by executing $R(a)$ for all $a \in \text{BaseSet}(A)$ such that $g(a)$ holds (before they are executed) in parallel, if they are mutually consistent. Otherwise, the execution of the UML Machine stops.
Iteration: If \( R \) is a rule then

\[
\text{iterate}(R)
\]

is a rule that iteratively executes the rule \( R \) until executing \( R \) gives no change (or until execution of \( R \) causes the execution to stop).

Note that the \texttt{do-in-parallel} rule may be expressed in terms of the \texttt{for all} rule, as in the Abstract State Machine setting [Gur97].

For convenience, we define some more transition rules which may be defined in terms of the ones above:

Loop through list: If \( v \) is a variable, \( X \) is a finite sequence of values in \( A \), and \( R(v) \) is a rule, then

\[
\text{loop } v \text{ through list } X \quad R(v)
\]

is a rule that iteratively chooses all elements \( x \in X \) if \( X \neq \emptyset \) (according to the content and order of \( X \) before executing the rule) and executes \( R(x) \) (or acts as \texttt{skip} if \( X = \emptyset \)).

Loop through set: If \( v \) is a variable, \( X \) is a multi-set name, and \( R(v) \) is a rule, then

\[
\text{loop } v \text{ through set } X \quad R(v)
\]

is a rule that iteratively chooses all multi-set elements \( x \in X \) (as \( X \) was before executing the rule, with the correct multiplicities, and in a non-deterministic order) and executes \( R(x) \) (or does nothing if \( X = \emptyset \)). This rule is used instead of \texttt{loop } v \texttt{ through list } X \ R(v) in situations where the order of the chosen elements does not matter, to avoid overspecification at the level of abstract modeling.

While: If \( g \) is a closed first-order formula and \( R \) is a rule then

\[
\text{while } g \text{ do } R
\]

is a rule. The rule \( R \) is executed while \( g \) holds.

Case distinction: If \( v \) is a variable, \( X_1, \ldots, X_n \) are mutually disjoint subsets of the base set of \( A \), and \( R_1, \ldots, R_n \) are rules, then the following is a rule:

\[
\text{case } v \text{ of } \\
\quad v \in X_1 : \texttt{do } R_1 \\
\quad \ldots \\
\quad v \in X_n : \texttt{do } R_n \\
\quad \text{else } S
\]

The rule is executed by evaluating \( v \) and executing one of the rules \( R_1, \ldots, R_n \) depending on the value of \( v \). \( v \in \{ x_i \} : \texttt{do } R_i \) may be abbreviated to \( x_i : \texttt{do } R_i \).
We define the following syntactic shortcuts, where \( A \) is a UML Machine and \( X \) is a multiset. We write \( \equiv \) for syntactic equality between (parts of) transition rules.

\[
\text{toutQu}_A(X) \equiv \text{outQu}_A := \text{outQu}_A \uplus X
\]

\[
\text{tinqu}_A(X) \equiv \text{inQu}_A := \text{inQu}_A \uplus X
\]

A run \( r \in \text{Run} A \) of a UML Machine \( A \) is a finite or infinite sequence \( S_0, S_1, \ldots \) of states such that the following conditions are satisfied:

- \( S_0 \) is the initial state \( \text{init} A \).
- For each \( n \in \mathbb{N} \), if \( S_n \) is the last element of the sequence \( r \) then
  - any consistent application of the transition rule \( \text{Rule}_A \) at state \( S_n \) leaves the state \( S_n \) unchanged, or
  - there exists an inconsistent application of \( \text{Rule}_A \) at state \( S_n \).
- For each \( n \in \mathbb{N} \), if \( S_n \) is not the last element of the sequence \( r \), then there exists a consistent application of \( \text{Rule}_A \) in \( S_n \) which \( S_{n+1} \) is the result of, and we have \( k > n \) such that \( S_k \neq S_n \).

Thus the definition of runs models termination by a finite sequence; runs can only be infinite if the state keeps on changing (eventually).

The idea of this definition is that the (hypothetical) "machine" which executes the UML Machine models chooses an update set non-deterministically. If it turns out to be inconsistent, the machine cannot proceed and the run stops. Choosing the update set is an internal action of the hypothetical execution machine which cannot be observed by the environment.\(^2\)

Note that due to the non-determinism introduced for example by the choose with do rule, there may be a non-singleton set of runs of a UML Machine \( A \). In particular, some of the branches of the choose with do rule may lead to an inconsistent application of a rule, while others may lead to a consistent rule application. In such a situation, there are runs that terminate at that choose with do rule while others continue (with the consistent rule application).

We define two UML Machines \( A \) and \( B \) to be equivalent if \( \text{Voc} A = \text{Voc} B \) and \( \text{Run} A = \text{Run} B \).

One may observe the input/output behavior of a UML Machine as follows. Given a UML Machine \( A \), tuples \( i \) and \( o \) of input and output names, and a sequence \( I \) of \( i \)-indexed multi-set tuples, consider the UML Machine

\(^2\) For readers familiar with process algebras this is, for example, just like the unobservable \( \tau \) action in the process algebra CCS [Mil89]. Intuitively, in CCS notation, the situation of having both consistent and inconsistent update sets at a point would thus look like \( \tau + p \) where after firing \( \tau \), the machine gets stuck, while after choosing \( p \), the execution may continue.
7.1 UML Machines 163

- the vocabulary \( \text{Voc}_\text{Behav}_{i,o}(A(I)) = \text{Voc}_A \cup \text{outlist}(A) \) (assuming \( \text{outlist}(A) \notin \text{Voc}_A \), if necessary by renaming),
- the initial state \( \text{Init}_\text{Behav}_{i,o}(A(I)) \) defined as \( \text{Init}_A \) and such that \( \text{outlist}(A) = [] \), and
- the transition rule \( \text{Rule}_\text{Behav}_{i,o}(A(I)) \) as given in Fig. 7.2.

**Fig. 7.2.** Behavior of UML Machine

Here \( f := I_f \) means that the input name \( f \) is assigned the \( f \)-component of the \( i \)-indexed tuple \( I \) of input multi-sets. \( \text{outlist}(A)_f := \text{outlist}(A)_f \) means to append the current value of the name \( f \) to the \( f \)-component of the sequence \( \text{outlist}(A) \). After completion of any possible run of this rule starting from an initial state where \( \text{outlist}(A) \) evaluates to the empty list [], \( \text{outlist}(A) \) contains a sequence of \( o \)-indexed tuples of multi-sets of values. Intuitively, this is the set of possible sequences of multi-set tuples of output values in the output names in \( o \) by iteratively adding each multi-set in \( I_{\text{inQu}} \) to \( \text{inQu}_A \) and assigning each multi-set in \( I_f \) to \( f \) for \( f \neq \text{inQu}_A \), then calling \( A \), and recording the multi-set of output values from \( \text{outQu}_A \) and the values of the output names in \( \text{outlist}(A) \).^1

**Definition 7.3.** Given a UML Machine \( A \) and tuples \( i \) and \( o \) of input and output names, the \( (i,o) \)-input/output behavior of \( A \) is a function \( [A]_{i,o}(I) \) from finite sequences of multi-sets of values to sets of sequences of multi-sets of values obtained by defining \( [A]_{i,o}(I) \) to consist of the possible contents of \( \text{outlist}(A) \) for each run of \( \text{Behav}_{i,o}(A(I)) \). For \( i = o = \emptyset \), we simply write \( [A]() \) and call it the input/output behavior of \( A \).

**Example**

As an example consider the UML Machine \( \text{SnDr} \) (a simplified version of the one considered in Sect. 8.1.3 in Fig. 8.7) with

- vocabulary \( \text{Voc}_\text{SnDr} = \{ \text{currState}, \text{inQu}_\text{SnDr}, \text{outQu}_\text{SnDr} \} \), where \( \text{currState} \) is a set name,
• the initial state \textbf{Init} \textit{Sndr} defined by \texttt{currState} = \texttt{inQu}_{\textit{Sndr}} = \emptyset, and
• whose transition rule \textbf{Rule} \textit{Sndr} is given in Fig. 7.3.

\begin{verbatim}
Rule Sndr :
  case currState of
    Wait: do
      choose \( e \) with \( e \in \texttt{inQu}_{\textit{Sndr}} \) do
        do in parallel
          if \( e = \texttt{send} \) then currState := Send
          \texttt{inQu}_{\textit{Sndr}} := \texttt{inQu}_{\textit{Sndr}} \setminus \{e\}
        enddo
    enddo
    Send: do
      do in parallel
        currState := Wait
        to\texttt{outQu}_{\textit{Sndr}}(\{\texttt{transmit} \})
      enddo
  enddo
\end{verbatim}

\textbf{Fig. 7.3. Example UML Machine}

The resulting input/output behavior can be characterized as follows.

\textbf{Fact 7.4.} For each sequence \((I_1, \ldots, I_n)\), \(\|\textit{Sndr}\|((I_1, \ldots, I_n))\) consists of those sequences \((O_1, \ldots, O_n)\) that fulfill the following conditions, for each \(i \in \{1, \ldots, n\}:\)

• \(O_i \subseteq \{\texttt{transmit} \}\).
• \(\#(O_1 \cup \ldots \cup O_i) \leq \#(I_1 \cup \ldots \cup I_{i-1}) \setminus \{\texttt{send} \}\).
• The conditions that
  \[\#(I_1 \cup \ldots \cup I_{i-1}) - \#(I_1 \cup \ldots \cup I_{i-1}) \setminus \{\texttt{send} \}\) \(< i - j - 2 \times \#(O_j \cup \ldots \cup O_{i-1})\)
  for each \(j < i\) and that \(\#(O_1 \cup \ldots \cup O_{i-1}) < \#(I_1 \cup \ldots \cup I_{i-1}) \setminus \{\texttt{send} \}\)
  imply \(\#O_i > 0\).

The last point means that an output of \texttt{transmit} is produced at time \(i\) (\(\#O_i > 0\)) provided that not all inputs of \texttt{send} that have previously been received have already prompted outputs (\(\#(O_1 \cup \ldots \cup O_{i-1}) < \#(I_1 \cup \ldots \cup I_{i-1}) \setminus \{\texttt{send} \}\)) and provided any other input received has already been consumed (which can be expressed as

\[\#(I_1 \cup \ldots \cup I_{i-1}) - \#(I_1 \cup \ldots \cup I_{i-1}) \setminus \{\texttt{send} \}\) \(< i - j - 2 \times \#(O_j \cup \ldots \cup O_{i-1})\)

because each processing of a \texttt{send} input takes two cycles, and during the remaining number of cycles, the system must have processed other inputs, provided there were any inputs at all). Note that in this example, it is not the
7.2 UML Machine Systems

To define the concept of UML Machine Systems, we first define the set Events from the previous section, containing the communication events, in more detail.

We assume a set MsgNm of message names consisting of finite sequences $n_1 : n_2 : \ldots : n_k$ of names where $n_1, \ldots, n_{k-2} \in \text{UMNames}$ are UMSs, $n_{k-1} \in \text{UMNames}$ is a UML Machine name, and $n_k \in \text{locMsgNm}$ is the local name of the message. We assume that MsgNm is partitioned into sets of operation names Op, signal names Sig, and return message names Ret, such that $n_1 : n_2 : \ldots : n_k \in \text{Op}$ (resp. Sig resp. Ret) if and only if $n_k \in \text{Op}$ (resp. Sig resp. Ret). For each operation name $op \in \text{Op} \cap \text{locMsgNm}$ there is a corresponding return message name $\text{return}_{op} \in \text{Ret} \cap \text{locMsgNm}$. We write $\text{retop}(\text{return}_{op}) \stackrel{def}{=} op$ for the operation name $op$ with which $\text{return}_{op}$ is associated. Messages with names in Op are called synchronous, those in Sig asynchronous.

We then define the set Events of events to consist of terms of the form $msg^{sdr}(exp_1, \ldots, exp_n)$ (for an arbitrary number $n$) where:

- $msg \in \text{MsgNm}$ is an $n$-ary message name,
- $sdr = []$ if $msg \in \text{Sig} \cup \text{Ret}$ and otherwise $sdr = n_1 :: n_2 :: \ldots :: n_k$ where $n_1, \ldots, n_{k-1} \in \text{UMNames}$ are names of UML Machines and $n_k \in \text{UMNames}$ is a name of a UML Machine, and
- $exp_1, \ldots, exp_n \in \text{Exp}$ are expressions, the parameters, or arguments of the event (for a given set of expressions Exp).

If the superscript $sdr$ is equal to the empty list $[]$, it may be omitted. We define $\text{Args}(m) \stackrel{def}{=} (exp_1, \ldots, exp_n)$ to be the sequence of the arguments of $m = msg^{sdr}(exp_1, \ldots, exp_n)$, $\text{msgnm}(m) \stackrel{def}{=} msg$ to be the name of its message, and $\text{snmd}(m) = sdr$ to be its sender. We provide the possibility to include the name of the sender in a message to enable handling of synchronous messages in the context of UML, see Sect. 8.1. We write $\text{head}(msg^{sdr}(exp_1, \ldots, exp_n))$ for $\text{head}(msg)$ and $\text{tail}(msg^{sdr}(exp_1, \ldots, exp_n))$ for $\text{tail}(msg)$.

**Definition 7.5.** A UML Machine System (UMS) $A = (\text{Name}_A, \text{Comp}_A, \text{Sched}_A, \text{Links}_A, \text{Msgs}_A)$ is given by the following data:

- A name Name$_A \in \text{UMNames}$ from a fixed set of UML Machine names UMNames.
A finite set \( \text{Comp}_A \subseteq \text{UMNames} \) of components. Each component \( C \in \text{Comp}_A \) has associated finite sets \( \text{Act}_C^A \) of activities and \( \text{Att}_C^A \) of local attributes. Each activity in \( \text{Act}_C^A \) has an associated UML Machine (possibly associated with a UMS from which it arises as defined in Fig. 7.4).

- A UML Machine \( \text{Sched}_A \), the scheduler.
- A set \( \text{Links}_A \) of links \( l = \{C, D\} \subseteq \text{Comp}_A \).
- A set \( \text{Msgs}_A \subseteq \text{MsgNim} \) of names of messages (see below) that the UMS is ready to receive.

Each activity UML Machine \( A \) of a component \( C \) is assumed to have the attributes of \( C \) as names, and also a flag \( \text{finished}_A \). Similarly, the scheduler UML Machine \( \text{Sched}_A \) has a flag \( \text{finished}_{\text{Sched}_A} \). The activities of a component \( C \) are required to have separate name-spaces except for the input and output queues \( \text{inQu}_C \) and \( \text{outQu}_C \) and the attributes. The components and the scheduler are required to have separate name-spaces except for the input and output queues.

Also, the scheduler may read each flag \( \text{finished}_A \).

The intuition is that a UMS models a computer system that is divided into a set of components that may communicate by sending messages through bi-directional communication links. \( \{C, D\} \in \text{Links}_A \) means that there is a communication link between the components \( C \) and \( D \) (where possibly \( C = D \)). The behavior of each component is specified using activities that are coordinated using the scheduler that calls the respective UML Machines, and which have access to the input and output queues and local attributes of the component. These UML Machines specifying the activities may themselves arise from UMSs as defined below. In that case, each UMS \( A \) specifying an activity of a component \( C \) has its own link set \( \text{Links}_A \), rather than sharing the links with the UMSs specifying other activities of \( C \). We will only consider UML Machines where the set of iteratively contained components is finite.

The execution of a UMS \( A \) is the joint execution of the UML Machines giving the component activities of the UMS, scheduled by the UML Machine giving the scheduler. It is modeled as a UML Machine \( \text{Exec}(A) \) with

- the vocabulary \( \text{VocExec}(A) \) which is the union of the vocabularies of the UML Machines modeling the activities in \( \text{Act}_C^A \) and of the scheduler UML Machine \( \text{Sched}_A \), with the set \( \{\text{linkQu}_A(l) : l \in \text{Links}_A\} \) of link queue names and which additionally contains the name \( \text{finished}_{\text{Exec}(A)} \);
- the initial state \( \text{InitExec}(A) \) defined as the initial states of the UML Machines modeling the activities in \( \text{Act}_C^A \) and of the scheduler UML Machine \( \text{Sched}_A \), and
- the transition rule \( \text{RuleExec}(A) \) given in Fig. 7.4.

The rule processes each incoming message, provided it is in the set of messages accepted by \( A \). In the case of an operation call, one needs to keep track of the message sender by adjusting the sender name attached to the message name. The message is then forwarded to the relevant component of \( A \). Then, similarly, the messages sent over links within \( A \) are processed, the scheduler
Rule Exec$(A)$:

\begin{enumerate}
  \item \textbf{loop} $e$ \textbf{through set} $\{ e \in \text{inQu}_C : \text{msgnm}(e) \in \text{Msgs}_A \land \text{head}(e) \in \text{Comp}_A \}$
  \begin{enumerate}
    \item if $\text{msgnm}(e) \in \text{Op}$ then $e' := \text{msgnm}(e)^{A, \text{ndir}(e)}(\text{Args}(e))$
    \item $e' := e$
  \end{enumerate}
  to $\text{inQu}_{\text{Exec}(\text{head}(e))}(\{ \text{tail}(e') \})$

  \item \textbf{loop} $e$ \textbf{through set} $\{ e \in \bigcup_{l \in \text{Links}_A} \text{linkQu}_{\text{Exec}(A)}(l) : \text{head}(e) \in \text{Comp}_A \}$
  \begin{enumerate}
    \item to $\text{inQu}_{\text{Exec}(\text{head}(e))}(\{ \text{tail}(e') \})$
  \end{enumerate}

  \item \textbf{forall} $l$ with $l \in \text{Links}_A$
  \begin{enumerate}
    \item $\text{linkQu}_{\text{A}}(l) := \emptyset$
  \end{enumerate}

  \item Rule Sched$_A$
  \begin{enumerate}
    \item $\text{finished}_{\text{Exec}(A)} := \text{finished}_{\text{Sched}_A}$
    \item $\text{forall} S$ with $S \in \text{Comp}_A$
    \begin{enumerate}
      \item \textbf{loop} $e$ \textbf{through set} $\text{outQu}_{\text{Exec}(S)}$
      \begin{enumerate}
        \item if $\text{msgnm}(e) \in \text{Op}$ then $e' := \text{msgnm}(e)^{S, \text{ndir}(e)}(\text{Args}(e))$
      \end{enumerate}
      \begin{enumerate}
        \item $e' := e$
        \item if $\text{head}(e) = A$ then $\text{toOutQu}_C(\{ \text{tail}(e') \})$
        \item $\text{outQu}_{\text{Exec}(S)} := \emptyset$
      \end{enumerate}
    \end{enumerate}
  \end{enumerate}

\end{enumerate}

Fig. 7.4. Behavior of UMS

is called, and finally the output messages from the components in $A$ are processed (again by adjusting the sender name in the case of operation calls). Note that if the name $A$ itself appears in the list of ASM names determining the sender (resp. receiver) of a message while this message is actually within $A$, this signifies that the message comes from (resp. goes to) an ASM outside $A$.

Precisely, a message $n_1 :: n_2 :: \ldots :: n_k$ sent by a UMS which is part of the system $n$ is delivered as follows (as the UML Machine Exec$(A)$ describes below in detail for a UMS $A$):

\begin{itemize}
  \item If $n_1 = n$ then the system $n$ sends out the tail $n_2 :: \ldots :: n_k$ of the message within the UML Machine containing $n$.
  \item If $n_1$ is a part of $n$, then the tail $n_2 :: \ldots :: n_k$ of the message is delivered to $n_1$.
\end{itemize}

Note that the definition of Rule Exec$(A)$ is complicated by the handling of synchronous messages $op \in \text{Op}$ and their returns, introduced since these are offered by UML. To use UML Machines just with asynchronous messages $\text{sig} \in \text{Sig}$ (which will be our main usage), the simpler definition in Fig. 7.5 suffices. Here $\text{links}_S \equiv \{ \{ A, B \} \in \text{Links}_A : A = S \}$ is the set of links connected to $S$. It is a relatively straightforward formal exercise to show that, if all messages are asynchronous, the rule in Fig. 7.5 indeed defines the same UML Machine as that in Fig. 7.4.
Rule \text{Exec}(A) :
forall \ S \ with \ S \in \text{Comp}_A \ do
\text{inQu}_{\text{Exec}(S)}(\{\{\text{tail}(e) \ : \ e \in (\text{inQu}_c \setminus \text{Msgs}_A)\}\cup\bigcup_{l \in \text{links}_A} \text{linkQu}_{\text{Exec}(A)}(l) \land \text{head}(e) = S\}) ;
inQu_c := \emptyset ;
forall \ l \ with \ l \in \text{Links}_A \ do
\text{linkQu}_A(l) := \emptyset ;
Rule \text{Sched}_A ;
\text{finished}_{\text{Exec}(A)} := \text{finished}_{\text{Sched}_A} ;
forall \ l \ with \ l \in \text{Links}_A \ do
\text{linkQu}_{\text{Exec}(A)}(l) := \{e \in \text{outQu}_{\text{Exec}(S)} : S \in \text{Comp}_A \land l = \{\text{head}(e), S\}\} ;
\text{toQu}_c(\bigcup_{S \in \text{Comp}_A} \{\text{tail}(e) : e \in \text{outQu}_{\text{Exec}(S)} \land \text{head}(e) = \text{Exec}(A)\}) ;
forall \ S \ with \ S \in \text{Comp}_A \ do
\text{outQu}_{\text{Exec}(S)} := \emptyset ;

\textbf{Fig. 7.5. Behavior of UMS (only asynchronous messages)}

An example for a scheduler $\text{Sched}_A$ is the one given in Fig. 7.6 that, given $n$ activities as UML Machines $A_1, \ldots, A_n$, simply executes them in parallel. Each of these UML Machines may in turn be defined as the behavior $\text{Exec}(A)$ of a UMS $A$, as defined above.

Rule \text{Sched}_A :
\begin{align*}
do & \quad \text{in - parallel} \\
& \text{Rule } A_1 \\
& \ldots \\
& \text{Rule } A_n \\
\end{align*}
enddo;
\text{finished}_{\text{Sched}_A} := \text{finished}_{\text{Exec}(A_1)} \land \ldots \land \text{finished}_{\text{Exec}(A_n)} ;

\textbf{Fig. 7.6. Simple scheduler}

\subsection{7.3 Refinement}

A useful paradigm of system development is that of \textit{stepwise refinement}. One starts with an abstract specification and refines it in several steps to a concrete specification which is implemented. For more discussion on the role of refinement in system development see Sect. 8.2.1.

The \texttt{choose} rule defined in Sect. 7.1 allows us to use underspecification to postpone design decisions to a later stage of system development. Correspondingly, we will define a notion of \textit{refinement} that allows us to proceed from abstract to more concrete specifications in a well-defined way. It is inspired
by a definition given in [Bro99, BS01] (in the setting of stream-processing functions). We will also define a notion of delayed refinement, which is a relaxation of refinement by allowing delays to be inserted. It allows a more flexible treatment, while still offering convenient structural properties.

First we give some preliminary definitions for delayed refinement.

The following definition is inspired by the treatment in [AL93].

**Definition 7.6.** For sets \( S, T \) of sequences of event multi-sets we write \( S \subseteq T \) if for each multi-set \( s \in S \) there is a multi-set \( t \in T \) such that \( s = t \), where for a sequence \( s \) of event multi-sets, \( \tilde{s} \) is the sequence obtained from \( s \) by leaving out all empty multi-sets.

Two sets \( S, T \) of sequences of event multi-sets are stutter-equivalent if \( S \subseteq T \) and \( T \subseteq S \). Two sequences \( s, t \) of event multi-sets are stutter-equivalent if the singleton sets \( \{s\}, \{t\} \) are.

For a set \( S \) of finite sequences of event multi-sets and a set \( \mathcal{E} \) of events, we define

\[
S \bowtie \mathcal{E} \overset{\text{def}}{=} \{(M_1 \setminus \mathcal{E}, \ldots, M_i \setminus \mathcal{E}) : (M_1, \ldots, M_i) \in S\}.
\]

**Definition 7.7 (Behavioral refinement of UML Machines).** Suppose we are given UML Machines \( A, A' \), tuples \( i \) and \( o \) of input and output names, and a set \( \mathcal{E} \subseteq \text{Events} \).

We say that \( A' \bowtie (i, o) \)-refines \( A \) if for each sequence \( I_1, \ldots, I_n \) of event multi-set tuples with \( \bigcup_i |I_i| \subseteq \mathcal{E} \), we have

\[
[A']_{i, o}(I_1, \ldots, I_n) \bowtie \mathcal{E} \subseteq [A]_{i, o}(I_1, \ldots, I_n) \bowtie \mathcal{E}.
\]

We say that \( A' \) delayed \( \mathcal{E} \)-refines \( A \) if for each sequence \( I_1, \ldots, I_n \) of event multi-sets with \( \bigcup_i |I_i| \subseteq \mathcal{E} \), we have

\[
[A']_{i, o}(I_1, \ldots, I_n) \bowtie \mathcal{E} \subseteq [A]_{i, o}(I_1, \ldots, I_n) \bowtie \mathcal{E}.
\]

For \( \mathcal{E} = \text{Events} \) and \( i = o = \emptyset \), we say \( A' \) (delayed) refines \( A \) if \( A' \) (delayed) \( \mathcal{E} \)-(i, o)-refines \( A \).

We say that \( A \) is stutter-invariant if for each two stutter-equivalent sequences \( s, t \) of event multi-sets, the sets \( [A]_{i, o}(s) \) and \( [A]_{i, o}(t) \) are also stutter-equivalent. A UML \( A \) is called stutter-invariant if its scheduler and activity UML Machines are.

One can thus use the set of events \( \mathcal{E} \) in order to hide the events not contained in it with respect to the refinement (this is inspired by the corresponding operators in CSP [Hoa85] and CCS [Mil89] although not quite the same).

**Example**

Let \( R(0), R(1) \) be any two rules. The UML Machine \( B \) with the rule **Rule B** = \( R(1) \) refines the UML Machine \( A \) with the following rule **Rule A** (assuming that they have the same initial state):
choose \(b\) with \(b \in \{0, 1\}\) do

\[
R(b)
\]

**Fact 7.8.** (Delayed) \(\mathcal{E}(i, o)\)-refinement of UML Machines is a preorder for each set of events \(\mathcal{E} \subseteq \text{Events}\) and tuples \(i\) and \(o\) of input and output names.

We extend the definition of refinement to UMSs. Here we are interested in a kind of “white-box” refinement that preserves the system structure (such as the links between components). We do not consider hiding in refinement (as in the case of refinement of UML Machines). Variations are, however, possible and may be useful in a given situation.

**Definition 7.9 (Behavioral refinement of UMSs).** A \(\text{UMS} \mathcal{A}' = (\text{Comp}_{\mathcal{A}'}, \text{Sched}_{\mathcal{A}'}, \text{Links}_{\mathcal{A}'}, \text{Msgs}_{\mathcal{A}'})\) is a (delayed) refinement of a \(\text{UMS} \mathcal{A} = (\text{Comp}_{\mathcal{A}}, \text{Sched}_{\mathcal{A}}, \text{Links}_{\mathcal{A}}, \text{Msgs}_{\mathcal{A}})\) if \(\text{Msgs}_{\mathcal{A}} \subseteq \text{Msgs}_{\mathcal{A}'}\) and if there are bijections \(b : \text{Comp}_{\mathcal{A}} \rightarrow \text{Comp}_{\mathcal{A}'}\) and \(b_C : \text{Act}_{\mathcal{A}} \rightarrow \text{Act}_{\mathcal{A}'}(\text{for each } C \in \text{Comp}_{\mathcal{A}})\) such that

- for each component \(C \in \text{Comp}_{\mathcal{A}}\) and activity \(A \in \text{Act}_{\mathcal{A}}\), the UML Machine of \(b_C(A)\) is a (delayed) \((i, o)\)-refinement of the \(\mathcal{A}\) Machine where \(i = \text{Att}_C\) and \(o = i \cup \{\text{finished}_A\}\), and

- \(\text{Sched}_{\mathcal{A}} = \text{Sched}_{\mathcal{A}'}\) and \(\text{Links}_{\mathcal{A}} = \text{Links}_{\mathcal{A}'}\) (each up to renaming of components using the bijections \(b, b_C\)).

For delayed refinement we additionally require that \(\mathcal{A}\) and \(\mathcal{A}'\) are stutter-invariant.

We show that refinement of UMSs behaves well with respect to defining their behavior using UML Machines. Note that, in the case of delayed refinement, the stutter-invariance of the UML Machines in the UMSs is essential. Intuitively, it ensures that a delay in one of the components, which is permitted to be introduced during refinement, does not lead to a change of behavior in the other components, except possibly for some further delay.

**Fact 7.10.** If the UMS \(\mathcal{A}'\) is a (delayed) refinement of the UMS \(\mathcal{A}\) then the UML Machine \(\text{Exec}(\mathcal{A}')\) is a (delayed) refinement of the UML Machine \(\text{Exec}(\mathcal{A})\).

Next, the structural properties of UML Machine refinement carry over to the UMS case.

**Fact 7.11.** (Delayed) refinement of (stutter-invariant) UMSs is a preorder.

We show that (delayed) refinement of UMSs is preserved by substitution (and is thus a precongruence with respect to composition by system formation).

A parameterized UMS \(\mathcal{A}(\mathcal{Y}_1, \ldots, \mathcal{Y}_n)\) is a UMS specification where \(n\) of the component activities are replaced by variables \(\mathcal{Y}_1, \ldots, \mathcal{Y}_n\). For UMSs \(\mathcal{A}_1, \ldots, \mathcal{A}_n\), \(\mathcal{A}(\mathcal{A}_1, \ldots, \mathcal{A}_n)\) is the UMS obtained by substituting in \(\mathcal{A}\) the UML Machine \(\text{Exec}(\mathcal{A}_i)\) defining the behavior of \(\mathcal{A}_i\) for \(\mathcal{Y}_i\), for each \(i\).
Fact 7.12. Suppose we are given a parameterized UMS $\mathcal{A}(V_1, \ldots, V_n)$, where the activity variable $V_i$ belongs to the component $C_i$, for each $i = 1, \ldots, n$, and that we are given UMSs $\mathcal{A}_i$ and $\mathcal{A}_i'$ for each $i$.

If for each $i = 1, \ldots, n$, $\text{Exec}(\mathcal{A}_i)$ is a $(i_i, o_i)$-refinement of $\text{Exec}(\mathcal{A}_i)$ where $i_i = \text{Att}^\mathcal{A}_i$ and $o_i = i_i \cup \{\text{finished}_{\text{Exec}(\mathcal{A}_i)}\}$ then $\mathcal{A}(\mathcal{A}_1, \ldots, \mathcal{A}_n)$ is a refinement of $\mathcal{A}(\mathcal{A}_1, \ldots, \mathcal{A}_n)$.

Suppose further that the scheduler and all activity UML Machines in $\mathcal{A}(V_1, \ldots, V_n)$ are stutter-invariant, as well as all $\text{Exec}(\mathcal{A}_i)$ and $\text{Exec}(\mathcal{A}_i')$, for each $i$. If for each $i = 1, \ldots, n$, $\text{Exec}(\mathcal{A}_i)$ is a delayed $(i_i, o_i)$-refinement of $\text{Exec}(\mathcal{A}_i)$ where $i_i = \text{Att}^\mathcal{A}_i$ and $o_i = i_i \cup \{\text{finished}_{\text{Exec}(\mathcal{A}_i)}\}$ then $\mathcal{A}(\mathcal{A}_1, \ldots, \mathcal{A}_n)$ is a delayed refinement of $\mathcal{A}(\mathcal{A}_1, \ldots, \mathcal{A}_n)$.

Theorem 7.13. (Delayed) refinement of (stutter-invariant) UMSs is a pre-congruence with respect to composition by system formation using parameterized UMSs.

Definition 7.14 (Behavioral equivalence). Two UMSs $\mathcal{A}$ and $\mathcal{A}'$ are (delayed) equivalent if $\mathcal{A}$ is a (delayed) refinement of $\mathcal{A}'$ and $\mathcal{A}'$ is a (delayed) refinement of $\mathcal{A}$.

Corollary 7.15. (Delayed) equivalence of (stutter-invariant) UMSs is a congruence with respect to composition by system formation.

We provide a more flexible concept of refinement (inspired by the definition in [BS01]).

Definition 7.16 (Interface refinement). Given UMSs $\mathcal{A}$ and $\mathcal{A}'$ and a parameterized UMS $I(Y)$, $\mathcal{A}'$ is a (delayed) $I$-interface refinement of $\mathcal{A}$ if $\mathcal{A}'$ is a (delayed) refinement of $I(\mathcal{A})$.

Theorem 7.17. Each (stutter-invariant) UMS $\mathcal{A}$ is a (delayed) $I$-interface refinement of itself, where $I(Y) \overset{\text{def}}{=} Y$.

For all UMSs $\mathcal{A}$, $\mathcal{A}'$, and $\mathcal{A}''$ such that $\mathcal{A}'$ is a (delayed) $I$-interface refinement of $\mathcal{A}$ and $\mathcal{A}''$ is a (delayed) $I'$-interface refinement of $\mathcal{A}'$, $\mathcal{A}''$ is a (delayed) $I' \circ I(Y) \overset{\text{def}}{=} I'(I(Y))$-interface refinement of $\mathcal{A}$.

7.4 Rely-Guarantee Specifications

To reason about system specifications in a modular way, one may usefully employ rely-guarantee specifications (also called assume/guarantee). The following definition follows a corresponding notion in [Bro98, BS01].

Definition 7.18. Given a UML Machine $\mathcal{A}$ and sets $R, G$ of sequences of event multi-sets, we say that $\mathcal{A}$ fulfills the rely-guarantee specification $(R, G)$ if for any $(I_1, \ldots, I_n) \in R$, we have $[\mathcal{A}](I_1, \ldots, I_n) \subseteq G$. A rely-guarantee specification $(R, G)$ where $R$ is the set of all sequences of event multi-sets accepted by $\mathcal{A}$ is also called a trace property.
We say that a set \( S \) of sequences of event multi-sets are stutter-closed if it contains every sequence of events that is stutter-equivalent to a sequence in \( S \).

**Theorem 7.19.** Suppose that the UML Machine \( A \) fulfills the rely-guarantee specification \( (R,G) \) where \( R \cap \mathcal{E} = R \) and \( G \cap \mathcal{E} = G \), and suppose \( E = \{ \mathbf{1} : I \cap \mathcal{E} = \mathbf{1} \} \).

If the UML Machine \( A' \) \( E \)-refines \( A \) and \( A' \) fulfills the rely-guarantee specification \( (R,E) \) then \( A' \) fulfills the rely-guarantee specification \( (R,G) \).

If the UML Machine \( A' \) delayed \( E \)-refines \( A \), \( G \) is stutter-closed, and \( A' \) fulfills the rely-guarantee specification \( (R,E) \), then \( A' \) fulfills the rely-guarantee specification \( (R,G) \).

In particular, the above theorem implies that (delayed) refinement preserves (stutter-closed) trace properties. Note that we do not require \( R \) to be stutter-closed; intuitively, the reason is that the above statement holds for each fixed sequence of input event multi-sets.

### 7.5 Reasoning About Security Properties

We now use the notions from Sect. 3.3.3 in the context of security analysis using UML machines. The notion of UMSs allows a rather natural modeling of potential adversary behavior. We can model specific types of adversaries that can attack different parts of the system in a specified way. For example, an attacker of type insider may be able to intercept the communication links in a company-wide local area network. We model the actual behavior of the adversary by defining a class of UML Machines that can access the communication links of the system in a specified way. To evaluate the security of the system with respect to the given type of adversary, we consider the joint execution of the system with any UML Machine in this class. This way of reasoning allows an intuitive formulation of many security properties. Since the actual verification is rather indirect this way, we also give alternative intrinsic ways of defining security properties below, which are more manageable, and show that they are equivalent to the earlier ones.

Security evaluation of specifications is done with respect to a given type \( A \) of adversary. The capabilities of an adversary of a given type are captured as follows.

Firstly, given a UMS \( A \) we define the set \( int_A \) of (recursively) contained components:

- for a UML Machine \( A \), \( int_A := \{ A \} \) and
- for a UMS \( A \), \( int_A := \bigcup_{B \in \text{Comp}_A} int_B \).

Similarly, for a UMS \( A \) we define the set \( \text{lbs}_A \) of (recursively) contained links:
• for a UML Machine \( A \), \( \text{lks}_A := \emptyset \) and
• for a UMS \( \mathcal{A} \), \( \text{lks}_A := \text{lks}_S \cup \bigcup_{B \in \text{Comp}\mathcal{A}} \text{lks}_B \).

To capture the capabilities of an attacker, we assume that, given a UMS \( \mathcal{A} \), we have a function \( \text{threats}^A(x) \) that takes a component or link \( x \in \text{int}_A \cup \text{lks}_A \) and a type of adversary \( A \) and returns a set of strings \( \text{threats}^A(x) \subseteq \{\text{delete}, \text{read}, \text{insert}, \text{access}\} \) under the following conditions:

• for \( x \in \text{int}_A \), we have \( \text{threats}^A(x) \subseteq \{\text{access}\} \),
• for \( x \in \text{lks}_A \), we have \( \text{threats}^A(x) \subseteq \{\text{delete}, \text{read}, \text{insert}\} \), and
• for \( l \in \text{lks}_A \) with \( i \in l \) and \( \text{threats}^A(i) = \{\text{access}\} \), the equation \( \text{threats}^A(l) = \{\text{delete}, \text{read}, \text{insert}\} \) holds.

The idea is that \( \text{threats}^A(x) \) specifies the threat scenario against a component or link \( x \) in the UMS \( \mathcal{A} \) that is associated with an adversary type \( A \). On the one hand, the threat scenario determines which data the adversary can obtain by accessing components; on the other hand, it determines which actions the adversary is permitted by the threat scenario to apply to the links concerned. See Chap. 4 for examples of functions \( \text{threats}() \).

Then each function \( \text{threats()} \) gives rise to the set \( \mathcal{K}^0_A \subseteq \text{Exp} \) of accessible knowledge which contains knowledge that may arise from accessing components and is defined to consist of all expressions \( E \in \text{Exp} \) appearing in the specification for any \( i \in \text{int}_A \) with \( \text{access} \in \text{threats}^A(i) \).

Next, for an adversary type \( A \), one has to specify a subalgebra \( \mathcal{K}^p_A \subseteq \text{Exp} \) of previous knowledge of adversaries of type \( A \). Then we define the set \( \mathcal{K}^0_A \subseteq \text{Exp} \) of the initial knowledge of any adversary of type \( A \) to be the \( \text{Exp} \)-subalgebra generated by \( \mathcal{K}^0_A \cup \mathcal{K}^p_A \) (thus it is closed under application of the algebra operations).

**Definition 7.20.** An adversary \( \text{adv} \in \text{Advers}_\mathcal{A}(A) \) of type \( A \) with previous knowledge \( \mathcal{K}^p_A \subseteq \text{Exp} \) against the UMS \( \mathcal{A} \) is a UML Machine such that

• \( \text{knows} \in \text{Voc adv} \) for a set name \( \text{knows} \) and \( \text{Voc adv} \cap \text{Voc} \mathcal{A} = \{\text{linkQu}_{\mathcal{A}}(l) : \text{threats}^A(l) \subseteq \{\text{access}\}\} \),
• \( \text{at the initial state Init adv} \) we have \( \text{knows} = \mathcal{K}^0_A \) as defined above, and
• the transition rule \( \text{Rule adv} \) fulfills the following three conditions (where \( f \) denotes the value of the name \( f \) before, and \( \text{adv}(f) \) the value after executing \( \text{Rule adv} \)):

  - \( \text{adv}(\text{knows}) = (\text{knows} \cup \{E \in \text{Args}(e) : e \in \bigcup_{d \in \text{threats}^A(l)} \text{linkQu}_{\mathcal{A}}(l)\}) \) (where \( \langle S \rangle \) is the \( \text{Exp} \)-subalgebra generated of a set \( S \subseteq \text{Exp} \), as defined in Sect. 3.3.3)
  - \( \text{adv}(\text{linkQu}_{\mathcal{A}}(l)) \subseteq \text{adv}(\text{knows}) \) or \( \text{adv}(\text{linkQu}_{\mathcal{A}}(l)) \subseteq \text{linkQu}_{\mathcal{A}}(l) \cup \text{adv} \) (knows)
  - delete \( \notin \text{threats}^A(l) \) implies \( \text{linkQu}_{\mathcal{A}}(l) \subseteq \text{adv}(\text{linkQu}_{\mathcal{A}}(l)) \)
  - insert \( \notin \text{threats}^A(l) \) implies \( \text{adv}(\text{linkQu}_{\mathcal{A}}(l)) = \text{linkQu}_{\mathcal{A}}(l) \) or \( \text{adv}(\text{linkQu}_{\mathcal{A}}(l)) = \emptyset \)
The intuition behind this definition is that an adversary may initially know only the data contained in $K_A^0$ (arising from previous knowledge and the data gained from accessing unprotected system parts) and may apply only those actions to the link queues which arise from the threats() sets defined above. The definition accommodates the fact that an adversary $A$ can be able to remove all values sent over the link $l$ (that is, $delete \in threats_A^e(l)$) may not be able to selectively remove a value $e$ with known meaning from $l$. For example, the messages sent over the Internet within a virtual private network are encrypted. Thus, an adversary who is unable to break the encryption may be able to delete all messages indiscriminately, but not a single message whose meaning would be known to the adversary.

The execution $A_{adv}$ of $A$ in the presence of the attacker $adv \in Advers_A(A)$ of type $A$ is modeled by a UML Machine $A_{adv}$ such that

- $\text{Voc}_{A_{adv}} = \text{Voc}_A \cup \text{Voc}_{adv}$,
- the initial state $\text{Init}_{A_{adv}}$ is defined as $\text{Init}_A$, and in addition $\text{Knows}_A = K_A^0$, and
- the transition rule $\text{Rule}_{A_{adv}}$ is defined as

$$\text{Rule}_{A_{adv}} \triangleq \text{RuleExec}(A) \cup \text{Rule} adv.$$

We note that our notion of refinement is preserved by the above construction.

**Fact 7.21.** Suppose we are given UMSs $B$ and $A$ such that $B$ is a refinement of $A$. Suppose we are given an adversary $adv$ of a given type $A$. Then the UML Machine $B_{adv}$ is a $(\emptyset, \{\text{Knows}\})$-refinement of the UML Machine $A_{adv}$.

Suppose we are given a UMS $A$ and an adversary $adv \in Advers_A(A)$ of type $A$. For any execution $e \in \text{Run}_{A_{adv}}$ of length $n \in \mathbb{N}$ (as defined in Sect. 7.1), we define the knowledge set $K_{adv}^e(A)$ of expressions known to $adv$ after execution of $e$:

- $K_{adv}^{[1]}(A)$ is the set $K_A^0$ defined above,
- $K_{adv}^{e\cdot e}(A) = (K_{adv}^e(A) \cup \bigcup_{\text{read} \in \text{reads}_A(l)} \text{link}_A^e(l))$ where $\text{link}_A^e(l)$ is the set of elements in the multi-set $\text{linkQu}_A(l)$ in state $e$.

We have the following characterizations of the adversary’s capabilities.

**Fact 7.22.** Suppose we are given a UMS $A$, an adversary $adv \in Advers_A(A)$ of type $A$, and an execution $e \in \text{Run}_{A_{adv}}$. Then after execution of $e$, $\text{Knows}$ evaluates to $K_{adv}^e(A)$.

We consider the overall adversary knowledge, which is simpler to handle and often sufficient. The overall knowledge of expressions known to any adversary $adv$ of type $A$ after $n$ iterations of $\text{Rule}_{A_{adv}}$ is $K_{adv}^n(A) \triangleq \bigcup_{adv \in \text{Advers}_A(A)} K_{adv}^e(A)$ (where $adv \in Advers_A(A)$ ranges over all adversaries of type $A$ and $e$ over all executions of $A_{adv}$ of length $n$). We write $K_A(A) \triangleq \bigcup_{n \in \mathbb{N}} K_{adv}^n(A)$.
Note that the definition of $\mathcal{K}_{\text{adv}}(A)$ does not depend on any particular adversary, only on the adversary type $A$. This allows one to give equivalent intrinsic definitions for the security properties considered in the following subsections, which simplifies reasoning.

**Fact 7.23.** Given a system $A$, the set knows of $A_{\text{adv}}$ evaluates to a subset of $\mathcal{K}_{A}(A)$, at any point.

This means that, during the execution of $A_{\text{adv}}$, an adversary of type $A$ will get to know, and thus can add to a link only expressions in $\mathcal{K}_{\text{adv}}(A)$.

Suppose we are given a UMS $A$ with a name $v$ and an adversary $\text{adv} \in \text{Advers}_{A}(A)$ of type $A$. For any execution $e \in \text{Run}_{A_{\text{adv}}}$ of length $n \in \mathbb{N}$ of $A_{\text{adv}}$, we define the influence set $\mathcal{I}_{\text{adv}}^{e}(A, v)$ of expressions that may be written to $v$ after execution of $e$:

- $\mathcal{I}_{\text{adv}}^{[1]}(A, v) \equiv \emptyset$.
- $\mathcal{I}_{\text{adv}}^{e}(A, v)$ is the union of $\mathcal{I}_{\text{adv}}^{e}(A, v)$ and the value to which $v$ evaluates at state $e$.

We consider the overall influence set, which is simpler to handle and often sufficient. The overall influence of any adversary $\text{adv}$ of type $A$ on a variable $v$ after $n$ iterations of Rule $A_{\text{adv}}$ is $\mathcal{I}_{A}^{n}(A, v) \equiv \bigcup_{e \in \text{Run}_{A_{\text{adv}}}}\mathcal{I}_{\text{adv}}^{e}(A, v)$ (where $\text{adv} \in \text{Advers}_{A}(A)$ ranges over all adversaries of type $A$ and $e \in \text{Run}_{A_{\text{adv}}}$ over all executions of $A_{\text{adv}}$ of length $n$). We write $\mathcal{I}_{A}(A, v) \equiv \bigcup_{n \in \mathbb{N}}\mathcal{I}_{A}^{n}(A, v)$.

**Fact 7.24.** Given a UMS $A$ with a name $v$, then during any run $e \in \text{Run}_{A_{\text{adv}}}$, the name $v$ evaluates to an element of $\mathcal{I}_{A}(A, v)$, at any point.

This means that, during the execution of $A_{\text{adv}}$, an adversary $\text{adv}$ of type $A$ can only cause $A$ to write to $v$ expressions in $\mathcal{I}_{A}(A, v)$.

One possibility to specify security requirements is to define an idealized UMS where the required security property evidently holds (for example, because all links and components are secure according to the kind of adversary under consideration), and to prove that the analyzed UMS is behaviorally equivalent to the idealized one (using Definition 7.14, and following an idea in [Aba00]). For an example see Sect. 5.1.

In the following two subsections, we consider ways of specifying the important security properties, secrecy and integrity which do not require one to explicitly construct such an idealized system.

### 7.5.1 Secrecy

We consider a secrecy property following the standard approach of [DY83] that is defined in an intuitive way by incorporating the attacker model, and we give an equivalent internal characterization of the property which allows easier verification. The secrecy property considered here relies on the idea
that a system specification preserves the secrecy of a piece of data \( d \) if the system never sends out any information from which \( d \) could be derived, even in interaction with an adversary. This definition is relatively coarse in that it may not prevent implicit information flow, but it is rather easy to verify and seems to be sufficient in practice [Aba00].

**Definition 7.25.** Given a UMS \( \mathcal{A} \), we say that an adversary of type \( A \) may eventually find out an expression \( E \in \text{Exp} \) from \( \mathcal{A} \) given inputs in \( \mathcal{E} \subseteq \text{Events} \) if there exists \( \text{adv} \in \text{Advers}_\mathcal{A}(A) \), an input sequence \( I \) whose multisets only contain elements in \( \mathcal{E} \), and a sequence \( s \in \bigcup_{\text{adv}} [A_{\text{dfr}}]_{[0, \text{length}]}(I) \) such that one of the knowledge sets in \( s \) contains \( E \). Otherwise we say that \( \mathcal{A} \) preserves the secrecy of \( E \) from adversaries of type \( A \) given inputs in \( \mathcal{E} \).

Given a variable \( v \), we say that \( \mathcal{A} \) preserves the secrecy of the variable \( v \) from adversaries of type \( A \) given inputs in \( \mathcal{E} \subseteq \text{Events} \) if for every expression \( E \) which is a value of the variable \( v \) at any point, \( \mathcal{A} \) preserves the secrecy of \( E \) from adversaries of type \( A \) given inputs in \( \mathcal{E} \).

We say that \( \mathcal{A} \) preserves the secrecy of \( E \) from adversaries of type \( A \) if \( \mathcal{A} \) preserves the secrecy of \( E \) from adversaries of type \( A \) given inputs in \( \text{Events} \).

Note that, alternatively, one could formulate this definition using the concept of a "most general adversary" (which non-deterministically behaves like any possible adversary).

**Examples**

- The UML Machine that sends the expression \( \{m\}_K : K \in \text{Exp} \) over an Internet link does not preserve the secrecy of \( m \) or \( K \) against attackers eavesdropping on the Internet, but the UML Machine that sends \( \{m\}_K \) (and then terminates) does, assuming that it preserves the secrecy of \( K \) against attackers eavesdropping on the Internet.

- The UML Machine \( \mathcal{A} \) that receives a key \( K \) encrypted with its public key \( K_A \) over a communication link and sends back \( \{m\}_K \) over the link does not preserve the secrecy of \( m \) against attackers eavesdropping on and inserting messages on the link, because such an adversary can insert \( \{K_0\}_K \) for a key \( K_0 \) known to him into the communication to \( \mathcal{A} \) and then decrypt the message \( \{m\}_K \) received back. \( \mathcal{A} \) does preserve the secrecy of \( m \) against attackers that cannot insert messages on the link.

We give an intrinsic characterization of preservation of secrecy in terms of knowledge sets, which makes it easier to verify.

**Theorem 7.26 (Secrecy Characterization).** \( \mathcal{A} \) preserves the secrecy of \( E \) against adversaries of type \( A \) if and only if \( E \notin \mathcal{K}_A(\mathcal{A}) \).

There is a similar characterization of preservation of secrecy given inputs in \( \mathcal{E} \), which, however, we will not need in the following.
This theorem is especially useful if one can give an upper bound for $K_A(A)$, which is often possible when the security-relevant part of the specification of $A$ is given as a sequence of command schemas of the form \texttt{await event e \rightarrow check condition g \rightarrow output event e'} (as for example when using UML sequence diagrams together with their UML Machine semantics defined in Chap. 8).

\textbf{Refinement}

When refining specifications, the concrete specification must have all relevant properties of the initial specification. This is indeed the case for system properties that can be expressed as properties on traces (taking refinement to be reverse inclusion on trace sets). A classical example is the Alpern–Schneider framework of safety and liveness properties [AS85].

However, many security properties proposed in the literature are properties on trace sets rather than traces and give rise to the refinement problem, which means that these properties are not preserved under many standard definitions of refinement (for non-interference this is pointed out in [McL94, McL96]; a similar situation might arise if one tried to use the current notion of refinement with equivalence-based notions of secrecy explained for example in [Aba00]).

For such properties, developing secure systems in a stepwise manner requires one to redo security proofs at each refinement step. More worryingly, since an implementation is necessarily a refinement of its specification, an implementation of a secure specification may not be secure. Thus the results of verifying such properties on the level of specifications need to be applied with care, as pointed out in [RS98].

We show that our notion of secrecy (which follows a standard approach) is preserved by the standard refinement operators defined above. To understand the cause of the refinement problem and how to fix it, we first make some general remarks on non-determinism and refinement.

In the specification of systems one may employ non-determinism in different ways, including the following:

Underspecification: to simplify design or verification of systems. Certain details may be left open in the early phases of system development or when verifying system properties, to simplify matters or because they may not be known (for example the particular scheduler used to resolve concurrency).

Unpredictability: to provide security. For example, keys or nonces are chosen in a way that should make them unguessable.

While the first kind of non-determinism is merely a tool during the development or verification, the second is a vital part of the functionality of a system. When one identifies the two kinds of non-determinism one faces the refinement problem mentioned above.

We separate the two kinds of non-determinism in the following way. The non-determinism of functional importance for the system is \textit{only} modeled
by specific mechanisms (for example, key generation is modeled by having separate sets of private keys for the different actors). Thus the security of a system does not rely on non-deterministic choice in the formal model.

It is quite common in the formal modeling of security to provide special primitives for operations such as key generation, encryption, etc. However, security properties for non-deterministic specifications often also use the non-deterministic choice operators to provide unpredictability (since they generally do not seek to provide a security-preserving refinement). Our security property rules this out.

The following result is an adjustment of a result in [Jür01g] to our current model.

**Theorem 7.27 (Secrecy Preservation).** If $A$ preserves the secrecy of $E$ from adversaries of type $A$ given inputs in $\mathcal{E}$ and $B$ (delayed) refines $A$ then $B$ preserves the secrecy of $E$ from adversaries of type $A$ given inputs in $\mathcal{E}$.

Intuitively, this holds because the definition of preservation of secrecy refers to the existence of an output sequence revealing $E$, and the possibility of this existence is decreased by refinement.

### 7.5.2 Integrity

We now formalize that definition that a system preserves the integrity of a variable if there is no adversary such that at some point during the execution of the system in presence of the adversary, the variable has a value different from the ones it should have.

**Definition 7.28.** Suppose we are given a variable $v$ in a UMS $A$ and a set $E \subseteq \text{Exp}$ of acceptable expressions. We say that an adversary of type $A$ may eventually violate integrity of the variable $v$ in $A$ with respect to $E$, given inputs in $\mathcal{E} \subseteq \text{Events}$, if there exists $\text{adv} \in \text{Advers}_A(A)$ and an input sequence $\mathbf{1}$ whose multi-sets only contain elements in $\mathcal{E}$, such that at some point of the execution of $A_{\text{adv}}$ on the inputs $\mathbf{1}$, $v$ takes on a value not contained in $E$. Otherwise we say that $A$ preserves the integrity of $v$ from adversaries of type $A$ given inputs in $\mathcal{E}$. If $E = \text{Exp} \setminus \mathcal{K}^0$, we simply say that $A$ preserves the integrity of an attribute $a$ from adversaries of type $A$.

The idea of this definition is that $A$ preserves the integrity of $v$ if no adversary can make $v$ take on a value different from the ones it is supposed to have, in interaction with $A$. Intuitively, it is the “opposite” of secrecy, in the sense that secrecy prevents the flow of information from protected sources to untrusted recipients, while integrity prevents the flow of information in the other direction. Again, it is a relatively simple definition, which may, however, not prevent implicit flows of information.

For simplified verification, we give a sufficient condition for preservation of integrity in terms of influence sets.
Theorem 7.29 (Integrity Characterization). $A$ preserves the integrity of a variable $v$ with respect to a set $E \subseteq \text{Exp}$ of acceptable expressions against adversaries of type $A$ if $I_A(A,v) \subseteq E$.

To derive a preservation result for integrity, we need to refine with respect to the variable that is supposed to be protected.

Theorem 7.30 (Integrity Preservation). Suppose we are given UMSs $A$ and $B$. Suppose that $A$ preserves the integrity of $v$ with respect to a set $E \subseteq \text{Exp}$ of acceptable expressions from adversaries of type $A$ given inputs in $E$ and that the UMS $B (\emptyset, \{v\})$-refines the UMS $A$. Then $B$ preserves the integrity of $v$ with respect to $E$ from adversaries of type $A$ given inputs in $E$.

Intuitively, this holds because the definition of preservation of integrity refers to the existence of an execution corrupting $v$, and the possibility of this existence is decreased by refinement.

7.5.3 Authenticity

We now formalize our notion of message authenticity, which is supposed to secure the information on the message origin.

Definition 7.31. Suppose we are given variables $v$ and $o$ in a UMS $A$, where $o$ is supposed to store the origin of the message stored in $v$. We say that $A$ provides (message) authenticity of $v$ with respect to its origin $o$ from adversaries of type $A$ given inputs in $E \subseteq \text{Events}$ if for all $\text{adv} \in \text{Advers}_A(A)$ and each input sequence $\text{i}$ whose multi-sets only contain elements in $E$, at any point of the execution of $A_{\text{adv}, \text{i}}$ on the inputs $\text{i}$, $v$ takes on a value which appeared first within the execution $\text{outQu}_o$, of all output queues and link queues in $A$.

The idea of this definition is that $A$ provides authenticity of $v$ with respect to its origin $o$ if no adversary can make $v$ take on a value not originating from $o$, in interaction with $A$. Intuitively, it is the "opposite" of secrecy, in the sense that secrecy prevents the flow of information from protected sources to untrusted recipients, while integrity prevents the flow of information in the other direction. Again, it is a relatively simple definition, which may, however, not prevent implicit flows of information.

We can derive a preservation result for authenticity similar to that for integrity, which can be proved in the same way.

Theorem 7.32 (Authenticity Preservation). Suppose we are given UMSs $A$ and $B$. Suppose that $A$ provides authenticity of $v$ with respect to its origin $o$ from adversaries of type $A$ given inputs in $E$ and that the UMS $B$ is a $(\emptyset, \{v,o\} \cup \text{outQu}_A \cup \text{linkQu}_A)$-refinement of the UMS $A$, where $\text{outQu}_A = \{\text{outQu}_O : O \in \text{Comp}_A\}$ and $\text{linkQu}_A = \{\text{linkQu}_A(l) : l \in \text{Links}_A\}$. Then $B$ provides authenticity of $v$ with respect to its origin $o$ from adversaries of type $A$ given inputs in $E$.

We recall that message authenticity is closely related to data integrity (see Sect. 3.3.5).
7.5.4 Freshness

We define freshness of data for UMSs.

**Definition 7.33.** An atomic value data \( d \in \textbf{Data} \cup \textbf{Keys} \) in a UMS \( A \) is fresh within a component \( D \) contained in \( A \) if the value \( d \) appears in the specification \( A \) only within the specification of \( D \) (the scope of data in \( A \)).

To support this definition, we note that our formal model captures the fact that security-critical data such as keys and nonces are usually assumed to be independent; that is, that no equations should hold between them from which an adversary could derive information (such as \( K = K' + 1 \) for two different keys \( K, K' \in \textbf{Keys} \)). This follows from the fact that the algebra of expressions is the quotient of a free algebra under the equations given in Sect. 3.3.3, in particular, only equations that follow from these equations hold in \( \textbf{Exp} \). We will make this more precise.

**Definition 7.34.** An expression \( E \in \textbf{Exp} \) is independent of a set of expressions \( \mathcal{E} \subseteq \textbf{Exp} \) if \( E \) is not an element of the subalgebra of \( \textbf{Exp} \) generated by \( \mathcal{E} \).

We establish some simple facts about this definition with regard to atomic expressions (which we call those in \( \textbf{Data} \cup \textbf{Var} \cup \textbf{Keys} \)). For this, we first need an additional definition regarding such expressions.

**Definition 7.35.** An expression \( E \in \textbf{Data} \cup \textbf{Var} \cup \textbf{Keys} \) is a subexpression of an expression \( E' \in \textbf{Exp} \) if for each term \( t' \) over the operations in \( \textbf{Exp} \) that is evaluates to \( E' \) in \( \textbf{Exp} \), the unique term \( t \in \textbf{Data} \cup \textbf{Var} \cup \textbf{Keys} \) which evaluates to \( E \) is a subterm of \( t' \).

For example, \( E \in \textbf{Data} \) is a subexpression of \( \{E\}_K \): Any term \( t' \) which evaluates to \( \{E\}_K \) has \( E \) as a subterm, because none of the equations which define \( \textbf{Exp} \) in Sect. 3.3.3 would eliminate \( E \) from \( \{E\}_K \). \( E \in \textbf{Data} \) is not a subexpression of \( \text{head}(E' :: E) \) for \( E' \in \textbf{Data} \) with \( E' \neq E \), since \( \text{head}(E' :: E) = E' \) by definition in \( \textbf{Exp} \), and \( E \) is not a subterm of \( E' \) since both are atomic and \( E' \neq E \).

**Fact 7.36.** For any expression \( E \in \textbf{Data} \cup \textbf{Var} \cup \textbf{Keys} \) and any set of expressions \( \mathcal{E} \), \( E \) is independent of \( \mathcal{E} \) if there exists no expression \( E' \in \mathcal{E} \) such that \( E \) is a subexpression of \( E' \).

Note that the converse does not hold: For example, \( E \in \textbf{Data} \) is independent of and also a subexpression of \( \{E\}_K \).

**Fact 7.37.** For any expression \( E \in \textbf{Data} \cup \textbf{Var} \cup \textbf{Keys} \) and any set of expressions \( \mathcal{E} \subseteq \textbf{Data} \cup \textbf{Var} \cup \textbf{Keys} \), \( E \) is independent of \( \mathcal{E} \) if and only if \( E \notin \mathcal{E} \).
Thus it is sufficient to require of fresh values that they do not appear in the specification \( S \) outside their scope \( D \), as in Definition 7.33, because by Fact 7.37, they are then independent of all atomic values outside of \( D \) in \( S \).

Note also that a value data that is fresh within a component \( D \) in the UMS \( S \) appears as a subexpression in the trace of messages exchanged within \( S \) only after it has been sent out by \( D \) as a message argument:

**Fact 7.38.** Suppose we are given an atomic value data \( \in \text{Data} \cup \text{Keys} \) in a UMS \( A \) which is fresh within a component \( D \) contained in \( A \), an adversary type \( A \) which does not have access to \( D \), and an adversary adv of type \( A \). Then during any run \( e \in \text{Run}_{A_{\text{adv}}} \), if at any state \( S \) in \( e \) an output or link queue outside \( D \) contains data as a subexpressions, then there exists a state \( S' \) preceding \( S \) in \( e \) where outQu\(_D\) contains data as a subexpressions.

Here the restriction on the adversary is necessary: if the adversary had access to the component containing the fresh value, he could inject it into a link queue outside this component before it is sent out by the component.  

7.5.5 Secure Information Flow

We explain an alternative way of specifying secrecy- and integrity-like requirements, which gives protection also against partial flow of information, but can be more difficult to deal with, especially when dealing with encryption (for which further refinements of the notion are possible, but not needed in the following).

Given a set of message names \( H \subseteq \text{MsgNm} \) and a sequence \( m \) of event multi-sets, we write:

- \( m^H \) for the sequence of event multi-sets derived from those in \( m \) by deleting all events the message names of which are not in \( H \), and
- \( m_J \) for the sequence of event multi-sets derived from those in \( m \) by deleting all events the message names of which are in \( H \).

**Definition 7.39.** Given a UML Machine \( A \) and a set of high message names \( H \subseteq \text{MsgNm} \), we say that:

- \( A \) prevents down-flow with respect to \( H \) if for any two sequences \( i, j \) of event multi-sets and any two output sequences \( o \in [A](i) \) and \( p \in [A](j) \), \( 1^H = 1_H \) implies \( o_H = p_H \), and
- \( A \) prevents up-flow with respect to \( H \) if for any two sequences \( i, j \) of event multi-sets and any two output sequences \( o \in [A](i) \) and \( p \in [A](j) \), \( 1^H = j^H \) implies \( o^H = p^H \).

Intuitively, to prevent down-flow means that outputting a non-high (or low) message does not depend on high inputs (this can be seen as a secrecy requirement for messages marked as high). Conversely, to prevent up-flow
means that outputting a high value does not depend on low inputs (this can be seen as an integrity requirement for messages marked as high).

This notion of secure information flow is a generalization of the original notion of non-interference for deterministic systems in [GM82] to non-deterministic systems. Many such generalizations have been proposed in the literature; the current one is motivated by the fact that it should be preserved under refinement (essentially, a UML Machine prevents down-flow if and only if each refinement to a deterministic UML Machine satisfies non-interference).

Recall that we use non-determinism for underspecification (rather than for a functional kind of non-determinism that should still exist on the implementation level). Thus a non-deterministic specification of a system that may output any value to the untrusted environment is not seen to prevent down-flow, because a legal implementation of it may output low messages that depend on high inputs. If instead one wanted to model “functional” (or possibilistic) non-determinism, the current definition might turn out to be rather strong.

We show that this formulation of secure information flow is also preserved under refinement, which again is possible since non-determinism is supposed to represent underspecification, while security-providing non-determinism is modeled through the generation of random values (such as keys). Note that computers currently in use are in fact deterministic, apart from special components that produce such random values.

**Theorem 7.40.** Suppose that the UML Machine $A$ prevents down-flow (resp. up-flow) with respect to the set $H \subseteq \text{MsgNm}$ and that the UML Machine $B$ refines $A$. Then $B$ prevents down-flow (resp. up-flow) with respect to $H$.

Note that secure information flow is not preserved under delayed refinement, since an introduced time delay may be used to leak information.

### 7.6 Notes

A notion of “interacting ASMs” has been introduced in [MIB98, Sch98], but without the possibility to construct ASM specifications in the same way as using our UMS.

The idea of extending ASMs with input and output queues is similar to that of algebraic state machines [BW00, Jir03a].

There has been extensive research in using formal models to verify secure systems. Their main areas of application in this domain include security policies and security protocols. Early influential work was given in [DY83, MCF87, Mea91], and [BAN89] (introducing the so-called “BAN-logic”). The process algebra CSP has been employed successfully, for example in [Low96, LR97, Low98, Low99, HL01]. [AG99] introduces the spi calculus. The inductive method of proving protocols correct using a mechanical proof assistant is explained for example in [Pau98b]. There has been some work
using ASMs reported in [BR97, BR98]. Work on tool-supported verification of cryptographic protocols is reported in [Eck98]. In [Lot00], threat scenarios are used to formally develop secure systems using the stream-based specification language Focus [BS01]. [KAH99, Hei01] use the Software Cost Reduction to set. [WW01] presents security extensions to the CASE tool AutoFocus.

[Mea95, Aba00, Mea00] give short overviews and point out open problems. For an overview of the work on verifying security protocols with a focus on the process algebra CSP, see [RSG+01].

Notions of secrecy similar to ours have been used for example in [DY83, CGG00, SV00].

[Jür01g] extends Focus by cryptographic operations including symmetric and asymmetric encryption and signing (and proves the preservation by the refinement result reported here). [Jür01b] examines composability issues of the secrecy property considered here in that setting. [Jür00] considers secure information flow in a similar model.

[Sch96] gives a confidentiality property preserved under refinement. However, cryptographic primitives are not considered and it is pointed out that their treatment may be less straightforward.

[HPSS01] gives a necessary and sufficient condition for a notion of confidentiality to be preserved under refinement in a probabilistic extension of CSP.

[Lam73] drew attention to covert channels; this initiated early influential work on secure information flow in [GM82, GM84]. An overview of secure information flow (and other formal security models) can be found in [McL94].

For a discussion on the refinement of secure information flow properties, see [GCS91, Mea92, McL94, McL96]. [RWW94] avoids the “refinement problem” by giving a security property that requires systems to appear deterministic to the untrusted environment. Special refinement operators that preserve information flow security are considered for example in [Man01].

The problem of how far formal models of cryptography are faithful to computational models is considered in [AJ00, AJ01]. Similar investigations regarding failure probabilities in the setting of safety-critical systems are presented in [Jür01a].

7.7 Discussion

After recalling some concepts about UML Machines, we introduced some extensions, UML Machines and UML Machine Systems, to model the interaction between a system and its environment and to build up UML Machine specifications in a modular way. In the following chapters, we will use them to define a formal semantics for a (restricted and simplified) fragment of UML. Also, we defined notions of refinement and rely-guarantee specifications for UML Machines (where rely-guarantee specifications were showed to be pre-
served under refinement), allowing stepwise and modular development, whose
use will also be shown in later examples.

We explained how we use UML Machines to specify security-critical sys-
tems, exploiting the fact that UML Machine Systems are designed to allow
the treatment of external influences on the system beyond the planned inter-
action. This allows a rather natural modeling of potential adversary behavior
that specifies which parts of the system are assumed to be accessible to an
adversary in which way. The adversary behavior is again modeled by a class
of UML Machines with the specified capabilities. This gives us an evaluation
of the system's security in a natural way, by considering the joint execution
of the system with any UML Machine in this class. Also, the security prop-
erties considered, secrecy and integrity, could thus be formalized intuitively.
Verifying the properties in this formulation is rather indirect, so we gave alter-
native ways of defining these security properties that do not refer to particular
adversaries and proved them to be equivalent to the earlier formulations. Ad-
ditionally, we defined notions of secure information flow. We will use this
approach in examples of security analyses (using UML Machines as a formal
semantics for a fragment of UML) in later chapters.

We addressed one of the issues for the formal development of security-
critical systems raised in the introduction, namely the so-called refinement
problem: we showed secrecy, integrity, and information flow to be preserved
under refinement. The goal is to be able to use stepwise development of
security-critical systems without having to redo security proofs at each re-
finement step, and to raise confidence that an implementation conforming to
the verified specification (which is necessarily a refinement of it) is also secure.
At the same time, the security properties considered are defined in a standard
way, as well as the refinement relation (rather than giving definitions specially
tailored to avoid the refinement problem).
Formal Systems Development with UML

We use UML Machines and UML Machine Systems to give a formal semantics for a (restricted and simplified) part of UML to enable advanced tool support for UMLsec. It allows one to use subsystems in a specific way to group together several kinds of diagrams, giving a simplified formal semantics of a restricted version of UML subsystems and their interactions. Objects, and more generally system components, can communicate by exchanging messages with parameters, which can be used in the subsequent execution. The behavior of actions and activities can be modeled explicitly. Since our semantics builds on UML Machine Systems, it allows us to make use of the treatment of security-critical systems in Sect. 7.5 to evaluate UML specifications for security aspects. We give consistency conditions for different diagrams in a UML specification. We define notions of refinement and behavioral equivalence, and investigate structural properties (such as substitutivity). Finally we consider rely-guarantee properties for UML specifications and their structural properties.

The proofs for statements in this chapter are given in Appendix C.

8.1 Formal Semantics for a Fragment of UML

The semantics of UML is given only in prose form [Obj03], leaving room for ambiguities (a problem especially when providing tool support or trying to establish behavioral properties of UML specifications). To reason about system behavior in a precise way, however, we need a precise (mathematical) semantics for the behavioral model elements of UML.

There has been a considerable amount of work to generally provide a formal semantics for UML (see Sect. 8.3). To use UML for critical systems development, we need a semantics that, more specifically, supports different views of a system, including its logical structure, its physical environment (including attack scenarios), and its behavior (focussing on interaction rather
than computation). At the same time, the semantics should be sufficiently simple to allow its use for manual, and eventually mechanical, reasoning.

We provide a formal semantics for a (restricted and simplified) part of UML that allows one to use a restricted version of UML subsystems to group together several diagrams. It is formulated using UML Machines. The statecharts semantics builds on part of the statechart semantics from [BCR00]. The formal semantics for subsystems incorporates the formal semantics of the diagrams contained in a subsystem. Actions and internal activities are modeled explicitly (rather than treating them as atomic given events), and they are also actually executed (for example, assigning new values to attributes). In particular, objects, and more generally system components, can communicate by exchanging messages with parameters, which can be used in the subsequent execution. In particular, we show how to compose our subsystems by including them in other subsystems. To our knowledge, there has not been a formal semantics of UML subsystems and their interactions published in the literature so far.

The aim is to prepare the ground for tool support based on this precise semantics.

Note that we only consider a restricted and simplified fragment of the UML syntax. In particular, the notion of subsystem considered here is restricted, for example, in the kinds and numbers of diagrams that may be contained. Also, following [KW01, p. 15], we do not model the creation and deletion of objects explicitly. A sufficient number of required objects are assumed to exist at the start of the execution; the activation of objects is controlled by the activity diagram in the subsystem. An object that reaches a final state within its top state is terminated (and may be reactivated). It would be interesting further work to examine the possibility of giving a more sophisticated account of object creation (note, however, that [LP99] argues that the creation and destruction of objects could lead to unbounded behaviors that would be impossible to verify automatically with a model-checker). This was not attempted here in order to keep the treatment as simple as possible to focus on our use of the formal semantics for security analysis. Furthermore, in our approach, we identify the objects in the runtime system with UML objects. We thus aim to provide an executable semantics for UML models to allow simulation. We feel that simulatability of a model can be of value for use in industrial practice, because it may assist understanding (although non-executable specifications also have their value), see [Rum02] for a supporting discussion. Also, our main goal for providing a formal semantics is the use for security analysis of UML models, and some of the security properties considered later refer to an execution trace of the model. However, code generation of the models is not our goal here, we do not aim to propose a visual programming language, and the models one may construct for the purpose of security analysis do not necessarily closely reflect the detailed design of a later implementation of the system.
One should also note that the semantics does not attempt to support simultaneous modeling of several overlapping aspects of the system behavior in different parts of the UML model. That is, in our approach, at any one time the behavior of a given thread of an object is represented by only one diagram. For example, our semantics enforces that different statecharts contained in a UML specification are always mapped to disjoint state sets of distinct sub-state machines of the overall semantics. This way our approach sidesteps questions that would arise from having different parts of a UML specification model the same part of the system behavior, which are interesting but beyond the scope of the current treatment.

However, the semantics provides a possibility to check whether such overlapping aspects are consistent; by creating two separate models which are identical apart from the non-overlapping parts and then establishing whether they are behaviorally equivalent using the corresponding definition in Sect. 8.2.1.

There are further simplifications whose explanation requires more detailed knowledge of the diagrams and which therefore will be explained in detail in the respective sections.

There are several reasons for the simplifications. The more immediately compelling, pragmatic reason is the space that the formal semantics could usefully be allowed to occupy in the present work, given that it is only used as a means to an end. A complete formalization of UML, the prose description of which in [Obj06] occupies more than 400 pages, would seem hard to achieve under these circumstances. Note also that our emphasis is on providing a system model that can be analyzed for security requirements. Therefore, our treatment is tailored toward this main goal, which may require compromises with regard to ease of implementability of the models we consider here.

A more principled consideration would be the question whether it would be at all feasible to provide tool support of the kind that is the long-term goal of the formalizations presented here by referring to a complete formalization of UML. It may in principle be possible to provide a complete formalization of UML and to use it to reason about (say, behavioral) properties of UML specifications using a computer-aided deduction system (which could in principle also offer assistance for the so-called variation points, where the UML definition leaves aspects underspecified). However, evaluating UML specifications in this way would be relatively expensive in practice, due to the relatively high amount of necessary human interaction. As an alternative, one could use model checking technology to evaluate UML specifications in a more automatic way. However, due to the inherent complexity of model checking, it would seem at present that this approach would be considerably facilitated when using only a considerably simplified fragment of the UML definition.

For these reasons it may seem reasonable to concentrate on a simplified fragment of UML. It could still be argued that developers who may already know UML could be taught relatively easily to use only a fragment of UML, rather than having to be taught a completely different formal method. Also, this could be seen to be in accordance with the view that UML might be
seen as a "family of languages" [Coo00], each for a specific purpose, such as tool-supported validation, but sharing a common core. To demonstrate that our choice of a subset of UML is reasonable for our present needs and our semantics of sufficient interest, we presented several case studies in Chap. 5 which only make use of those parts of UML whose semantics is defined here. Some of these originate from an industrial context and are therefore more realistic in size and complexity than some of the protocols often considered in the academic literature; thus they may demonstrate that the fragment of UML used in our work seems to be sufficient for our present needs.

The main goal of the semantics presented here is to prepare the ground for constructing tool support for our proposed use of UML for secure systems design. In particular, the executable semantics using UML machines was designed with a view toward facilitating the construction of tool support, rather than ease of human consumption (although it is not clear if there could actually be a formal semantics for a substantial part of UML that would be easily read and understood, given its complexity).

Since our semantics uses UML Machines and UML Machine Systems (UMSs) defined in Sects. 7.1 and 7.2, it allows us to make use of the treatment of security-critical systems in Sect. 7.5 (see Chap. 4). In particular, UML specifications can be evaluated using the attacker model from Sect. 7.5, which incorporates the possible attacker behaviors, to find vulnerabilities. For the trivial kind of adversary who is not able to access any part of the system, our approach gives us the usual (restricted and simplified) UML semantics without security considerations.

*Diagrams*

Our formal semantics includes restricted and simplified versions of the following kinds of diagrams: static structure diagrams (which are class or object diagrams that may also contain subsystems), statechart diagrams, sequence diagrams, activity diagrams, deployment diagrams, and subsystems.

The semantics for statecharts presented here is based on part of [BCR00], which, however, had to be extended to incorporate the features mentioned above.

*Consistency*

We give some conditions for consistency between different kinds of diagram in a UML specification (such as static versus behavioral diagrams).

*Equivalence*

We define a notion of behavioral equivalence between UML specifications. This can be used for example to verify the consistency of two of our kinds of subsystem specifications that are supposed to describe the same behavior, one of which uses statecharts to specify object behavior, and the other sequence diagrams.
8.1 Formal Semantics for a Fragment of UML

Refinement

In UML, refinement denotes a certain kind of dependency relation between model elements [Obj03, p. 2-18]. There is no constraint on the semantic relationship between the model elements (also in the heuristics for state machine refinement on [Obj03, p. 2-172]).

When trying to establish system properties, behavioral conformance of refinement can help to save effort (properties may be easier to establish at a more abstract level; preservation by refinement means that this is in fact sufficient).

We aim for a trade-off between flexibility of a refinement relation and the gain from establishing that a specification refines another by considering two kinds of refinement for UML specifications. The first of these, property refinement, provides full behavioral conformance, and thus preserves all safety properties. The second, interface refinement, allows some degree of control over the extent to which the structure and behavior of the system is preserved. Both were inspired by notions of refinement in [BS01]. For both kinds of refinement, we define a relaxation, called delayed refinement, that allows time delays to be introduced during refinement.

Rely-Guarantee Specifications

Finally, we consider rely-guarantee specifications (following [BS01]) in the setting of UML and prove some results regarding them.

8.1.1 General Concepts

We consider a restricted and simplified fragment of UML to simplify the treatment and because it is sufficient for our needs. We believe that extending the work to include some (but not necessarily all) other aspects should be possible in principle. Note, however, that this would cause an increase in complexity and therefore possibly an increased challenge when trying to provide tool support. It remains a topic of further research to determine to what an extent the UML definition could feasibly and usefully be given a more complete formalization. Note that there are some aspects that are omitted simply because they are not used in the sequel, such as associations in class diagrams, and which one should be able to add in a relatively straightforward fashion. Generally, for our present needs it is sufficient to remain on the instance level, as for other non-functional requirements [Wat02, sl. 4]. For example, [LGS01] points out some security problems arising in CORBAsec from its emphasis on types rather than instances. It would be interesting to try to add type-level concepts to our approach, but we will not need to.

For our intended use in security analysis, we only need the abstract syntax of the static modeling elements given below, while for the behavioral diagrams, we need a formal semantics. Note, however, that the abstract syntax of the
structural diagrams is needed to define the formal semantics of subsystems containing them, for example, because the semantics depends on it.

The UML specification document [Obj03] gives the abstract syntax of the UML notation using a fragment of the UML notation. The logical cycle arising from this could be avoided by giving a separate definition of the abstract syntax of that fragment. For simplicity, we define the abstract syntax of the fragment of UML we use entirely in basic set-theoretical terms.

In our approach, we view an object or component as an entity characterized by a unique name, which may have associated information such as its attributes and their values which may change during its execution (which may be specified as a statechart). Thus we identify the objects or components in the runtime system with UML objects or components. Thereby, we aim to provide a simplified executable semantics for a restricted kind of UML models. Note that in more general use of UML, the relation between UML objects and system objects may not be functional in either direction.

Objects, and more generally system components, can communicate by exchanging messages. These consist of the message name and possibly arguments to the message (which are assumed to be elements of a set \( \text{Exp} \); in our later use this will be the set defined in Sect. 3.3.3). Message names may be prefixed with object or subsystem instance names (analogous to the names of UML Machines or UMLSubs, see Sect. 7.1).

Messages can be synchronous (meaning that the sender of the message passes the thread of control to the receiver and receives it back together with the return message) or asynchronous (meaning that the thread of control is split in two, one each for the sender and the receiver, unless they already had separate threads of control). Exchanging a synchronous (resp. asynchronous) message is called “calling an operation” (resp. “sending a signal”). Accordingly, we partitioned (in Sect. 7.2) the set of message names \( \text{MsgNm} \) into sets of operation names \( \text{Op} \), signal names \( \text{Sig} \), and return message names \( \text{Ret} \).

Note that the UML specification in some parts makes a distinction between the term “Stimulus” and the term “Message”, which is “a specification of a Stimulus” [Obj03, 3.63.1]. However, in other places distinction is again removed or blurred:

- Firstly, in the case of the usage of “message name”: According to [Obj03, 3.72.2.5], a message name is “the name of the Operation to be applied to the receiver, or the Signal that is sent to the receiver”.
- Secondly, the glossary defines:
  - receive (a message): The handling of a stimulus passed from a sender instance.
  - send (a message): The passing of a stimulus from a sender instance to a receiver instance.

To avoid confusion, we do not use the term “Stimulus” at all, but use the term “message” (or “message instance” for emphasis) to denote the actual
message that is exchanged (as in Sect. 7.2), and “message specification” for the specification of a message.

An event is “the specification of a significant occurrence that has a location in time and space” [Obj03, p. Glos.-7]. Here we consider the events arising from the reception of an operation call or a signal. Accordingly, we defined the set Events to consist of messages $msg^{nd}(exp_1, \ldots, exp_n)$ for $msg \in \text{MsgNm}$, $snd = n_1 \vdots \ldots \vdots n_k$ with $n_1, \ldots, n_k \in \text{UMNames}$, and $exp_i \in \text{Exp}$ (see Sect. 7.1). In our model, every object or subsystem instance $O$ has associated multi-sets $\text{inQu}_O$ and $\text{outQu}_O$ (event queues). As explained in detail in Sect. 7.2, our formal semantics using UMLSs models sending a message $msg = op(exp_1, \ldots, exp_n) \in \text{Events}$ from an object or subsystem instance $S$ to an object or subsystem instance $R$ as follows:

1. $S$ places the message instance $R.msg$ into its multi-set $\text{outQu}_S$.
2. The “virtual machine” $\text{Exec}(A)$ for an ASM system $A$ defined in Sect. 7.2 distributes the message instances from output queues to the intended input queues (while removing the message head); in particular, $R.msg$ is removed from $\text{outQu}_S$ and $msg$ added to $\text{inQu}_R$.
3. $R$ removes $msg$ from its input queue and processes its content.

In the case of operation calls, we also need to keep track of the sender to allow sending return signals. As defined in Sect. 7.2, this is done by associating the sender name as a superscript of the name of a message instance.

This way of modeling communication allows for a relatively flexible treatment; for example, we can modify the behavior of the scheduler to take account of knowledge of the underlying communication layer (for example, regarding security (see Sect. 7.2) or performance issues).

Note that messages with the same name and possibly the same arguments can appear several times at different places in a UML specification. As mentioned above, our semantics does not attempt to support overlapping model parts. Whenever two such messages are sent during a given model execution, they are interpreted as two different message instances created by distinct system events (the corresponding method called by the calling objects), and they are later also consumed by distinct system events (the events in the UML sense at the called objects). Thus, whenever a message instance is sent in a UML model, our semantics models this by adding a new element to the $\text{outQu}$ multi-set, as explained above. This directly implies that, conversely, for any element of an input or output queue, there is a unique occurrence of this message instance in the UML model from which it originates. Thus, in our approach, at each point of a given execution of a system, the same message instance in the running system is only represented once in the UML diagrams, and hence only once in our semantics. More concretely, each time an expression $\text{call}(msg)$ or $\text{send}(msg)$ appears as an action in a statechart diagram, or a message $msg$ appears at a connection in a sequence diagram that is “executed”, it is interpreted as a different message instance (which may happen to have the same name and the same arguments as a previous
message instance). Thus, this message instance is newly added to the output queue using the macro `tooutQu()`, as defined in Sects. 8.1.3 and 8.1.4. Since these are the only ways message instances are introduced during the execution of a model, this ensures that a message instance appears only once in the semantics, by definition.¹ This observation is presented in more detail in Fact 8.1.

The mechanism for handing on the message instances explained above is performed locally at the subsystems and objects; where it will be sent depends on its place and on the relative addressing of the recipient. For example, assume we have subsystem instances $S$ (resp. $S'$) each with objects $S$ and $R$. Then the object $S.S$ (resp. $S'.S$) may each be specified to send the message instance $R.msg$ (for example, in two different statecharts contained in $S$ (resp. $S'$)). Then the message instance $msg$ sent by $S.S$ will be delivered to $S.R$, while the message instance $msg$ sent by $S'.S$ will be delivered to $S'.R$.

We model a synchronous operation call by sending two asynchronous signals – the message and its return value. By imposing restrictions on statecharts and sequence diagrams (see the respective sections) we can model the passing of control implicitly. The semantics does keep track of the sender of a synchronous operation, so that the return message can be delivered.

Note that an object may receive several synchronous messages calling the same operation $op$ before sending back a corresponding return value. To enable sending back the return value to the sender, the statechart and sequence diagram semantics include last-in-first-out buffers containing the names of the senders of the message calls (assuming that the calls and their returns are “well-bracketed” in a sense detailed below). On the level of the semantics using UMLs, the sender names are attached to the messages sent (see Sect. 7.2). When return messages are sent out, the recipients of these messages are taken from that buffer.

A cycle of invocations (when the client’s statechart invokes another object’s operation, with the called object calling others, and so on) that leads back to the same thread of the same object instance is not permitted, and an attempt to execute it will result in deadlock (similar to the treatment in [HG97]). This restriction seems to be inherent in the current UML run-to-completion semantics, as pointed out in [TS02], which also offers a solution for this problem (which cannot be incorporated here because of space restrictions). Details and discussions are given in Sects. 8.1.3 and 8.1.4.

Note also that there is only one input buffer (and one output buffer) for a given object or component. This buffer may be accessed in various ways – for example, concurrent substates of a statechart diagram read from the same input buffer and write to the same output buffer. That this happens

¹ This might be compared to abstraction in the lambda-calculus, where one may have two appearances of the same variable on the syntax level, which, however, evaluate to different entities on the semantics level.
consistently is ensured by the semantics (for example, concurrent substates are executed interleavingly).

The UML semantics includes some semantic variation points to allow adjusting the semantics to a given application domain. For example, the order of dequesing events at an object or component is not defined. Similarly, in the case of statechart diagrams, the order in which enabled transitions are executed is left open (except that transitions with innermost source states have highest priority, see Sect. 8.1.3 for an explanation of these concepts). In both cases, we use the non-deterministic choice operator of UML Machines to determine the order. The intention here is not to prejudice any view over what the UML specification document prescribes; by using the non-deterministic choice operator, it is made sure that in our use of the formal semantics for formulating our concepts regarding security, we do not make use of any additional properties in our semantics that are not specified in the UML specification document regarding this issue (such as a partial specification of order), which might lead to problems when using our ideas with a different semantics. When implementing this semantics in the form of a tool, either this non-determinism could be preserved by giving a probabilistic interpretation, in order to keep designers using the tool from making use of any specified order; or, alternatively, one could define the non-deterministic choice operator by an operator determining any kind of choice based on the situation at hand. Thus, by using the non-deterministic choice to determine the order, we are covering all possibilities of choosing an order, so that the results based on our definitions will automatically cover all such more detailed elaborations (following a standard idea in refinement-based methods).

Note also that we follow the UML specification in that we do not make any fairness assumptions on the input queue of an object; thus dispatching an event can be delayed indefinitely provided the event queue contains more than one event at each point during the execution (this could also be changed easily; for example, by taking the event queue to be a list).

Objects may have attributes, which are variables that are local to the object and whose names are given in the set \( \text{Attribute} \subseteq \text{Var} \cup \text{Keys} \cup \text{Data} \) (where the names in \( \text{Keys} \cup \text{Data} \) denote constant attributes with the same value). We will not consider situations where changing attributes may lead to unexpected side-effects (such as changes to object references). We assume that attribute names are only used for attributes and that the attributes of an object can only be changed by the object itself (and only indirectly by other objects, namely through sending messages).

Each element in \( \text{Var} \) has an associated UML Machine variable with the same name which represents it on the semantics level (by storing assigned values). Initially, all variables are set to the value \( \text{undef} \).

An action is “the specification of an executable statement that forms an abstraction of a computational procedure. An action typically results in a change in the state of the system” [Obj03, p. Glos-2]. We consider the actions sending a message to an object or component and modifying the value of an
attribute. Thus actions and events are related in that the execution of an action at one object may or may not cause the occurrence of an event at another object. We write Action for the set of actions which are expressions of the following forms:

Call action: call\((op(a_1,\ldots,a_n))\) for an n-ary operation \(op \in Op\) and expressions \(a_i \in Exp\) (called the arguments of \(op\)).
Send action: send\((sig(a_1,\ldots,a_n))\) for an n-ary signal \(sig \in Sig\) and argument \(a_i \in Exp\).
Return action: send\((return_{op}(a))\) for an operation \(op \in Op\) with return value \(a \in Exp\).
Assignment: \(att := exp\) where \(att \in Attribute\) is an attribute and \(exp\) is a term evaluating to an expression in \(Exp\).
Void action: \(nil\).

Before we define the semantics of these actions in terms of a UML Machine, we need an additional binary function \(sender()\) in its vocabulary. Given a parameter \(S\) (such as the state in a statechart diagram) and a synchronous operation \(op\), the function \(sender_S(op)\) returns the list of previous senders of \(op\) which are needed when sending back return messages. The parameter \(S\) allows parallel processing of several operations with the same name. We define the following syntactic shortcut for sending back return messages, where \(A\) is a UML Machine, \(op \in Op\), and \(S\) is a parameter.

\[
\text{retMsg}_{A,S}(op) \equiv \begin{cases} \text{tooutQu}_A(\\{\text{head}(\text{sender}(S, op))\},\text{return}_{op}(\text{args})\}) \quad \text{if} \quad \text{sender}(S, op) \neq [] \\
\text{tail}(\text{sender}(S, op)) \quad \text{else} 
\end{cases}
\]

Then for any action \(a\) executed at an object or component \(O\) and for a parameter \(S\), we define the expression \(ActionRule_S(a)\), where \(op \in Op\) and \(msg \in Sig\):

Call action: \(ActionRule_S(\text{call}(op(\text{args}))) \equiv \text{tooutQu}_O(\\{\text{op}(\text{args})\})\)
Send action: \(ActionRule_S(\text{send}(msg(\text{args}))) \equiv \text{tooutQu}_O(\\{msg(\text{args})\})\)
Return action: \(ActionRule_S(\text{send}(\text{return}_{op}(\text{args}))) \equiv \text{retMsg}_{O,S}(op)\)
Assignment: \(ActionRule_S(\text{att} := \text{exp}) \equiv \text{att} := \text{exp}\)
Void action: \(ActionRule_S(\text{nil}) \equiv \text{skip}\)

Note that in our usage of these rules to define a formal semantics for a simplified fragment of UML statecharts and sequence diagrams below, the assumption is that whenever a rule \(ActionRule_S(a)\) for a call or send action \(a\) is fired during a single execution of a given UML specification, a new message is created and added to the relevant output queue (although messages with the same name and the same arguments may already be in use during this execution). This is also realized by the rules defined above. Note also that these rules are the only way that messages are created in the execution of a UML specification. Thus, during a single execution of a particular specification,
8.1 Formal Semantics for a Fragment of UML

Each message is only created once, and only at one particular location of the specification (and note also that in the definition of a behavior of UMSs in terms of UML Machines, no messages are newly created or duplicated, but only transferred between different input, output, and link queues). The only other place a message is referred to in a UML specification is the event in the statechart or sequence diagram of its recipient where it is consumed. Again, in each given execution of a specification, there can be at most one such event, because when a message is consumed, it is removed from the queues of a specification. This is a restriction in so far as diagrams cannot be used in a way that permits “overlapping” in time (more details about this are given in Fact 8.1). It is a simplification for us in that related questions of consistency within a single specification are avoided.

We fix a set **Activity** of **activity names** such that each activity has an associated UML Machine representing the activities in a UML specification. Each such UML Machine $A$ has a Boolean $finished_A \in out$ as one of its output values (set to $true$ by $A$ when it is finished). These UML Machines may themselves be given as the formal semantics of UML diagrams defined in the following sections, or they may be defined directly using UML Machine rules (for example, an assignment to an attribute). We assume that there is an activity $Nil \in Activity$ representing absence of activity, whose interactive ASM has the following rule:

**Rule Exec( Nil ) :**

$$finished_{Nic} := true$$

We assume a set **Stereotypes** of **stereotype** names to be given, as well as a function mapping each stereotype to its set of associated tags and its constraint. In a UML diagram, stereotypes are written in double angle brackets « ». For examples see Chap. 4 where we present the stereotypes used for the extension UMLsec of UML for secure systems development.

The set of **Boolean expressions** $BoolExp$ is the set of first-order logical formulas with equality statements between elements of $Exp$. They will be used for example as guards in UML diagrams. Here we may write $x \neq y$ as short for $\neg x = y$.

The set of **Boolean expressions** $BoolExp$ is the set of first-order logical formulas with equality statements between elements of $Exp$ as atomic formulas. They will be used for example as guards in UML diagrams.

In the following sections, we will define the abstract syntax of the various UML diagrams considered here using mathematical notation, and then give a precise semantics of the modeled system behavior for each of the diagram kinds using UML Machines. In Sect. 8.1.7, we explain how to use the different kinds of diagram in the context of a UML system specification, and we put the formal semantics of the various diagram types together to form one formal semantics for a UML system specification.
8.1.2 Class Diagrams

We define the abstract syntax for class and object diagrams.

A message specification \( O = (oname, args, otype) \) is given by:

- an operation or signal name \( oname \in \text{Op} \cup \text{Sig} \),
- a set \( \text{args} \) of arguments of the form \( A = (argname, argtype) \) where \( argname \) is the argument name and \( argtype \) its type, and
- the type \( otype \) of the return value.

Note that the set of arguments may be empty, and that the return type may be the empty type \( \emptyset \) denoting absence of a return value. We assume the “default” types \( \text{Exp} \) for arguments and \( \emptyset \) for return values, which may be omitted to increase readability.

An object \( O = (oname, cname, stereo, aspec, mspec, int) \) is given by:

- an object name \( oname \),
- a class name \( cname \),
- a set \( stereo \subseteq \text{Stereotypes} \) of stereotypes,
- a set of attribute specifications \( aspec \) of the form \( A = (aname, gtype) \) where \( aname \in \text{Attribute} \) is the attribute name and \( gtype \) the attribute type,
- a set of message specifications \( mspec \), and
- a set of interfaces \( int \) of the form \( I = (iname, mspec) \) where \( iname \) is the interface name and \( mspec \) a set of message specifications, such that messages with the same name in different interfaces have the same type.

As a convention, we denote constant attributes by underlining the attribute type. We use the further convention that constant attributes are named by their value (thus we can leave out the explicit assignment of initial values to constant attributes). For example, \( (K, \text{Keys}) \) specifies a constant of value \( K \in \text{Keys} \) (this is written \( K : \text{Keys} \) in concrete syntax).

A class is an “object” (as defined above) \( C = (oname, cname, stereo, aspec, mspec, int) \) where \( oname \) is the empty string.

A dependency is a tuple \( (dep, indep, int, stereo) \) consisting of:

- the names \( dep \) of the dependent and \( indep \) of the independent class (signifying that \( dep \) depends on \( indep \)),
- an interface name \( int \) (the interface of the class \( indep \) through which instances of \( dep \) accesses instances of \( indep \); if the access is direct this field contains the name of the independent class), and
- a stereotype \( stereo \in \{\text{call}, \text{send}\} \).

A class diagram \( D = (\text{Classes}(D), \text{Dep}(D)) \) is given by a set \( \text{Classes}(D) \) of classes and a set \( \text{Dep}(D) \) of dependencies.\(^2\) We require that the names of the classes are mutually distinct.

\(^2\) Again, we omit modeling elements such as associations, specific notation for active objects, other stereotypes, and other modeling elements only because they are not used in the following; they can be added without complication.
8.1 Formal Semantics for a Fragment of UML

An object diagram \( O = (\text{Objects}(D), \text{Dep}(D)) \) is given by a set \( \text{Objects}(D) \)
of objects and a set \( \text{Dep}(D) \) of dependencies, such that object specifications from the
same class coincide up to the object name, and the names of different objects are mutually distinct. Note that
on the level of abstract syntax, it is sufficient to specify dependencies between classes rather than objects,
although on the level of concrete syntax of object diagrams, dependencies are
drawn between objects.

Note that in UML, class (resp. object) diagrams may contain subsystems
(resp. subsystem instances) rather than classes (resp. objects). To avoid confusion, we use the term static structure diagram (see Sect. 8.1.7) in this case,
following a suggestion in [Obj03, p. 3-34].

8.1.3 Statechart Diagrams

Usually statecharts are used to describe the behavior of classes of objects
rather than single objects, for simplicity (although this has to be handled
with care because often subclassing does not preserve behavior, so that a
statechart may not actually give the behavior of an object of the class it is
associated with, if the object is in a subclass with different behavior). When
assigning a meaning to a UML specification, one eventually has to associate
statecharts with objects, because objects, rather than classes, execute the
behavior modeled by statecharts. To simplify the treatment, in the following
we assume that this step to the instance level has already been made, and
associate statecharts with objects already. From the perspective of our usage
of UML, this suggests itself also because the difficulties that we will focus on
(such as security properties of a system) are typically not closely related to
the fact that there are a high number of objects in a given class, but rather
they are related to the behavior of a few objects or components that has
to be analyzed rather carefully. This is similar to the situation with other
non-functional requirements, which are usually analyzed on the instance level
[Wat02]. Therefore, it seems to make sense to explicitly model these instances,
and thus to remain on the instance level, for our purposes (and let the user of
our approach determine how far objects in the same class are supposed to
have the same behavior). It would be interesting to try to add type-level
ccepts to our approach, but we will not need to. Note that statecharts may
also define other model elements (for example, we will later use statecharts to
define activities), rather than complete object behavior.

We extend the formal semantics for statecharts from part of [BCR00] in
the following respects:

- Events can carry parameters. This is also one of the major differences from
Harel's statecharts [Obj03, p. 2-174].
- We incorporate a dispatching mechanism for events and the handling of
actions.
To keep the treatment accessible, we give the formal semantics for statecharts that are simplified as follows, which is sufficient for our present needs:

- Events cannot be deferred.
- There are no history states.
- Transitions may not cross boundaries within or across composite states; transitions from composite states must be completion transitions.
- A cycle of invocations (when the client’s statechart invokes another object’s operation, with the called object calling others, and so on) that leads back to the same thread of the same object instance is not permitted, and an attempt to execute it will result in deadlock (similar to the treatment in [HG97]). An example of such a situation is recursion. This restriction seems to be inherent in the current UML run-to-completion semantics, as pointed out in [TS02], which also offers a solution for this problem (which cannot be incorporated here because of space restrictions). More details are given below.

Also, [Cav00] explains how fork-join and junction pseudostates, and submachine, stub, and synth states, can be defined using the constructs treated here; we therefore omit their treatment, as well as that of time and change events.

**Abstract Syntax of Statechart Diagrams**

We define the abstract syntax of statechart diagrams.

A *statechart diagram* $D = (\text{Object}_D, \text{States}_D, \text{Top}_D, \text{Transitions}_D)$ is given by an object name $\text{Object}_D$, a (finite) set of states $\text{States}_D$, the top state $\text{Top}_D \in \text{States}_D$, and a set $\text{Transitions}_D$ of transitions, defined in the following. We use $\text{Object}_D$ to provide the context of a statechart diagram which links a state machine to another model element [Obj03] (this is usually not part of the concrete statechart syntax but needed when giving a formal semantics to complete specifications as in Sect. 8.1.7).

$\text{States}_D$ is a set that is disjointly partitioned into the sets $\text{Initial}_D$ of initial states in $D$, $\text{Final}_D$ of final states, $\text{Simple}_D$ of simple states, $\text{Conc}_D$ of concurrent states, and $\text{Sequ}_D$ of sequential states in $D$, together with the following data for each $S \in \text{States}_D$:

- a string $\text{name}(S)$ of characters called the name of the state,
- an action $\text{entry}(S) \in \text{Action}$ called the entry action,
- a set of states $\text{state}(S) \subseteq \text{States}_D$, the set of substates of $S$,
- an activity $\text{internal}(S) \in \text{Activity}$ called the internal activity (or do-activity) of the state, and
- an action $\text{exit}(S) \in \text{Action}$ called the exit action,

under the following conditions:

- We have $\text{Top}_D \in \text{Conc}_D \cup \text{Sequ}_D$. 
• For every $S \in \text{Sequ}_D$ there exists exactly one $T \in \text{state}(S) \cap \text{Initial}_D$ (which we write as $\text{init}(S)$).
• $S \in \text{Simple}_D \cup \text{Final}_D \cup \text{Initial}_D$ implies $\text{state}(S) = \emptyset$ and $S \in \text{Conc}_D$ implies that $\text{state}(S)$ has at least cardinality 2.
• $T \in \text{Conc}_D$ and $S \in \text{state}(T)$ implies $S \in \text{Conc}_D \cup \text{Sequ}_D$.
• For all $S, T \in \text{States}_D$, state $(S) \cap \text{state}(T) \neq \emptyset$ implies $S = T$.
• For $S \in \text{Initial}_D \cup \text{Final}_D \cup \{\text{Top}_D\}$, we have $\text{entry}(S) = \text{nil}$, $\text{internal}(S) = \text{Nil}$, and $\text{exit}(S) = \text{nil}$.
• Let the relation $\prec$ on states $S \in \text{States}_D$ be defined by $S \prec T$ if there exist states $S_1, \ldots, S_n$ with $n \geq 1$ such that $S_1 = S$, $S_n = T$, and $S_i \in \text{state}(S_{i+1})$ for $i < n$. Then $\prec$ is acyclic (in particular irreflexive), and fulfills the condition that for all $S, T, U \in \text{States}_D$ with $S \prec T$ and $S \prec U$ we have $T \prec U$ or $U \prec T$. $\text{Top}_D$ is the largest element in $\text{States}_D$ with respect to $\prec$.

Intuitively, therefore, a state $S \in \text{States}_D$ in a statechart $D$ may be an initial, final, simple, concurrent, or sequential state. A state has a name, and may have entry and exit actions executed when entering and exiting it, and an internal activity executed while the state is active (unless it is initial or final). Concurrent and sequential states have substates, which in the case of concurrent states are again concurrent or sequential states (and the sets of substates are mutually disjoint). The state relation fulfills certain soundness conditions that follow directly from the definition of statecharts at the concrete syntax level (such as the non-existence of substate cycles). Note that the name of the state has no semantic significance; it may be omitted (then $\text{name}(S)$ is the empty string). We allow the same internal activity to be used in different states; it is initialized whenever such a state is entered and is executed at each current state of which it is the internal activity, until it finishes. For technical reasons, there exists a “top” state which includes all other states as substates (possibly in a nested way). At the beginning, the initial state which is a direct substate of the top state is entered.

$\text{Transitions}_D$ is a set with subset $\text{Internal}_D \subseteq \text{Transitions}_D$ such that for $t \in \text{Transitions}_D$, we have the following data:

• a state $\text{source}(t) \in \text{States}_D$, the source state of $t$,
• an event $\text{trigger}(t) \in \text{Events}$, the triggering event of $t$,
• a Boolean expression $\text{guard}(t) \in \text{BoolExp}$ called the guard of $t$,
• an action $\text{effect}(t) \in \text{Action}$ (to be performed when firing $t$), and
• a state $\text{target}(t) \in \text{States}_D$, the target state of $t$

under the following conditions for each $t \in \text{Transitions}_D$:

• $\text{source}(t) \notin \text{Final}_D \cup \{\text{Top}_D\}$ (final states and the top state have no outgoing transitions).
• $\text{target}(t) \notin \text{Initial}_D \cup \{\text{Top}_D\}$ (initial states and the top state have no incoming transitions).
• $\text{source}(s) = \text{source}(t) \in \text{Initial}_D$ implies $s = t$ for any $s, t \in \text{Transitions}_D$. 
• \(\text{source}(t) \in \text{Initial}_D\) implies \(\text{trigger}(t) = \text{ComplEv}\) and \(\text{guard}(t) \equiv \text{true}\) (syntactic equality).

• For any \(S \in \text{States}_D\), \(\text{source}(t) \in \text{state}(S)\) implies \(S \in \text{Sequ}_D\) and \(\text{target}(t) \in \text{state}(S)\).

• \(\text{trigger}(t)\) must be of the form \(op(\text{exp}_1, \ldots, \text{exp}_n) \in \text{Events}\) where \(\text{exp}_1, \ldots, \text{exp}_n \in \text{Var}\) are variables (called \text{parameters}), which must be mutually distinct.

• If \(t \in \text{Internal}_D\) then \(\text{source}(t) = \text{target}(t)\).

• Multiple completion transitions leaving the same state must have mutually exclusive guard conditions. For \(s, t \in \text{Transitions}_D\) such that \(\text{source}(s) = \text{source}(t)\) and \(\text{trigger}(s) = \text{trigger}(t) = \text{ComplEv}\), the condition \(\text{guard}(s) \land \text{guard}(t)\) evaluates to \text{false} for any variable valuations [Obj03, p. 2-159].

The intuition is that transitions describe how to proceed from one state (the source state of the transition) of an object to another (the target state). Firing a transition is caused by its triggering event (which is an event whose message has mutually distinct variables as arguments); the transition is only fired if its guard is currently fulfilled. In that case, the effect of the transition is also executed. There are some consistency restrictions on the abstract syntax: final states and the top state have no outgoing, and initial states and the top state no incoming, transition; an initial state has only one outgoing transition, which is a completion transition the guard of which is the constant true. A state with outgoing transition can be a substate only of a sequential state, which also contains the target state of that transition. For internal transitions the source and target states coincide. Multiple completion transitions leaving the same state must have mutually exclusive guard conditions.

As in [BCR00], we assume a special completion event \(\text{ComplEv} \in \text{Events}\) (with no parameters). A transition \(t\) with \(\text{trigger}(t) = \text{ComplEv}\) is called a completion transition; the trigger \(\text{ComplEv}\) is not written explicitly in the diagram.

A guard consisting of the expression \text{true} may be omitted in the diagram, as well as any occurrence of the action nil (in both states and transitions) or the internal activity Nil. If \(t \in \text{Internal}_D\) then \(t\) is called an internal transition, otherwise it is called \text{external}\(^3\).

To model the passing of control, we assume that return messages \(\text{return}_{op}\) are given explicitly in the diagrams and that the following conditions are fulfilled:

• A target state \(S'\) of a transition whose action \(op\) is a synchronous call operation has no internal activities and exactly one outgoing transition, and this transition carries the corresponding return event \(\text{return}_{op}\), and no guards or actions, as follows:

\(^3\) Note that there can be external transitions with the same source and target states and that these are different from internal transitions, because triggering the latter does not involve executing entry or exit actions of the corresponding state.
8.1 Formal Semantics for a Fragment of UML

- For any transition \( t \) with \( \text{effect}(t) = \text{return}_{op} \) for some \( op \in \text{Op} \), we have \( \text{internal}(\text{target}(t)) = \text{Nil} \) and \( \text{trigger}(s) \neq \text{CompEv} \) for any transition \( s \) with \( \text{source}(s) = \text{target}(t) \).

The first condition ensures that, within a concurrent substate, an object that makes a synchronous operation call hands over the control thread and waits until the return message arrives. The second condition ensures that, within a concurrent substate, an object gives back the thread of control when passing back the return message of a synchronous operation call. Note that both conditions apply to each concurrent substate of a statechart separately. For example, this means that after sending a synchronous message (and while waiting for the return message), an object may be specified to accept an other message (through a different concurrent substate). For an example see Fig. 8.1 (and the explanation there). If confusion is impossible, the subscript \( op \) on return messages may be omitted in the diagram.

This treatment is a departure from the standard UML statecharts definition, which itself does not treat recursive calls properly, as pointed out in [TS02], which also offers a solution for this problem. While our approach (similar to the treatment in [HG97], see also below) does not permit recursive calls either, we choose this way of modeling synchronous calls for space restrictions. Incorporating the solution in [TS02] remains interesting further work.

Nevertheless, for the assignment of return values, one may use the notation

\[
\begin{align*}
S \xrightarrow{ev[g]} & S' \\
/\alpha := \text{op(arg)} & \quad \text{return}_{op}(a) \quad T
\end{align*}
\]

as a shortcut for

\[
\begin{align*}
S \xrightarrow{ev[g]} & S' \xrightarrow{\text{return}_{op}(a)} \\
/\alpha := \text{op(arg)} & \quad T
\end{align*}
\]

(where \( ev \) is an event, \( gd \) a guard, and \( S' \) is a simple state with no other incoming or outgoing transitions, and no entry nor exit actions, nor internal activities). We emphasize that we treat this simply as a syntactic shortcut; a more detailed examination of the mechanism of assigning return values in UML is beyond the current scope but can be found for example in [TS02].

As a further syntactic shortcut, one may use pattern matching with constants as arguments in the event specifications for increased readability; these event specifications should be replaced by events with variables as arguments and the relevant guard conditions before assigning the formal semantics.

Behavioral Semantics

We give a formal semantics of statechart diagrams using UML Machines. It is based on part of [BCR00], which, however, had to be extended to incorporate
Fig. 8.1. Example: parallel invocations

features such as explicit modeling of the passing of messages with their arguments between different objects or components, and use of the arguments in the subsequent execution.

The central part of the UML statechart semantics is the run-to-completion step, which means that events are processed one at a time, and that the current event is completely executed before the next event is dispatched. Any dispatched event that does not trigger any transition is lost.

We explain how our statechart semantics, for multi-threaded objects, treats the fact that an object may receive several synchronous messages calling the same operation \( op \) before sending back a corresponding return value, in different concurrent substates. To enable sending back the return value to the sender, each state \( S \) containing substates that accept \( op \) has an associated last-in-first-out buffer \( \text{sender}(S, op) \) containing the names of the senders of the message calls (see the vocabulary of \([D]_{SC} \) defined below). This buffer is updated by the macro \( \text{execEv}(t, e) \) given below. When return messages are sent out from within \( S \), the recipients of these messages are taken from that buffer, as defined in \( \text{ActionRule}_S(\text{send}(\text{return}_op(\text{arg})) \) below. Thus the assumption is that, within a state \( S \), a return message for \( op \) corresponds to the last call of \( op \) received beforehand. Also, we assume that return messages are sent while the execution is still in a (direct) substate of \( S \); otherwise, the return message is lost (to avoid confusion with return messages from concurrent substates). A simple example of such a situation is given in Fig. 8.1. A typical execution of this statechart would be to wait for reception of the synchronous operation \( \text{call} \) with argument \( z1 \) (handled by the first of the two concurrent substates), to call itself first with the asynchronous message \( \text{store} \) with argument \( z2 \) and then with the operation \( \text{call} \) with argument \( z4 \) (both by the first substate, while the reception is handled by the second substate), and to send the return value \( z6 \) back to itself and finally the return value \( z8 \) back to the sender of the first \( \text{call} \) message.

Note that we do not consider call-backs within one state (rather than across several concurrent substates), for example for recursion. We refer to the comment in [HG97, p. 39]: “When the client’s statechart invokes another object’s operation, its execution freezes in midtransition, and the thread of control is passed to the called object. Clearly, this might continue, with the called object calling others, and so on. However, a cycle of invocations that
leads back to the same object instance is illegal, and an attempt to execute it will abort.” In fact, recursive call-backs within one thread cannot be handled properly within the current official UML statechart run-to-completion semantics, as pointed out in [TS02], which is why [TS02] proposes a solution that introduces an additional kind of model element. It would be interesting further work to inquire whether we could include this extension in our present work. In the meanwhile, our above restriction may be tolerable when modeling security-critical parts of systems, where the emphasis is on interaction rather than computation.

We fix a statechart diagram \( D \) together with a set \( \text{Att}_D \subseteq \text{Attribute} \) of used attributes and give its behavioral semantics as a UML Machine \([D]^{SC}\) with name \( O \) def Name\(_A = \text{Object}_D \) and the following functions:

- the set name \( \text{curState} \) (storing the set of currently active states),
- the multi-set names \( \text{inQu}_O, \text{outQu}_O \) (the input and output queues),
- the function name \( \text{sender}(S, op) \) mapping each concurrent or sequential state \( S \in \text{Conc}_D \cup \text{Seq}_D \) and each synchronous operation name \( op \in \text{Op} \)
  to a list of sender names each of the form \( n_1 : \ldots : n_k \) where \( n_1, \ldots, n_{k-1} \)
  are names of subsystems and \( n_k \) is the name of an object,
- the function name \( \text{finished} : \{[D]^{SC}\} \cup \text{States}_D \rightarrow \text{Bool} \) (indicating whether \( D \) or a given state is finished),
- all variable names in \( \text{trigger}(t) \) for all \( t \in \text{Transitions}_D \), and
- the attribute names in \( \text{Att}_D \).

Recall from the definition of UML Machines in Sect. 7.1 that whenever several UML Machines are executed in parallel (for example, those arising from different statecharts), they may generally share only input and output queues and their finished flag, if necessary by suitable renaming (the only exception occurring in the semantics for subsystems in Sect. 8.1.7 where different UML Machines modeling activities of the same object may share input and output queues and attributes). Therefore, to improve readability, we refrain from explicitly parameterizing names such as \( \text{curState} \) and \( \text{Completed} \) (see below), for example, although they are associated with a specific statechart diagram. On the other hand, each value \( \text{finished}_S \) for a state \( S \in \text{States}_D \) of \( D \) is shared between the interactive ASM \([D]^{SC}\) modeling \( D \) and the interactive ASM modeling the internal activity of \( S \) (if any exists), by assumption on activities (see Sect. 8.1.1).

The Boolean \( \text{finished}_S \) may be set to \( \text{true} \) by the rule \( \text{Exec}(\text{internal}(S)) \)
of an internal activity at state \( S \) to indicate that the activity has finished. In particular, the UML Machine \([D]^{SC}\) sets \( \text{finished}_{D}^{SC} \) to \( \text{true} \) at the end of its execution.

For each state \( S \in \text{States}_D \setminus \{\text{Top}_D\} \), we write \( \text{upState}(S) \) for the unique state of which \( S \) is a direct substate.

At the initial state of the UML Machine \([D]^{SC}\), we define:

- \( \text{inQu}_O \) and \( \text{outQu}_O \) to be equal to \( \emptyset \),
- currState \(= \{ \text{Top}_S \} \cup (\text{Initial}_D \cap \text{state}(\text{Top}_D)) \),
- finished\(_{DSC}\) and finished\(_A\) for any internal activity \(A\) to be equal to false, and
- \(\text{sender}_S(op) := []\) for each state \(S\) and operation name \(op \in \text{Op}\).

The UML Machine \([D]^{SC}\) has the rule \(\text{Exec}(D)\) given below using macros defined in the rest of the subsection. It selects the event to be executed next (where priority is given to the completion event) and executes it, and then executes the rules for the internal activities in a random order.

\[
\text{Rule Exec}(D) : \\
\text{if currState } \subseteq \text{Final}_D \cap \text{state}(\text{Top}_D) \text{ then } \text{finished}\(_{DSC}\) := \text{true} \\
\text{else} \\
\text{if Completed } \neq \emptyset \text{ then } \text{eventExecution(ComplEv)} \\
\text{else} \\
\text{choose } e \text{ with } e \in \text{inQu}_O \text{ do} \\
\text{inQu}_O := \text{inQu}_O \setminus \{e\}; \\
\text{eventExecution}(e); \\
\text{loop } S \text{ through set currState} \\
\text{Exec(internal}(S))
\]

Fig. 8.2. Statechart rule

The idea behind the statechart rule in Fig. 8.2 is the following. Firstly, it is checked whether all active states are final and direct substates of the top state, in which case the execution of the statechart is finished, which is indicated by setting \(\text{finished}_{DSC} := \text{true}\). Otherwise, an event is executed. If there is a state that is \(\text{completed}\) (and thus contained in the set \(\text{Completed}\) defined below), the completion event is executed. Otherwise, an event \(e\) is dispatched from the input queue (provided it is non-empty) which is executed. After the event execution, a further iteration of the internal activities of the active states is performed.\(^4\)

Thus our semantics is "based on the premise that a single run-to-completion step applies to the entire state machine and includes the concurrent steps taken by concurrent regions in the active state configuration" [Obj03, p. 2-162].

Here we make use of the macro \(\text{Completed}\). With \(\text{macro}\) we mean a name that is just introduced for presentation purposes; it is not a function updated by \([D]^{SC}\), but instead its definition is included in the rule at each of its occurrences when executing \([D]^{SC}\). This remark also applies to the other

\(^4\) Recall that internal activities are themselves modeled as UML Machines, which means that executing \(\text{Exec}(\text{internal}(S))\) for a state \(S\) does not \textit{restart} the activity \(\text{internal}(S)\) (this is done only when entering a state by firing \(\text{Init}(\text{internal}(S))\)), but only executes a further cycle of that activity.
macros used in the following. Completed is defined to be syntactically equal to the following expression:

\[
\text{Completed} \equiv \\
\{ S \in \text{currState} : \exists t \in \text{Transitions}_D. (\text{source}(t) = S \land \text{trigger}(t) = \text{ComplEv}) \land \\
( S \in \text{Initial}_D \\
\lor \text{finished}_{\text{internal}}(S) \\
\lor (S \in \text{Seq}_D \cup \text{Conc}_D \land \text{state}(S) \cap \text{currState} \subseteq \text{Final}_D) ) \}
\]

\text{eventExecution}(e) (for an event \( e \)) is defined to be syntactically equal to the expression given in Fig. 8.3.

\[
\text{eventExecution}(e) \equiv \\
\text{loop } T \text{ through set } \text{FirableTrans}(e) \\
\text{choose } t \text{ with } t \in T \text{ do} \\
\text{if } t \in \text{Internal}_D \text{ then } \text{execEv}(t, e) \\
\text{else} \\
\text{execState(source}(t)); \\
\text{execEv}(t, e); \\
\text{enterState(target}(t))
\]

\textbf{Fig. 8.3. Event execution rule}

Note that at any given point in time, an event may fire several transitions in different concurrent regions. The rule \text{loop } T \text{ through set } \text{FirableTrans}(e) in the \text{eventExecution}(e) \text{ rule ensures that this is done consistently by sequentializing it.}

\text{FirableTrans}(e) \text{ is defined as follows (while it is straightforward to define it as a syntactic macro, we give a more descriptive version for readability). For any transition } t \text{ we define enabled}(t, \text{ComplEv}) \equiv \text{true} \text{ if the following conditions are fulfilled (otherwise it is false):}

- \text{trigger}(t) = \text{ComplEv},
- \text{guard}(t) \text{ is } \text{true}, \text{ and}
- \text{source}(t) \in \text{currState} \cap \text{Completed}.

For any transition \( t \) and any event \( e \neq \text{ComplEv} \) we define \text{enabled}(t, e) \equiv \text{true} \text{ if the following conditions are fulfilled (otherwise it is false):}

- the operation or signal names of \text{trigger}(t) \text{ and } e \text{ coincide: }
  \text{msgnm} \text{trigger}(t) = \text{msgnm} \text{e},
- \text{guard}(t) \text{ evaluates to } \text{true} \text{ when its variables are substituted with the arguments of } e, \text{ and}
- \text{source}(t) \in \text{currState}.

Let \text{FirableStates}(e) \text{ be the set of } \sim \text{-minimal elements in the set } \{ S \in \text{States}_D : \exists t. \text{enabled}(t, e) \land \text{source}(t) = S \}. \text{ Then } \text{FirableTrans}(e) \equiv \{ t \in
Transitions\textsubscript{D} : enabled(t, e) \land source(t) = S \rightarrow S \in \text{FirableStates}(e) \} (the set of sets of enabled transitions with the same, innermost state).

We define the macro exitState(S) for a state S in Fig. 8.4.

\[
\text{exitState}(S) \equiv \\
\text{if state}(S) \cap \text{currState} \neq \emptyset \\
\text{then} \\
\text{loop } T \text{ through set state}(S) \cap \text{currState} \\
\text{exitState}(T) \\
\text{else} \\
\text{currState} := \text{currState} \setminus \{S\}; \\
\text{ActionRule}(\text{exit}(S))
\]

\textbf{Fig. 8.4. Exit state rule}

The intuition behind this rule is the following. First the substates of the state S to be exited are exited recursively. Then S is exited by removing it from the set of current states and by firing its exit rule.

The macro \text{execEv}(t, e) (for a transition t and an event e) is defined in Fig. 8.5. In \text{Args}(\text{trigger}(t)) := \text{Args}(e), each of the variables in \text{trigger}(t) is assigned the respective input value in \text{Args}(e).

\[
\text{execEv}(t, e) \equiv \\
\text{Args}(\text{trigger}(t)) := \text{Args}(e); \\
\text{if msgnm}(e) \in \text{Op} \text{ then} \\
\text{sender}(\text{upState}(\text{source}(t)), \text{msgnm}(e)) := \\
\text{sndr}(e).\text{sender}(\text{upState}(\text{source}(t)), \text{msgnm}(e)); \\
\text{ActionRule}_{\text{upState}(\text{source}(t))}(\text{effect}(t))
\]

\textbf{Fig. 8.5. Execute event rule}

We define the macro enterState(S) for a state S in Fig. 8.6. The intuition behind this rule is the following. First S is added to the set of current states, its entry action is executed, and its internal activity is initialized. \text{sender}(S, op) is initialized to the empty list for each operation op. If S is a sequential state, its initial state is entered. Otherwise (that is, if S is a concurrent state or if its set of substates is empty), its substates are entered recursively.

\textbf{Example}

The interpretation \([\text{Snadr}]^{SC}\) defined above of the statechart \text{Snadr} given in Fig. 8.7 which describes the behavior of the object \text{Snadr} is equivalent (in the sense of Sect. 7.1) to the UML Machine ([\text{Snadr}]^{SC}, \{\text{inQu}_{\text{Snadr}}\},

\[ enterState(S) \equiv \\
\text{currState} := \text{currState} \cup \{S\}; \\
\text{ActionRule}_{\text{opState}(S)}(\text{entry}(S)); \\
\text{forall op with } \text{op} \in \text{Op} \text{ do} \\
\text{sender}(S, \text{op}) = []; \\
\text{if } S \in \text{Seq} \text{ then enterState(init(S))} \\
\text{else loop } T \text{ through set state}(S) \\
\text{enterState}(T) \]

Fig. 8.6. Enter state rule

\[ \begin{array}{c}
\text{Wait} \\
\text{transmit}(d) \\
\rightarrow \\
\text{Send} \\
\end{array} \]

Fig. 8.7. Example sender statechart

\{\text{outQu}_{\text{Sndr}}, \text{finished}_{\text{Sndr}}^e \} \text{ with the initialization rule } \text{Init}(\text{Sndr}) \text{ and } \\
\text{whose main rule is given in Fig. 8.8.} \text{ Further examples are given in Chap. 5.} \]

\begin{verbatim}
case currState of
  \{\text{Top}_{\text{Sndr}}, \text{Initial}_{\text{Sndr}}\}: \text{do currState := } \{\text{Top}_{\text{Sndr}}, \text{Wait}\}
  \{\text{Top}_{\text{Sndr}}, \text{Wait}\}: \text{do}
    \text{choose } e \text{ with } e \in \text{inQu}_{\text{Sndr}} \text{ do}
    \text{do - in - parallel}
    \text{inQu}_{\text{Sndr}} := \text{inQu}_{\text{Sndr}} \setminus \{ e \}
    \text{if msgnm}(e) = \text{send then}
    \text{do - in - parallel}
    \text{currState := } \{\text{Top}_{\text{Sndr}}, \text{Send}\}
    d := \text{Arg}(e)
  \text{endo}
  \{\text{Top}_{\text{Sndr}}, \text{Send}\}: \text{do}
    \text{do - in - parallel}
    \text{currState := } \{\text{Top}_{\text{Sndr}}, \text{Wait}\}
    \text{tooutQu}_{\text{Sndr}}(\{ \text{transmit}(d) \})
  \text{endo}
\end{verbatim}

Fig. 8.8. Example interpretation
8.1.4 Sequence Diagrams

We emphasize again that we give a formal semantics only for a rather restricted fragment of sequence diagrams, for reasons explained in Sect. 8.1, which, however, is sufficient for our present needs.

Recall that in our approach, we view an object as an entity characterized by a unique name. We thus identify the objects in the runtime system with UML objects. Also, recall that at each point of a given execution of a system, the same message instance in the running system is only represented once in the UML diagrams.

Following [Obj03] we assume that no two events happen exactly at the same time. This implies that some behavior that could be viewed as concurrent may be sequentialized, for example, if two subsequent messages involve completely distinct components as senders and receivers. Note that this design decision in [Obj03] is not a restriction in practice; in particular, in the example mentioned, the formal semantics given below allows the two messages to be sent in an arbitrary order.

Also, for simplicity, we omit the possibility to specify time information in sequence diagrams (either given explicitly or by using non-horizontal connections). Furthermore, UML sequence diagrams in their full generality allow one to use branching lifelines to specify conditional branching. It has been argued, however, that branching lifelines can become confusing when the system under consideration has a significant amount of conditional branching (for example, [FS97] takes this view). Thus, in our approach we do not consider branching lifelines in sequence diagrams, but use statecharts when necessary to model conditional behavior. Similarly, we do not use sequence diagrams (but instead statecharts) if we want to describe concurrent behavior within a given component. Sequence diagrams are mainly used to describe behavior exemplarily.

Under the above assumptions, all connections in a sequence diagram can be ordered strictly by their occurrence (that is, by their horizontal position in the sequence diagram, which relies on the above assumption that for physical reasons two events do not happen exactly at the same time, and that therefore only one event is specified to happen at any point in time).

Recall also that following [KW01, p. 15], we do not model the creation and deletion of objects explicitly. In particular, we do not have creation or deletion messages in sequence diagrams.

Note that a sequence diagram $S$ with branching lifelines can be presented as a set $S$ of sequence diagrams without branching lifelines, if in the sequence diagrams in $S$ one marks the branching points from $S$ in a way that allows one to associate each branching point in $S$ to the respective places in the sequence diagrams in $S$. In this way one can conveniently express branching behavior (as suggested in [FS97]), rather than by using branching lifelines, which can make a diagram difficult to read. A detailed discussion of this is omitted.
Note also in our intended application domain of security-critical systems that sequence diagrams have to be used carefully [Aba00]; precisely, a message \textit{msg} on a connection from an object \textit{O} to an object \textit{P}, where \textit{O} and \textit{P} are connected by an untrusted network, means that:

- \textit{O} sends \textit{msg} to the network with intended recipient \textit{P}, and
- if \textit{P} receives a message \textit{msg}' with the same message name as \textit{msg}, it will proceed with its protocol part using the arguments of \textit{msg}'.

There is no guarantee that \textit{P} will ever receive a message with the same name as \textit{msg}, or that \textit{msg}' contains the same arguments as \textit{msg}. Therefore we treat the sent and received arguments as potentially different entities in the sequence diagram. We do this below by using, for each object \textit{O}, each message \textit{msg} accepted by \textit{O}, and each number \textit{n} up to the number of arguments of \textit{msg}, a local variable \textit{O}.\textit{msg}\textsubscript{\textit{n}} of \textit{O} that denotes the \textit{n}th argument of the most recent instance of the message \textit{msg} that is supposed to be received by the object \textit{O} according to the sequence diagram (and these variables may have different values from those intended by the protocol, depending on a possible adversary).

Here, as usual, the prefix \textit{O} may be omitted if no confusion will arise. See Sect. 5.2 for a discussion of these issues in the specific situation of modeling security protocols.

Also, an object has no information about the sender of a message. So at any point during the execution of the sequence diagram, the object may actually receive the expected message, which may or may not come from the expected sender (from inside or outside the sequence diagram).

An example of a sequence diagram is given in Fig. 3.4 (which is discussed in more detail in Sect. 5.2). Here, to increase readability, we use the notation \textit{var} := \textit{exp} (where \textit{var} is a local variable not used for any other purpose and \textit{exp} may not contain \textit{var}) as a syntactic shortcut. Before assigning a semantics to the diagram, the variable \textit{var} should be replaced by the expression \textit{exp} at each occurrence.

### Abstract Syntax of Sequence Diagrams

A sequence diagram \(D = (\text{Obj}(D), \text{Cncts}(D))\) is given by:

- a set \(\text{Obj}(D)\) of pairs \((O, C)\) where \textit{O} is an object of class \textit{C} whose interaction with other objects is described in \(D\), and
- a (finite) sequence \(\text{Cncts}(D)\) consisting of elements of the form \(l = (\text{source}(l), \text{guard}(l), \text{msg}(l), \text{target}(l))\) (so-called \textit{connections}) where
  - \textit{source}(\textit{l}) is the source object of the connection,
  - \textit{guard}(\textit{l}) \in \text{BoolExp} is a Boolean expression that is the guard of the connection,
  - \textit{msg}(\textit{l}) \in \text{Events} is the message of the connection, and
  - \textit{target}(\textit{l}) \in \text{Obj}(D)\) is the target object of the connection,
such that for each \( l \in \text{Cncts}(D) \), we have \( \text{source}(l) \in \text{Obj}(D) \) or \( \text{target}(l) \in \text{Obj}(D) \) (or both).

Again a guard syntactically equal to true may be omitted in the diagram.

Note that our semantics for sequence diagrams given below supports the joint use of different sequence diagrams \( D, D' \) where \( \text{Obj}(D) \cap \text{Obj}(D') \neq \emptyset \) provided that the parts of \( D \) and \( D' \) referring to the same object \( O \) relate to different parts of the possible behavior of \( O \) separated in time or depending on mutually exclusive preconditions (and not to different aspects of parts of its behavior that may overlap over a period in time). That is, at any one time the behavior of a given thread of an object is represented by only one diagram.

To model the passing of control, we assume that return messages \( \text{return}_{op} \) are given explicitly in the diagrams and that the following condition is fulfilled for each sequence \( l \) of connections at nodes in \( \text{Cncts}(D) \): the number of return messages \( \text{return}_{op} \) for an operation \( op \) sent from an object \( O \) is at any time bounded by the number of calls of \( op \) received up to that time (that is, no \( \text{return}_{op} \) message is sent without previously receiving a corresponding \( op \) call).

If confusion is impossible, the subscript \( op \) on return messages may be omitted in the diagram.

**Behavioral Semantics**

We present the formal semantics for sequence diagram behavior. It supports explicit modeling of the passing of messages with their arguments between different objects or components, and further use of the arguments in the subsequent execution (for example, the guards of the transitions may refer to the input arguments and the attributes may be assigned values received as input). Note that we do not consider possible variations of an object’s behavior at the same point in time during its behavior through diagrams overlapping in content.

Note that collaboration diagrams contain similar information as sequence diagrams. A formal semantics for collaboration diagrams using UML machines is contained in [Cav00]. We cannot make use of this for our semantics for sequence diagrams, however, since the above-mentioned features are not supported.

In the semantics defined below, the sequence \( l \) of connections in a sequence diagram is split into "views" \( \mathfrak{I}_O \) for each of the involved objects \( O \), consisting of the connections going out from or coming into \( O \), as defined more precisely below. For each such view, we define a UML Machine modeling the behavior of \( O \) as defined by the sequence diagram.

As with statecharts, we model the order of dequeuing events from the event queue using the non-deterministic choice operator of UML Machines, to cover the different possibilities of this semantic variation point. A motivation for this treatment is given in Sect. 8.1.3.

As in the case of statecharts, we also have to account for the possibility that an object may receive several synchronous message instances calling
the same operation $op$ before sending back a corresponding return value. To enable sending back the return value to the sender, each UML Machine representing an object $O$ in the sequence diagram that accepts $op$ has an associated last-in-first-out buffer $sender_{Top}(op)$ (or shorter $sender(op)$) because a sequence diagram has only one control state $Top$ containing the names of the senders of the message calls (see the definition of UML Machines in Sect. 7.1). When return messages are sent out from $O$, the recipients of these messages are taken from that buffer, according to the definition of the macro $ActionRule_{Top}(return_{op}(args))$ in Sect. 8.1.1 (and the condition on the return messages in the above subsection on the abstract syntax ensures that the buffer is not empty at that point). Thus the assumption is that a return message from $O$ for $op$ corresponds to the last call of $op$ received by $O$ beforehand.

Note that there is only one input buffer (and one output buffer) for a given object or component, even if it occurs in several sequence diagrams. See Sect. 8.1.7 for more information regarding the joint use of several behavioral diagrams.

We fix a sequence diagram $D$ and an object $O \in \text{Obj}(D)$. We give the behavior of $O$ as defined in $D$ as a UML Machine $[D,O]^{SD}$.

We assume that for each object $O$, each message $msg$ accepted by $O$, and each number $n$ up to the number of arguments of $msg$, the set $\text{Var}$ (see Sect. 7.5) contains an element $O.msg_n$ that will store the $n$th argument of the most recent instance of the message $msg$ that is supposed to be received by the object $O$ according to the sequence diagram. We define $O.msg = [O.msg_1, \ldots, O.msg_k]$ (where the operation $msg$ is assumed to have $k$ arguments).

The signature of $[D,O]^{SD}$ has the following function names:

- the multi-set names $\text{inQu}_O$, $\text{outQu}_O$ (the input and output queues),
- the name $\text{finished}_{[D,O]}$ $\in \text{Bool}$ (indicating whether $[D,O]^{SD}$ is finished),
- a name $\text{cncts}$ (the subsequence of $\text{Cncts}(D)$ consisting of the connections relevant to $O$ that are still to be processed),
- the function name $\text{sender}(op)$ (as shorthand for $sender_{Top}(op)$ since a sequence diagram has only one control state $Top$) mapping each synchronous operation name $op \in \text{Op}$ to a list of sender names each of the form $n_1 \cdots n_k$ where $n_1, \ldots, n_k$ are the names of subsystems and $n_k$ is the name of an object, and
- the names $O.msg_n$.

Sending a synchronous message $op \in \text{Op}$, asynchronous message $msg \in \text{Sig}$, or return message $return_{op}$ is modeled as the actions $a = \text{call}(op)$, $a = \text{send}(sig)$, and $a = \text{send}(return_{op})$, respectively, with the UML machine rules $ActionRule_{Top}(a)$ defined in Sect. 8.1.1 (or shorter $ActionRuleSD(m)$).
Given a sequence of connections \( l \) and an object \( O \), we define \( l \|_O \) to be the subsequence \( l \) of those elements \( l \) with source\((l) = O \) or target\((l) = O \) (called the object \( O \)'s view of the connections).

At the initial state of the UML Machine \( [D,O]^{SD} \), we define:

- \( \text{inQu}_O \) and \( \text{outQu}_O \) to be equal to \( \emptyset \),
- \( \text{cncts} \triangleq \text{Cncts}(D)\|_O \), and
- \( \text{finished} := \text{false} \).

The rule of the UML Machine \( [D,O]^{SD} \) is given in Fig. 8.9.

**Rule Exec\((D,O)\):**

if \( \text{cncts} = [] \) then \( \text{finished}_{D,O} := \text{true} \)
else
  if source\((\text{head}(\text{cncts})) = O \land \text{guard}(\text{head}(\text{cncts}))
    \text{then }
      \text{ActionRuleSD}(\text{msg}(\text{head}(\text{cncts})));
      \text{if target}(\text{head}(\text{cncts})) \neq O \text{ then } \text{cncts} := \text{tail}(\text{cncts});
    \text{if target}(\text{head}(\text{cncts})) = O \text{ then } \text{choose } e \text{ with } e \in \text{inQu}_O
      \land \text{msgnm}(\text{msg}(\text{head}(\text{cncts})) = \text{msgnm}(e) \text{ do }
        \text{inQu}_O := \text{inQu}_O \setminus \{e\};
        O.\text{msg}(\text{head}(\text{cncts})) := \text{Arg}(e);
        \text{if } \text{msgnm}(e) \in \text{Op} \text{ then }
          \text{sender}(\text{msgnm}(e)) := \text{sndr}(e).\text{sender}(\text{msgnm}(e));
        \text{cncts} := \text{tail}(\text{cncts})
  \text{end if}
\text{end if}
\text{end if}
\text{end if}

**Fig. 8.9.** UML Machine rule for sequence diagram

Thus the sequence \( \text{cncts} \) of connections with source or target \( O \) is processed from the beginning to the end. If the connection \( c \) under consideration has \( O \) as its source and the guard of \( c \) evaluates to true, the message of \( c \) is sent out, and, unless \( c \) has \( O \) also as its target, the next connection is examined. If the guard of \( c \) evaluates to false, the execution of the sequence diagram does not proceed. If the connection \( c \) under consideration has \( O \) as its target, an event with the same message name as the message of \( c \) is chosen and dispatched from the input queue (if existent), and its arguments are stored in the variable \( O.\text{msg}(\text{head}(\text{cncts})) \). If the specified system executes as planned there will be such a message in our input queue put there by another object in the diagram under consideration, that is the one put there by the object from whose point of view \( c \) is an outgoing message. If such an event does not currently exist, the input queue is checked at each iteration round until it does exist. When the sequence \( \text{cncts} \) is reduced to the empty list, \( \text{finished}_{D,O} \) is set to \( \text{true} \) and no further processing is done.

Note that it is not checked whether an object actually uses up all the contents of its input queue. Also, this semantics automatically enforces the
(realistic) assumption that the behavior of an object after reception of a message does not depend on the identity of the sender of this message. In particular, at any point during the execution of the sequence diagram, the object may actually receive the expected message, which, however, may or may not originate from the expected sender, which may or may not be part of the sequence diagram (in that sense, there may be the possibility to define a notion of composition of sequence diagrams based on the present semantics, which, however, is not investigated here).

Note that, although we will not actually exploit this feature in the present work, our way of modeling sequence diagrams would support the joint use of different sequence diagrams $D, D'$ where $\text{Obj}(D) \cap \text{Obj}(D')$ provided that the parts of $D$ and $D'$ referring to the same object $O$ relate to different parts of the possible behavior of $O$ separated in time or depending on mutually exclusive preconditions (and not to different aspects of parts of its behavior that may overlap over a period in time). That is, the assumptions on our semantics need to enforce that at any one time the behavior of a given thread of an object is represented by only one diagram.

We do not consider possible variations of an object’s behavior at the same point in time during its behavior through diagrams overlapping in content.

8.1.5 Activity Diagrams

In our treatment, the only actions admitted in activity diagrams are assignments, since messages are processed by the activities in the activity diagram.

As in the statechart case, fork-join and junction pseudostates, and submachine, stub, and synch states, can be reduced to the constructs treated here [Cav00]. Note that this, in particular, requires that the activity diagrams are well-structured in the sense that they can be viewed as statecharts, following what the UML standard requires on [Obj03, p. 2-178]. Note, however, that the UML definition document is not entirely consistent with regard to this point. In particular, some additional features of activity diagrams mentioned in the UML definition may contradict its requirement that activity diagrams should be a special kind of statechart (for example, multiple parallel invocations of the same activity). In our simplified account of activity diagrams, we do not consider such features. Despite the different notation, the same well-formedness rules on states as in state-machines then apply [Obj03, p. 2-178]. Since we only considered a simplified fragment of statecharts, we thus also only consider a simplified fragment of activity diagrams (which, however, is sufficient for our needs).

We do not consider the additional concept of object flow states, since we will not need it.

To keep the presentation concise, we remain on the level of abstract (rather than concrete) syntax. The concrete syntax form of an activity diagram can be derived from its abstract syntax in the way outlined above.
Thus the abstract syntax of activity diagrams is defined as follows (which is a simplification of the abstract syntax of statecharts from Sect. 8.1.3).

An activity diagram \( D = (\text{States}_D, \text{Top}_D, \text{Transitions}_D) \) is given as a (finite) set of \( \text{States}_D \), the top state \( \text{Top}_D \in \text{States}_D \), and a set \( \text{Transitions}_D \) of transitions, defined in the following.

\( \text{States}_D \) is a set that is disjointly partitioned into the sets \( \text{Initial}_D, \text{Final}_D, \text{Simple}_D, \text{Conc}_D, \text{Sequ}_D \), together with the following data for each \( S \in \text{States}_D \):

- a string \( \text{name}(S) \) of characters called the \text{name} of the state,
- an action \( \text{entry}(S) \in \text{Action} \) called the \text{entry action},
- a set of states \( \text{state}(S) \subseteq \text{States}_D \), the set of \text{substates} of \( S \),
- an activity \( \text{internal}(S) \in \text{Activity} \) called the internal activity (or do-activity) of the state,
- an action \( \text{exit}(S) \in \text{Action} \) called the exit action, and
- the name \( \text{swim}(S) \) of the swimlane containing \( S \),

under the conditions given in Sect. 8.1.3 and the condition that access to attributes applies only to attributes of the object in the relevant swimlane.

\( \text{Transitions}_D \) is a set with subset \( \text{Internal}_D \subseteq \text{Transitions}_D \) such that for \( t \in \text{Transitions}_D \) we have the following data:

- a state \( \text{source}(t) \in \text{States}_D \), the source state of \( t \),
- a Boolean expression \( \text{guard}(t) \in \text{BoolExp} \) called the \text{guard} of \( t \),

and

- a state \( \text{target}(t) \in \text{States}_D \), the target state of \( t \)

under the conditions given in Sect. 8.1.3. Transitions in activity diagrams do not have events or actions; they are triggered by completion events.

Then following the statechart semantics in Sect. 8.1.3, an activity diagram \( D \) with a set \( \text{Att}_D \subseteq \text{Attribute} \) of attributes (used by the activities in \( D \)) and a set \( \mathcal{S} \) of swimlanes representing objects or components are modeled by a UML Machine

\[
( [[D]^{AD}, \{\text{inQu}_O : O \in \mathcal{S} \}, \{\text{outQu}_O : O \in \mathcal{S} \} \cup \{\text{finished}_{D^{sc}} \})
\]

with the rules \( \text{Init}(D) \) and \( \text{Main}(D) \) as in the definition of \( [[D]^{SC} \), except that there is no access to the input and output queues, which happens on the activity level. For completeness, the simplified rules are repeated below.

An internal activity \( S \) in an activity diagram can for example be given as \( D \) (for a statechart \( D \)) or \( D.O \) (for a sequence diagram \( D \) and specified object \( O \)), where \( \text{Object}_D \) or \( O \) is the name of the object or component \( \text{swim}(S) \) labeling the swimlane containing \( S \) (for more details on how this could be done and on the restrictions we impose when doing it, see Sect. 8.1.7). In this way our statechart semantics deals with the fact that activity diagrams can contain several objects or components in different swimlanes. In each case, the rule \( \text{Exec}(D) \) or \( \text{Exec}(D,O) \) is executed. By general assumption on the joint execution of UML Machines (see Sect. 7.1), these generally have their own
namespaces, except for those variables in their input and output signatures. Thus different UML Machines modeling activities of the same object may share input and output queues and attributes.

We give the rules for the activity diagram semantics, a simplified version of those for the statecharts in Sect. 8.1.3 (see there for explanations):

**Rule Init(D):**
```
seq
  currState := Ø
  finished₁D₁c := false
  enterState(Topₐ)
endseq
```

**Rule Main(D):**
```
if currState ⊆ Final₁D₁ ∩ state(Topₐ) then finished₁D₁c := true
  loop S through set currState
    Main(internal(S))
endseq
```

Completed is defined as in Sect. 8.1.3.

eventExecution(e) (for an event e) is defined to be syntactically equal to the following expression:

\[
\text{eventExecution}(e) \equiv
\text{loop } T \text{ through set FirableTrans}(e)
\text{choose } t \text{ with } t \in T \text{ do}
  \text{if } t \in \text{internal}_D \text{ then execEv}(t, e)
  \text{else}
    \text{seq}
    \quad \text{exitState(source}(t))
    \quad \text{enterState(target}(t))
  \text{endseq}
\text{exitState}(S) \text{ is defined as in Sect. 8.1.3.}
```

We define the macro enterState(S) for a state S:

\[
\text{enterState}(S) \equiv
\text{seq}
  \text{currState := currState } \cup \{S\}
  \text{ActionRuleSC}_{\text{upState}(S)}(\text{entry}(S))
  \text{Init(}\text{internal}(S))
  \text{if } S \in \text{Sequ then enterState(}\text{init}(S))
  \text{else loop } T \text{ through set state}(S)
    \text{enterState}(T)
\text{endseq}
\]
8.1.6 Deployment Diagrams

A deployment diagram is a "diagram that shows the configuration of runtime processing nodes and the components, processes, and objects that live on them" [Obj03, p. Gloss-6].

We give the abstract syntax of deployment models.

A node \( N = (\text{loc}, \text{comp}) \) is given by:

- the name \( \text{loc} \) of its location and
- a set of contained \( \text{components}^5 \) \( \text{comp} \) of the form \( C = (\text{name}, \text{int}, \text{cont}) \) where \( \text{name} \) is the component name, \( \text{int} \) a (possibly empty) set of interfaces, and \( \text{cont} \) a set of subsystem instance and object names (those contained in the component).

A deployment diagram \( D = (\text{Nodes}(D), \text{Links}(D), \text{Dep}(D)) \) is given by:

- a set \( \text{Nodes}(D) \) of nodes,
- a set \( \text{Links}(D) \) of links of the form \( l = (\text{nds}(l), \text{ster}(l)) \) where \( \text{nds}(l) \subseteq \text{Nodes}(D) \) is a two-element set of nodes being linked and where \( \text{ster}(l) \subseteq \text{Stereotypes} \) is a set of stereotypes, and
- a set \( \text{Dep}(D) \) of dependencies of the form \( (\text{clt}, \text{spl}, \text{int}, \text{ster}) \) where \( \text{clt} \) and \( \text{spl} \) are component names (the client and supplier of the dependency), \( \text{int} \) is the interface of \( \text{spl} \) accessed by the dependency (with \( \text{int} = \text{spl} \) if the access is direct), and \( \text{ster} \subseteq \text{Stereotypes} \) is a set of stereotypes.

We assume that for every dependency \( D = (C, S, I, S') \) there is exactly one link \( L_D = (N, N') \) such that \( N = \{C, S\} \) for the set of linked nodes.

8.1.7 Subsystems

Recall that we only consider a simplified fragment of the UML syntax. A general motivation for the simplifications made is given in the introduction to Sect. 8.1. In particular, the notion of subsystem considered here is restricted, for example in the kinds and numbers of diagrams that may be contained. In the UML definition document, there is relatively little restriction on the kinds of diagrams a subsystem may contain and on the relation the diagrams should have to each other. Therefore, giving a formal semantics for this unrestricted use of UML subsystems, which assigns a formal meaning to the diagrams contained in a subsystem as well, could be a challenge comparable to giving a formal semantics to all of UML. For reasons explained in the introduction to Sect. 8.1, this is not attempted in the present work. To demonstrate that our use of UML subsystems is reasonable and our semantics of sufficient interest, we present several non-trivial case studies in Chap. 5.

Thus in our treatment, a system part \( C \) given by a subsystem instance \( S \) may contain sub-parts \( C_1, \ldots, C_n \), given in a so-called static structure diagram

\[5\] With components we mean component instances; we do not consider component (or node) types which are optionally allowed by the UML syntax specification.
(see below). Note that the name static structure diagram used at the instance level may be misleading in so far as in the implemented system, objects may be created at runtime. The diagram \( S \) contains an activity diagram that describes the activities performed by the sub-parts: each swimlane in the activity diagram gives the behavior of the sub-part \( C_i \) whose name labels the swimlane (which may be an object or may itself contain other system parts). Each activity in the activity diagram may be specified either itself as a subsystem instance, or its behavior is described directly as a UML Machine rule (such as an assignment to an attribute) or using a set of statecharts or a sequence diagram (for example, if the swimlane describes an object).

Each statechart describes the behavior of one activity (following the semantics in Sect. 8.1.3). The name of the activity (the context of the statechart, for which the UML specification currently does not offer a special notation\(^6\) [Obj03, 3.74.2]) is written under the initial state of the statechart.

Alternatively, the sequence diagram describes the behavior of a set of activities that interact during their execution (as explained in Sect. 8.1.4). To achieve this, the sequence diagram is split up into different views of the objects or components described in it, as explained in Sect. 8.1.4. Each such view may then describe an activity in the swimlane of the relevant object or component. Again the context of the sequence diagram is written under the diagram, such that the name of a corresponding activity in the activity diagram is the name written under the sequence diagram followed by the name of the object or system part carrying out the activity.

Below we discuss possible complications that may arise when using sequence diagrams to specify activities. Note that it is possible to generalize the use of sequence diagrams to using a set of sequence diagrams for the activities in one activity diagram. Since we do not need this generalization in the following, we leave it out here for reasons of space restriction. Thus the recommended approach here is generally to use sets of statecharts to specify activities in an activity diagram (although we will also make use of the possibility to use a single sequence diagram in our case studies).

Recall that in our approach, we view an object or component as an entity characterized by a unique name, which may have associated information such as its attributes and their values which may change during its execution (which may be specified as a statechart). Thus we identify the objects or components in the runtime system with UML objects or components.

In modeling non-atomic activities using statecharts we follow [Obj03, 2.13.2.7] which requires that at a subactivity state “an associated subactivity graph is executed” (since activity diagrams are special kinds of state machines, we are being more general by allowing the use of statecharts, but one can of course restrict oneself to activity diagrams).

We will explain the idea behind this way of modeling activities. Firstly, note that we take activity diagrams to be special kinds of statecharts, in ac-

\(^6\) This may change with UML 2.0.
cordance with the UML specification (see Sect. 8.1.5). Also, one may observe that within a statechart one may view the sequential substates of a given state $S$ to form a statechart themselves, namely one that describes a certain activity performed at state $S$ (provided that the statechart is well-structured as the ones we consider here are). Thus, conversely, one may use statecharts to define the activities in activity diagrams in a rather natural way: when giving a meaning to an activity diagram (seen as a statechart $C$), the activities of which are defined using statecharts, one essentially inserts the statecharts as substates of the states in the statechart $C$. Intuitively, then, the statecharts defining the activities appearing in the swimlane belonging to an object or component $C$ could be put together to give a larger statechart describing the behavior of $C$. Since from a sequence diagram, the formal semantics given in Sect. 8.1.4 derives a state machine for each of the involved objects (corresponding to the object's view of the sequence diagram), one may also use sequence diagrams to describe activities.

Note also that we do not currently make any restrictions that would prevent a designer from creating a model that may not be particularly intuitive or useful. For example, the following situations might occur. Suppose we are given a sequence diagram $D$ containing objects $O$, $P$, and $Q$. Suppose there is an activity $s.O$ in the activity diagram, but no activity $s.P$. Then, in our formal model, the sequence diagram is translated to three UML machines modeling the behavior of $O$, $P$, and $Q$, and the first of them, but not the second, is executed when executing the system (what this means for the overall behavior of the system depends on the sequence diagram; it may mean that object $O$ waits for a message from $P$ indefinitely). Suppose now that, instead, the activities $s.O$, $s.P$, and $s.Q$ all occur in the activity diagram, but in such a way that they are not concurrent. Then the three UML Machines are executed, but not concurrently (again what this means for the overall behavior of the system depends on the sequence diagram; it may mean that object $O$ waits for a message from $P$ or $Q$ indefinitely and that these are never executed, and that therefore the activity modeled by $O$ never finishes).

Although according to the UML specification statecharts can be used to describe the behavior of various kinds of model elements (such as activities), they are often used to describe the complete behavior of the objects in a given class. This can also be achieved with our approach, by using an activity diagram which for each of the objects involved contains exactly one activity (the behavior of which is given by a statechart), and these activities are synchronized in parallel using synchronization bars.

Furthermore, a subsystem instance contains a deployment diagram specifying the physical layer of the system. This information is exploited when analyzing UML specifications under security aspects (see Chap. 4, using concepts from Sect. 7.5). A subsystem instance may specify a set of accepted messages, and may also offer interfaces.

One can distinguish between realization and specification elements in a subsystem; visualizing this distinction is optional and not considered here.
Abstract Syntax

A subsystem (instance) $S = (\text{name}(S), \text{Msgs}(S), \text{Ints}(S), \text{Ssd}(S), \text{Dd}(S), \text{Ad}(S), \text{Sc}(S), \text{Sd}(S))$ is given by:

- the name $\text{name}(S)$ of the system part modeled by the subsystem,\(^\text{7}\)
- a (possibly empty) set $\text{Msgs}(S) \subseteq \text{MsgNm}$ of names of offered operations and accepted signals,
- a (possibly empty) set $\text{Ints}(S)$ of subsystem interfaces,
- a static structure diagram $\text{Ssd}(S)$ (defined below),
- a deployment diagram $\text{Dd}(S)$,
- an activity diagram $\text{Ad}(S)$, and
- for each of the activities in $\text{Ad}(S)$, a corresponding interactive UML Machine act $\in \text{Activity}$ specifying the behavior of objects appearing in $\text{Ssd}(S)$ by defining the activities in the activity diagram. They may be given directly as UML Machine rules, or as UML machines arising as the formal semantics from the following kinds of diagrams: a (possibly empty) set $\text{Sc}(S)$ of statechart diagrams, possibly a single sequence diagram $\text{Sd}(S) = \{S\}$, and the subsystems in $\text{Ssd}(S)$. Each diagram $D \in \text{Sc}(S) \cup \text{Sd}(S)$ has an associated name $\text{pckname}(D)$ (which in the concrete syntax is written underneath it).

A static structure diagram [Obj03, p. 3-34] $D = (\text{SuSys}(D), \text{Dep}(D))$ is given by a set $\text{SuSys}(D)$ consisting of objects or subsystem instances, and a set $\text{Dep}(D)$ of dependencies ($\text{dep}$, $\text{indep}$, $\text{int}$, $\text{stereo}$) defined in Sect. 8.1.2 (except that $\text{dep}$ and $\text{indep}$ may now be subsystems, rather than objects). We require that the names of the subsystems or objects are mutually distinct.

Note that in UML, static structure diagrams are called class or object diagrams (see Sect. 8.1.2) even though they may contain not just class or objects, but also subsystems. In our usage here, we follow a suggestion on [Obj03, p. 3-34]. Thus an object diagram is a particular kind of static structure diagram, although this again may be confusing in so far as in the implemented system, objects may be created at runtime (and may thus not be considered static).

Scope of Objects or Subsystem Instances

We define the notion of scope of an object or subsystem instance within a subsystem which will be of use for the treatment of freshness of data in Sect. 4.1.2.

The idea is that a piece of data $d$ within a subsystem diagram $S$ belongs to the scope of an object or subsystem instance $C$ contained in $S$ if $d$ is (initially) under the control of $C$. More precisely, $d$ belongs to the scope of $C$ in $S$ if $d$ occurs within $S$ at most in:

\[^\text{7}\] By subsystem in the following we always mean subsystem instance.
- the object or subsystem instance representing \( C \) in the static structure diagram contained in \( S \),
- the swimlanes belonging to \( C \) in the activity diagram contained in \( S \),
- the statechart diagrams contained in \( S \) that model parts of the behavior of \( C \), or
- \( C \)'s view \( 1_C \) of the sequence of connections \( l \) in the sequence diagram contained in \( S \), as defined in Sect. 8.1.4.

**Consistency Between UML Diagrams**

A subsystem \( S \) is called *consistent* if the following conditions are fulfilled.

**Activities**

For every activity \( act \in \text{Activity} \) in a swimlane labeled \( O \) in the activity diagram exactly one of the following holds:

- There is a subsystem \( S \in \text{SuSys}(Ssd(S)) \) with \( \text{name}(S) = act \).
- There is a statechart \( D \in \text{Sc}(S) \) with \( O = \text{Object}_D \) and \( \text{pckname}(D) = act \).
- We have \( act = D,O \) where \( D \in Sd(S) \) is a sequence diagram with \( O \in \text{Obj}(D) \) and \( \text{pckname}(D) = act \).
- The activity is defined directly as an interactive ASM which accesses only the input and output queues and attributes in its own swimlane.

Note that, in particular, several activities can be modeled by statecharts; the above condition ensures that an activity is not modeled in more than one way and enforces that at any one time the behavior of a given thread of an object is represented by only one diagram. However, activities in the same swimlane may access the same attributes of the object specified by the swimlane.

**Names of Behavioral Diagrams**

For any two diagrams \( D, D' \in \text{Sc}(S) \cup \text{Sd}(S) \), the condition \( \text{pckname}(D) = \text{pckname}(D') \) implies \( D = D' \).

**Object Communication**

Each object modeled by a swimlane in \( S \) must appear exactly once in the deployment diagram.

Each subsystem in the deployment diagram and each object in the deployment diagram must appear in the static structure diagram. For any «call» or «send» dependency between subsystems or objects in the static structure diagram there must be the same dependency between the components containing the corresponding subsystems or objects in the deployment diagram.

For each statechart diagram \( S \in \text{Sc}(S) \) the following conditions must hold:
8.1 Formal Semantics for a Fragment of UML 221

- For each call action call\((\text{obj}, e)\) (resp. send action send\((\text{obj}, e)\)) in \(S\) (for an object name \(\text{obj}\)), the object diagram \(C\) in \(S\) must have a «\text{call}» (resp. «\text{send}») dependency from the object \(\text{Object}_{S}\) to the object \(\text{obj}\) or one of its interfaces supplying the operation \(\text{msgm}(e)\) (resp. able to receive the signal \(\text{msgm}(e)\)). The types of the message specifications in the class diagrams and those of the values in the statechart diagrams must match.

- For each assignment action \(\text{att} := \text{exp}\) in \(S\), \(\text{att}\) is contained in the set of attributes of \(\text{Object}_{S}\) given in \(C\).

Similarly, for each sequence diagram \(S \in \text{Se}(S)\) the following condition must hold: for each call action call\((\text{obj}, e)\) (resp. send action send\((\text{obj}, e)\)) sent out from an object \(O \in \text{Obj}(D)\) in \(S\) (for an object name \(\text{obj}\)), the object diagram \(C\) in \(S\) must have a «\text{call}» (resp. «\text{send}») dependency from the object \(O\) to the object \(\text{obj}\) or one of its interfaces supplying the operation \(\text{msgm}(e)\) (resp. able to receive the signal \(\text{msgm}(e)\)). The types of the message specifications in the class diagrams and those of the values in the sequence diagram must match.

**Behavioral Semantics of Subsystems**

The different subsystems and objects have their own input and output queues. Recall that there is only one input buffer (and one output buffer) for a given object or component. This buffer may be accessed in various ways – for example, concurrent substates of a statechart diagram read from the same input buffer and write to the same output buffer. That this happens consistently is ensured by the semantics.

The run-to-completion step for each subsystem is performed in parallel, each with its own dispatcher only dispatching events prefixed by the subsystem name. This joint run-to-completion step is composed sequentially with the execution of the scheduler that takes the events from the output queues of the client subsystems requesting a service from another object and distributes them to the input queues of the server subsystems requested to provide the service. For the formal semantics of this we use the corresponding concepts for UMSs defined for this purpose in Sect. 7.2.

Recall that following [KW01, p. 15], we do not model the creation and deletion of objects explicitly. A sufficient number of required objects is assumed to exist at the start of the execution; the activation of objects is controlled by the activity diagram in the subsystem. An object that reaches a final state within its top state is terminated (and may be reactivated).

We give the formal behavioral interpretation for subsystems.

Suppose we are given a consistent subsystem \(\mathcal{S}\). The behavioral interpretation of \(\mathcal{S}\) is defined to be the UMS \([\mathcal{S}] = (\text{name}(\mathcal{S}), \text{Comp}, \text{Sched}, \text{Links}, \text{Msgs})\) where:

- the set \(\text{Comp} \subseteq \text{UMNames}\) of UMS components is the set consisting of the names of the components in the deployment diagram \(\text{Dd}(\mathcal{S})\), where
for each $\mathcal{N} \in \text{Comp}$, its set $\text{Act}_\mathcal{N}$ of activities consists of the activities appearing in the activity diagram $\text{Ad}(\mathcal{S})$, and its set of attributes $\text{Att}_\mathcal{N}$ is the union of the sets of attributes of its activities,

- $\text{Sched}$ is the interactive ASM $[\text{Ad}(\mathcal{S})]_{AD}$ modeling the activity diagram $\text{Ad}(\mathcal{S})$,
- $\text{Links}$ is the set consisting of the links $l \in \text{Links}(\text{Dd}(\mathcal{S}))$ in the deployment diagram $\text{Dd}(\mathcal{S})$ and a link $l_{ST}$ for any two (possibly coinciding) subsystems or objects $S, T$ residing on the same node in $\text{Dd}(\mathcal{S})$, and
- $\text{Msgs}$ is the set $\text{Msgs}(\mathcal{S})$ of messages accepted by the subsystem $\mathcal{S}$.

By the assumption regarding activity diagrams in the section on consistency between UML diagrams above, activities can thus be defined as subsystems, statecharts, or object views of sequence diagrams contained in the subsystem $\mathcal{S}$, or can be defined directly as a UML Machine.

As usual, we assume that the names (except for the specified input and output names) of the interactive ASMs involved in the above definition are renamed to avoid unwanted name-clashes. For the attributes referred to in any statecharts above, we require more specifically that they are renamed by prefixing the attribute name with the name of the object it belongs to. This way, different statecharts modeling activities of the same object can all access its attributes. Note that no conflicts arise from the shared access to attributes even by concurrent activities, because according to the activity diagram semantics defined in Sect. 8.1.5, they are executed by interleaving them.

*Properties of the Semantics*

We discuss an important property of our semantics.

**Fact 8.1.** During each given execution of a UML specification, each occurrence of a message is created at at most one location in the specification.

Note also that each occurrence of a *call* or *send* action in a statechart or a message sent out in a sequence diagram actually adds a new occurrence of the corresponding message to the communication queues of the ASM system modeling the UML specification (rather than refer to an existing one), by definition of the rules $\text{ActionRuleSC}_S(a)$ for a *call* or *send* action $a$ and $\text{ActionRuleSD}(msg)$ for a message $msg$. Also, each occurrence of a message is consumed at at most one location in the specification.

In that sense, a message cannot be referred to more than once. This feature of our semantics is a restriction in so far as diagrams cannot be used in a way that permits “overlapping” in time. It is a simplification for us in that related questions of consistency within a single specification are avoided.
8.2 Development with UML

8.2.1 Refinement

System development is about turning an idea of what a system should accomplish into a product implementing the idea. This may be achieved by constructing a first abstract system specification satisfying the given requirements and by applying a number of successive transformations that add more detail while preserving the relevant requirements. This has been followed in the approach of stepwise development [Dij68, Wir71] (also called the top-down approach).

Changes to the system specification during the development process are supported by refinements. A refinement relates two descriptions of the same thing at two levels of detail, of which the more concrete one realizes the more abstract one.

Thus we have the corresponding notion of stepwise refinement: a complex problem is decomposed into smaller subproblems (and thereby simplified); subproblems are refined step by step and integrated to solve the original problem.

In practice, one often has to modify a part of a system to account for changes in the environment of this part or in the requirements on it: development is an “incremental production of a series of prototypes, which eventually evolve into the final implementation” [Boc91] (iterative development process).

In the latter case, refinement is usually not assumed to provide full behavioral conformance [HG97, p. 40] or even to preserve the exact structure of the refined subsystem. This applies in particular to the kinds of refinement proposed in the context of UML:

- In UML, refinement denotes a certain kind of dependency relation\(^8\) between model elements [Obj03, p. 2-18]. There is no constraint on the semantic relationship between the model elements. Examples of refinements in this general sense are state machine refinement and substitution [UML01, p. 2-178]. Some heuristics on how state machines can be refined are given on [UML01, p. 2-177]. Refining state machines corresponds to specializing the model elements whose behavior the state machines model.

- There is a related kind of dependency called realization which specifies a relationship between a specification model element and a model element that implements it. The implementation model element is required to support all of the operations or received signals that the specification model declares. Again there is no other constraint on the semantic relationship between the model elements.

On the other hand, in situations requiring high confidence that certain properties of a system are fulfilled, behavioral conformance of refinement can help to save effort to gain this confidence (be it by theorem proving, model

\(^8\) More precisely, it is a kind of abstraction.
checking, simulation, testing, etc.). For example, this is the case when there are stringent requirements on the security or safety of a system. The reasons are the following:

(1) It is often easier to verify system properties at a rather high degree of abstraction.

(2) If one has to make changes to the specification during the development process, without any behavioral conformance one would have to redo all the verification work which (for reasons pointed out in the previous point) has been done earlier in the process.

Thus, formal methods research has traditionally focussed on refinements that do preserve behavioral properties (see for example [Mil71, Hoa72, Jon72, Jon87, AL91]). In the context of object subtyping, this has been advocated for example in [LW94, PH97].

There seems to be a tension between flexibility of a refinement relation and the gain from establishing that a specification refines another (the trivial refinement relation that declares any system to be a refinement of any other system can be applied quite widely but is not very useful).

Since our focus is on the development of systems satisfying critical requirements (such as security requirements), we try to find the right trade-off by giving several kinds of refinement, some of which strictly preserve the behavior of the system, and others which allow for a modification in the behavior which is controlled in a way that allows one to reuse established knowledge on critical properties of the system. Of these, the more liberal kinds of refinement are especially useful in the early parts of system development (when the system is still subject to much change), and from one iteration in an iterative development process to the next. The stricter kinds are more useful in the later parts (when some properties have already been established that should be preserved), and within one iteration in an iterative process.

We introduce several kinds of refinement by referring to the corresponding definitions in Chap. 7 through the formal semantics defined in previous sections (in particular the UMS \([s]\) defined for a UML subsystem \(S\) in Sect. 8.1.7 and the associated UML Machine \(\text{Exec}(\([s]\))\) from Sect. 7.2).

The strictest kind of refinement is called behavioral refinement. This is essentially refinement by reverse subset inclusion of the sets of inputs and outputs (with variations on what part of the system behavior is included, called black-box refinement and white-box refinement). It reduces the possible behaviors of the overall system and preserves all safety properties [AS85] (since these are sets of input/output sequences), and thus also those security properties which may be expressed by safety properties (for a discussion of subtleties regarding this see Sect. 7.5.1). Its applicability may, however, be restricted, as mentioned above.

**Definition 8.2 (Black-box refinement).** For a set \(E \subseteq \text{Events of messages}, we say that the UML subsystem \(S'\) is a (delayed) \(E\)-black-box refinement of
the UML subsystem $S$ if the derived UML Machine $(\text{Exec}(S'), \text{inQu}_{\text{Exec}}(S'))$, \text{outQu}_{\text{Exec}}(S'))$ is a (delayed) $E$-refinement of the UML Machine $(\text{Exec}(S), \text{inQu}_{\text{Exec}}(S), \text{outQu}_{\text{Exec}}(S))$.

For example, given a set $M \subseteq \text{MsgNm}$ of message names, one may consider $E$-black-box refinements where $E \overset{\text{def}}{=} \{e \in \text{Events} : \text{msgnm}(e) \in M\}$. One can thus use the set of message names $M$ in order to hide the events with different message names with respect to the refinement (this is inspired by the corresponding operators in CSP [Ho85] and CCS [Mil89]).

**Fact 8.3.** (Delayed) $E$-black-box refinement of UML Machines is a preorder for each set of events $E \subseteq \text{Events}$.

While white-box refinement (defined below) preserves the internal structure of subsystems and thus the capabilities of a given kind of adversary, this is not so for black-box refinement. We define a notion of black-box refinement that is relative to types of adversaries. Note that for delayed refinement of UML Machines, the UML Machines involved are not assumed to be stutter-invariant (as in the case for UMSs); therefore it makes sense to consider delayed refinement of the composition of a system with an adversary (that cannot be assumed to be stutter-invariant).

**Definition 8.4.** Given a set of events $E \subseteq \text{Events}$, the subsystem $B$ is a (delayed) $E$-black-box refinement of the subsystem $A$ given adversaries of type $A$ if for every adversary $b$ of type $A$ for the UMS $[B]$ such that the derived UML Machine $(\text{Exec}(B'), \text{inQu}_{\text{Exec}}(B'), \text{outQu}_{\text{Exec}}(B'))$ is a (delayed) $E$-refinement of the UML Machine $(\text{Exec}(B), \text{inQu}_{\text{Exec}}(B), \text{outQu}_{\text{Exec}}(B))$.

We have the following result similar to Theorem 7.27.

**Theorem 8.5.** If the subsystem $S$ preserves the secrecy of $E$ from adversaries of type $A$ and $T$ (delayed) refines $S$ given adversaries of type $A$ then $T$ preserves the secrecy of $E$ from adversaries of type $A$.

The next kind of refinement, white-box refinement, preserves the system structure (such as the links between components), and considers the behavior of the components in a UML subsystem. In contrast, the black-box refinement defined above only considers externally visible behavior.

**Definition 8.6 (White-box refinement).** The UML subsystem $S'$ is a (delayed) white-box refinement of the UML subsystem $S$ if the derived UMS $[S']$ is a (delayed) refinement of the UMS $[S]$.

**Theorem 8.7.** (Delayed) white-box refinement of (stutter-invariant) UML subsystems is a precongruence with respect to composition by subsystem formation.
Definition 8.8 (White-box equivalence). Two subsystem specifications \( S \) and \( S' \) are (delayed) white-box equivalent if \( S' \) is a (delayed) white-box refinement of \( S \) and \( S \) is a (delayed) white-box refinement of \( S' \).

White-box equivalence can be used for example to verify consistency of two subsystem specifications that are supposed to describe the same behavior, for instance, one of which uses statecharts to specify object behavior, and the other a sequence diagram.

Corollary 8.9. (Delayed) white-box equivalence of (stutter-invariant) UML subsystems is a congruence with respect to composition by subsystem formation.

In practice, one often needs more flexible refinements that allow one to modify the subsystem’s interface (in the general sense). Interface refinement is a looser kind of refinement which allows a change in the external interface of the part of the system under refinement. To exhibit the extent to which behavioral properties are preserved under the refinement, interface refinement is parameterized by system parts relating a system to its refinement.

Definition 8.10 (Interface Refinement). Given UML subsystems \( S \) and \( S' \) and a parameterized UML subsystem \( I'(Y) \), \( S' \) is a (delayed) \( I' \)-interface refinement of \( S \) if \( S' \) is a (delayed) white-box refinement of \( I'(S) \).

This definition allows one to handle the trade-off between the generality of a refinement relation and the degree to which it preserves system properties in a very flexible way. It is motivated by the observation that, in practice, subsystems are often reused as part of their refinements (a well-known example is the wrapper facade pattern where subsystems are refined by encapsulating them in other subsystems [SSRB00]).

Theorem 8.11. Each (stutter-invariant) UML subsystem \( S \) is a (delayed) \( Id \)-interface refinement of itself, where \( Id(Y) \) \( \equiv Y \).

For all UML subsystems \( S \), \( S' \), and \( S'' \) such that \( S' \) is a (delayed) \( I \)-interface refinement of \( S \) and \( S'' \) is a (delayed) \( I' \)-interface refinement of \( S' \), \( S'' \) is a (delayed) \( I' \circ I \)-interface refinement of \( S \), where \( I' \circ I(Y) \equiv I'(I(Y)) \).

A more liberal kind of refinement is that of a pattern-based transformation. Patterns [GHJV95] encapsulate the design knowledge of software engineers in the form of recurring design problems. Here the developer may construct a refinement by applying a predefined transformation together with results on the preservation of behavior provided by this transformation (which may be either defined for this purpose or reused from other work). This kind of refinement is the most application-dependent; we consider it in the context of secure systems development in Sect. 4.3.

An extended example of the application of refinement is given in Sect. 5.1.
8.2.2 Rely-Guarantee Specifications

To reason about system specifications in a modular way, one may usefully employ rely-guarantee specifications. The following definitions are again adapted from Chap. 7.

**Definition 8.12.** Given a UML subsystem $S$ and sets $R, G$ of sequences of event multi-sets, we say that $S$ fulfills the rely-guarantee specification $(R, G)$ if the derived UML Machine $(\text{Exec}(S), \text{inQu}_{\text{Exec}(S)}, \text{outQu}_{\text{Exec}(S)})$ fulfills $(R, G)$.

**Theorem 8.13.** Suppose that the UML subsystem $S$ fulfills the rely-guarantee specification $(R, G)$ and that $R \cap \mathcal{E} = R$ and $S \cap \mathcal{E} = S$.

If the UML subsystem $S'$ E-black-box refines $S$ then $S'$ fulfills the rely-guarantee specification $(R, G)$.

If the UML subsystem $S'$ delayed E-black-box refines $A$ and $G$ is stutter-closed then $S'$ fulfills the rely-guarantee specification $(R, G)$.

In particular, (delayed) white-box refinement of UML machine systems preserve rely-guarantee specifications by Theorem 7.19.

8.3 Notes

There has been a considerable amount of work on a formal semantics for various parts of UML. [FELR98, EFLR99] discuss some fundamental issues concerning a formal foundation for UML; [KER99, RW99, RACH00] point out some related problems. [BGH'98] uses a framework based on stream-processing functions. [GPP98] employs graph transformations. [LP99] gives a formalization of UML state machines. It does not include the other kinds of diagrams considered here, and does not treat arguments of actions. [RACH00, RCA00] give an approach using algebraic specification. [MS00] translates UML class diagrams to B abstract machines. [ML02] defines transformation rules for OCL constraints into the formal method B. [BD00b] gives a translation of statecharts into the process algebra CSP. [BCR00] uses ASMs for UML statecharts; [Cav00] also contains formal semantics for other kinds of diagrams. [OP00] considers interacting UML subsystems, but without giving a formal semantics. [Ste01a] give a semantics for use case diagrams based on transition systems. A combined formal semantics for UML statecharts and class diagrams has been given in [RCA01]. [Mer02, SKM01] gives a semantics for statecharts and show exemplarily how to check whether a set of statecharts satisfies a collaboration. [Krü02] gives a formal foundation for services in UML and UML-RT. [DGH02] defines a translation of UML statecharts to UPPAAL timed automata. [ZG02] defines an extension of OCL with temporal logic. [Jan02] proposes a probabilistic version of UML statecharts. [WS02] gives results on compile-time scope resolution for statechart
transitions. [BB03] defines a formal notion of refinement of UML diagrams. [VP03] applies an approach for automated formal verification of model transformations to a transformation from UML statecharts to Petri nets. [Stō03] considers assertion, negation, and refinement in UML interaction specifications. [BG03] examines UML actions and activities.

[Jür02a] presents the statechart semantics and [Jür02d] the semantics for UML subsystems given in this chapter.

There has also been a significant amount of work on the semantics for formalisms related to UML, including work regarding Message Sequence Charts in [Krü00].

Refinements have been investigated in the object-oriented setting, for example in [DB00, DS00a], where the introduced structural refinement has a similar motivation to our interface refinement; for an overview see [DB01]. A further discussion on refinement in UML can be found in [HK98].

8.4 Discussion

We defined a formal semantics for a (restricted and simplified) part of UML using UML Machines and UML Machine Systems. It gives a precise meaning to groups of diagrams of various kinds gathered in a restricted version of UML subsystems. Actions and internal activities are modeled explicitly (rather than treating them as atomic given events). In particular, objects, and more generally system components, can communicate by exchanging messages with parameters, which can be used in the subsequent execution. The formal semantics for subsystems and their interactions seems to be the only one published so far, although again the notion of subsystem considered here is restricted, for example in the kinds and numbers of diagrams that may be contained.

This approach aims to prepare the ground for executable modeling of larger parts of UML specifications.

We gave supplementary results for formal UML development, such as consistency conditions for different diagrams in a UML specification, notions of refinement, behavioral equivalence, and rely-guarantee specifications, with nice structural properties (such as substitutivity).

A long-term goal of this research is to raise industry acceptance of formal methods, by integrating formal methods with a standard development method, as postulated for example in [CW96]. Also, since UML is part of many computer science and software engineering curricula and since it largely comprises already established kinds of notations, one might expect developers to know the notation, thus reducing costs. A UML specification of a system to be analyzed may already be available, or can be delivered. Anecdotal evidence suggests that at least parts of a system specification are formulated in a notation that is at least similar to UML (for example, the protocols in the CEPS considered in Sect. 5.3 are specified in a notation similar
to sequence diagrams). Also, the formal semantics defined here aims to contribute toward the development of tool support for UML specifications able to check behavioral properties. With such tool support, there might be more incentive for developers to write a UML specification for critical parts of their systems. This might then contribute to our more general aim to widen the impact of formalism on software development processes in practice, beyond the application domain of security-critical systems.

The formal semantics delivers the required mathematical foundation to reason about subtle behavioral properties such as security requirements. Since our semantics builds on UML Machine Systems, it allows us to make use of the treatment of security-critical systems in Sect. 7.5 to evaluate UML specifications for security in the following chapters.

UML Machines appear to be an adequate tool to handle the complexities in defining the semantics, due to their flexibility and expressiveness.

While we considered only a restricted version of the UML syntax, we believe that extending the work to include other aspects should be possible in principle (but note that this may cause an increase in complexity and therefore possibly an increased challenge when performing manual reasoning or trying to provide tool support). Since the UML definition itself is inconsistent in several ways, it may not be possible to extend our semantics consistently to all of UML as it is presently defined. It remains a topic of further research to determine to what an extent the UML definition could feasibly and usefully be given a more complete formalization. To demonstrate that our choice of a subset of UML is reasonable and our semantics of sufficient interest, we present several case studies in Chap. 5, some of which are taken from industrial applications and therefore of a more realistic size and complexity than some of the examples of security-critical systems considered in the literature.
Part IV

Epilogue
Further Material

In this chapter, we give a short overview of material related to the content of this book.

We start by listing some more material within the UMLsec approach that had to be omitted for space reasons and then give an overview of other approaches.

9.1 More on UMLsec

The following further material related to UMLsec has to be omitted here:

- [Jür03b] gives more detail on specification-based testing for critical systems with UML.
- [Jür02e] uses UMLsec to provide formally based development methods for CORBA-based applications.
- [Jür02b, Jür03c, Jür03d, Jür04] demonstrate how to generalize the approach presented here to develop systems with other criticality requirements (such as safety-critical or performance-critical systems) using an appropriate extension of UML.
- [Jür02g] presents applications of UMLsec in the telemedicine application domain.
- [JPW02] gives more results about using security patterns in model-based development in the context of UMLsec.
- [JPW03, PJWB03, BBH+03] give some results on development methods for security-critical systems using UMLsec. In particular, they give an extension of a process for the use-case-based elaboration of requirements for security-critical systems. They show a methodical concept for the development of security-critical systems and the modeling of security aspects in the application core with UMLsec. Furthermore, they introduce security use cases for the development of security aspects in conjunction with behavioral modeling.
• [Jür03a] explores the notion of algebraic state machines (similar to the UML Machines considered here) and applications to security.
• [JG03] gives a case study for applying UMLsec in the automotive domain.
• [JHO3, HJG3] show how to combine the use of UMLsec with model-based risk assessment.

9.2 Other Approaches to Security Engineering

We give an overview of other approaches to security engineering using formal methods or UML. Here we only give a general overview of topics relevant to the main focus of the UMLsec approach; more specific references are given in the notes sections of the preceding chapters.

9.2.1 Formal Methods Applied to Security

There has been extensive research in using formal models to verify secure systems. Early influential work was given in [DY83].

[Sch90] gives a confidentiality property preserved under refinement. However, cryptographic primitives are not considered and it is pointed out that their treatment may be less straightforward.

[HPS01] gives a necessary and sufficient condition for a notion of confidentiality to be preserved under refinement in a probabilistic extension of CSP.

[Lam73] drew attention to covert channels; this initiated early influential work on secure information flow in [GM82, GM84]. An overview of secure information flow (and other formal security models) can be found in [McL94].

For a discussion of the refinement of secure information flow properties cf. [GCS91, Mea92, McL94, McL96]. [RWW94] avoids the “refinement problem” by giving a security property that requires systems to appear deterministic to the untrusted environment. Special refinement operators that preserve information flow security are considered for example in [Man01].

The problem in how far formal models of cryptography are faithful to computational models is considered in [AJ01].

Many references attribute the approach for using formal methods to analyze abstract models of cryptographic software as influenced by the early reference [DY83].

[Mea95] gives short overviews and points out open problems. For an overview of the work on verifying security protocols with a focus on the process algebra CSP see [RSG+01].

Overviews of applications of formal methods to security protocols can be found in [Mea95, GSG99, Aba00, Mea00] and [RSG+01, Chap. 9] (see there for more details than we can present here). Our overview is based on that in [GJW03]. Because of the amount of material in this area, it is virtually impossible to present a complete overview, and this is not attempted here.
9.2 Other Approaches to Security Engineering

Roughly, one can try to classify the different approaches into the following categories:

**Intensional methods** model the behavior of the protocol participants together with an attacker (who can perform well-defined actions such as eavesdropping, storing, deleting, and inserting messages at a communication link, but is usually assumed not to be able to break cryptographic mechanisms). In these approaches, one often starts with the specification of the protocol, which is usually relatively straightforward. The security requirements are formulated by referring to this specification. It is then established whether the possible behaviors of the model give rise to violations of security goals. For this goal, one can use different techniques.

- **State-space search** is an approach for constructing and analyzing all possible attack scenarios used for example in model checking. Examples include [MCF87, Kem89, Men96, FGG97, MMS97, GL97, Eck98, DFG99, JW01b]. The process algebra CSP has been employed successfully, for example in [Low96, LR97, Low98, RSG+01]. There exist specification languages tailored to security protocols (including [Low98, BMM99]), which translate abstract protocol models to low-level specifications that can then be verified.

- **Proof-construction methods** are used to establish the absence of attacks relevant to the adversary model by mathematical proof (which may be mechanically assisted). Examples include [Sch96, THG98]. [AG99] introduces the spi calculus. An inductive method of proving protocols correct using a mechanical proof assistant is explained for example in [Pau98b]. [KAH99, Hei01] use the Software Cost Reduction toolset. There has been some work using ASMs reported in [BR97, BR98]. In [Lot00], threat scenarios are used to formally develop secure systems using the stream-based specification language **FOCUS** [BS01].

**Extensional methods** focus more on the security requirements, rather than the protocol specification. They often use specialized logics to model and analyze security protocols, often by modeling the changing knowledge and beliefs of the protocol participants during the execution of the protocol. Formulating security properties is often intuitive and elegant, while specifying the protocol may be more indirect than with intensional methods. The most famous example is probably the BAN logic (named after its inventors Michael Burrows, Martin Abadi, and Roger Needham) [ABN89]. In the BAN logic, one can formulate statements such as “P sees X” (meaning that P has received X), “P believes X” (P is led to believe that X is true), “X is fresh”, “K is a good key for communication between P and Q”, and so on. One can use logical inferences to construct the set of statements which hold about a protocol according to the logic. For example, if P receives X encrypted under the key K, and also believes that K is a good key for communication with Q, then P believes that Q said X. In particular, one can verify that a certain security
property, formalized as a BAN logic statement, holds for the protocol according to the logic. There exist several extensions to the BAN logic, including [GNY90, SvO94, KN98].

A formal approach close to UMLsec is that using the computer-aided software engineering (CASE) tool AutoFocus [HMR+98, SH99, RJW+03]. Similar to the UMLsec approach presented in this book, cryptographic systems can be specified with diagrams similar to UML sequence diagrams and statecharts and examined for security weaknesses using the model-checker SMV included in AutoFocus [JW01b, WW01]. Additionally, the specifications can be simulated or tested. In a particular application [VWW02], a secure electronic purse application for personal digital assistants (PDAs) has been developed using the AutoFocus approach following a development process based on the Common Criteria [CC01].

Related to the work using AutoFocus is research using the formal method FOCUS involving stream-processing functions, on which AutoFocus is based. [Jür01g] extends FOCUS by cryptographic operations including symmetric and asymmetric encryption and signing. [Jür01b] examines composability of secrecy. [Jür00] considers secure information flow.

Whereas, traditionally, most of the research has been applied to security protocols (or the secure flow of information in multi-level secure systems) and has been confined to an academic environment, there have been advances to widen the scope of application of formal methods in security, and to put more emphasis on usability of the research in industry. For example, [Hei01] stresses the need for formal approaches to be practical in their application. [GHR03] uses a model-checker to analyze Linux configurations.

### 9.2.2 Software Engineering and Security

Compared to research done using formal methods, less work has been done more generally using software engineering techniques for computer security (examples are [Eck95, EM97, DFS98], for an overview of the topic see [DS00b]). [And94] suggests using software engineering techniques to ensure security. Security of object-oriented systems has been considered in [JKS95, SBCJ97].

### 9.2.3 Other Approaches Using UML

By now, there exist several lines of research toward using UML for security systems development. To the extent known to the present author, they differ from the one herein that they, currently, usually aim to cover a less comprehensive set of security requirements (mostly focussing on role-based access

---

1 A German introduction is contained in [BS03b].
control requirements). Also, most of them do not attempt to perform an analysis of the security requirements based on a formal semantics of a simplified fragment of UML.

[HF97] extends use cases and interaction diagrams to support distributed system architecture requirements. [FH97] proposes a method determining role-based access rights. Use cases are extended with rights specifications and the rights of a role are derived from the use cases. The method thus enforces the design principle of least privilege. Work on security patterns using UML includes [Fer99, FP01]. [Low00] uses UML for requirements capture for security protocols.

Some other approaches have been discussed at recent workshops on the topic [JCF+02, JFS04, JRFF03]:

- [GFR02] demonstrates how to use UML for aspect-oriented development of security-critical systems. Design-level aspects are used to encapsulate security concerns that can be woven into the models. In [GFR03], authentication mechanism models are considered in an abstract aspect model and more detailed models are created from these. The models can be composed with primary decomposition models, allowing system architects to analyze different mechanisms to realize a particular concern, such as authentication. [RFGL03] proposes to use aspect-oriented modeling for addressing access control concerns. Functionality that addresses a pervasive access control concern is defined in an aspect. The remaining functionality is specified in a so-called primary model. Composing access control aspects with a primary model then gives a system model that addresses access control concerns. [RLKF03] uses a variant of UML to model Role Based Access Control and Mandatory Access Control and to compose access control policy frameworks.

- [HdBL02] uses UML for the risk assessment of an e-commerce system within the CORAS framework for model-based risk assessment [DRR+02, FKG+02, AddB+02]. This framework is characterized by an integration of aspects from partly complementary risk assessment methods. It incorporates guidelines and methodology for the use of UML to support and direct the risk assessment methodology as well as a risk management process based on standards such as AS/NZS 4360 and ISO/IEC 17799. It uses a risk documentation framework based on RM-ODP together with an integrated risk management and system development process based on UP and offers a platform for tool inclusion based on XML. [HH03] presents SecurityAssessmentUML, a UML profile for model-based security assessments, as well as a security assessment process and its associated documentation framework. The main objective is to support documentation of output based on risk identification and risk analysis in a security assessment. The profile supports specification of concrete scenarios demonstrating how attacks may occur, as well as a combination of fault trees and activity diagrams for analyzing the frequency of risks.
9 Further Material

- [FMMMP02] uses UML for the design of secure databases. It proposes an extension of the use case and class models of UML using their standard extension mechanisms (stereotypes, tagged values, and constraints) designing secure databases. It uses an OCL-based language for specifying security constraints called OSCL. The paper demonstrates how to use the methodology to classify information into different sensibility levels and to specify which user roles will be able to access the information.

- [KPP02, BKL02] demonstrate how to deal with access control policies in UML. The specification of access control policies is integrated into UML. A graph-based formal semantics for the UML access control specification permits one to reason about the coherence of the access control specification.

[Blo02] uses UML for modeling security-critical systems in the health sector. [LBD02, BDL03] show how UML can be used to specify access control in an application and how one can then generate access control mechanisms from the specifications. The approach is based on role-based access control and gives additional support for specifying authorization constraints. [AW03] suggests a method for specifying access control policies with UML use cases and proposes a methodology to resolve some issues of consistency and completeness of access control specifications.

Internationally, several research projects exist on the topic of secure systems development with UML (including the European working groups CORAS [Sto01], NEPTUNE [nep01], and DEGAS [DEG01], as well as one German project [arc01]).

9.2.4 Other Non-functional Requirements

[LLK+02] proposes a basis for partially automated risk analysis in early development phases with UML. [PMP01, HG02, HTB03] propose to use UML for safety-critical systems development. [Pat02] presents a methodology to extend the OMG General Resource Modeling sub-profile to model component faults in a UML design. [Pet02] proposes a strategy for development of critical control systems using UML. [Sel03] discusses the use of modeling techniques in safety-critical systems design. [CLM+03, LMM03, vKSZ03] report on the usage of UML for real-time systems. [XLL03] deals with software performance prediction based on UML models. [RJW+03] applies the AutoFocus tool (see Sect. 9.2.1) to embedded systems.
Outlook

The method for model-based development of security-critical systems proposed here has been successfully applied in security-critical projects in industry. Examples include the following:

- The project FairPay [Fa01] funded by the German Ministry of Economics and involving the two largest German banks and the largest German telecommunications company, among others. The use of UMLsec there included an analysis of the Common Electronic Purse Specifications under development by Visa International and others.
- The project Verisoft [Ve03] funded by the German Ministry of Education and Research and involving the largest German telecommunications company, the largest German chip manufacturer, and a large German car company.
- Projects with a large German bank.

In particular, these experiences indicate that the approach is adequate for use in practice, once developers are taught how to use it. In that respect, it is important to note that, as a research-level book, a significant part of the material presented here is not targeted at normal users of UMLsec and the associated tool-support, but at researchers interested in extending UML in a similar way, and in themselves providing sophisticated tool-support for UML. It seems reasonable that basic usage of UMLsec can in fact be taught to developers in industrial practice. A beginning has already made in a series of tutorials on UMLsec [Jür04].

The ideas presented here are by no means exhaustive. They can, for example, be combined with many of the other approaches for using UML for developing security-critical systems mentioned in Sect. 9.2.3 (advances in this direction include [FJ02, HJ03b, JH03]). Efforts are under way to create a standard UML extension for secure systems development.

Given the current state of computer security in practice, with many vulnerabilities reported continually, it seems to be a promising idea to apply model-driven development to security-critical systems, since it enables developers
who are not experts in security to make use of security engineering knowledge encapsulated in a widely used design notation. Since there are many highly subtle security requirements which can hardly be verified with the “naked eye”, even security experts may profit from this approach. Although the approach explained here puts some emphasis on the weaknesses arising from the design level, it can also be used for analyzing code for security weaknesses using model-based security testing (as explained in Sect. 6.4), and combined with tools analyzing data arising during the deployment of a system (such as access rights, as demonstrated in Sect. 6.3 in the context of SAP systems). And although the fragment of UML considered here is restricted, and its use more disciplined than in average current industrial usage of UML, in order to allow treatment using advanced tool support, the industrial case studies mentioned above indicate that our usage of UML is sufficient for our needs.

Since UML is widely taught, even a relatively restricted and more disciplined use of UML may be easier to learn and to use than a completely different formal method. When analyzing a system for security requirements, there may already be a specification in UML or enough knowledge in UML available to enable constructing one without too much further training, reducing costs. Thus a more disciplined use of a restricted fragment of UML for secure systems development as proposed in this book may be more cost-effective than having to teach a completely different formal method. In particular, this is supported by the tools for automatic analysis of UMLsec models currently in development (some of which we presented in Sect. 6.2). This may assist in transferring ideas and results from formal methods to industrial practice, at least in a way that is complementary to the usual methods of quality assurance by testing. Thus one can avoid mistakes that are difficult to find by testing alone (such as breaches of subtle security requirements), as well as the disadvantages of the “penetrate-and-patch” approach. Since considering security aspects early in the system life-cycle may significantly reduce costs, there is a potential for developing secure systems in a cost-efficient way.

Beyond security, the UMLsec approach has also been successfully generalized to Critical Systems Development with UML (CSDUML), covering other application domains involving intricate non-functional requirements, including safety-critical [Jü03c, Jü02b, Jü04, JH03, Jü03b, Jü03d] and mobile systems [JK03].
Part V

Appendices
Towards UML 2.0

At the time of writing, the next major version of the Unified Modeling Language, UML 2.0, is being finalized. In this chapter, we shortly sketch how to accommodate the changes from the current version, UML 1.5, with respect to the approach proposed in this book. A good introduction to UML 2.0 and a list of the changes between the different UML versions is contained in [Fox04].

Most changes do not have a significant impact on our approach: For example, object diagrams, which already exist in UML 1.5, are in UML 2.0 more explicitly defined. In UML 1.5, packages can be contained in class diagrams, while in UML 2.0 such diagrams are now independently named as package diagrams. The UML 1.5 collaboration diagrams are called communication diagrams in UML 2.0. The UML 2.0 interaction overview diagrams are a new kind of diagram integrating activity and sequence diagrams. While in UML 1.5, the activities in activity diagrams can already be defined using other diagrams (such as sequence diagrams), this link can be made more explicitly in UML 2.0 by actually including the sequence diagrams in the respective activity states. Timing diagrams are a kind of diagram which is entirely new to UML with version 2.0. These diagrams, which will be familiar to many hardware engineers from electronic engineering, do not seem to be particularly specific to secure software engineering.

Composite structure diagrams have been included in UML 2.0 from the real-time UML-RT extension. They contain parts represented by rectangles that may be connected by connectors drawn as lines between parts. These diagrams are useful for specifying component structures and hierarchies within components. In that, they are similar to the static structure diagrams of UML 1.5, which can contain subsystems that may themselves in turn contain static structure diagrams, and to component models, which can be part of deployment diagrams. Thus, stereotypes such as `secrecy`, `integrity`, `authenticity`, and `high` defined for dependencies in static structure diagrams and deployment diagrams can also be applied to the connectors in composite structure diagrams. Also, parts can be marked as `critical` (such as class models or subsystems in static structure diagrams). Although the
UMLsec notation can be extended quite nicely to UML 2.0 composite structure diagrams, one should note that this information can already be expressed in UML 1.5 static structure and deployment diagrams, so using UML 1.5 is not a restriction in this respect. Note that while deployment diagrams and component diagrams are integrated in UML 1.5, they are written in different diagrams in UML 2.0.

UML 2.0 also adds quite a few new model elements for existing diagram types. These include state machine extensions, gates in interaction diagrams, and power types in class diagrams. They are not very particular to secure software engineering but are also not in conflict with the UMLsec notation and can be used within the context of the approach presented in this book without any problems. Similarly, some existing model elements have been changed, but most of these do not appear in our treatment in this book at all. In particular, we do not make use of any UML 1.5 model elements that have been dropped in UML 2.0. In UML 2.0 sequence diagrams, interaction frames extend the guards used in UML 1.5 sequence diagrams to specify conditional behavior. Again, this extension can be used with UMLsec as well, although it remains to be seen in which situations the added expressivity outweighs the increase of complexity in the notation. Stereotypes are in UML 2.0 more tightly defined than before and exclude a previous usage as a kind of keywords (but include the usage of stereotypes in UMLsec). Activity diagrams are defined more liberally in UML 2.0 than before: While UML 1.x views activity diagrams formally as a special case of statechart diagram, this imposes some constraints on the structure of a diagram that are removed in UML 2.0, for example that forks and joins have to match. To accommodate this liberalization, the semantics is now formulated in a Petri-net style by referring to token flows. It seems that this is done in a way that preserves backward compatibility; in the sense that the semantics of a UML 2.0 activity diagram that is also a well-defined activity diagram in UML 1.5 is logically equivalent to its UML 1.5 semantics (although it would require deeper investigation to confirm this). New Swimlanes can be multidimensional in UML 2.0 and are called partitions.
The Semantics of UML Machine Rules

The following definitions are based on those for Abstract State Machines in [SSB01, BS00].

We write $\zeta[x \mapsto a]$ for the variable assignment which coincides with $\zeta$ except that it assigns the element $a$ to the variable $x$. Thus:

- $\zeta[x \mapsto a](v) = a$ if $v = x$
- $\zeta[x \mapsto a](v) = \zeta(v)$ otherwise.

We also write $\text{BaseSet}(A)$ for the base set and $\text{Voc } A$ for the set of function names of a UML Machine $A$.

**Definition B.1 (Update).** An update for a UML Machine $A$ is a triple $(f, a_1, \ldots, a_n, b)$, where $f$ is an $n$-ary function name, and $a_1, \ldots, a_n$ and $b$ are elements of $\text{BaseSet}(A)$.

Thus an update means that the interpretation of the function $f$ in $A$ has to be changed at the arguments $a_1, \ldots, a_n$ to the value $b$. An update set is a set of updates.

For two update sets $U, V$, we define the update set $U; V$ (a followed by $V$) as follows: $U; V \overset{\text{def}}{=} \{ (f, a, b) \in U : \exists c. (f, a, c) \in V \} \cup V$.

For a rule $R$ and $n \geq 1$, we write $R_n$ for the rule seq $R \ldots R$ endseq that iterates $R$ $n$ times.

A transition rule of a UML Machine produces, in any given state, an update set for each variable assignment. Recursive calls to other rules are allowed; thus it is possible that a rule has no well-defined semantics at all. We give a calculus to define the semantics of transition rules in Fig. B.1.

**Definition B.2 (Semantics of transition rules).** The semantics of a transition rule $R$ of a given UML Machine in a state $S$ with respect to a variable assignment $\zeta$ is defined if and only if there exists an update set $U$ such that $[R]_\zeta^A \vdash U$ can be derived in the calculus in Fig. B.1.
\[
\begin{align*}
\text{[skip]} & \xrightarrow{\gamma} \emptyset \\
U(t) & \xrightarrow{[f(a,b)]} [R_1] \xrightarrow{\cdots} [R_n] \xrightarrow{U_n} \\
\text{do-in-parallel} & \ x_1 \ldots x_n \ x_{n+1} \ x_{n+2} \ x_{n+3} \ x_{n+4} \xrightarrow{U_1} \\
\text{if } & \ x_2 \ \text{then } R_1 \ \\
\text{if } & \ x_3 \ \text{else } S_1 \ \xrightarrow{U} \\
\text{if } & \ x_2 \ \text{then } R_2 \ \xrightarrow{\cdots} U \\
\text{choose } & \ x_2 \ \text{with } g \ \xrightarrow{U} \\
\text{forall } & \ x_2 \ \text{do } R \ \xrightarrow{U} \\
\text{seq } & \ x_2 \ \text{and } \ x_3 \ \text{do } R \ \xrightarrow{U} \\
\text{iterate } & \ x_2 \ \text{with } \ x_{n+1} \ \xrightarrow{U} \\
\text{if } \ & a = [t] \text{ and } b = [s] \\
\text{if } & \ [g] = \text{true} \\
\text{if } & \ [g] = \text{false} \\
\text{if } & \ \exists a \ \text{with } [g] = \text{true} \\
\text{if } & I = \{ a \in \text{BaseSet}(A) \mid [g] = \text{true} \} \\
\text{if } & \ \exists n \geq 0 : U_n = U_{n+1} \\
\end{align*}
\]

Fig. B.1. The semantics of UML Machine rules

It is possible that the update set \([R]_A\) contains several updates for the same function name \(f\). Then the updates have to be consistent, otherwise the execution stops.

Note that there can be different update sets \(U\) such that \([R]_A \triangleright U\) is derivable in the calculus (because of the non-determinism introduced by the choose with do rule).

**Definition B.3 (Consistent update set).** An update set \(U\) is called consistent if it satisfies the following property:

\[
\text{If } (f, (a_1, \ldots, a_n), b) \in U \text{ and } (f, (a_1, \ldots, a_n), c) \in U, \text{ then } b = c.
\]

Thus a consistent update set contains for each function and each argument tuple at most one value.

If an update set \(U\) is consistent, it can be fired in a given state. The result is a new state in which there may be function names the interpretations of which are changed according to \(U\).

**Definition B.4 (Firing of updates).** The result of firing a consistent update set \(U\) in a state \(S\) is a new state \(T\) with the same base set as \(S\) satisfying the following two conditions for the interpretations of function names \(f\) of Voc \(A = \text{Voc } B\):

- If \((f, (a_1, \ldots, a_n), b) \in U\), then \(f^T(a_1, \ldots, a_n) = b\).
- If there is no \(b\) with \((f, (a_1, \ldots, a_n), b) \in U\), then \(f^T(a_1, \ldots, a_n) = f^S(a_1, \ldots, a_n)\).
Definition B.5 (Run of a UML Machine). Let $M$ be a UML Machine with vocabulary $\Sigma$, initial state $S$, and main rule name $R$. Let $\zeta$ be a variable assignment. A run $r \in \text{Run}_M$ of $M$ is a finite or infinite sequence $S_0, S_1, \ldots$ of states for $\Sigma$ such that the following conditions are satisfied:

- $S_0 = S$.
- For each $n \in \mathbb{N}$, if $S_n$ is the last element of the sequence $r$ then
  - for any update set $U$ with $[R]_\zeta^A \triangleright U$, applying $U$ leaves the state $S_n$ unchanged, or
  - there exists an inconsistent update set $U$ with $[R]_\zeta^A \triangleright U$.
- For each $n \in \mathbb{N}$, if $S_n$ is not the last element of the sequence $r$, then there exists a consistent update set $U$ with $[R]_\zeta^A \triangleright U$ in $S_n$ such that $S_{n+1}$ is the result of firing $U$, and we have $k > n$ such that $S_k \neq S_n$. 

C

Proofs

We give here the formal proofs for the statements from Chaps. 8 and 5.

Note that it is not intended to propose manual reasoning as in the proofs below to be used in the context of security engineering with UMLsec in practice. Instead, tool support for analyzing UMLsec specifications should be used as discussed in Chap. 6. Manual proofs are presented here to demonstrate that UMLsec is suitable overall to express important security properties in a way that allows detailed formal security analysis.

C.1 Secure Channels

**Proposition 5.1.** $C$ preserves the secrecy of the variable $d$ from adversaries of type $A = \text{default}$ with specified previous knowledge $K^n_A$, given inputs from $\text{Data \setminus K}^n_A$. Note that, intuitively, this proposition is obvious, because the adversary cannot read the channels. We give the proof to illustrate how to apply the formal framework.

*Proof.* We have to show that for every expression $E$ which is a value of $d$ at any point, $C$ preserves the secrecy of $E$.

We use Theorem 7.26. Since an adversary of type $\text{default}$ cannot access any of the components or links in $C$, we have $K_A(C) = K^n_A$ (because there is no read access), and $d$ takes values only in $\text{Exp \setminus K}^n_A$ (because there is no write access).

Thus for every expression $E$ which is a value of $d$ at any point, $C$ preserves the secrecy of $E$, by definition of preservation of secrecy.

**Proposition 5.2.** $C'$ is a delayed black-box refinement of $C$ given adversaries of type $A = \text{default}$ with

$$K^n_A \cap \{K_S^{-1} : (K_R') \cup \{K_n, \{x \vdash n\}_{K_n} : x \in \text{Exp} \land n \in \mathbb{N}\} = \emptyset$$

and for which $\text{Sign}_{K_R^{-1}}(K' \vdash n) \in K^n_A$ implies $K' = K_n$. 
Proof. For readability, we give the proof (as the other proofs in this appendix) at a rather abstract level, but in a mathematically precise way, because of space restrictions, and because it may be considered more accessible than a formal derivation at the level of the formal semantics using ASMs. The proofs make use of elementary techniques from universal algebra (see [Jur01f] and references therein). Tool-supported reasoning (such as the model checking of security properties [JW01b]), which is the long-term goal of this work but currently beyond the present scope, would of course need to refer to the formal semantics more directly.

We have to show that for every adversary $b$ of type $A$ for the ASM system $[C']$ there exists an adversary $a$ of type $A$ for the ASM system $[C]$ such that the derived interactive ASM $(\text{Exec}([C'], b), \{\text{inQu}_{\text{Exec}}([C']), \{\text{outQu}_{\text{Exec}}([C']), \})$ is a delayed black-box refinement of the interactive ASM $(\text{Exec}([C], a), \{\text{inQu}_{\text{Exec}}([C]), \{\text{outQu}_{\text{Exec}}([C]), \})$.

Note that $\mathcal{K}_A(C')$ is contained in the algebra generated by $\mathcal{K}_A^0 \cup \{\text{Sign}_{K^+_R}(k_j \cdot j)\}_{K_S}$ and the expressions $\{d :: n\}_K$ for inputs $d$. Firstly, the adversary can obtain no certificate $\{\text{Sign}_{K^+_R}(k :: j)\}_{K_S}$ for $k \neq k_j$, because the Receiver object only outputs the certificates $\{\text{Sign}_{K^+_R}(k_j \cdot j)\}_{K_S}$ (for $j \in \mathbb{N}$) to the Internet. Secondly, the sender outputs only messages of the form $\{d :: n\}_K$ to the Internet, for inputs $d$ and any $k \in \text{Keys}$ for which a certificate $\{\text{Sign}_{K^+_R}(k :: n)\}_{K_S}$ has been received. Here $k$ must be $K_n$ since no other certificate can be produced (since the key $K^+_R$ is never transmitted). Note also that $\mathcal{K}_A^p = \mathcal{K}_A^0$ since there are no components accessed by the adversary.

Also, the values that an adversary for $C'$ may insert into the Internet link may only delay the behavior of the two objects regarding $\text{outQu}_C$, since the adversary has no other certificate signed with $K^+_R$ and does not have access to the key $K^+_R$, and because of the transaction numbers used. Thus any other value inserted is ignored by the two objects.

For any adversary $b$ for $C'$ we can thus derive an adversary $a$ for $C$ by omitting insert and read commands such that the interactive ASM $(\text{Exec}([C'], b), \{\text{inQu}_{\text{Exec}}([C']), \{\text{outQu}_{\text{Exec}}([C']), \})$ is a delayed black-box refinement of the interactive ASM $(\text{Exec}([C], a), \{\text{inQu}_{\text{Exec}}([C]), \{\text{outQu}_{\text{Exec}}([C]), \})$ (since the outputs to and $\text{outQu}_C$ (resp. $\text{outQu}_{C'}$) are stutter-equivalent).

**Proposition 5.3.** $C'$ preserves the secrecy of the variable $d$ from adversaries of type $A = \text{default}$ with

$$\mathcal{K}_A^p \cap (\{K^+_S, K^-_R, K\} \cup \{x :: n\}_K : x \in \text{Exp} \land n \in \mathbb{N}) = \emptyset$$

and for which $\text{Sign}_{K^+_R}(K') \in \mathcal{K}_A^p$ implies $K = K'$, given inputs from $\text{Data} \setminus \mathcal{K}_A^p$.
C.2 A Variant of the Internet Protocol TLS

The Flaw

Theorem 5.4. For given $C$ and $i$, $T$ does not preserve the secrecy of $C, s_i$ from adversaries of type $A = \text{default}$ with $K_A, K_A^{-1} \in \mathcal{K}_A'$. This means that the protocol does not provide its intended security requirement, secrecy of $s_i$, against a realistic adversary.

Proof. We prove the existence of a successful attacker $adv$. We fix instances $C$ and $S$ with execution rounds $i$ and $j$ (where $S_i = S$) and denote the link between $C$ and $S$ as $l_{CS}$.

The adversary $adv$ proceeds as follows:

- A message of the form $S.\text{init}(N_i, K_C, \text{Sign}_{K_C^{-1}}(C :: K_C))$ in $l_{CS}$ is replaced by the message $S.\text{init}(N_i, K_A, \text{Sign}_{K_A^{-1}}(C :: K_A))$; that is, the public key $K_C$ of $C$ is replaced by the public key $K_A$ of $A$ at each occurrence and as the signature key.
- When $S$ then sends back the message $\text{resp} \{ \text{Sign}_{K_C^{-1}}(k_j :: \text{arg}_{S,1,1}) \} K_A$, $\text{Sign}_{K_C^{-1}}(S :: K_S)$, using $K_A$ to encrypt the session key $k_j$, $adv$ can obtain $k_j$ and replace the message by $\text{resp} \{ \text{Sign}_{K_C^{-1}}(k_j :: \text{arg}_{S,1,1}) \} K_C$, $\text{Sign}_{K_C^{-1}}(S :: K_S)$.
- When $C$ subsequently returns $\{ s_i \} k_j$, $adv$ can extract the secret $s_i$ (and forward the message).

An ASM that achieves this is defined as follows:

Rule Main($adv$):

- \text{do} \: \text{in} \: \text{parallel}
  - \text{if} \: \text{linkQu}_{T}(l_{CS}) = \{ e \} \: \text{and} \: \text{msgnm}(e) = S.\text{init}
    - \text{then} \: \text{linkQu}_{T}(l_{CS}) := \{ S.\text{init}(\text{Arg}_1(e), K_A, \text{Sign}_{K_C^{-1}}(\text{Ext}(\text{Arg}_2(e) :: (\text{Arg}_3(e)))) :: K_A) \} \}
  - \text{if} \: \text{linkQu}_{T}(l_{CS}) = \{ e \} \: \text{and} \: \text{msgnm}(e) = C.\text{resp} \: \text{then}
    - \text{do} \: \text{in} \: \text{parallel}
      - \text{linkQu}_{T}(l_{CS}) := \{ C.\text{resp} \{ \text{Sign}_{K_C^{-1}}(\text{Arg}_1(e)) \} K_C, \text{Arg}_2(e) \} \}
      - \text{local} := \{ \text{Ext}(K_2(\text{Dec}_{K_A^{-1}}(\text{Arg}_1(e)))) \} \}
    - \text{enddo}
  - \text{if} \: \text{linkQu}_{T}(l_{CS}) = \{ e \} \: \text{and} \: \text{msgnm}(e) = S.\text{chal} \: \text{then}
\[ \text{outQu}_{\text{adv}} := \{ \text{Dec}_{\text{local}}(\text{Arg}_1(c)) \} \]

\text{enddo}

One can convince oneself that \text{adv} eventually outputs \( s \).

**The Fix**

**Theorem 5.5.** Suppose we are given a particular execution of the repaired TLS variant subsystem \( \mathcal{T}' \) (including all clients and servers), a client \( C \), and a number \( I \) with \( S = S_I \), and suppose that the server \( S \) is in its \( I \)th execution round in the current execution when \( C \) in its \( I \)th execution round initiates the protocol (that is, \( C_i = I \) and \( S_j = J \)). Then this execution of \( \mathcal{T}' \) preserves the secrecy of \( C.s_I \) against adversaries of type \( A = \text{default} \) whose previous knowledge \( \mathcal{K}^A_N \) fulfills the following conditions:

- we have

\[
\left\{ \{ C.s_I, \mathcal{K}_C^{-1}, \mathcal{K}_S^{-1} \} \cup \{ S.k_j : j \geq J \} \right\} \cap \mathcal{K}_N^A = \emptyset,
\]

- for any \( X \in \text{Exp} \), \( \text{Sign}_{\mathcal{K}_C^{-1}}(C \rightarrow X) \in \mathcal{K}_N^A \) implies \( X = \mathcal{K}_C \), and

- for any \( X \in \text{Exp} \), \( \text{Sign}_{\mathcal{K}_C^{-1}}(S \rightarrow X) \in \mathcal{K}_N^A \) implies \( X = \mathcal{K}_S \).

**Proof.** We use Theorem 7.26 from Sect. 7.5.1. We show the following claim.

**Claim.** For each knowledge set \( \mathcal{K}^A_{\text{adv}}(A) \) for an adversary \( \text{adv} \) of type \( A \) after an overall execution \( e \) of \( \mathcal{T}' \), whose previous knowledge \( \mathcal{K}_N^A \) satisfies the conditions in the above statement of the theorem, there exists a subalgebra \( X_0 \) that is minimal with respect to the subset relation among the subalgebras \( X \) of \( \text{Exp} \) fulfilling the following two conditions:

1. such that \( X_0 \) contains \( \mathcal{K}^A_{\text{adv}}(A) \).

Firstly, the following condition (1) is required to hold:

\[
\mathcal{K}_{\text{N}}^A \cup \left\{ c.N_i, K_c, \text{Sign}_{\mathcal{K}_c^{-1}}(c \rightarrow K_c) : i \in \mathbb{N} \land c \in \text{Client} \right\}
\]

\[
\cup \left\{ \text{Sign}_{\mathcal{K}_C^{-1}}(s \rightarrow K_s) : s \in \text{Server} \land (s = S \Rightarrow K_s = K_S) \right\}
\]

\[
\cup \left\{ c.s_i k : k \in \text{Keys} \land i \in \mathbb{N} \land c \in \text{Client} \right\}
\]

\[
\land \exists K \in \text{Keys}, E \in \text{Exp}, E' \in X
\]

\[
\left\{ \text{Sign}_{\mathcal{K}_C^{-1}}(E) \in X \land \text{fst}(E) = c.s_i \land \text{snd}(E) = K
\right\}
\]

\[
\land \exists K \in \text{Keys}, E \in \text{Exp}, E' \in X
\]

\[
\left\{ \text{Sign}_{\mathcal{K}_C^{-1}}(E') \in X \land \text{fst}(E') = c.s_i \land \text{snd}(E') = K
\right\}
\]

\[
\subseteq X.
\]

\( \mathcal{K}^A_{\text{adv}}(A) \) is not contained in every such subalgebra \( X \), because the actual messages exchanged may differ depending on the adversary behavior.
Condition (2) requires that for each \( j \in \mathbb{N} \) and \( s: \text{Server} \) and for an associated fixed key \( k_{j,s} \in \text{Keys} \cap X \), a fixed expression \( x_{j,s} \in \text{Exp} \), and a fixed nonce \( n_{j,s} \in \text{Data} \cap X \) with \( \text{Sign}_{K_{j,s}^{-1}}(x_{j,s} :: k_{j,s}) \in X \), we have
\[
\{ \text{Sign}_{K_{j,s}^{-1}}(s.k_j :: n_{j,s} :: k_{j,s}) \}_{k_{j,s}} \in X.
\]

Note that in the second condition, it can be the case that \( k_{j,s}^{-1} \notin K_e \) for any client \( c \) (because \( K_{j,s}^{-1} \notin K_e \) and \( K_c^{-1} \) is never sent out), and \( c \) will notice that something is wrong in the corrected protocol (and because the counter \( j \) is increased, the adversary cannot make the server publish another signature with the same \( k_j \) and the correct \( K_c \)).

Proof of claim. Intuitively, the above claim holds because each knowledge set \( K_e \) is by definition the subalgebra of the algebra of expressions \( \text{Exp} \) built up from \( K_A \) in interaction with the protocol participants during the protocol run \( e \). To argue in more detail, we have to consider what knowledge the adversary can gain from interaction with the protocol participants. From the first message of a client \( c \), the adversary can learn the expressions \( c.N_s, K_c \) and \( \text{Sign}_{K_c^{-1}}(c :: K_c) \). From the first message of a server \( s \), the adversary can firstly learn \( \text{Sign}_{K_c^{-1}}(s :: K_c) \). Secondly, for each encryption key \( K \in \text{Keys} \) in the knowledge of the adversary such that the adversary knows \( \text{Sign}_{K_c^{-1}}(x :: K) \) for some \( x \in \text{Exp} \), and for each \( N \) known to the adversary, the adversary learns \( \{ \text{Sign}_{K_c^{-1}}(s.k_j :: N :: K) \}_{K} \in X \), but only a unique such expression for a given server \( s \), protocol run \( e \), and transaction number \( j \), because the transaction number \( j \) is increased as long as the protocol is iterated (this is reflected by the fact that \( X_0 \) is required to be minimal). From the second message from a client \( c \), for each encryption key \( K \in \text{Keys} \) such that
\[
\begin{align*}
&\text{Sign}_{K_c^{-1}}(E) \text{ is known to the adversary for an } E \in \text{Exp} \text{ with } \text{fst}(E) = c.S_i \text{ and } \text{snd}(E) = K, \text{ and such that} \\
&\text{there exists } E' \in \text{Exp} \text{ which is known to the adversary and such that} \\
&\mathcal{E}xt_K(\mathcal{D}ec_{K_c^{-1}}(E')) = (k, c.N_s, K_c) \text{ for some } k \in \text{Keys},
\end{align*}
\]
the adversary learns \( \{ c.s_i \}_K \in X \).

Since there are no other messages sent out by the specified system, the claim holds by the definition of the adversary knowledge as the algebra generated by the exchanged messages and the initial adversary knowledge. This completes proof of the claim.

Thus it is sufficient to show that \( c.s_I \notin X_0 \) for every \( X_0 \) defined above, because \( K_A(A) \) is contained in the union of all such \( X_0 \) by the above argument. We aim for a contradiction by fixing such an \( X_0 \) and assuming that \( c.s_I \in X_0 \). \( X_0 \) is defined to be a minimal subalgebra...
satisfying the conditions (1) and (2) above. Recall that from the definition of the algebra of expressions \( \text{Exp} \) in Sect. 3.3.3 as a free algebra it follows that \( C.s_I \) is different from any other expression not containing it, since no equation with such an expression is defined. In particular, we have \( C.s_I \neq c.s_i \) for any client \( c \) and number \( i \) with \( c \neq C \) or \( i \neq I \). Thus the only occurrence in the conditions defining \( X_0 \) in a minimal way, where \( C.s_I \) may be introduced as a subterm, is in the requirement that \( X_0 \) contains \( \{ C.s_I \}_k \) for each key \( k \in \text{Keys} \) for which there exist \( K \in \text{Keys}, E \in \text{Exp}, E' \in X_0 \) such that

\[
\text{Sign}_{K_{C^{-1}}} (E) \in X_0 \land \text{fst}(E) = S \land \text{snd}(E) = K \\
\wedge \text{Ext}_K (\text{Dec}_{K_{C^{-1}}} (E')) = (k, C.N_I, K_C)
\]

in condition (1). The assumption \( C.s_I \in X_0 \) thus implies that there exists a key \( k \in \text{Keys} \) for which there exist \( K \in \text{Keys}, E \in \text{Exp}, E' \in X_0 \) such that

\[
\text{Sign}_{K_{C^{-1}}} (E) \in X_0 \land \text{fst}(E) = S \land \text{snd}(E) = K \\
\wedge \text{Ext}_K (\text{Dec}_{K_{C^{-1}}} (E')) = (k, C.N_I, K_C).
\]

By definition of \( X_0 \) and assumption on \( K_{A}^{p} \), the condition \( \text{Sign}_{K_{C^{-1}}} (E) \in X_0 \land \text{fst}(E) = S \land \text{snd}(E) = K \) implies that \( K = K_S \) (since any expression of this form in \( K_{A}^{p} \) must satisfy this, and also any such expression introduced in \( X_0 \)). Similarly, \( E' \in X_0 \) with \( \text{Ext}_K (\text{Dec}_{K_{C^{-1}}} (E')) = (k, C.N_I, K_C) \) implies \( k = S.k_j \) for some \( j \), because \( E' \notin K_{A}^{p} \) by assumption on the previous adversary knowledge \( K_{A}^{p} \), because \( K_{S}^{-1} \) is never communicated, and because the expression \( \{ \text{Sign}_{K_{S}^{-1}} (S.k_j :: n_{j,s} :: k_{j,s}) \}_{k_{j,s}} \) (in condition (2)) is the only expression with a subterm of the form \( \text{Sign}_{K_{S}^{-1}} (k :: n_{j,s} :: k_{j,s}) \) that is introduced (and we can also conclude that \( n_{j,s} = C.N_I \) and \( k_{j,s} = K_C \) in this term). Furthermore, we can conclude \( j \geq J \) by the assumption that \( S \) is in its \( J \)th execution round when \( C \) is in its \( J \)th round, and by the requirement that the \( C.N_i \) should be fresh (that is, each distinct from any other occurring value). Thus by assumption on the previous adversary knowledge \( K_{A}^{p} \), we have \( S.k_j \notin K_{A}^{p} \) since \( j \geq J \), and thus the adversary must have learned \( S.k_j \) in a protocol interaction. By the freshness assumption on \( S.k_j \), the only message containing \( S.k_j \) is a term of the form \( \{ \text{Sign}_{K_{S}^{-1}} (S.k_j :: n_{j,s} :: k_{j,s}) \}_{k_{j,s}} \).

By condition (2) and the minimality of \( X_0 \), we know that \( n_{j,s} = C.N_I \) and \( k_{j,s} = K_C \) for any such term by the above observation. Therefore, this term has to be decrypted with \( K_{C}^{-1} \) in order to get the \( S.k_j \). The only protocol participant that possesses \( K_{C}^{-1} \) and that could thus provide this service for the adversary is \( C \) (since the other participants do not have \( K_{C}^{-1} \) in their initial knowledge, and \( K_{C}^{-1} \) is never exchanged). However, none of the values in \( \{ \text{Sign}_{K_{S}^{-1}} (S.k_j :: C.N_I :: K_C) \}_{K_C} \) is ever sent out to the network by \( C \). Thus we must conclude that \( K_{C}^{-1} \in K_{A}^{p} \), which contradicts the initial assumption about \( K_{A}^{p} \).
One can see as follows that the adversary knowledge before each iteration of the system satisfies these conditions as well:

(1) In the \( t \)th execution round of the client \( C \), no data of the form \( X, s_i \) is output except \( C, s_I \), which, as the theorem shows, is kept secret from the adversary. The secret keys \( K_C^{-1}, K_S^{-1} \) (for each \( C, S \)) are never output at all. The key \( S.K_I \) is only sent out during the \( J \)th executing round of \( S \), and it follows from the above theorem that in that round, the key is not leaked to the adversary (because otherwise the adversary would gain knowledge of \( C, s_I \) by decrypting the contents of the xchd message). Similarly, an expression of the form \( \{ \text{Sign}_{K_C^{-1}}(X :: C.N_I :: K_C) \} \) (for \( X \in \text{Keys} \)) is only output in the \( I \)th execution round of \( C \) (and is of no use in any later round).

(2) For any \( X \in \text{Exp} \), \( \text{Sign}_{K_C^{-1}}(C :: X) \) is sent out only for \( X = K_C \) (and \( K_C^{-1} \) is not sent out at all).

(3) For any \( X \in \text{Exp} \), \( \text{Sign}_{K_C^{-1}}(S :: X) \) is sent out only for \( X = K_S \) (and \( K_C^{-1} \) is not sent out at all).

**Corollary 5.6.** Any execution of \( T' \) over all clients and servers and all execution rounds preserves the secrecy of each \( C, s_I \) (for \( C : \text{Client and } 1 \leq I \leq l \)) against adversaries of type \( A = \text{default} \) whose previous knowledge \( \mathcal{K}_A^p \) before the overall execution of \( T' \) fulfills the following conditions:

- we have
  \[
  \left( \{ K_C^{-1}, K_S^{-1}, c, s_i, s, k_j, \{ \text{Sign}_{K_C^{-1}}(X :: c.N_i :: K_C) \} \} : c : \text{Client} \land s : \text{Server} \land 1 \leq i \leq l \land 1 \leq j \geq l \land X \in \text{Keys} \right) \cap \mathcal{K}_A^p = \emptyset,
  \]
- for any \( X \in \text{Exp} \) and any \( c : \text{Client} \), \( \text{Sign}_{K_C^{-1}}(c :: X) \in \mathcal{K}_A^p \) implies \( X = K_C \), and
- for any \( X \in \text{Exp} \) and any \( s : \text{Server} \), \( \text{Sign}_{K_C^{-1}}(s :: X) \in \mathcal{K}_A^p \) implies \( X = K_s \).

**Proof.** Suppose we are given an execution \( e \) of \( T' \), a client \( C \), and a number \( I \). Then we have \( S_I = S \) for a server \( S \), and within the execution \( e \), at the point where \( C.i = I \), we have \( S.j = J \) for a number \( J \). Since the conditions on the previous adversary knowledge in the current corollary imply those of the previous theorem, we can thus directly apply the theorem.

### C.3 Common Electronic Purse Specifications

#### C.3.1 Purchase Transaction

**Vulnerability**

**Theorem 5.7.** \( P \) does not provide merchant security against insider adversaries with \( \{ \text{Sign}_{K_C^{-1}}(ID.C :: K_C) \} \subseteq \mathcal{K}_A^p \).
Proof. We prove the existence of a successful attacker \( adv \). We assume that the adversary has a certificate \( Sign_{K_{C'}}(ID_{C'} : K_{C'}) \) and the corresponding private key \( K_{C'}^{-1} \) (this should of course not be linked to the identity of the adversary to avoid identification). We write \( l_{CP} \) (resp. \( l_{DP} \)) for the link between \( C \) and \( P \) (resp. \( P \) and \( D \)). \( l_{AP'} \) is a link between the attacker and the PSAM \( P' \). Thus:

**Rule Main(\( adv \)) :**

if link\(_{QuP}(l_{CP}) = \{ e \} \land msgnm(e) = P.Ccert \)

then seq

\[
\text{link}_{QuP}(l_{AP'}) := \text{link}_{QuP}(l_{CP}) \\
\text{link}_{QuP}(l_{CP}) := \{ P.Ccert(ID_{C'}, K_{C'}, Sign_{K_{C'}^{-1}}(ID_{C'} : K_{C'})) \}
\]

endsq

else seq

if link\(_{QuP}(l_{CP}) = \{ e \} \land msgnm(e) = C.Deb

then \( m := \text{fst}(Dec_{K_{C'}^{-1}}(\text{Arg}_2(e))) \)

if link\(_{QuP}(l_{AP'}) = \{ e \} \land msgnm(e) \in \{ C.Pcert, C.Deb \}

then \( \text{link}_{QuP}(l_{CP}) := \text{link}_{QuP}(l_{AP'}) \)

if link\(_{QuP}(l_{CP}) = \{ e \} \land msgnm(e) = P'.Resp

then do - in - parallel

\[
\text{link}_{QuP}(l_{AP'}) := \text{link}_{QuP}(l_{CP}) \\
\text{link}_{QuP}(l_{DP}) := \{ D Disp(m) \}
\]

endo

endsq

Note that again we give a simplified presentation of the ASM for increased readability. For example, according to the definition of an adversary in Sect. 7.5, the command \( \text{link}_{QuP}(l_{AP'}) := \text{link}_{QuP}(l_{CP}) \) has to be realized by using commands of the form \( \text{read}_{\text{C}_{P'}}(m) \equiv m := \text{link}_{QuP}(l_{CP}) \) and \( \text{insert}_{\text{L}_{AP'}}(e) \equiv \text{link}_{QuP}(l_{AP'}) := \text{link}_{QuP}(l_{AP'}) : \{ e \} \), in a suitable iteration.

We explain how the attacker ASM proceeds. If a message with name \( P.Ccert \) is sent over \( l_{CP} \), the adversary copies it to \( l_{AP'} \) and replaces it in \( l_{CP} \) by \( P.Ccert(ID_{C'}, K_{C'}, Sign_{K_{C'}^{-1}}(ID_{C'} : K_{C'})) \). Otherwise, if a message with name \( C.Deb \) is sent over \( l_{CP} \), the adversary extracts the amount \( \text{fst}(Dec_{K_{C'}^{-1}}(\text{Arg}_2(e))) \) from it and stores it in \( m \). A message with name \( C.Pcert \) or \( C.Deb \) in \( l_{AP'} \) is copied to \( l_{CP} \). If \( l_{CP} \) consists of a message with name \( P'.Resp \), the content of \( l_{CP} \) is copied to \( l_{AP'} \) and the message \( D Disp(m) \) is sent to \( l_{DP} \).

The above condition of merchant security is clearly violated: when executing \( P \) in the presence of \( adv, D \) receives the value \( M_{NT} \), but \( P \) is not in possession of \( Sign_{K_{C'}^{-1}}(ID_{C} : ID_{P} : M_{NT} : NT) \).
Proposed Solution

**Proposition 5.8.** \( \mathcal{P}' \) provides secrecy of \( K_C^{-1}, K_P^{-1} \) and integrity of \( K_C^{-1}, K_C, K_{CA}, ID_C, K_P^{-1}, K_P, M_{NT}, SK_{NT}, NT \) against insider adversaries with \( \mathcal{P}_A' \cap \{ K_C^{-1}, K_P^{-1} \} = \emptyset \).

**Proof.** For an adversary to gain knowledge of \( K_C^{-1}, K_P^{-1} \), the adversary would have to read these expressions from one of the two communication links. We therefore have to consider, if at any point any of the two expressions is communicated over any of the two communication links. According to the specification, none of the values is output by any of the protocol participants at any time. Therefore secrecy of \( K_C^{-1}, K_P^{-1} \) is provided since these values are never sent outside the smart cards (which under the current threat scenario are assumed to be impenetrable).

For the adversary to violate the integrity of any of the attributes \( K_C^{-1}, K_C, K_{CA}, ID_C, K_P^{-1}, K_P, M_{NT}, SK_{NT} \), the adversary would have to cause their values to take on an atomic value in \textbf{Data}^a, during the interaction with the protocol participants. In particular, their values would have to change. From the protocol specification, we can see that the value of none of these attributes is changed at all during the execution of the protocol. Thus their integrity is preserved.

Similarly, for the adversary to violate the integrity of the attribute \( NT \), the adversary would have to cause its value to take on an atomic value in \textbf{Data}^a, during the interaction with the protocol participants. From the protocol specification, we can see that the value of \( NT \) is changed only to take on values of the form \( 0, 0+1, 0+1+1, \) etc., all of which are not in \textbf{Data}^a. Thus the integrity of \( NT \) is preserved.

**Theorem 5.9.** Consider adversaries of type \( A = \text{insider with} \)

\[
\mathcal{K}_A \cap \left\{ \{ K_C^{-1}, K_P^{-1}, K_{CA} \} \cup \{ SK_{NT} : NT \in \mathbb{N} \} \right. \\
\left. \cup \{ \text{Sign}_{K_C^{-1}}(E) : E \in \text{Exp} \} \cup \{ \text{Sign}_{K_P^{-1}}(E) : E \in \text{Exp} \} \\
\cup \{ \text{Sign}_{SK_{NT}}(E) : E \in \text{Exp} \land NT \in \mathbb{N} \} \right\} = \emptyset
\]

and such that for each \( X \in \text{Exp} \) with \( \text{Sign}_{K_{CA}^{-1}}(X : K) \in \mathcal{K}_A \), \( X = ID_C \) implies \( K = K_C \) and \( X = ID_P \) implies \( K = K_P \). The following security guarantees are provided by \( \mathcal{P}' \) in the presence of adversaries of type \( A \):

**Cardholder security:** For all \( ID_C, ID_P, M_{NT}, NT, K_C^{-1} \) such that \( K_C \) is valid for \( ID_C \), if \( P \) is in possession of \( \text{Sign}_{K_C^{-1}}(ID_C : ID_P : M_{NT} : NT) \) then \( C \) is in possession of \( \text{Sign}_{K_{CA}^{-1}}(M_{NT} : SK_{NT} : ID_P : ID_C : NT) \) (for some \( SK_{NT} \) and \( K_P^{-1} \) such that the corresponding key \( K_P \) is valid for \( ID_P \)).

**Merchant security:** Each time \( D \) receives the value \( M_{NT} \), \( P \) is in possession of \( \text{Sign}_{K_C^{-1}}(ID_C : ID_P : M_{NT} : NT) \) for some \( ID_C, K_C^{-1} \), and a new value \( NT \).
Card issuer security: After each completed purchase transaction, let S be the sum of all M\textsubscript{NT} in the sequence consisting of the processed elements of the form \(\text{Sign}_{K_{C}^{-1}}(ID_{C} :: ID_{F} :: M_{\text{NT}} :: NT)\) (with possibly varying ID\textsubscript{C}, ID\textsubscript{F}, and K\textsubscript{C}^{-1}, such that the corresponding key K\textsubscript{C} is valid for ID\textsubscript{C} and where the NT are mutually distinct for fixed C). Also, let S' be the sum of all M'\textsubscript{NT'} in the sequence of processed \(\text{Sign}_{K_{C}^{-1}}(M'_{\text{NT}'} :: SK_{\text{NT}'} :: ID_{F'} :: ID_{F'} :: M_{\text{NT}'} :: NT')\) (with possibly varying ID\textsubscript{F'}, ID\textsubscript{F''}, and K\textsubscript{F''}^{-1}, such that the corresponding key K\textsubscript{F'} is valid for ID\textsubscript{F'}, and where the NT' are mutually distinct for fixed F'). Then S is no greater than S'.

Proof

Cardholder security: We proceed by contraposition. Suppose that (for any SK\textsubscript{NT}, K\textsubscript{F}^{-1} such that the corresponding key K\textsubscript{F} is valid for ID\textsubscript{F}) C is not in possession of \(\text{Sign}_{K_{C}^{-1}}(M_{\text{NT}} :: SK_{\text{NT}} :: ID_{F} :: ID_{C} :: NT)\). We would like to show that for every K\textsubscript{C}^{-1} such that the corresponding key K\textsubscript{C} is valid for ID\textsubscript{C}, P is not in possession of \(\text{Sign}_{K_{C}^{-1}}(ID_{C} :: ID_{F} :: M_{\text{NT}} :: NT)\). We fix such ID\textsubscript{C}, K\textsubscript{C}, and K\textsubscript{C}^{-1}.

We consider:

- the joint knowledge set K\textsubscript{A} of all participants except C (that is, the objects P and D, and any given adversary, which according to the threat scenario are not able to penetrate the smart card on which C resides) and
- the knowledge set K\textsubscript{C} of C.

Claim. K\textsubscript{A} is contained in every subalgebra X of \textbf{Exp} containing

\[
\textbf{Keys} \setminus \{K_{C}^{-1}\} \cup K_{A} \cup \textbf{Data} \cup \\
\{\text{Sign}_{K_{C}^{-1}}(ID_{C} :: id_{P} :: m :: nt)\}_{sk},
\text{Sign}_{sk}(m :: \{\text{Sign}_{K_{C}^{-1}}(ID_{C} :: id_{P} :: m :: nt)\}_{sk} ) : \text{id}_{P}, k_{P}, m, sk, nt, E \in K_{C} \land \text{Sign}_{K_{C}^{-1}}(id_{P} :: k_{P}) \in K_{C} \\
\land \text{Ext}_{k_{P}}(E) = m :: sk :: id_{P} :: ID_{C} :: nt\}.
\]

Note that \(\text{Sign}_{sk}(m :: \{\text{Sign}_{K_{C}^{-1}}(ID_{C} :: id_{P} :: m :: nt)\}_{sk})\) is actually redundant, but included for explicitness. Note also that it is not claimed that K\textsubscript{A} is actually the intersection of such algebras. For example, any of the above algebras (and thus their intersection) contains the key K\textsubscript{C}^{-1}, although K\textsubscript{A} does not. The latter fact is nevertheless used in the proof (below when using the claim). A similar remark applies to terms of the form \(\text{Sign}_{K_{C}^{-1}}(ID :: K)\). Note that K\textsubscript{A} contains SK\textsubscript{NT}, but not K\textsubscript{C}^{-1} (as shown below).

The above claim holds because the knowledge set K\textsubscript{A} is by definition the subalgebra of the algebra of expressions \textbf{Exp} built up from the initial
knowledge by the protocol participants except C and any adversary in interaction with C. We thus have to consider what knowledge the other participants can gain from interaction with C. The expressions learned from the first message from C are contained in X because X is assumed to contain all keys $K \in \textbf{Keys} \setminus \{K^{-1}_C\}$, and all data in $\textbf{Data}$. The expressions learned from the second message from C are contained in X because X is assumed to contain $\{\text{Sign}_{K^{-1}_C}(ID_C :: id_P :: m :: nt)\}_{sk}$ and $\text{Sign}_{sk}(m :: \{\text{Sign}_{K^{-1}_C}(ID_C :: id_P :: m :: nt)\}_{sk})$ for all $id_P, k_P \in K_C$ with $\text{Sign}_{K^{-1}_C}(id_P :: k_P) \in K_C$ and $m, sk, nt, E \in K_C$ with $\text{Sign}_{sk} (E) = m :: sk :: id_P :: ID_C :: nt$, and because C must receive the values $id_P, k_P, \text{Sign}_{K^{-1}_C}(id_P :: k_P), m, sk, nt, E$ before sending out the messages $\{\text{Sign}_{K^{-1}_C}(ID_C :: id_P :: m :: nt)\}_{sk}$ and $\text{Sign}_{sk}(m :: \{\text{Sign}_{K^{-1}_C}(ID_C :: id_P :: m :: nt)\}_{sk})$.

In particular, we have $K^{-1}_C \notin K$, because the initial knowledge of P, D, and the adversary does not include $K^{-1}_C$, and it (or anything it could be derived from) is not transmitted.

Under the above assumption that $\text{Sign}_{K^{-1}_C}(M_{NT} :: SK_{NT} :: ID_P :: ID_C :: NT) \notin K_C$ (for any $SK_{NT}, K^{-1}_P$ such that the corresponding key $K_P$ is valid for $ID_P$), we prove that such a subalgebra X with $\text{Sign}_{K^{-1}_C}(ID_C :: ID_P :: M_{NT} :: NT) \notin X$ exists. Let X be the $\text{Exp}$ subalgebra generated by

$$\begin{align*}
G := \textbf{Keys} \setminus \{K^{-1}_C\} \cup \textbf{Data} \cup \\
\{\text{Sign}_{K^{-1}_C}(id_C :: id_P :: m :: nt)\}_{sk}, \\
\text{Sign}_{sk}(m :: \{\text{Sign}_{K^{-1}_C}(id_C :: id_P :: m :: nt)\}_{sk}) : \\
(id_C, id_P, m, nt) \notin (ID_C, ID_P, M_{NT}, NT).
\end{align*}$$

By construction, X fulfills the above conditions, using the fact that the adversary does not have access to $K^{-1}_C$ (since it is not in the adversary’s initial knowledge and it (or anything it could be derived from) is never transmitted) and thus does not have access to terms of the form $\text{Sign}_{K^{-1}_C}(id_P :: k_P)$ unless $k_P$ is valid for $id_P$. Also, we have $\text{Sign}_{K^{-1}_C}(ID_C :: ID_P :: M_{NT} :: NT) \notin X$.

Thus we have $\text{Sign}_{K^{-1}_C}(ID_C :: ID_P :: M_{NT} :: NT) \notin K$.

**Merchant security:** Each time D receives the value $M_{NT}$, P is in possession of $\text{Sign}_{K^{-1}_C}(ID_C :: K_C)$ and $\text{Sign}_{K^{-1}_C}(ID_C :: ID_P :: M_{NT} :: NT)$ for some $ID_C, K^{-1}_C$, and a new value $NT$.

By the specification of $P$ (and the assumption of a secure communication link between $P$ and $D$), D receives the value $M_{NT}$ only after $P$ has checked the conditions in its part of the protocol; that is, $P$ is in possession of $\text{Sign}_{K^{-1}_C}(id_C :: k_C)$ and $\text{Sign}_{K^{-1}_C}(id_C :: ID_P :: M_{NT} :: NT)$ for some $id_C$. Newness of $NT$ in this expression is guaranteed since $P$ creates the value itself by incrementing it between different runs of the protocol, and because the value is prevented from rolling over.

**Card issuer security:** This follows from *cardholder security.*
C.3.2 Load Transaction

Vulnerabilities

Theorem 5.10. $L$ does not provide load acquirer security against adversaries of type insider with $\{cep, lda, m_n\} \subseteq K^E_A$.

Proof. An attacker may proceed as follows. The attack assumes a threat scenario where the attacker may be (or collaborate with) the card issuer. Thus it suffices to give a modification of the card issuer behavior that achieves the goal of the attack, that is to successfully complete the protocol and to possess a signature of the form $ml$ but with the changed amount $\tilde{m}$ in the end. The following modified card issuer specification $J$ simply stores $Sign_r(cep' :: nt' :: lda' :: \tilde{m} :: s1' :: he'nt :: hl' :: h2l')$ instead of $ml$ in the logging object $\mathcal{C}Log$:

Rule Main($J$) :
\begin{enumerate}
    \item case currState of
        \begin{enumerate}
            \item init: do trans$_f$(Load, (cep, lda, m, nt, s1, R, ml, hl, h2l),
                \begin{align*}
                    \text{valid}(cep) \land \text{Ext}_{K_{CI}}(s1) = cep :: lda :: m :: nt \\
                    \land \text{Ext}_{K_{CI}}(ml) = cep :: nt :: lda :: m :: s1 :: \\
                    \tilde{\text{hash}}(lda :: cep :: nt :: rcnt) :: hl :: h2l;
                \end{align*}
            \end{enumerate}
        \end{enumerate}
    \end{enumerate}
Load: do trans$_f$(Comp, (cep, lda, m, nt, r2l, s3), true;
\begin{enumerate}
    \item \text{iLog}(cep, lda, m, nt, r, ml, r2l), Final;
\end{enumerate}
Fail: do trans$_f$([], [], true; \text{iLog}(cep, lda, 0, nt, r, ml, 0), Final;)
Final: do finished := true

Here we use the macro
\[ \tilde{m} \equiv \text{Sign}_r(cep :: nt :: lda :: \tilde{m} :: s1 :: hcnt :: hl :: h2l) \]

Proposed solution

Proposition 5.11. $L'$ provides secrecy of $K_{CI}, K^{-1}_L, K^{-1}_I$ and integrity of $K_{CI}, K^{-1}_L, K^{-1}_I, cep, nt, rcnt, lda, n, rl_n, nt, r2ln, m_n$ against insider adversaries with $K^E_A \cap \{K_{CI}, K^{-1}_L, K^{-1}_I\} = \emptyset$.

Proof. Secrecy is evident since these values are never sent outside the smart cards (which are under the current threat scenario assumed to be impenetrable).

Similarly, integrity of $K_{CI}, K^{-1}_L, K^{-1}_I, cep, rcnt, lda, rl_n, r2ln, m_n$ is evident since these values are not changed during the execution of the specification. Note that the secure definition of $m_{nt}$ (which is outside the current specification) again relies on a secure connection between the terminal where the cash is entered and the LSAM. Also, the creation of the random values $rcnt, rl_n, r2ln$ is outside the current scope. Finally, integrity of $nt$ (resp.
n) in the sense of Sect. 4.1.2 follows from the fact that the card (resp. the LSAM) changes the value of \(nt\) (resp. \(n\)) during the protocol irrespective of the behavior of the environment.

**Theorem 5.12.** In the presence of adversaries of type \(A = \text{insider}\) with

\[
K_A^P \cap \{K_{CI}, K_L^{-1}, K_T^{-1}\} \cup \{r_{nt} : nt \in \mathbb{N}\} \cup \{rl, r2ln : n \in \mathbb{N}\} = \emptyset
\]

the following security guarantees are provided by \(\mathcal{C}'\):

**Cardholder security:** For any message \(\text{Clog}(lda, m, nt, s2, rl)\) sent to \(c : \text{CLog}\), if \(m \neq 0\) (that is, the card seems to have been loaded with \(m\)) then \(rl \neq 0\) and

\[
\text{Ext}_{K_{CI}}(s2) = cep : nt :: \text{Sign}_{K_{CI}}(cep :: lda :: m :: nt) :: \text{Hash}(lda :: cep :: nt :: rl)
\]

holds (that is, the card issuer certifies \(rl\) to be a valid proof for the transaction). For any two messages \(\text{Clog}(lda, m, nt, s2, rl)\) and \(\text{Clog}(lda', m', nt', s2', rl')\) sent to \(c : \text{CLog}\), we have \(nt \neq nt'\).

**Load acquirer security:** Suppose that we have \(ml_n \in K\) and \(rl_n \in K\) where

\[
ml_n = \text{Sign}_{K_T^{-1}}(cep :: nt :: lda :: m_n :: s1 :: y :: hl_n :: h2l_n) \text{ with } hl_n = \text{Hash}(lda :: cep :: nt :: rl_n) \text{ and } h2l_n = \text{Hash}(lda :: cep :: nt :: rl_n),
\]

for some \(cep, nt, s1, y\). Then at the end of an execution of \(L\) either of the following two conditions hold:

- a message \(\text{Llog}(cep, lda, m_n, nt, x)\) has been sent to \(l : \text{LLog}\) (which implies that \(L\) has received and retains \(m_n\) in cash) or
- a message \(\text{Llog}(cep, lda, 0, nt, x)\) has been sent to \(l : \text{LLog}\), for some \(x\) (that is, the load acquirer assumes that the load failed and returns the amount \(m_n\) to the cardholder) and we have \(x' \in K_L\) and \(z \in K\) with \(z = \text{Sign}_{K_L^{-1}}(cep :: lda :: m_n :: nt :: y')\) where \(y' = \text{Hash}(lda :: cep :: nt :: x') = y\) (that is, the load acquirer can prove that the load was aborted).

**Card issuer security:** For each message \(\text{Clog}(lda, m, nt, s2, rl)\) sent to \(c : \text{CLog}\), if \(m \neq 0\) and

\[
\text{Ext}_{K_{CI}}(s2) = cep : nt :: \text{Sign}_{K_{CI}}(cep :: lda :: m :: nt) :: \text{Hash}(lda :: cep :: nt :: rl)
\]

holds for some \(lda\), then the card issuer has a valid signature \(ml_n\) corresponding to this transaction.

**Proof.**

**Cardholder security:** Suppose that the message \(\text{Clog}(lda, m, nt, s2, rl)\) has been sent to \(c : \text{CLog}\), where \(m \neq 0\). We need to show that \(rl \neq 0\) and that
\[ \text{Ext}_{K_C}(s2) = \text{cep} :: \text{nt} :: \text{Sign}_{K_C}(\text{cep} :: \text{lida} :: \text{m} :: \text{nt}) :: \text{Hash}(\text{lida} :: \text{cep} :: \text{nt} :: \text{rl}) \]

holds. By assumption, the connection between C : Card and c : CLog is secure (since the objects are on the same smart card). This implies that C actually sent the message CLog(lida, m, nt, s2, rl). According to the specification of C, this can only happen if rl ≠ 0 and if \( \text{Ext}_{K_C}(s2) = \text{cep} :: \text{nt} :: s1 :: \text{hl} \) holds, where \( s1 = \text{Sign}_{K_C}(\text{cep} :: \text{lida} :: \text{m} :: \text{nt}) \) and \( \text{hl} = \text{Hash}(\text{lida} :: \text{cep} :: \text{nt} :: \text{rl}) \).

Suppose the two messages CLog(lida, m, nt, s2, rl) and CLog(lida', m', nt', s2', rl') have been sent to c : CLog. We need to show that nt ≠ nt'. Again, by the threat scenario we can conclude that C sent the two messages to c. Suppose without loss of generality that CLog(lida, m, nt, s2, rl) was sent first, According to the statechart specification for C, C reaches the final state immediately afterwards. According to the overall activity diagram given in the specification, C starts a new protocol run only after nt is incremented (and rolling over is not possible). Thus we have nt' ≥ nt + 1, in particular nt ≠ nt'.

**Load acquirer security:** Suppose that we have ml_n, rl_n ∈ K where ml_n = \( \text{Sign}_{K_C}(\text{cep} :: \text{nt} :: \text{lida} :: \text{m} :: \text{nt} :: \text{y} :: \text{h2l}_n) \) with \( \text{hl}_n = \text{Hash}(\text{lida} :: \text{cep} :: \text{nt} :: \text{rl}_n) \) and \( \text{h2l}_n = \text{Hash}(\text{lida} :: \text{cep} :: \text{nt} :: \text{rl}_n) \), for some cep, nt, s1, and y, and that a message LLog(cep, 0, nt, x) has been sent to l : LLog, for some x. We need to show that there exist \( x' \in K_L \) and \( z \in K \) with \( z = \text{Sign}_{K_C}(\text{cep} :: \text{lida} :: \text{m} :: \text{nt} :: \text{y}') \) where \( y' = \text{Hash}(\text{lida} :: \text{cep} :: \text{nt} :: x') = y \).

By the assumed threat scenario, the communication link between L and l is secure (and according to the specification only L can send messages to l). This implies that the message LLog(cep, 0, nt, x) to l : LLog originated at L. According to the specifications of L, this implies that L previously received a message RespC(s3, x') with \( x' = x \), \( x' ≠ 0 \), and such that \( \text{Hash}(\text{lida} :: \text{cep} :: \text{nt} :: x') = y' \) for a value y' received in the message Resp(cep, nt, s1, y') previously in the same protocol run, and such that for the second argument of the message RespL(s2, z) received immediately before RespC(s3, x'), \( \text{Ext}_{K_I}(z) = \text{cep} :: \text{lida} :: \text{m} :: \text{nt} :: y' \) holds (in particular we have \( x', z \in K_L \)).

**Card issuer security:** Suppose that the message CLog(lida, m, nt, s2, rl) was sent to c : CLog, where \( m ≠ 0 \) and \( \text{Ext}_{K_C}(s2) = \text{cep} :: \text{nt} :: \text{Sign}_{K_C}(\text{cep} :: \text{lida} :: \text{m} :: \text{nt}) :: \text{Hash}(\text{lida} :: \text{cep} :: \text{nt} :: \text{rl}) \) holds for some lida. We need to show that the card issuer has a valid signature ml_n corresponding to this transaction.

From the specification of C we see that C has received the message Credit(s2, rl) just before in the same protocol run, and that \( \text{Ext}_{K_C}(s2) = \text{cep} :: \text{nt} :: s1 :: \text{hl} \) holds, where \( s1 = \text{Sign}_{K_C}(\text{cep} :: \text{lida} :: \text{m} :: \text{nt}) \) and \( \text{hl} = \text{Hash}(\text{lida} :: \text{cep} :: \text{nt} :: \text{rl}) \). Since the key \( K_C \) is kept secret by C and I (see Proposition 5.11), we may conclude that I created s2. According to the specification of I, this can only be the case if ml ∈ K_I with \( \text{Ext}_{K_I}(ml) = \text{cep} :: \text{nt} :: \text{lida} :: \text{m} :: \text{s1} :: \text{hc} :: \text{ht} :: \text{hl} :: \text{h2l} \).
C.4 UML Machines

Example

**Fact 7.4.** For each sequence $(I_1, \ldots, I_n)$, $\langle Snd \rangle ((I_1, \ldots, I_n))$ consists of those sequences $(O_1, \ldots, O_n)$ that fulfill the following conditions, for each $i \in \{1, \ldots, n\}$:

- $O_i \subset \{transmit\}$.
- $\#(O_1 \cup \ldots \cup O_i) \leq \#(I_1 \cup \ldots \cup I_{i-1}) \setminus \{send\}$.
- The conditions that
  \[\#(I_j \cup \ldots \cup I_{i-1}) - \#((I_j \cup \ldots \cup I_{i-1}) \setminus \{send\}) < i - j - 2 + \#(O_j \cup \ldots \cup O_{i-1})\]
  for each $j < i$ and that $\#(O_1 \cup \ldots \cup O_{i-1}) < \#(I_1 \cup \ldots \cup I_{i-1}) \setminus \{send\}$ imply $\#O_i > 0$.

**Proof.** To see that the above characterization of the behavior of $\langle Snd \rangle ((I_1, \ldots, I_n))$ is correct, one has to convince oneself that the given conditions are necessary and sufficient for a sequence $(O_1, \ldots, O_n)$ to be contained in $\langle Snd \rangle ((I_1, \ldots, I_n))$ (for any sequence $(I_1, \ldots, I_n)$).

We first consider necessity. The first condition is necessary, because $\langle Snd \rangle ()$ only outputs messages transmit, on any input. The second condition is necessary, because the ASM only outputs a message for each send that is received. The third condition is necessary because the ASM will output a message at execution round $i$ provided that, firstly, there is still a send message in the input queue that has not yet prompted a transmit output, and, secondly, we have $currState = Send$, because any other input received apart from send messages has already been consumed.

To consider sufficiency of the conditions, suppose we are given sequences $(O_1, \ldots, O_n)$ and $(I_1, \ldots, I_n)$ such that the three conditions are fulfilled. Then $(O_1, \ldots, O_n)$ is contained in $\langle Snd \rangle ((I_1, \ldots, I_n))$, because from the two sequences we can construct an internal behavior of the ASM SnM with the sequence $C_i$ of contents of currState which produces the sequence of outputs $(O_1, \ldots, O_n)$ given the sequence of inputs $(I_1, \ldots, I_n)$: for each $i$, if $O_i = \{transmit\}$ then $C_i = Send$, otherwise $C_i = Wait$.

C.5 Refinement

**Fact 7.8.** (Delayed) $E$-(i, o)-refinement of UML Machines is a preorder for each set of events $E \subseteq Events$ and tuples i and o of input and output names.

**Proof.** We show that (delayed) $E$-(i, o)-refinement is reflexive for each set of events $E \subseteq Events$ and tuples i and o of input and output names. For any UML Machine $A$, any set $E \subseteq Events$, tuples i and o of input and
output names, and sequence \( I \) of event multi-sets, we have \([A]_{i,o}(I) \cap \mathcal{E} \subseteq [A]_{i,o}(I) \cap \mathcal{E}\) and \([A]_{i,o}(I) \cap \mathcal{E} \subseteq [A]_{i,o}(I) \cap \mathcal{E}\) since \( \subseteq \) and \( \subseteq \) are reflexive.

We show that (delayed) \( \mathcal{E} - \langle i, o \rangle \)-refinement is transitive for each set of events \( \mathcal{E} \subseteq \text{Events} \) and tuples \( i \) and \( o \) of input and output names. Suppose we are given the UML Machines \( A, A', \) and \( A'' \), tuples \( i \) and \( o \) of input and output names, and a set \( \mathcal{E} \subseteq \text{Events} \), such that \( A' \) (delayed) \( \mathcal{E} - \langle i, o \rangle \)-refines \( A \) and \( A'' \) (delayed) \( \mathcal{E} - \langle i, o \rangle \)-refines \( A' \). To show that \( A'' \) (delayed) \( \mathcal{E} - \langle i, o \rangle \)-refines \( A \), suppose we are given a sequence \( I = I_1, \ldots, I_n \) of event multi-sets with \( \bigcup I_i = \mathcal{E} \). We have to show that \([A']_{i,o}(I) \cap \mathcal{E} \subseteq [A]_{i,o}(I) \cap \mathcal{E}\) and \([A'']_{i,o}(I) \cap \mathcal{E} \subseteq [A']_{i,o}(I) \cap \mathcal{E}\). By assumption, we know that we have \([A]_{i,o}(I) \cap \mathcal{E} \subseteq [A]_{i,o}(I) \cap \mathcal{E}\) and \([A'']_{i,o}(I) \cap \mathcal{E} \subseteq [A']_{i,o}(I) \cap \mathcal{E}\), and \([A']_{i,o}(I) \cap \mathcal{E} \subseteq [A]_{i,o}(I) \cap \mathcal{E}\) and \([A'']_{i,o}(I) \cap \mathcal{E} \subseteq [A']_{i,o}(I) \cap \mathcal{E}\). We can conclude by transitivity of \( \subseteq \) and \( \subseteq \).

**Fact 7.10.** If the UMS \( A' \) is a (delayed) refinement of the UMS \( A \) then the UML Machine \( \text{Exec}(A') \) is a (delayed) refinement of the UML Machine \( \text{Exec}(A) \).

**Proof.** Suppose we are given UMSs \( A' \) and \( A \) such that \( A' \) is a (delayed) refinement of \( A \). We need to show that the UML Machine \( A' \) is a (delayed) refinement of the UML Machine \( A \).

The link structures of \( A \) and \( A' \) are the same by definition of (delayed) refinement for UMSs. It is thus sufficient to show that each sequence of contents of the family of link queues \((\text{linkQu}_{A'}(i))_{i \in \text{Links}_{A'}}\) of multi-set names is also (stutter-equivalent to) a sequence of contents of the family \((\text{linkQu}_{A}(i))_{i \in \text{Links}_{A}}\) (since in the case of delayed refinement the UMSs are assumed to be stutter-invariant). This follows from the assumption that \( A' \) is a refinement of \( A \) and from the definition of refinement of UMSs, which implies that there are bijections \( b \) and \( b' \) as in Definition 7.9 such that for each activity \( C, b_{\mathcal{M}}(C) \) is a refinement of \( C \).

**Fact 7.11.** (Delayed) refinement of (stutter-invariant) UMSs is a preorder.

**Proof.** We show that (delayed) refinement of ASM systems is reflexive. For any ASM system \( A = (\text{Comp}_A, \text{Sched}_A, \text{Links}_A, \text{Msgs}_A) \), the identity functions \( b = \text{id} : \text{Comp}_A \rightarrow \text{Comp}_A \) (and similarly the \( b_{\mathcal{M}} \)) fulfill the required conditions by Fact 7.8.

We show that (delayed) refinement of UMSs is transitive. Suppose we are given UMSs \( A = (\text{Comp}_A, \text{Sched}_A, \text{Links}_A, \text{Msgs}_A), A' = (\text{Comp}_{A'}, \text{Sched}_{A'}, \text{Links}_{A'}, \text{Msgs}_{A'}), \) and \( A'' = (\text{Comp}_{A''}, \text{Sched}_{A''}, \text{Links}_{A''}, \text{Msgs}_{A''}) \), such that \( A' \) (delayed) refines \( A \) and \( A'' \) (delayed) refines \( A' \). Thus we have bijections \( b : \text{Comp}_A \rightarrow \text{Comp}_{A'} \) and \( b' : \text{Comp}_{A'} \rightarrow \text{Comp}_{A''} \) (and similarly for the \( b_{\mathcal{M}} \)) fulfilling the above conditions. To show that \( A'' \) (delayed) refines \( A \), we note that the bijection \( b' \circ b : \text{Comp}_A \rightarrow \text{Comp}_{A''} \) (and similarly the \( b_{\mathcal{M}} \)) fulfills the conditions as well, by Fact 7.8.
Fact 7.12. Suppose we are given a parameterized UMS $\mathcal{A} = (\mathcal{Y}_1, \ldots, \mathcal{Y}_n)$, where the activity variable $\mathcal{Y}_i$ belongs to the component $C_i$, for each $i = 1, \ldots, n$, and that we are given UMSs $\mathcal{A}_i$ and $\mathcal{A}'_i$ for each $i$.

If for each $i = 1, \ldots, n$, $\text{Exec}(\mathcal{A}_i')$ is a $(i, o_i)$-refinement of $\text{Exec}(\mathcal{A}_i)$ where $i_i = \text{Att}_{C_i}^i$ and $o_i = i_i \cup \{\text{finished}_{\text{Exec}(\mathcal{A}_i)}\}$ then $\mathcal{A}(\text{Exec}(\mathcal{A}_1'), \ldots, \text{Exec}(\mathcal{A}_n'))$ is a refinement of $\mathcal{A}(\text{Exec}(\mathcal{A}_1), \ldots, \text{Exec}(\mathcal{A}_n))$.

Suppose further that the scheduler and all activity UML Machines in $\mathcal{A}(\mathcal{Y}_1, \ldots, \mathcal{Y}_n)$ are stutter-invariant, as well as all $\text{Exec}(\mathcal{A}_i)$ and $\text{Exec}(\mathcal{A}_i')$, for each $i$. If for each $i = 1, \ldots, n$, $\text{Exec}(\mathcal{A}_i')$ is a delayed $(i, o_i)$-refinement of $\text{Exec}(\mathcal{A}_i)$ where $i_i = \text{Att}_{C_i}^i$ and $o_i = i_i \cup \{\text{finished}_{\text{Exec}(\mathcal{A}_i)}\}$, for each $i = 1, \ldots, n$. We have to show that $\mathcal{A}(\text{Exec}(\mathcal{A}_1'), \ldots, \text{Exec}(\mathcal{A}_n'))$ is a refinement of $\mathcal{A}(\text{Exec}(\mathcal{A}_1), \ldots, \text{Exec}(\mathcal{A}_n))$.

Firstly, we have $\text{Mgs}_{\mathcal{A}(\mathcal{A}_1, \ldots, \mathcal{A}_n)} = \text{Mgs}_{\mathcal{A}} = \text{Mgs}_{\mathcal{A}(\mathcal{A}_1', \ldots, \mathcal{A}_n')}$ by construction.

Secondly, we have the bijections sending $\mathcal{A}_i$ to $\mathcal{A}_i'$ for each $i$ and being the identity on the other activities; they fulfill the required conditions by supposition on the $\mathcal{A}_i$ and $\mathcal{A}_i'$.

For delayed refinement, the proof is similar, using the stutter invariance assumption.

Theorem 7.13. (Delayed) refinement of (stutter-invariant) UMSs is a pre-congruence with respect to composition by system formation.

Proof. This follows from Facts 7.11 and 7.12.

Corollary 7.15. (Delayed) equivalence of (stutter-invariant) UMSs is a congruence with respect to composition by system formation.

Proof. This follows from Theorem 7.13.

Theorem 7.17. Each (stutter-invariant) UMS $\mathcal{A}$ is a (delayed) $I$-interface refinement of itself, where $I(\mathcal{Y}) \overset{\text{def}}{=} \mathcal{Y}$.

For all UMSs $\mathcal{A}$, $\mathcal{A}'$, and $\mathcal{A}''$ such that $\mathcal{A}'$ is a (delayed) $I$-interface refinement of $\mathcal{A}$ and $\mathcal{A}''$ is a (delayed) $I$-interface refinement of $\mathcal{A}'$, $\mathcal{A}''$ is a (delayed) $I' \circ I(\mathcal{Y}) \overset{\text{def}}{=} I'(I(\mathcal{Y}))$.

Proof. Suppose we have a UMS $\mathcal{A}$ and define $I(\mathcal{Y}) \overset{\text{def}}{=} \mathcal{Y}$. Then we have $I(\mathcal{A}) = \mathcal{A}$ which is a (delayed, if $\mathcal{A}$ is stutter-invariant) refinement of $\mathcal{A}$ by reflexivity of (delayed) refinement (see Theorem 7.13). Thus $\mathcal{A}$ is a (delayed) $I$-interface refinement of itself.
Suppose we have UMSs $\mathcal{A}$, $\mathcal{A}'$, and $\mathcal{A}''$ such that $\mathcal{A}'$ is a (delayed) $\mathcal{I}$-interface refinement of $\mathcal{A}$ and $\mathcal{A}''$ is a (delayed) $\mathcal{I}$-interface refinement of $\mathcal{A}'$, and define $\mathcal{I}' \circ \mathcal{I}(Y) \equiv \mathcal{I}(\mathcal{I}(Y))$. Then we have $\mathcal{I}' \circ \mathcal{I}(\mathcal{A}) = \mathcal{I}(\mathcal{I}(\mathcal{A}))$. By assumption, we know that $\mathcal{A}'$ is a (delayed) refinement of $\mathcal{I}(\mathcal{A})$ and that $\mathcal{A}''$ is a (delayed) refinement of $\mathcal{I}'(\mathcal{A}')$. By substitutivity of (delayed) refinement, we derive that $\mathcal{I}'(\mathcal{A}')$ is a (delayed) refinement of $\mathcal{I}'(\mathcal{I}(\mathcal{A}))$, and by transitivity of (delayed) refinement, this implies that $\mathcal{A}''$ is a (delayed) refinement of $\mathcal{I}'(\mathcal{I}(\mathcal{A}))$ (see Theorem 7.13). Thus $\mathcal{A}''$ is a (delayed) $\mathcal{I}' \circ \mathcal{I}$-interface refinement of $\mathcal{A}$.

### C.6 Rely-Guarantee Specifications

**Theorem 7.19.** Suppose that the UML Machine $A$ fulfills the rely-guarantee specification $(R, G)$ where $R \cap E = R$ and $G \cap E = G$, and suppose $E = \{ I : I \cap E = I \}$.

If the UML Machine $A'$ $E$-refines $A$ and $A'$ fulfills the rely-guarantee specification $(R, E)$ then $A'$ fulfills the rely-guarantee specification $(R, G)$.

If the UML Machine $A'$ delayed $E$-refines $A$, $G$ is stutter-closed, and $A'$ fulfills the rely-guarantee specification $(R, E)$, then $A'$ fulfills the rely-guarantee specification $(R, G)$.

**Proof.** Suppose that the UML Machine $A$ fulfills the rely-guarantee specification $(R, G)$ and the UML Machine $A'$ $E$-refines $A$, with $R \cap E = R$ and $G \cap E = G$. We need to show that $A'$ fulfills the rely-guarantee specification $(R, G)$. Suppose we are given $I \in R$. We need to show that $[A'](I) \subseteq G$. By assumption on $A$, we know that $[A](I) \subseteq G$. By assumption on $A'$, we have $[A'](I) \subseteq [A'](I) \cap E \subseteq [A](I) \cap E$. Thus we may conclude that $[A'](I) \subseteq G$, as required, since $G \cap E = G$.

The proof for delayed refinement is analogous, using the fact that $G$ is stutter-closed to conclude that $[A'](I) \subseteq G$ from the fact that $[A'](I) \subseteq [A](I) \subseteq G$.

### C.7 Reasoning About Security Properties

**Fact 7.36.** For any expression $E \in \text{Data} \cup \text{Var} \cup \text{Keys}$ and any set of expressions $E$, $E$ is independent of $E$ if there exists no expression $E' \in E$ such that $E$ is a subexpression of $E'$.

**Proof.** Suppose we are given $E$ and $E$ as above such that there exists no expression $E'$ of which $E$ is a subexpression. We show that $E$ is independent of $E$, that is $E$ is not an element of the subalgebra $\mathcal{A}$ of $\exp$ generated by $E$. $\mathcal{A}$ is defined to be the subset of values $\exp$ obtained by recursively applying all operations of $\exp$ starting with the set $E$. 
For each set of expressions $A \subseteq \text{Exp}$ let $p(A)$ be the property that there exists no expression $E'$ in $A$ such that $E$ is a subexpression of $E'$. We prove inductively that $E \notin A$ by showing that $p(\varepsilon)$ holds and that the validity of $p(A)$ is preserved by applying the operations of $\text{Exp}$ pointwise to $A$:

- We have $p(\varepsilon)$ by assumption.
- Assuming $p(A)$, we show by contraposition that for all $a_1, a_2 \in A$, $E$ is not a subexpression of $a_1 :: a_2$. Suppose $E$ is a subexpression of $a_1 :: a_2$ for some $a_1, a_2 \in A$. Without loss of generality, suppose that $E$ is not a subexpression of $a_1$. Thus, there exists a term $t_1$ which is equal to $a_1$ in $\text{Exp}$ such that $E$ is not a subterm of $t_1$. However, by assumption, $E$ is a subterm of $t_1 :: t_2$ for every term $t_2$ which is equal to $a_2$ in $\text{Exp}$. Since $E \in \text{Data} \cup \text{Var} \cup \text{Keys}$ by assumption, $E$ is thus a subterm of every such $t_2$, by definition of the equations in $\text{Exp}$. Thus, $E$ is a subexpression of $a_1$.
- Suppose $p(A)$ holds. Then for every $a \in A$, $E$ is not a subexpression of $\text{head}(a)$. If $E$ was a subterm of every term $h$ that is equal to $\text{head}(a)$ in $A$, then $E$ is also a subterm of every term $t$ that is equal to $a$ in $A$, because the head of every such term $t$ is such an $h$. An analogous argument applies to $\text{tail}(\_)$.
- The cases for $\{\_\}$, $\text{Sign}(\_)$, and $\text{Hash}(\_)$ can be treated analogously to the one for $\_ :: \_$. For $\text{Dec}(\_)$ and $\text{Est}(\_)$ one needs to choose $a_1, a_2, t_1, t_2$ such that $t_1$ and $t_2$ are minimal in length.

**Fact 7.36.** For any expression $E \in \text{Data} \cup \text{Var} \cup \text{Keys}$ and any set of expressions $\varepsilon \subseteq \text{Data} \cup \text{Var} \cup \text{Keys}$, $E$ is independent of $\varepsilon$ if and only if $E \notin \varepsilon$.

**Proof.** This follows from Fact 7.36 since for $E, E' \in \text{Data} \cup \text{Var} \cup \text{Keys}$, $E$ is a subexpression of $E'$ only if $E = E'$.

**Fact 7.21.** Suppose we are given UMSs $B$ and $A$ such that $B$ is a refinement of $A$. Suppose we are given an adversary $adv$ of a given type $A$. Then the following hold:

- The UML Machine $(B_{adv}, \{\text{inQu}_B\}, \{\})$ is a refinement of the UML Machine $(A_{adv}, \{\text{inQu}_A\}, \{\})$.
- The UML Machine $(B_{adv}, \{\text{inQu}_B\}, \{\text{outQu}_B\})$ is a refinement of the UML Machine $(A_{adv}, \{\text{inQu}_A\}, \{\text{outQu}_A\})$.

**Proof.** Suppose we are given UMSs $B$ and $A$ such that $B$ is a refinement of $A$, and an adversary $adv$ of a given type $A$. We need to show that the UML Machine $(B_{adv}, \{\text{inQu}_B\}, \{\})$ (resp. $(B_{adv}, \{\text{inQu}_B\}, \{\text{outQu}_B\})$) is a refinement of the UML Machine $(A_{adv}, \{\text{inQu}_A\}, \{\})$ (resp. $(A_{adv}, \{\text{inQu}_A\}, \{\text{outQu}_A\})$).

Since the link structures of $A$ and $B$ are the same by definition of refinement for UMSs, it is sufficient to show that each possible sequence of contents
of the family \((\text{link}_{QU_B}(l))_{l \in \text{links}_B}\) of multi-set names is also a possible sequence of contents of the family \((\text{link}_{QU_A}(l))_{l \in \text{links}_A}\). This follows from the assumption that \(B\) is a refinement of \(A\) and from the definition of refinement of UMSs, which implies that there are bijections \(b^*_N\) such that for each \(C \in \text{Comp}_A\), \(b^*_N(C)\) is a refinement of \(C\).

**Fact 7.22.** Suppose we are given a UMS \(A\), an adversary \(\text{adv} \in \text{Advers}_A(A)\) of type \(A\), and an execution \(\text{e} \in A_{\text{adv}}\). Then after execution of \(\text{e}\), any expression \(E \in \text{Exp}\) in any occurrence of \(\text{insert}_i(E)\), or any command of the form \(\text{:=} E\) in the specification of an adversary of type \(A\), evaluates to an element of \(K^\text{e}_{\text{adv}}(A)\).

**Proof.** Suppose the execution \(\text{e}\) of \(A_{\text{adv}}\) has been executed. Suppose we are given a link \(l\). We need to show that for any occurrences of \(\text{insert}_i(E)\), \(E\) evaluates to an element of \(K^\text{e}_{\text{adv}}(A)\). By definition of adversaries, the expression \(E\) is contained in the subalgebra of expressions generated by \(K^0_A \cup \text{Var}\). To show that \(E\) evaluates to an element of \(K^\text{e}_{\text{adv}}(A)\) when instantiating the variables, it is sufficient to show that the contained variables evaluate to elements of \(K^\text{e}_{\text{adv}}(A)\) (since \(K^\text{e}_{\text{adv}}(A)\) is by definition closed under generation of subalgebras). This is, however, the case by definition of \(K^\text{e}_{\text{adv}}(A)\).

**Fact 7.23.** Given a system \(A\), any expression \(E \in \text{Exp}\) in any occurrence of \(\text{insert}_i(E)\), or any command of the form \(\text{:=} E\) in the specification of an adversary of type \(A\), evaluates to an element of \(K_A(A)\), at any point.

**Proof.** Suppose we are given a system \(A\) and an expression \(E \in \text{Exp}\) in an occurrence of \(\text{insert}_i(E)\), or any command of the form \(\text{:=} E\) in the specification of an adversary of type \(A\). We need to show that \(E\) evaluates to an element of \(K_A(A)\). Suppose we are given an adversary \(\text{adv} \in \text{Advers}_A(A)\) and an execution \(\text{e} \in A_{\text{adv}}\). By Fact 7.22, we know that after execution of \(\text{e} E\) evaluates to an element of \(K^\text{e}_{\text{adv}}(A)\). Therefore, \(E\) evaluates to an element of \(K_A(A)\), by definition of \(K_A(A)\).

**Fact 7.24.** Given a UMS \(A\), any expression \(E \in \text{Exp}\) in any occurrence of a command of the form \(\text{:=} E\) in the specification of an adversary of type \(A\) evaluates to an element of \(I_A(A,v)\), at any point.

**Proof.** Suppose we are given a UMS \(A\) and an expression \(E \in \text{Exp}\) in an occurrence of a command of the form \(\text{:=} E\) in the specification of \(A\). We need to show that \(E\) evaluates to an element of \(I_A(A,v)\). Suppose we are given an execution \(\text{e}\) of \(A\). By the definition of \(I^\text{e}_{\text{adv}}(A,v)\), we know that, after execution of \(\text{e}\), \(E\) evaluates to an element of \(I^\text{e}_{\text{adv}}(A,v)\). Therefore, \(E\) evaluates to an element of \(I_A(A,v)\), by definition of \(I_A(A,v)\).

**Theorem 7.26.** \(A\) preserves the secrecy of \(E\) against adversaries of type \(A\) if and only if \(E \notin K_A(A)\).
Proof. Suppose that we are given a UMS $\mathcal{A}$, an expression $E \in \text{Exp}$, and an adversary type $A$.

Firstly, we show that if $\mathcal{A}$ preserves the secrecy of $E$ against adversaries of type $A$ then $E \not\in \mathcal{K}_{\mathcal{A}}(A)$. We proceed by contraposition. We assume that we have $E \in \mathcal{K}_{\mathcal{A}}(A)$. We need to show that $\mathcal{A}$ does not preserve the secrecy of $E$ against adversaries of type $A$. By definition of preservation of secrecy, it is sufficient to show that there is an adversary $adv \in \text{Advers}_{\mathcal{A}}(A)$, an input sequence $1$, and an output sequence $o \in [[(\mathcal{A}_{adv})]](1)$ such that one of the multisets in $o$ contains an event which has $E$ as an argument. By the assumption $E \in \mathcal{K}_{\mathcal{A}}(A)$ and the definition of $\mathcal{K}_{\mathcal{A}}(A)$, we know that we have $E \in \mathcal{K}_{\mathcal{A}}(A)$ for some $n \in \mathbb{N}$. Thus there is an adversary $adv$ and an execution $e$ of length $n$ such that $E \in \mathcal{K}_{\mathcal{A}}(A)$. Thus $adv$ can output $E$ after the $n$th iteration of $\text{Main}(\mathcal{A}_{adv})$.

Secondly, we show that if $E \not\in \mathcal{K}_{\mathcal{A}}(A)$ then $\mathcal{A}$ preserves the secrecy of $E$ against adversaries of type $A$. Suppose that $E \not\in \mathcal{K}_{\mathcal{A}}(A)$. We need to show that $\mathcal{A}$ preserves the secrecy of $E$ against adversaries of type $A$; that is, for every adversary $adv$ of type $A$, input sequence $1$, output sequence $o \in [[(\mathcal{A}_{adv})]](1)$, and multi-set $M$ in $o$, $M$ does not contain an event which has $E$ as an argument. Suppose we are given such an adversary $adv$, input sequence $1$, output sequence $o$, and multi-set $M$. To show that $M$ does not contain $E$, it is sufficient to see that for any $exp$ in $adv$, $exp$ does not evaluate to $E$ at any point. This follows from Fact 7.23, since we have $E \not\in \mathcal{K}_{\mathcal{A}}(A)$ by assumption on $E$.

**Theorem 7.27.** If $\mathcal{A}$ preserves the secrecy of $E$ from adversaries of type $A$ given inputs in $I$ and $B$ (delayed) refines $\mathcal{A}$ then $B$ preserves the secrecy of $E$ from adversaries of type $A$ given inputs in $I$.

**Proof.** Suppose we are given a UMS $\mathcal{A}$ that preserves the secrecy of a given expression $E$ from adversaries of type $A$ given inputs in $I$ for $I \subseteq \text{Events}$. Suppose that the UMS $B$ refines $\mathcal{A}$. We need to show that $B$ preserves the secrecy of $E$ from adversaries of type $A$ given inputs in $I$.

Suppose we are given $adv \in \text{Advers}_{\mathcal{A}}(A)$, an input sequence $1$ whose multisets only contain elements in $I$, an output sequence $o \in [[(\mathcal{A}_{adv})]](1)$, and a multi-set $M$ in $o$. We need to show that $M$ does not contain an event which has $E$ as an argument.

Since $B$ refines $\mathcal{A}$, we have $adv \in \text{Advers}_{\mathcal{A}}(A)$ and $o \in [[(\mathcal{A}_{adv})]](1)$ (up to stutter-equivalence, in the delayed case). Since $\mathcal{A}$ is assumed to preserve the secrecy of $E$, we conclude that $M$ does not contain an event which has $E$ as an argument.

**Theorem 7.29.** $\mathcal{A}$ preserves the integrity of a variable $v$ against adversaries of type $A$ if $\mathcal{I}_{\mathcal{A}}(\mathcal{A}, v) \subseteq \text{Data}_{\mathcal{A}}$.

**Proof.** Suppose that we are given a UMS $\mathcal{A}$, a variable $v$, and an adversary type $A$. We show that if $\mathcal{I}_{\mathcal{A}}(\mathcal{A}, v) \subseteq \text{Data}_{\mathcal{A}}$ then $\mathcal{A}$ preserves the integrity of
v against adversaries of type A. Suppose that $I_A(A, v) \subseteq \text{Data}_A$. To show that $A$ preserves the integrity of $v$ against adversaries of type $A$, it is sufficient to show that for every adversary $a$ of type $A$ and every input sequence $i$, $v$ does not contain a value $a \notin \text{Data}_A$ at any point. Suppose we are given such an adversary $a$, an input sequence $i$, and an assignment $v := E$ in $A$ where $E$ evaluates to $a$. By Fact 7.24 we can conclude that $a \in \text{Data}_A$ since this was assumed for every $a \in I_A(A, v)$.

**Theorem 7.30.** Suppose we are given UMSs $A$ and $B$. Suppose that $A$ preserves the integrity of $v$ from adversaries of type $A$ given inputs in $I$, that the UMS $B$ refines the UMS $A$, and that the UML Machine $(\text{Exec}(B), \{\text{in}Q_{\text{Exec}(B)}\} \cup l_{\text{Exec}(B)}; \{v\})$ refines the UML Machine $(\text{Exec}(A), \{\text{in}Q_{\text{Exec}(A)}\} \cup l_{\text{Exec}(A)}; \{v\})$. Then $B$ preserves the integrity of $v$ from adversaries of type $A$ given inputs in $I$.

**Proof.** Suppose we are given a UMS $A$ that preserves the integrity of a given expression $v$ from adversaries of type $A$ given inputs in $I$ for $I \subseteq \text{Events}$. Suppose that the UMS $B$ refines the UMS $A$, and that the UML Machine $(\text{Exec}(B), \{\text{in}Q_{\text{Exec}(B)}\} \cup l_{\text{Exec}(B)}; \{v\})$ refines the UML Machine $(\text{Exec}(A), \{\text{in}Q_{\text{Exec}(A)}\} \cup l_{\text{Exec}(A)}; \{v\})$. We need to show that $B$ preserves the integrity of $v$ from adversaries of type $A$ given inputs in $I$.

Suppose we are given $a \in \text{Advers}_A(A)$ and an input sequence $i$. We need to show that for any $a \in \text{Data}$ contained in $v$ at any point of the execution, $a$ originates in $A$.

Since $B$ refines $A$, and $(\text{Exec}(B), \{\text{in}Q_{\text{Exec}(B)}\} \cup l_{\text{Exec}(B)}; \{v\})$ refines $(\text{Exec}(A), \{\text{in}Q_{\text{Exec}(A)}\} \cup l_{\text{Exec}(A)}; \{v\})$, we have $a \in \text{Advers}_A(A)$, and $[[B_{adv}, \{\text{in}Q_A\}, \{v\}]](i) \subseteq [[A_{adv}, \{\text{in}Q_A\}, \{v\}]](i)$. Since $A$ is assumed to preserve the integrity of $v$, we conclude that no sequence in $[[B_{adv}, \{\text{in}Q_A\}, \{v\}]](i)$ contains any value not originating in $A$.

**Theorem 7.40.** Suppose that the UML Machine $A$ prevents down-flow (resp. up-flow) with respect to the set $H \subseteq \text{MsgNm}$ and that the UML Machine $B$ refines $A$. Then $B$ prevents down-flow (resp. up-flow) with respect to $H$.

**Proof.** Suppose we are given UML Machines $A, B$ and a set of message names $H \subseteq \text{MsgNm}$, such that $A$ prevents down-flow with respect to $H$ and that $B$ refines $A$.

We have to show that $B$ prevents down-flow with respect to $H$. Suppose that we are given input sequences $i, j$ of event multi-sets and output sequences $o \in [B](i)$ and $p \in [B](j)$ with $i_H = j_H$. We have to show that $o_H = p_H$. Since $B$ refines $A$, we know that $o \in [A](i)$ and $p \in [A](j)$. Since $A$ prevents down-flow with respect to $H$, this implies that $o_H = p_H$.

The case for prevention of down-flow is analogous.
C.8 Formal Systems Development with UML

**Fact 8.1.** During each given execution of a UML specification, each occurrence of a message is created at at most one location in the specification.

**Proof.** We observe that the only ways in which a message can be newly introduced into the communication queues of the ASM system is via a rule \texttt{ActionRuleSC}_{S(a)} for a \texttt{call} or \texttt{send} action \texttt{a} or a rule \texttt{ActionRuleSD}(\texttt{msg}) for a message \texttt{msg}; there is no usage in the formal semantics of the macro \texttt{toutQu()} except in these rules, the macro \texttt{toinQu()} is not used at all, and there are messages added directly to the multi-sets \texttt{inQu}, \texttt{outQu}, or \texttt{linkQu}(). Also, in the definition of the behavior of ASM systems in terms of interactive ASMs in Sect. 7.2, no messages are newly added to the communication queues (but only transferred between the queues).

Thus to any occurrence \texttt{m}_i of a message \texttt{m} in any of the input, output, or link queues of the ASM system modeling the specification at a given point of its execution, we can associate an occurrence \texttt{l}_i of a \texttt{call} or \texttt{send} action in a statechart or a message sent out in a sequence diagram, such that the occurrence \texttt{m}_i of \texttt{m} originated from the occurrence \texttt{l}_i of the action (for \texttt{i} = 1, 2).

Conversely, each such occurrence \texttt{l}_i when executed, adds a new occurrence \texttt{m}_i of \texttt{m} to the communication queues, by the definition of the associated action rule.

To see that each occurrence of a message is created at at most one location, it is sufficient to see that no occurrence of a message is removed during the execution of the ASM system except when messages are consumed by its recipient.

For this we observe that in the formal semantics defined in this chapter, an occurrence of a message is only removed from the communication queues of a UML specification when it is consumed while it fires a transition at its recipient. Also, in the definition of the behavior of ASM systems in terms of interactive ASMs in Sect. 7.2, no messages are removed from the communication queues (but only transferred between the queues).

Thus each occurrence of a message is created at at most one location.

**Fact 8.3.** (Delayed) $E$-black-box refinement of UML Machines is a preorder for each set of events $E \subseteq \text{Events}$.

**Proof.** This follows from Fact 7.8.

**Theorem 8.5.** If the subsystem $S$ preserves the secrecy of $E$ from adversaries of type $A$ and $T$ (delayed) refines $S$ given adversaries of type $A$ then $T$ preserves the secrecy of $E$ from adversaries of type $A$.

**Proof.** Suppose we are given a subsystem $S$ that preserves the secrecy of a given expression $E$ from adversaries of type $A$. Suppose that the subsystem $T$ refines $S$ given adversaries of type $A$. We need to show that $T$ preserves the secrecy of $E$ from adversaries of type $A$. 
Suppose we are given \( t \in \text{Advers}_T(A) \), an input sequence \( i \), an output sequence \( o \in \text{Exec}(\[[T]\])_t(i) \), and a multi-set \( M \in o \). We need to show that \( M \) does not contain an event which has \( E \) as an argument.

Since \( T \) refines \( S \) given adversaries of type \( A \), we have \( s \in \text{Advers}_S(A) \) and \( o \in \text{Exec}(\[[S]\])_t(i) \). Since \( S \) is assumed to preserve the secrecy of \( E \), we conclude that \( M \) does not contain an event which has \( E \) as an argument.

**Theorem 8.7.** *(Delayed) white-box refinement of (stutter-invariant) UML subsystems is a precongruence with respect to composition by subsystem formation.*

**Proof.** This follows from Theorem 7.13.

**Corollary 8.9.** *(Delayed) white-box equivalence of (stutter-invariant) UML subsystems is a congruence with respect to composition by subsystem formation.*

**Proof.** This follows from Theorem 8.7.

**Theorem 8.11.** Each *(stutter-invariant)* UML subsystem \( S \) is a *(delayed)* Id-interface refinement of itself, where \( \text{Id}(Y) \overset{\text{def}}{=} Y \).

For all UML subsystems \( S, S', \) and \( S'' \) such that \( S' \) is a *(delayed)* \( I \)-interface refinement of \( S \) and \( S'' \) is a *(delayed)* \( I' \)-interface refinement of \( S' \), \( S'' \) is a *(delayed)* \( I \circ I \)-interface refinement of \( S \), where \( I \circ I(Y) \overset{\text{def}}{=} I(I(Y)) \).

**Proof.** This follows from Theorem 7.17.

**Theorem 8.13.** Suppose that the UML subsystem \( S \) fulfills the rely-guarantee specification \( (R,G) \) and that \( R \land E = R \) and \( S \land E = S \).

If the UML subsystem \( S' \) \( E \)-black-box refines \( S \) then \( S' \) fulfills the rely-guarantee specification \( (R,G) \).

If the UML subsystem \( S' \) delayed \( E \)-black-box refines \( A \) and \( G \) is stutter-closed then \( S' \) fulfills the rely-guarantee specification \( (R,G) \).

**Proof.** This follows from Theorem 7.19.
References


[AW03] K. Alghathbar and D. Wijesekera. Consistent and complete access control policies in use cases. In Stevens [Ste03b], pages 373–387.


[Coo00] S. Cook. The UML family: Profiles, prefaces and packages. In Evans et al. [EKS00], pages 255–264.


[DB00] J. Derrick and E. Boiten. Refinement of objects and operations in Object-Z. In Smith and Talcott [ST00], pages 257–277.


[Dij68] E. W. Dijkstra. Stepwise program construction. Published as [Dij82], February 1968.

References


[FMMMP02] E. Fernández-Medina, A. Martínez, C. Medina, and M. Piattini. UML for the design of secure databases: Integrating security levels, user roles, and constraints in the database design process. In Jürjens et al. [JCF+02], pages 93–106.


[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns—Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, 1995.


References 285


References


[KPP02] M. Koch and F. Parisi-Presicce. Access control policy specification in UML. In Jürjens et al. [JCF02], pages 63–78.
References 287


[Krü02] I. Krüger. Towards precise service specification with UML and UML-RT. In Jürjens et al [JCF+02], pages 19–34.


References


[Mer02] S. Merz. From diagrams to semantics – and back (invited talk). In Júrgens et al. [JCF+02].


[MS00] É. Meyer and T. Santen. Behavioural conformance verification in an integrated approach using UML and B. In Grieskamp et al. [GSS00], pages 358–379.


[ÖP00] G. Övergaard and K. Palmkvist. Interacting subsystems in UML. In Evans et al. [EKS00], pages 359–368.


[Pat02] A. Patarica. From the general resource model to a general fault modeling paradigm? In Jürjens et al. [JCF+02], pages 163–170.


References


References


[Sch00] W. Schulte. Why doesn’t anyone use formal methods? In Grieskamp et al. [GSS00], pages 297–298.


[Sel03] B. Selic. The use of modeling techniques in safety-critical system design (invited talk). In Jurjens et al. [IRFF03], pages 1–2.


[Ste01a] P. Stevens. On use cases and their relationships in the Unified Modeling Language. In Hußmann [Hu01], pages 140–155.


References


[XLL03] Zhongfu Xu, J. Luehti, and A. Lehmann. Predicting software performance based on UML models during the unified software development process. In Jürjens et al. [JRFF03], pages 105–112.

# Index

\((\cdot)^{-1}\), 36  
\(\vdash\), 36  
\(\models\), 159  
\(|\cdot|\), 195  
\(\emptyset\), 37  
\([\cdot]\), 34  
\(\Pi ()\), 163  
\(\{\cdot\}\), 34  
\(\{\cdot\}^\omega\), 36  
\(\backslash\), 34  
\(\equiv\), 162  
\(\land\), 34  
\(\lor\), 169  
\(\{\cdot\}\), 34  
\(\subseteq\), 169  
\(\subseteq\), 34  
\(\hat{\cdot}\), 34  
\(\bar{\cdot}\), 34  
\(A_{adv}\), 174  
\(D_{ec}(\cdot)\), 36  
\(E_{adv}(\cdot)\), 36  
\(H_{adv}(\cdot)\), 36  
\(I(\cdot)\), 175  
\(T^A_{adv}(\cdot)\), 175  
\(T^A(\cdot)\), 175  
\(k^\omega\), 58  
\(k_\omega\), 75  
\(K^{\mathcal{A}}\), 173  
\(K^{\mathcal{A}}\), 38  
\(K^\mathcal{A}(\cdot)\), 174  
\(K^\mathcal{A}(\cdot)\), 39, 173  
\(K^\mathcal{A}(\cdot)\), 174  
\(K^\mathcal{A}(\cdot)\), 174  
\(K^\mathcal{A}(\cdot)\), 174  
\(\mathcal{P}(\cdot)\), 34  
\(S_{ign}(\cdot)\), 36  

\begin{align*}
\text{access}, & 173 \\
\text{access}, & 38 \\
\text{access control}, & 24, 25, 114 \\
\text{access decision objects}, & 25 \\
\text{accessible knowledge}, & 39, 173 \\
\text{Action}, & 194 \\
\text{action}, & 193 \\
\{\text{action}\}, & 52 \\
\text{action state}, & 31 \\
\text{ActionRule}(\cdot), & 194 \\
\text{ActionRuleSD}(\cdot), & 211 \\
\text{Activity}, & 195 \\
\text{activity}, & 195 \\
\text{activity diagram}, & 26, 30, 214 \\
\text{Advers}(\cdot), & 173 \\
\text{adversary}, & 173 \\
\{\text{adversary}\}, & 51, 52, 55, 59 \\
\text{adversary knowledge}, & 174 \\
\text{adversary type}, & 38 \\
\text{Arg}\text{e}(\cdot), & 165 \\
\text{assume/guarantee}, & 171 \\
\text{asymmetric key}, & 36 \\
\text{asynchronous message}, & 165 \\
\text{atomic}, & 180 \\
\text{attribute}, & 193 \\
\text{authentication}, & 24
\end{align*}
authenticity, 24, 42, 78, 179
{authenticity}, 58

bag, 34
Behav(), 162
behavioral refinement, 169, 170, 224
belongs to the scope of, 219
black-box refinement, 224
block rule, 160
Bool, 158
Boolean expression, 195
{BoolExp}, 195

«call», 27
case distinction rule, 161
{cert}, 52
choice rule, 160
choose with do, 160
class, 27, 196
class diagram, 26, 27, 196
collaboration diagrams, 26
communication diagrams, 243
complete mediation, 67
completion event, 200
completion transition, 200
component diagrams, 26
composite structure diagrams, 243
concurrent composite state, 29
concurrent state, 29
conditional rule, 159
confidentiality, 24
connections, 209
connector, 243
consistent subsystem, 220
consistent update rules, 160
constant attributes, 196
constraint, 32
control, 38
CORBA, 25
covariant channels, 25
«critical», 58
cryptographic protocol, 23

Data, 36
data origin authenticity, 24
«data security», 59
default attacker, 55
delayed \(\mathcal{E}\)-black-box refinement, 224
delayed \(\mathcal{E}\)-refinement, 169
delayed equivalence, 171
delayed refinement, 169, 170
delayed white-box equivalence, 226
delayed white-box refinement, 225
delete, 173
delete 38
dependency, 27, 196
deployment diagram, 26, 31, 216
do – in – parallel enddo, 160
do-activity, 198, 214

\(\mathcal{E}\)-(i, o)-refinement, 169
\(\mathcal{E}\)-black-box refinement, 224
economy of mechanism, 66
«encrypted», 53
tility authenticity, 24
entry action, 28
equivalence, 171
equivalent UML Machines, 162
event, 35, 165, 191
event queue, 191
Events, 35, 165
Exp(), 166
exit action, 28
Exp, 35, 36, 165, 190
Exp-subalgebra, 37
expression, 35, 36, 165, 190
fail-safe defaults, 66
fair exchange, 24
«fair exchange», 50, 51
false, 158
final state, 29
finished, 195
firing a transition, 28
forall with do, 160
forall rule, 160
fresh, 25, 43, 180
{fresh}, 58
freshness, 25
fst(), 37

{guard}, 63
guard objects, 25
«guarded», 63
guarded access, 25
«guarded access», 63
guarded object, 115
guards, 25
hash, 36
head(), 34, 165
head of list, 34
high, 25
«high», 56
{high}, 58

X-interface refinement, 171, 226
if then else, 160
independent, 37, 180
influence set, 175
initial adversary knowledge, 39
initial knowledge, 173
initial state, 29
input queue, 35, 158, 203
input/output behavior, 163
inQu, 35, 158
insert, 173
insert, 38
instance, 27
integrity, 24, 41, 178
«integrity», 56
{integrity}, 58
interaction frame, 244
interaction overview diagrams, 243
interface, 27
«interface», 27
interface refinement, 171, 226
internal activity, 198, 214
internal transition, 28
«Internet», 53
«issuer node», 53
iterate(), 161
iteration rule, 161
iterative development process, 223

Java Security Architecture, 25

key, 36
Keys, 36
knowledge set, 174

«LAN», 53
least common mechanism, 67
least privilege, 67
link queue, 36
loop-through-list rule, 161
loop-through-set rule, 161
loop through list, 161
loop through set, 161
low, 25

message, 35
message authenticity, 24
message name, 165
message specification, 196
msgnum(), 165
MsgNm, 35
MsgNm, 165
multi-level secure, 25
multi-set, 34

Nul, 195
«no down-flow», 58
«no up-flow», 58
non-interference, 25, 44, 182
non-repudiation, 24
nonce, 25, 36

object, 27, 196
object diagram, 26, 197
object diagrams, 243
Op, 165
open design, 67
operation, 165
output queue, 35, 158, 203
outQu, 35, 158

package, 26, 32
package diagrams, 243
parallel composition, 160
parameterized UML Machine System, 170
part, 243
pattern, 68, 226
pattern-based transformation, 226
permission, 115
«POS device», 53
preserving the secrecy of an expression, 176
preserving the secrecy of a variable, 176
preventing down-flow, 44, 181
preventing up-flow, 44, 181
previous knowledge, 39, 173
profile, 33
protection domain, 115
«provable», 52
psychological acceptability, 68
state, 28, 38, 158
state diagram, 26
state machine, 28
statechart diagram, 26, 28, 198
static structure diagram, 197, 219
stepwise development, 223
stepwise refinement, 223
Stereotypes, 195
stereotype, 32, 195

«call», 27
«critical», 58
«data security», 59
«encrypted», 53
«fair exchange», 50, 51
«guarded», 63
«guarded access», 63
«high», 56
«integrity», 56
«interface», 27
«Internet», 53
«issuer node», 53
«LAN», 53
«no down-flow», 58
«no up-flow», 58
«POS device», 53
«provable», 52
«rbac», 53
«secrecy», 56
«secure dependency», 57
«secure links», 55
«send», 27
«smart card», 53
«wire», 53

{stop}, 51
stutter-closed, 172
stutter-equivalent, 169
stutter-invariant, 169
subactivity state, 31
subalgebra, 37
subexpression, 180
substate, 29
subsystem, 26, 32
swimlane, 30
symmetric key, 36
synchronization bar, 30
synchronous message, 165
system, 32
tagged value, 32
Index 299

tags
{action}, 52
{adversary}, 51, 52, 55, 59
{authenticity}, 58
{cert}, 52
{fresh}, 58
{guard}, 63
{high}, 58
{integrity}, 58
{right}, 53
{role}, 53
{secrecy}, 58
{start}, 51
{stop}, 51
tail(), 34, 165
tail of list, 34
target state, 199, 214
term, 158
term evaluation, 158
thd(), 37
threat function, 173
threat scenario, 38, 173
Threats(), 38, 53
threats(), 38, 173
timing diagrams, 243
toinQu(), 162
tooutQu(), 162
top state, 198, 214
top-down development, 223
trace property, 171
transition, 28
transition rule, 158
triggering event, 199
true, 158
UML 2.0, 243
UML Machine, 157, 158
UML machine, 35
UML Machine name, 165
UML Machine System, 165
UMNames 35, 165
UMS, 165
undefined, 158
underspecification, 168, 177
unpredictability, 177
update rule, 159
use case diagrams, 26

Var, 36
variable assignment, 158
view, 212
Voc., 158
vocabulary, 158

while do, 161
while rule, 161
white-box equivalence, 226
white-box refinement, 225
«wire», 53
wrapper facade pattern, 226

NEU, IX–XI, 6, 9, 11–15, 17, 20, 21,
23–29, 32, 34, 35, 37–42, 44, 48,
50–53, 55, 58, 60, 62, 69, 71, 75,
77, 78, 80–82, 84, 86–88, 123,
132, 158, 162, 163, 165–167, 170,
172–174, 177–180, 183, 207, 209,
211, 236–239, 243