The AutoFOCUS 3 C0 Code Generator

Florian Hölzl
1 Pervasive Code Generation: a Necessary Tool for Model-based Development

Since the beginning of the scientific field of computer science and informatics, languages, in particular programming languages, were of central interest. Over the decades different languages with different levels of abstraction arose. The plethora spans from low-level, processor-dependent machine languages (usually called assembly languages), over structured languages, like C, to higher languages, like C++, Java or various Scripting Languages, like Perl or PHP.

1.1 Modeling Languages

Starting with these programming languages, computer science has evolved into the field of modeling languages over the last decade. The distinction of modeling languages and programming languages is to some extent blurry. In general, modeling languages provide an even more abstract view on the system under development. We no longer think of variables, statements, and functions as primary entities. Instead, modeling languages describe entities of the application domain in an abstract way. We usually describe the systems by characterizing the relevant parts in terms of their interaction and their behavior. This model of the system is mostly independent of concrete realizations on a given hardware platform.

Modeling languages capture the problem domain and its entities using terms of this domain instead of concentrating on the technical details that are used to represent these entities and their actions in a computer-based system. A typical example from the field of business information systems is a customer. Modeling languages usually describe the data associated with each customer, like its name or address. However, on the abstract level nothing is said about
the representation of this chunks of data in some database or the low-level SQL expression for creating a new customer entry in the database.

Modeling languages talk about the application or problem domain, especially at the early stage of the requirements elicitation. However, to obtain a complete and executable system, a development process must define the necessary steps from the abstract levels down to the concrete levels of the implementation, which finally result in a runnable system.

During this step-wise refinement, more and more information is added to obtain an implementation for some specific execution environment. Modern approaches of model-based development try to use generator technologies to provide an automatic support for deriving implementations from abstract descriptions.

In this context a special focus is set on the property preservation during the generation process. The abstract models, usually being smaller and more concentrated, are suitable to show certain properties of the system using validation and verification techniques, like testing or model-checking. However, this tests and proofs on the abstract level are useless, if the generator does not preserve the property. Such a generator must be considered incorrect because the produced system description, e.g. programs or executable code, violate properties that have been proved for the abstract model.

Developing a set of methods and tools to obtain a development process that supports the seamless development of systems, while also providing means to verify the development products is a grand challenge of computer science today.

1.2 Embedded Systems

One of the most important system classes and application domains of today is the field of embedded systems. Software-based electronic devices can be found in nearly every machine, be it production roboters, washing machines, modern cars and airplanes, or medical equipment. An embedded system is characterized as being a system that is placed in some environment (i.e. the embedding system), receives data from this environment via a set of sensors and can manipulate the environmental conditions via a set of actors. Such simple forms of embedded systems, usually executing a specific control task, are found in large parts of today’s daily life, e.g. in modern house heating installations or air-condition machines or traffic light systems.

With increasingly powerful processors and computers and decreasing costs per unit daily life can be supported in increasingly extensive manner with the use of computerized systems, which in term implement their supportive functions in software. Thus, embedded systems found in modern application domains are becoming more and more complex. They are usually constituted of distributed units that cooperate to fulfill complex control tasks. Hereby these parts communicate with each other and thus coordinate their actions to fulfill the desired effects.

Often this class of distributed systems is characterized by very specific timing constraints, usually defined as real-time constraints. Typical instances of this
class are engine control systems in cars or flight control systems in airplanes. For these kind of systems timing is critical. It is not sufficient that some computation result is eventually available, but if the result, although it might be numerically correct, is delivered too late, it is of no use or might cause fatal effects. Thus timing behavior must be specified and verified to guarantee a proper execution of the system.

Due to the complexity of today’s control tasks a thorough understanding of the embedded system and especially its software parts is needed. Even simple control systems cannot be understood at the machine language level any more because they consist of millions of elementary operations executed per second and in parallel on different devices. Without proper modeling tools that apply suitable abstractions these systems cannot be developed in a way that guarantees their safe operation.

1.3 Verification
A good part of modern embedded systems is characterized by a high level of safety and reliability requirements. For example, most systems in modern aircraft control must undergo thorough testing and reviews and if possible formal verification of its runtime behavior. Abstract models of the system can be of valuable help here. Using a suitable language to describe the system and its behavior, and suitable techniques to verify this behavior are the current challenges of computer science in the field of distributed, embedded, real-time systems.

1.4 Deployment
Abstract system descriptions are suitable during the course of development, however, in the end an executable system must be obtained. For embedded systems, this usually means to deploy platform-specific code into defined execution environments or devices, usually an electronic control unit, which consists of a processor, memory and suitable periphery for communication, sensor input and actuator output.

Today the task of deploying a system into its execution environment usually involves three steps: first, the platform-specific code (most often standard C or a subset thereof) is generated from the model; second, this code is compiled into executable programs for the given machine; third, this executable is transferred onto the control unit.

It is vital that during the deployment step properties of the system that have been proven for the abstract implementation are preserved when the system is executed by the runtime environment. However, this topic is out of scope of this report. It has been addressed by the Verisoft project [1].

1.5 AutoFocus 3 and Embedded C Code Generation
In the remainder of this document, we describe the AutoFocus 3 modeling environment for distributed, timed, reactive systems, its different modeling ar-
Furthermore, we specify the modeling language intended for embedded systems and precisely define the C code generation provided for this subset. We also proof the correctness of the generation by showing that the generated code exhibits the behavior of the AutoFocus 3 model, i.e. it is a valid refinement of the model.

The C language, as specified by the ISO 9899 standard [4], allows the use of C constructs and language features that are not suitable for embedded systems. Some features are known to have compiler-dependent behavior and thus are not reliable across platforms; other features, like floating point operations, deny formal verification techniques.

On the contrary, the C language is the current de-facto standard for implementing software supposed to run on digital hardware boards or electronic control units of embedded systems.

Therefore, we define the subset of the C language that suits both the necessities of embedded systems and formal verification techniques. We also describe the mapping of the modeling language subset onto the C language subset in detail. This mapping is the basic specification for the code generator. Note that we do not verify the code generator implementation in a formal, computer-aided way. Here, we only provide the mathematical proof in "paper and pencil" form.

With respect to the results of the first Verisoft project, the C language subset used here is also a subset of the C0 language defined during that project. The C0 language allows access to some kind of heap memory and also provides garbage collection support. However, we believe that using these options makes it much harder to reason about the generated programs. Furthermore, we believe that the class of embedded systems and especially the critical parts of embedded software should not make use of dynamic memory, but should be constructed with precise knowledge about both memory and processor cycle consumption. The AutoFocus 3 C0 code generator produces C0 code that has a compile-time computable consumption of memory (for both code and data) and processor cycles, i.e. resource requirements of the final system can be statically determined.

1.6 Running Example: A Digit Converter

We will use a running example during the course of our introductory explanations. The system uses two integer input values. The first one is interpreted as a natural number with base ten, and converted into its four digit representation with a base given as the second input parameter, which must be 2 at least. The result is output as a five tuple consisting of the four digits (from the least-significant on the left to the highest significant on the right) and a boolean indicator, which is true, if the conversion resulted in an overflow. For example, the binary conversion $s(7, 2) = (1, 1, 1, 0, false)$, while the ternary conversion $s(81, 3) = (0,0,0,0,true)$ overflows.

Furthermore, the digit converter communicates its current state by issuing a status message. The status is one of Ready, Busy, Complete, or Overflow.
2 The AutoFocus 3 Modeling Environment

In AutoFocus 3 we use three different views to build the model of the system under development. We define data types and basic functions using the **data definition view**. We specify the system's architecture using the **system structure view**. Finally, the system's behavior is defined by input / output automata in the **state transition behavior view**.

Data types will be used to describe the communication between the main system components and to define the behavior of each such component. A component is our basic notion of a part of the system, most often associated with a specific task like pre-processing user input or controlling an actor in an embedded system. Components receive input from their environment and compute outputs, which are sent to the environment and often cause physical effects like changing the engine torque or adjusting the pitch elevator. To handle complexity components can themselves consist of a set of communicating (sub-) components. In particular, the whole system itself can be seen as a component, which may be decomposed into sub-components (or sub-systems). Especially in embedded systems parts of these components are of mechanical or electrical nature, but increasingly many parts are also realized in software/hardware digital control units. For the remainder of this report, we concentrate on pure software components, although it is basically possible to describe other kinds of components at a certain level of abstraction (e.g. we could develop an AutoFocus 3 model of a communication bus, which is usually a piece of hardware and low-level software).

Each component has a syntactic input/output interface that is described by the set of typed input ports and the set of typed output ports. Communication paths between components are described using channels, which describe a directed connection between two ports. Thus, we obtain a network of communicating components: the static hierarchic system structure. We describe this part of the model with the **system structure view**. Note that hierarchic components are a way to structure our system for better readability and understandability of the model. From the semantic point of view the hierarchical structuring is irrelevant: the behavior of a composite component is defined by the behaviors of its child components and their communication links \[2\].

In this report we consider only the network of atomic components\[1\]. However, the hierarchic structure is also resembled in the generated C code, which means that the each component (atomic and hierarchic) is described in a separate source file and communication is forwarded across hierarchy boundaries using additional buffers.

In addition to the atomic components we need to consider a single parent component: the environment component. This component represents the border between the system under development and its environment. Its behavior is given by composing the atomic components with input, output and local chan-

\[1\] Note that there are extensions, like time refinement, that impose a different semantic meaning on hierarchic components. This topic is not covered here, but postponed to an extension report.
nels, which transmit information from and to the environment and between the
atomic components, respectively.

For each atomic component we have to define its behavior. We use state
transition automata for this. An automaton consists of a set of control states
and a set of typed variables defining its data state. We define the input/output
behavior of an automaton by specifying transitions between the control states.

A transition connects two control states and specifies a guard and an effect.
During execution each atomic component has a current control state and a data
state, i.e. the valuations of the input and output ports and a set of data state
variables. A component performs a state transition if the guard predicate of a
transition evaluates to true. During the state transition the output port values
and the data state variables are updated according to the effect specification
of the transition. Finally, the current control state is updated to be the target
state of the transition.

For the guard section of each such transition, we define the messages ob-
served on the component’s input ports and a set of logical preconditions over
these inputs and the current data state, i.e. the current values of the data state
variables. The effect section of the transition specifies the messages output via
the component’s output ports and the set of value assignments of the data state
variables. We can use the basic functions defined in the data definition view to
compute preconditions, output messages and data state variable values.

We take a closer look at these modeling views in the following sections.
We use the running example to illustrate the relevant topics. Note that the
modeling features presented here form a subset of the complete AutoFocus 3
modeling language. This subset was chosen to satisfy the requirements of em-
bedded systems, like static analyzability of memory consumption and worst
case execution times, while also providing the semantic basis for analysis and
verification methods.

2.1 Data Definition

Every software-based system works on some kind of data, which needs to be
specified and precisely defined. AutoFocus 3, as most other programming
languages, provides a set of pre-defined data types and the possibility to build
more complex, user-defined data types on top of these. The data definition
section is quite similar to function languages like ML. However, we restrict the
language: we do not support recursive data types or functions when building
models of embedded system software.

There are two basic data types available: the 32-bit integer numbers,
denoted by int, and the boolean type, denoted by boolean. Instances of the first
are denoted with the usual constants ..., -2, -1, 0, 1, 2, ..., while the boolean
constants are true and false.

We may specify enumeration types and tuple types. An enumeration
type consists of a finite set of constructors where each one forms an atomic
term. Internally, an enumeration type is isomorphic to a subset of the natural
numbers. However, it is more convenient to have meaningful names instead of
numeric representations of system messages in the model. A tuple type is used to define composite data elements quite similar to structures in C or records in Pascal. Tuple types consist of a single tuple constructor associated with a finite list of typed selectors, i.e. it aggregates a set of data elements. Tuple types might reference other tuple types in a selector, however, there must not be a cyclic dependency, i.e. tuple types are not recursive. We require the identifiers of user-defined types and constructors to be unique in the data definition, respectively. Furthermore, all of them must start with a capital letter avoiding identifier clashes with the pre-defined types.

We restrict the possible user-defined types to these options for reason of simplicity of translating these types to C language constructs. The resulting C code can be analyzed more easily because it does not use complex C constructs like pointers (or even worse pointer arithmetic). Furthermore, we can compute the memory consumption and worst-case execution times.

**Type Checking** Every term used must be type correct w.r.t. to the type definitions provided so far. Note that the term language also contains a special bottom atom to represent undefined computation results. This atom is not to be mistaken with the empty message signal we will introduce with the signal language later. The bottom symbol can usually be omitted completely, if the developer takes great care in avoiding possibly hazardous terms like unchecked division, possibly resulting in a division by zero error. In the future the bottom symbol might also be usable in function definitions explicitly (similar to null used in Java), however currently it is not.

**Example** For the example system, we define an enumeration data type for status messages with the following atomic constructors: StatusType = Ready | Busy | Complete | Overflow. The type is called StatusType and each of the four atoms describes a status message issued by the system. The Ready message is issued while the system waits for valid input, Busy means the system is currently computing the result, Complete denotes the completion of a successful computation, while Overflow indicates that the requested computation resulted in an overflow (i.e. there are more than four digits needed to represent the given number w.r.t. the given base).

This definition automatically generates eight functions that we can use in the behavior specifications later. There are four constructor functions with result type StatusType to instantiate the type atoms: Ready, Busy, Complete, Overflow. Furthermore, there are four discriminator functions to test a given StatusType term if it represents a specific atom. These functions return a boolean value. For example is_Ready is defined as follows:

```plaintext
is_Ready(_D:StatusType) : boolean =
| (Ready) -> true
| _  -> false
```

If the value of the parameter _D can be matched with Ready the discriminator returns true, otherwise it returns false. Again, pattern matching is typical
feature of functional languages, we use here. Note that in AUTOFOCUS 3 user-defined functions are defined in the same syntax, also using the matching feature to distinguish different cases.

For demonstration purposes, we introduce a tuple type for the results of the computation. For the example system, the result is represented as a five tuple: the four digits and the overflow indicator. We introduce a type \texttt{ResultType} to represent this tuple. This type is defined as follows:

\[
\text{ResultType} = \text{Result}(\text{digit0}: \text{int}, \text{digit1}: \text{int}, \text{digit2}: \text{int}, \\
\text{digit3}: \text{int}, \text{overflow}: \text{boolean})
\]

This specification introduces a constructor function \texttt{Result} that takes five arguments of the given types: four integer and one boolean value. As above, a single discriminator function is also introduced. Since we forbid alternative constructors when parameters are used, this discriminator is not used. The specification also introduces five selector functions, which can be used to extract the tuple’s argument values. They are named \texttt{digit0} to \texttt{digit3}, and \texttt{overflow}, respectively. For example, \texttt{overflow} is defined as follows:

\[
\text{overflow}(_S: \text{ResultType}) : \text{boolean} = \\
| (\text{Result}(\_, \_, \_, \_, \_0)) \rightarrow \_0
\]

Figure 1: The digit system data definitions

Note that this expression matches the function parameter \_S with the pattern \texttt{Result(\_, \_, \_, \_, \_0)}, thus binding \_0 to the fifth element of the tuple, and returning this value as the result. This function is undefined for other constructors. However, the restricted subset does not allow types with mixed constructors, so this case can never occur.
We now have defined all the types necessary to specify our simple digit conversion system. Fig. 1 taken from the AutoFocus 3 tool summarizes our data definition specification.

2.2 System Structure

With the system structure specification we define the static architecture of the system by means of communicating components. We build a network of components, which are linked to the inputs and outputs of the system environment or to each other.

**Weak and strong causality** Each component is classified as either weakly or strongly causal. Strongly causal components deliver their computation results to the receivers with a delay of one logical time unit. Weakly causal components deliver their computation results without a logical delay.

While in the AUTOFOCUS 3 model components work in parallel, the C0 program generated from the model executes components in a sequential manner. Strongly causal components can be sequentialized in arbitrary order because their outputs are delayed until the next step. In contrast, weakly causal components have to be treated with care: in a sequential execution a weakly causal component must be executed before any of its dependent components are executed. Furthermore, the network of components may not include cycles of weakly causal components.

**Example** While AUTOFOCUS 3 and the code generator support hierarchic structures, this example is restricted to a single layer of decomposition. The DigitSystem consists of two sub-components: the Digits4_Base component computes the conversion, while the ComposeNumber component collects the digits and issues the result tuple upon completion.

![Figure 2: The digit system external syntactic interface](image)

Let us first look at the external interface of the system. Fig. 2 shows the syntactic interface of the DigitSystem component.
Based on the external interface, we extend our system structure specification with its internal structure. We introduce the two sub-components (Fig. 3), specify their interfaces (Fig. 4), and connect the components with each other and the parent interface, respectively.

![System Structure](Image)

Figure 3: The system’s structure

Now, we have specified which component talks to which other system part and what messages are possibly transferred during this communication. This system structure gives us an understanding of work sharing and the different tasks of the parts of the system. Furthermore, the system structure specification adheres to the principle of information hiding. We have only specified communication paths and syntactic interfaces, nothing is said yet about the internal structuring of these main system components.

At this point, we have the option to proceed with the system structuring by further decomposing components into sub-component structures. There is no fixed rule to what extent or granularity the decomposition should be carried out. It largely depends on the purpose and complexity of each single part of the system and of course on the architect’s experience.

Other factors that might result in further decomposition are the readability and the size of the behavior specifications (sometimes a network of cooperating components can be specified more nicely than one big automaton with many states and complex transition specifications), or the size of the components’ state space. Smaller state spaces in general allow to apply verification methods like model checking in reasonable time.

For our example system, we have reached a sufficient level of granularity and we will continue to specify each component’s behavior with state transition automatons.
Figure 4: The sub-components’ syntactic interface

Figure 5: The digit conversion automaton
<table>
<thead>
<tr>
<th>Transition</th>
<th>Specification</th>
</tr>
</thead>
</table>
| waiting          | Input: number?; base?  
Precondition: true  
Output: status!Ready  
Postcondition: |
| startValidation  | Input: number?; N; base?; B  
Precondition: N > 0; B > 1  
Output: status!Busy  
Postcondition: _rest = _N; _base = _B |
| firstDigit       | Input: true  
Output: digit0!; rest % _base; status!Busy  
Postcondition: _rest = _rest / _base |
| overflow         | Input: _rest > 0  
Output: status!Overflow  
Postcondition: |
| complete         | Input: _rest == 0  
Output: status!Ready  
Postcondition: |

Table 1: Exemplary transitions of the Digits4Base component.

### 2.3 State Transition Behavior

The system structure defines the static aspects of the system under development: syntactic interfaces and communication links. To complete the system specification, we need to specify the dynamic aspects of each atomic (not further decomposed) component, i.e. its semantic interface or behavior. In AutoFocus 3 this specification is done using input/output automata as explained in this section.

#### 2.3.1 Simple Transition Input/Output Automata

An automaton specification consists of a set of control states (including one state tagged as initial), a set of typed variables forming the data state, and a transition function. Given a current control state, a current valuation of the data variables, and a valuation of the current component input, the transition function specifies the component’s output, its subsequent control state and its new data variable valuations.

Fig. 5 shows the automaton specification of the digit converter component. **Idle** denotes the initial state as indicated by the black dot in the upper left part of the oval. From here, **waiting** is an idling transition that is chosen
while the component receives no input or the input is not valid. Valid input is checked by the `startValidation` transition. Here, the precondition states that the input received as base must be greater than one, and the number to be digitized must be greater than or equal to zero. If this condition is satisfied the component stores the input values in local buffer variables and commences with the digitization. Since we use a four digit representation there are four transitions doing this computation. The final computation step is defined by the transitions labeled `complete` and `overflow`. Here, the component decides whether the digitization completed successfully or an overflow occurred.

Each transition specification is divided into four parts: an input section, a precondition section, an output section, and a postcondition section. The `input section` specifies input patterns that must be applicable to the current input of the component. The pattern matching possibly results in the binding of transition local variables (this is quite similar to pattern matching in functional languages). The `precondition section` specifies logical conditions that may refer to both data state variables and transition local variables, and that must be satisfied for the transition to be able to execute. Preconditions are also often referred to as guards. The `output section` specifies the externally visible reaction of the component, in case the transition is executed. Here, the output values are constructed using inputs, data variables and possible computations therefore (using basic functions defined in the data definition view). Finally, the `postcondition section` specifies the changes to internal data state by assigning new values to data state variables.

![Figure 6: Example of hierarchic state and transition segments.](image)

For the digitization example, Tab. 1 shows some of the interesting transitions of the automaton shown in Fig. 5. The `waiting` transition executes when both inputs are absent (indicated by `number?` and `base?`, respectively). Its precondition does not apply any further restrictions, thus its `status` output is set
to Ready (indicated by status!Ready). All other output values are not specified and are thus reset. Alternatively, we could have specified this reset explicitly using digit0!; digit1!; digit2!; digit3!. In contrast to underspecified outputs, which loose their last value, data state variables, which have not been assigned a new value in the postcondition section, hold their value, i.e. it is not necessary to specify rest = rest; base = base explicitly.

Note that in AUTOFOCUS 3 the semi-colon ; is not interpreted as sequential execution, but should be read as a logical conjunction. This is particularly relevant in the postcondition section where the left-hand side of the equations denote the updated value of the data state variables, while the right-hand side refers to their current value, e.g. a swapping of variable values could be specified either by a=b; b=a or by b=a; a=b. This syntax originates from FOCUS, which uses logical formulas to specify the relation between the current state and the next state.

The startValidation transition shows all the remaining features of transition specification in AUTOFOCUS 3. First, both input values must be present, indicated by the given non-empty pattern. Here, the pattern matching introduces two transition local variables N and B, which are given the respective values of the inputs. In the precondition section the values of the inputs are thus required to fulfill the given guard, e.g. the number must be positive and the base must be at least two. The output section specifies the status to be set to Busy and, as before, all other outputs are reset. Finally, the postcondition section updates the data state variables _rest and _base with the respective values of N and B.

2.3.2 Hierarchic States and Transition Segmentation

The example in Fig. 5 shows a rather small automaton, while real-world example will be more complex and the automata diagrams will easily become unreadable. Therefore AUTOFOCUS 3 provides two mechanisms to structure automata specifications to obtain more readable diagrams: hierarchic states and transition segmentation.

A hierarchic state consists of a set of (sub-)states, which might be again hierarchic, and a set of transitions (or more precise transition segments) internal to this parent state. Any transition entering or leaving a state is connected to it through some connector point (denoted by the white and dark gray circles on the state’s border). Fig. 6 shows the internal view of a hierarchic state displayed in a separate diagram in AUTOFOCUS 3. The border connectors are displayed on the background of the diagram (note that they swapped their color. Thus, transitions always run from black to white connectors within one diagram).

Furthermore, Fig. 6 shows a local connector point, denoted by the gray circle, which is also located on the diagram background. Both border connectors and local connectors allow segments of transitions to be connected in a consecutive manner. Thus, the result is a network of transition segments, which allows to

\footnote{It is a good convention to use capital letters for transition local variables and lower-case letters for data state variables to enhance the readability}
describe complex transition systems in a compact manner. However, such a
tool network of hierarchic states and transition segments is only well-formed, if the
following (statically checkable) constraints hold.

We consider a transition to be a sequence of one or more consecutive transi-
tion segments, i.e. if one segment is the successor of some other segment in this
sequence, there exists a connector point, which is the source point of the former
and the target point of the latter. Then the following constraints must hold for
all transitions of an automaton specification.

**Constraint 2.1** (Transitions Connect Atomic States). A transition starts at
an exit connector point of some atomic state, i.e. the first segment’s source
connector point is an exit connector point of some atomic state.
A transition ends at an entry connector point of some atomic state, i.e. the last
segment’s target connector point is an entry connector point of some atomic
state.

**Constraint 2.2** (Acyclic Transition Segment Network). A transition has finite
length, i.e. the graph formed by the transition segments as edges and connector
points as nodes is acyclic.

**Constraint 2.3** (Unique Transition Local Variable Binding). No variable iden-
tifier appears more than once in patterns of the input specifications of the tran-
sition’s segments, i.e. variable identifiers bound by pattern matching are unique
in a transition.

**Constraint 2.4** (Separation of Data State Variables and Transition Local Vari-
ables). Variable identifiers used for pattern matching are not equal to any data
state variable identifier, i.e. comparing inputs and data state variable values
must be done in the precondition section.

**Constraint 2.5** (Use of Declared Variable). Variables used in preconditions,
outputs, and right-hand sides of postconditions of some transition segment are
either data state variables or have been bound by pattern matching in some
input specification of the same segment, i.e. transition local variables are only
valid within the transition segment they were bound in.

**Constraint 2.6** (Unique Output Specification). No two output specifications
of a transition refer to the same output port, i.e. each output port is assigned a
value at most once in each transition.

**Constraint 2.7** (Unique Data State Variable Assignments). No two postcon-
dition specifications of a transition refer to the same data state variable in the
left-hand side of the postcondition equation, i.e. each data state variable is as-
signed a value at most once in each transition.

These constraints guarantee that for each automaton using hierarchic states
and transition segmentation, there exists a behaviorally equivalent automaton
that has a flattened structure like the one in Fig. 5.
This flattened automaton can be constructed as follows: we consider only the set of atomic states of the hierarchic automaton. Then, for each transition (i.e. each path of transition segments), we introduce a single transition segment in the flattened automaton leading from the source connector point of the first segment of the transition to the target connector point of the last segment of the transition. The input patterns, preconditions, outputs and postconditions are the union of the segment’s respective specifications.

We could use this flattening procedure to transform automata before generating code. However, this flattening leads to combinatorial explosion of transition segments. Therefore, the code generator uses procedure calls to provide a more efficient implementation of automata using the transitions’ segmentation. We will see this later when we discuss the mapping from AutoFocus 3 to C0 in detail.

### 2.4 Non-Determinism and Unique Object Identifiers

The automaton specifications of AutoFocus 3 are in general non-deterministic. For some current state there may be more than one transition executable. However, the C0 implementation of an automaton is by definition deterministic, since C0 executes a program sequentially. Thus, the non-determinism is resolved by the code generator in an unfair manner. Each transition segment has an unique object identifier stored in the AutoFocus 3 model. If the code generator reaches a connector point (or an atomic state) with multiple outgoing transition segments, the segments are tested in the sequence implied by the total order implied by the unique object identifiers. Note that this approach may lead to unreachable C0 code, if the model is non-deterministic.

Note that there exist AutoFocus 3 models, which are internally non-deterministic, but their externally visible behavior is deterministic. However, deterministic models are desired in embedded systems. We may use verification techniques like model-checking to verify that the AutoFocus 3 model is deterministic and thus the corresponding C0 program is also deterministic. Furthermore, careful design and intelligent use of the transition segments can help us to proof the AutoFocus 3 model to be deterministic. In particular, we could annotate connector points with predicates that restrict the automaton’s state and input space as defined by atomic state invariants and all sequences of segments that lead to this connector point. We thus obtain connector point invariants that can ease our proof.

Note also that this resolution mechanism is shared by other generators, in particular the Verisoft XT Isabelle/HOL exporter [7]. Thus, if some property does not hold for the AutoFocus 3 model due to non-determinism, but we can verify (for example in Isabelle) that the property holds for the exported deterministic model, we have verified that this property also holds for the corresponding deterministic C0 program. Furthermore, we can also import the generated C0 program into the Isabelle verification environment and show their equivalence by translation validation. Thus, resolving non-determinism in a similar way in different generators is crucial for reasoning and verification.
However, an in-depth discussion of these methodological considerations and approaches is out of scope of this report.

3 Semantics of AutoFocus 3

This section defines the AutoFocus 3 semantics.

We use following denotations:

- list $S$ denotes the set of ordered lists over the set $S$. We use lists primarily to iterate over the elements of $S$ in a specific order, i.e. we assume no duplicate elements in the list.

- $[\ ]$, $[\ 0 \ ]$ denotes the empty list and the list containing the element 0, respectively. For a non-empty list $l$, $l.first$ denotes the first element of the list, $l.last$ denotes the last element of the list, and $l.rest$ denotes the rest list obtained by removing the first element from $l$. We use $m @ l$ to denote the list concatenation and $|L|$ to denote the length of the list $L$. Sometimes, it is more convenient to directly access the lists elements via an index: $l.0$, $l.1$, and so on.

- We will use $\text{dom}.f$ to denote the domain of the mapping $f$, i.e. the set of values for which $f$ is defined, and $\text{rng}.f$ to denote the range of the mapping, i.e. the set of values that are mapped by some $x \in \text{dom}.f$.

- ID denotes the set of alphanumeric identifiers used as type names, variable names, and function names. For AutoFocus 3 models this also includes identifiers starting with the underscore symbol _ used for variable names.

3.1 Data Definition Semantics

3.1.1 Types

Definition 3.1 (Types). AutoFocus 3 types are defined by the following set of type denotations: $\text{TYPE}_{AF3} = \{\text{boolean}, \text{int}\} \cup \{\text{tuple}(i, m) \mid i \in \text{ID}, m \in \text{list ID} \times \text{TYPE}_{AF3}\} \cup \{\text{variant}(i, v) \mid i \in \text{ID}, v \in \text{list ID}\}$ is the set of pre-defined, user-defined tuple types, and user-defined variant types.

The set of types is statically defined by the data dictionary of an AutoFocus 3 model. Each user-defined type definition becomes a member of $\{\text{tuple}(i, m) \mid i \in \text{ID}, m \in \text{list ID} \times \text{TYPE}_{AF3}\}$ or $\{\text{variant}(i, v) \mid i \in \text{ID}, v \in \text{list ID}\}$, depending on the fact that the type definition specifies a single constructor with at least one selector (i.e. parameter) or a list of constructors without any selectors, respectively. In the second case, we assume this list to have less than $2^{32}$ constructors\(^3\), thus we can represent each constructor by a unique 32-bit integer value.

\(^3\)Variant types are usually used to replace simple integer values with more meaningful string literals, as we have done in our running example. Our assumption is thus only needed from the formal point of view; in practice large sets of variant type constructors are a hint on bad design.
The following constraints must hold for the data definitions.

**Constraint 3.1** (Unique user-defined type identifiers). In a given data dictionary all the types have disjunct names, i.e. any valid type identifier maps to an AutoFocus 3 type be it either boolean, int, or some tuple or variant type.

**Constraint 3.2** (Disjunct Constructor Identifiers). All constructors of all variant types of a given data dictionary have disjunct identifiers.

**Constraint 3.3** (Disjunct Selector Identifiers). All selectors of all tuple types of a given data dictionary have disjunct identifiers.

**Constraint 3.4** (Recursion Free Types). The types of the selectors of the tuple types of a given data dictionary may be chosen arbitrarily as long as there is no cyclic dependency, i.e. tuples can be nested in tuples, but recursive type definitions are not allowed.

With the statically checkable restrictions applied to the data definitions, it is easy to see that the set $\text{TYPE}_{AF3}$ is finite, although it is recursively defined.

**Type Values.** We denote with $VALUE$ the set of all values of all types contained in $\text{TYPE}_{AF3}$. We use the following syntax for the values of the respective types and assume that each value is uniquely associated with exactly one type, i.e. there exists a total mapping $\text{type}_{AF3} : VALUE \rightarrow \text{TYPE}_{AF3}$ that returns the type of the given value. We use following syntax for the value elements:

- $true$ and $false$ denote the usual boolean literal.
- $\ldots,-2,-1,0,1,2,\ldots$ denote the usual integer literals of type int. Valid values lie in the range of 32-bit signed integers.
- $I(p_1, \ldots, p_n)$ denotes the tuple type instance of type $\text{tuple}(I,m)$ with $p_1, \ldots, p_n$ being valid instances of the respective selector type defined in $m$. Note that the constructor used in type definitions for tuple types is irrelevant because there is only one. However, in expressions we use the constructor to distinguish between the instance and the type syntactically.
- $C$ denotes the variant type constructor of the type $\text{variant}(I,v)$ with $C$ being an element of $v$.

**Variable Type**

**Definition 3.2** (Variable Type). Let $V \subseteq ID$ be the set of variables declared by a given model in a given scope. Then $\text{type}_{AF3}^{var} : V \rightarrow \text{TYPE}_{AF3}$ is a mapping, which maps each variable to its specified type.

Atomic components may have data state variables defined in the automaton specification and local variables defined in transition segment input patterns. Each variable has a fixed typed as given by the mapping $\text{type}_{AF3}^{var}$. We assume
that in any scope a variable identifier is uniquely associated with some value (see below). Although, we consider scopes in several places, we do not introduce the concept of shadowing variables, i.e. sub-scopes are required to introduce only fresh variables, e.g. the variable was not declared in some parent scope.

### 3.1.2 Variable Environment

**Definition 3.3** (Variable Environment). A variable environment is a mapping $\nu_{AF}^{var} : ID \rightarrow VALUE$, which maps variable identifiers to their currently stored value.

$\Upsilon_{AF}^{var}$ denotes the set of all variable environments. Clearly, a variable environment is assumed to be type safe:

$$\forall v \in \text{dom.} \nu_{AF}^{var} : \nu_{AF}^{var}(v) \in \text{type}_{AF}^{var}(v)$$

Variable environments represent the current state of a set of variables. Thus, variable environments can be changed by updating variables. Formally, this is captured by the following function updating the variable’s current value.

**Definition 3.4** (Value Update Function). Let $\nu_{AF}^{var} \in \Upsilon_{AF}^{var}$ be some variable environment, $t \in TYPE_{AF}$ be some type, $x \in VALUE$ an arbitrary value of type $t$ and $u \in ID$, $v \in ID$, with $type_{AF}^{var}(u) = t$, arbitrary variables. Then the value update function is a mapping $update : \Upsilon_{AF}^{var} \times ID \times VALUE \rightarrow \Upsilon_{AF}^{var}$ defined as follows:

$\text{update}(\nu_{AF}^{var}, u, x)(v) = \left\{ \begin{array}{ll} x & \text{if } u = v \\ \nu_{AF}^{var}(v) & \text{if } u \neq v \end{array} \right.$

Sometimes we will need to update a set of disjunct variables (like function parameters). Therefore, we define a merge operator on variable environments.

**Definition 3.5** (Variable Environment Merge). Let $\nu, \nu' \in \Upsilon_{AF}^{var}$ with $\forall u \in (\text{dom.} \nu \cap \text{dom.} \nu') : \nu(u) = \nu'(u)$, and $w \in (\text{dom.} \nu \cup \text{dom.} \nu')$. Then

$$(\nu \cup \nu')(w) = \left\{ \begin{array}{ll} \nu(w) & , \text{if } w \in \text{dom.} \nu \\ \nu'(w) & , \text{otherwise} \end{array} \right.$$
• $\phi.defs \in \text{list}(\text{list}(\text{PTN}) \times \text{EXP})$ is the finite, non-empty list of function definition pairs. $\text{PTN}$ denotes the set of pattern expressions and $\text{EXP}$ denotes the set of expressions. Each pair consists of a list of patterns (exactly one for each function parameter) and a result expression.

$\Phi$ denotes the set of all function implementations. If $v0$ is a variable name, we denote by $v0 \in \phi.params$ that $v0$ represents a parameter variable of the function implementation $\phi$. For a given parameter $p \in \phi.params$, we use $p.name$ and $p.type$ to access the elements of the pair, respectively.

**Constraint 3.5** (Unique Parameter Identifiers). The parameter names of a given function implementation are unique with respect to this function implementation.

$\phi.defs$ is the list of function definition pairs with each pair $d$ consisting of a pattern matching expression (accessed by $d.pattern$) and a result expression (accessed by $d.exp$). Each pattern matching expression is an ordered list of patterns with exactly one pattern for each parameter in $\phi.params$. Alternatively, the pattern matching expression can be the universal matching, corresponding to the empty pattern list and syntactically denoted by $\_$. Each pattern may contain variables that are bound by the matching process (see below). We assume that the identifiers of function parameters and variable names bound by pattern matchings are distinct. The result expression can then make use of both the parameter variables and the pattern bound variables. **AutoFocus 3** expressions are statically typed and the code generator assumes type correctness of models.

**Constraint 3.6** (Pattern Matching Expression Length). In each function definition $d$ of some function implementation $\phi$ the pattern list is empty or its length is equal to the length of the parameter list, i.e. $d.pattern = [] \lor |d.pattern| = |\phi.params|$.

**Constraint 3.7** (Unique Pattern Variables). In each function definition $d$ the variables in $d.pattern$ are unique, i.e. they differ from the function parameter variables.

**Constraint 3.8** (Type Correctness). Pattern matching expressions and result expressions of each function definition pair must be type-correct.

**Constraint 3.9** (Recursion-free Function Calls). Function calls used in result expressions are required to be recursion-free, i.e. the call dependency graph of the user-defined functions of a given data dictionary is acyclic.

**Definition 3.7** (Function Table). The function table is a mapping $ftable : ID \rightarrow \Phi$, which maps function names to function implementations.
Table 2: Unary AutoFocus 3 Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Operand Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>!e₀</td>
<td>negation</td>
<td>boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>-e₀</td>
<td>unary minus</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>~e₀</td>
<td>bit-wise negation</td>
<td>int</td>
<td>int</td>
</tr>
</tbody>
</table>

For reasons of simplicity, we assume unique function names, i.e. the function name is sufficient to bind a function call to a function implementation. Usually, the complete function signature is used for this binding process. Uniqueness of function names is statically checked in AutoFocus 3. Furthermore, we assume that each function is total, i.e. for each possible set of arguments, there exists at least one function definition pair that matches this set of arguments in its matching expression. The evaluation selects the first possible matching in the list.

**Constraint 3.10** (Unique Function Identifiers). For a given data dictionary all the contained functions (including selectors and discriminators) have unique identifiers.

As shown in Sec. 2.1 the predefined functions for discriminators of variant type constructors and the selectors of a tuple type constructor can be expressed in a similar way like user-defined functions. Thus, from the semantic point-of-view we do not distinguish between pre-defined and user-defined functions.

The function table is defined by the data definition part of the AutoFocus 3 model.

### 3.1.4 Expression Evaluation

The following sections define the semantics of AutoFocus 3 expression evaluation. \( EXP \) denotes the set of all possible AutoFocus 3 expressions built from the basic expression given in this section and function calls as defined later. We assume all expressions to be type-safe when evaluated.

Given a variable environment \( \upsilon_{\text{var AF3}} \) and an expression \( e \in EXP \), the function \( eval_{\text{AF3}} \), defined as follows, is used to compute the expressions value:

- Let \( e = \text{atom} \) be a value of type \( \text{boolean}, \text{int} \) or some constructor in \( m \) from some variant type \( \text{variant}(V, m) \):
  \[
eval_{\text{AF3}}(e, \upsilon_{\text{var AF3}}) = \text{atom}
  \]
  i.e. atomic values evaluate to themselves.

- Let \( e = <uo>e₀ \) be an expression with unary operator \( <uo> \) from Tab. 2:
  \[
eval_{\text{AF3}}(<uo>e₀, \upsilon_{\text{var AF3}}) = <uo> \cdot \eval_{\text{AF3}}(e₀, \upsilon_{\text{var AF3}})
  \]
### Table 3: Binary AutoFocus 3 Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Operand Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e_1 + e_2$</td>
<td>addition</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>$e_1 - e_2$</td>
<td>subtraction</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>$e_1 * e_2$</td>
<td>multiplication</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>$e_1 / e_2$</td>
<td>division</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>$e_1 % e_2$</td>
<td>modulo</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>$e_1 &amp; e_2$</td>
<td>bit-wise and</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>$e_1</td>
<td>e_2$</td>
<td>bit-wise or</td>
<td>int</td>
</tr>
<tr>
<td>$e_1 ^ e_2$</td>
<td>bit-wise xor</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>$e_1 &lt;&lt; e_2$</td>
<td>shift left</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>$e_1 &gt;&gt; e_2$</td>
<td>shift right</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>$e_1 &gt;&gt;&gt; e_2$</td>
<td>unsigned shift right</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>$e_1 &lt; e_2$</td>
<td>less than</td>
<td>int</td>
<td>boolean</td>
</tr>
<tr>
<td>$e_1 &lt;= e_2$</td>
<td>less or equal</td>
<td>int</td>
<td>boolean</td>
</tr>
<tr>
<td>$e_1 &gt;= e_2$</td>
<td>greater or equal</td>
<td>int</td>
<td>boolean</td>
</tr>
<tr>
<td>$e_1 &gt; e_2$</td>
<td>greater than</td>
<td>int</td>
<td>boolean</td>
</tr>
<tr>
<td>$e_1 == e_2$</td>
<td>equal</td>
<td>int, boolean, variant</td>
<td>boolean</td>
</tr>
<tr>
<td>$e_1 != e_2$</td>
<td>not equal</td>
<td>int, boolean, variant</td>
<td>boolean</td>
</tr>
<tr>
<td>$e_1 &amp;&amp; e_2$</td>
<td>non-strict and</td>
<td>boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>$e_1 | e_2$</td>
<td>non-strict or</td>
<td>boolean</td>
<td>boolean</td>
</tr>
</tbody>
</table>

with the usual semantic interpretation of the unary operators w.r.t. the type instances.

- Let $e = e_0 <\text{bo}> e_1$ be an expression with binary operator $<\text{bo}>$ from Tab. 3:

$$eval_{\text{AF}3}(e_0 <\text{bo}> e_1, v^{\text{var}}_{\text{AF}3}) = eval_{\text{AF}3}(e_0, v^{\text{var}}_{\text{AF}3}) <\text{bo}> eval_{\text{AF}3}(e_1, v^{\text{var}}_{\text{AF}3})$$

with the usual semantic interpretation of the binary operators w.r.t. the type instances.

- Let $e = \text{if } b_0 \text{ then } e_0 \text{ else } e_1 \text{ fi}$ be an if-then-else expression:

$$eval_{\text{AF}3}(e, v^{\text{var}}_{\text{AF}3}) = \begin{cases} 
  eval_{\text{AF}3}(e_0, v^{\text{var}}_{\text{AF}3}), & \text{if } eval_{\text{AF}3}(b_0, v^{\text{var}}_{\text{AF}3}) = \text{true} \\
  eval_{\text{AF}3}(e_1, v^{\text{var}}_{\text{AF}3}), & \text{if } eval_{\text{AF}3}(b_0, v^{\text{var}}_{\text{AF}3}) = \text{false} 
\end{cases}$$

- Variables are evaluated using the variable environment, let $v \in ID$ be a variable identifier:

$$eval(v, v^{\text{var}}_{\text{AF}3}) = v^{\text{var}}_{\text{AF}3}(v)$$

#### 3.1.5 Pattern matching

Pattern matching is used at two occasions in AutoFocus 3: in function implementations and input sections of transition segments. Pattern matching allows
to compare some given value $x \in VALUE$ with some pattern $p \in PTN$.

Syntactically, a pattern $p \in PTN$ uses the syntax given above for type
instances (see Sec. 3.1.1). Additionally, a pattern may also include variable
identifiers and/or the universal matching placeholder $\_$. If the pattern matching succeeds then an updated variable environment $\upsilon_{AF3}$
is computed, which associates with every variable occurring in $p$ the matched
(sub-)value from $x$. Again, we assume that variables in $p$ are unique in the
respective scope and are not mapped to a value in $\upsilon_{AF3}$ already. We denote by
$\text{patvars}_p$ the set of pattern variables of the pattern $p$.

Semantically, pattern matching is defined by the functions

\[
\begin{align*}
\text{match} & : PTN \times VALUE \times \upsilon_{AF3} \rightarrow \text{boolean} \\
\text{bind} & : PTN \times VALUE \times \upsilon_{AF3} \rightarrow \upsilon_{AF3}
\end{align*}
\]
according to the following matching rules:

- Let $p = \_$ be the universal matching and $x$ an arbitrary value, then

\[
\text{match}(p, x, \upsilon_{AF3}) = \text{true} \\
\text{bind}(p, x, \upsilon_{AF3}) = \upsilon_{AF3}
\]

- Let $p = \text{atom}$ be some value of type boolean, int or some constructor in $m$
from some variant type $\text{variant}(V, m)$, then

\[
\text{match}(p, \text{atom}, \upsilon_{AF3}) = \text{true} \\
\text{bind}(p, \text{atom}, \upsilon_{AF3}) = \upsilon_{AF3}
\]

- Let $p = v$ be some variable identifier $v$ and $x$ an arbitrary value of type $t$,
then $\text{type}_{AF3}(v) = t$ and

\[
\text{match}(p, x, \upsilon_{AF3}) = \text{true} \\
\text{bind}(p, x, \upsilon_{AF3}) = \text{update}(\upsilon_{AF3}, v, x)
\]

- Let $p = \text{CON}(p_0, \ldots, p_n)$ with $\text{CON}$ the tuple constructor from some tuple
type $\text{tuple}(\text{CON}, m)$ and $p_0, \ldots, p_n$ be patterns. Furthermore, let $x = \text{CON}(x_0, \ldots, x_n)$ with $\text{CON}$ as before and $x_0, \ldots, x_n$
arbitrary type correct values. Then,

\[
\text{match}(p, x, \upsilon_{AF3}) = \text{true}, \text{if } \forall i \in \{0, \ldots, n\} : \text{match}(p_i, x_i, \upsilon_{AF3}) = \text{true} \\
\text{bind}(p, x, \upsilon_{AF3}) = \bigcup_{i \in \{0, \ldots, n\}} \text{bind}(p_i, x_i, \upsilon_{AF3})
\]

- In all other cases the pattern matching fails letting variable bindings un-
changed:

\[
\text{match}(p, x, \upsilon_{AF3}) = \text{false} \\
\text{bind}(p, x, \upsilon_{AF3}) = \upsilon_{AF3}
\]
match and bind are uniquely defined for some given model, since tuple types may be nested, but are required to be non-recursive.

We extend the match and bind function to lists of patterns and values (with arbitrary $R$ and $P \not= [] \land X \not= [] \land |P| = |X|$):

\[
\begin{align*}
\text{match}([], R, v) &= \text{true} \\
\text{match}(P, X, v) &= \text{match}(P.\text{first}, X.\text{first}, v) \land \text{match}(P.\text{rest}, X.\text{rest}, v)
\end{align*}
\]

\[
\begin{align*}
\text{bind}([], R, v) &= v \\
\text{bind}(P, X, v) &= \text{bind}(P.\text{first}, X.\text{first}, v) \cup \text{bind}(P.\text{rest}, X.\text{rest}, v)
\end{align*}
\]

Matching the empty pattern with an arbitrary $R$ corresponds to the universal matching introduced earlier. Such a matching succeeds without binding any variables. However, if the pattern $P$ is not empty then the value list $X$ must be of corresponding length and the binding process may result in additional variable bindings.

### 3.1.6 Function Call Evaluation

In order to simplify reasoning about function calls, we require that the following constraint holds.

**Constraint 3.11** (Universal Matching Constraint). We require that every user-defined function has at least one function definition pair and that the last pair in the list has an empty pattern list, i.e. the last pair matches universally.

\[
\forall \phi \in \text{rng.ftable} : \phi.\text{defs} \not= [] \land \phi.\text{defs}.\text{last.pattern} = []
\]

**Definition 3.8** (Function Definition Evaluation). Let $\phi$ be a function implementation, $d_0, \ldots, d_m$ be the function definition pairs in $\phi.\text{defs}$, $A = [a_0, \ldots, a_n]$ be the list of arguments, and $v$ a variable environment. Then the list of function definitions is evaluated as follows

\[
\text{eval}_{AF3}(\phi.\text{defs}, A, v) = \text{eval}_{AF3}(d_s.\text{exp}, \text{bind}(d_s.\text{pattern}, A, v))
\]

with

\[
s = \min\{i \mid i \in \{0, \ldots, m\} \land \text{match}(d_i.\text{pattern}, A, v)\}
\]

Thus, a function definition is executed by finding the first matching function definition pair and evaluating the pair expression after binding the pattern matching variables. With Constraint 3.11 such a function definition pair must exist.

**Definition 3.9** (Function Call Evaluation). A function call is evaluated by evaluating all the arguments and storing these results in a fresh variable context.
Then the function implementation as defined by the function table is executed with this variable context as follows.

\[
\phi = ftable(f) \\
eval_{AF3}(e0, v^{var}_{AF3}) = a_0 \\
\vdots \\
eval_{AF3}(en, v^{var}_{AF3}) = a_n
\]

\[
v = update(\ldots update(\emptyset, \phi.params.0.name, e_0) \ldots, \phi.params.n.name, e_n)
\]

\[
eval_{AF3}(f(e0, \ldots, en), v^{var}_{AF3}) = eval_{AF3}(\phi.defs, [a_0, \ldots, a_n], v)
\]

### 3.2 Behavior Semantics

Before we define the semantics of component networks connected via channels as described in the system structure specification of an AutoFocus 3 model we first define the semantics of atomic components.

We formally introduce the relevant structural elements. These are in particular: atomic components, their interface, automaton specifications, states, connector points, transition segments and data state variables. Afterwards, we define the execution semantics of atomic AutoFocus 3 components, which have an automaton specification as their behavior specification, based on these structural elements.

#### 3.2.1 Syntactic Component Interface

From the structural point of view a component is characterized by its syntactic interface, which consists of a set of typed input ports and a set of typed output ports each having an associated initial message.

**Definition 3.10 (Syntactic Component Interface).** Let \( C \) be the set of all components of a given AutoFocus 3 model. Let \( IP \) be the set of all input ports and \( OP \) be the set of all output ports in this model, respectively. Let \( MSG = VALUE \cup \{\epsilon\} \) and \( \epsilon \) represent the empty message. Then the syntactic component interface \( ifc^{syn}_{C} \) of a component \( C \in C \) is a 4-tuple \( \langle I_C, O_C, typeport, initport \rangle \) with

- \( I_C \subseteq IP \) being the input ports of the component.
- \( O_C \subseteq OP \) being the output ports of the component.
- \( typeport : I_C \cup O_C \rightarrow TYPE_{AF3} \) being the port typing.
- \( initport : O_C \rightarrow MSG \) being the initial values of the output ports.

Any port in \( IP \) and \( OP \) is an input or output port of exactly one component in the model. Type-correctness holds for the initial message.
Constraint 3.12 (Type-correct Initial Message). Non-empty initial messages of output ports must be type-correct.

\[ \forall o \in O_C : \text{init}^{port}(o) = \epsilon \lor \text{type}^{val}_{AF3}(o) = \text{type}^{port}(o) \]

We denote with \( C_{env} \in C \) the environment component in some given context (e.g. the component the code generator is executed with) and \( C_{atm} \subset C \) the set of atomic components in some given context (e.g. all atomic components hierarchically below the code generators start component).

3.2.2 Automaton State

First, we formally define the notion of an input pattern, a precondition expression, an output expression, and a postcondition assignment.

Definition 3.11 (Input Pattern). Let \( I_C \) be the set of inputs ports of some atomic component \( C \in C_{atm} \) and let \( PTN \) be the set of patterns. Then an input pattern is a tuple \( \text{inp} \in (I_C \times PTN \cup \{\epsilon\}) \) associating the given pattern or the empty message \( \epsilon \) with the input port.

\( I_C \) denotes the set of all input patterns for a given component \( C \). The input pattern must be type-correct w.r.t. the input port type.

We augment the pattern matching to messages using the following definition.

Definition 3.12 (Message Pattern Matching). Let \( \text{pat} \in (PTN \cup \{\epsilon\}) \) be a message pattern, \( m \in MSG \) be a message, and \( \upsilon \in \text{var}^\upsilon_{AF3} \) a variable environment. Then

\[
\begin{align*}
\text{msgmatch}(\text{pat}, m, \upsilon) &= \begin{cases} 
\text{match}(\text{pat}, m, \upsilon), & \text{if } \text{pat} \neq \epsilon \land m \neq \epsilon \\
\text{true}, & \text{if } \text{pat} = \epsilon \land m = \epsilon \\
\text{false}, & \text{otherwise}
\end{cases} \\
\text{msgbind}(\text{pat}, m, \upsilon) &= \begin{cases} 
\text{bind}(\text{pat}, m, \upsilon), & \text{if } \text{pat} \neq \epsilon \land m \neq \epsilon \\
\upsilon, & \text{otherwise}
\end{cases}
\end{align*}
\]

We extend the \( \text{msgmatch} \) and \( \text{msgbind} \) function to lists of patterns and messages (with arbitrary \( R \) and \( P \neq [] \land X \neq [] \land |P| = |X| ):

\[
\begin{align*}
\text{msgmatch}([], R, v) &= \text{true} \\
\text{msgmatch}(P, X, v) &= \text{msgmatch}(P.first, X.first, v) \land \text{msgmatch}(P.rest, X.rest, v) \\
\text{msgbind}([], R, v) &= v \\
\text{msgbind}(P, X, v) &= \text{msgbind}(P.first, X.first, v) \cup \text{msgbind}(P.rest, X.rest, v)
\end{align*}
\]

Definition 3.13 (Precondition Expression). Let \( EXP \) be the set of expressions as given above. A precondition expression is an expression \( \text{pre} \in EXP \).
\( \mathcal{PRE} \) denotes the set of all precondition expressions. Each precondition must be of type \textit{boolean}.

**Definition 3.14** (Output Expression). Let \( O_C \) be the set of outputs ports of some atomic component \( C \in \mathcal{C}_{atm} \). Then an output expression is a tuple \( \text{out} \in O_C \times \text{EXP} \cup \{\epsilon\} \) associating the given expression or the empty message \( \epsilon \) with the output port.

\( O_C \) denotes the set of all output expressions for a given component \( C \). Each output expression must be type-correct w.r.t. the output port type.

**Definition 3.15** (Postcondition Assignment). Let \( \mathcal{DSV} \subset \text{ID} \) be the set of data state variable identifiers (being part of the automaton specification as defined below) and \( \text{EXP} \) be the set of expressions. Then a postcondition assignment is a tuple \( \text{post} \in \mathcal{DSV} \times \text{EXP} \) associating the given expression with given data state variable.

\( \mathcal{POST} \) denotes the set of all postcondition assignments. The postcondition expression must be type-correct w.r.t. the data state variable type.

**Definition 3.16** (Automaton Specification). An automaton specification of a given atomic component \( C \) is a 7-tuple

\[
\text{Atm}_C = (\mathcal{S}, \mathcal{CP}, \mathcal{TS}, \mathcal{DSV}, \text{type}_{dsv}, s, \text{init}_{dsv})
\]

with

- \( \mathcal{S} \) being the finite set of atomic control states with \(|\mathcal{S}| < 2^{32}\)
- \( \mathcal{CP} \supset \mathcal{S} \) being the finite set of connector points
- \( \mathcal{TS} \) being the finite set of transition segments
- \( \mathcal{DSV} \subset \text{ID} \) being the finite set of data state variables
- \( \text{type}_{dsv} : \mathcal{DSV} \rightarrow \text{TYPE}_{AF3} \) being the data state variable typing.
- \( s \in \mathcal{S} \) being the initial control state.
- \( \text{init}_{dsv} : \mathcal{DSV} \rightarrow \text{VALUE} \) the data state variable initial values.

To keep things simple, we use atomic control states as a special kind of connector points. We do not consider any hierarchical state, but assume their connector points to be like local connector points, e.g. we consider only the flattened automaton. Yet we keep the concept of transition segments, since they provide means to partition transition specifications and their structure can be used to generate more compact C0 code.

**Definition 3.17** (Transition Segment). Let \( \mathcal{CP} \) be the finite set of connector points and \( C \) be an atomic component. Then a transition segment is 6-tuple \( t = (\text{src}, \text{trg}, I, G, O, A) \) with the following elements:
• \( src \in CP \) is the source connector point of the segment.

• \( trg \in CP \), with \((trg \neq src) \lor (src \in S \land trg \in S)\), is the target connector point of the segment.

• \( I \subset IC \) is the set of input patterns.

• \( G \subset PR \) is the set of precondition expressions.

• \( O \subset OC \) is the set of output expressions.

• \( A \subset POS \) is the set of postcondition assignments.

The following consistency constraints must hold for any automaton specification in the model:

**Constraint 3.13** (Acyclic Graph of Transition Segment). The graph formed by \( CP \setminus S \) and \( TS \) is acyclic, i.e. every path of connected transition segments is finite.

**Constraint 3.14** (No Dangling Transition Segments). There are no dangling transition segments, i.e. every local connector point has at least one incoming and at least one outgoing segment.

\[
\forall c \in CP \setminus S : \exists t_1 \in TS : \exists t_2 \in TS : t_1 \neq t_2 \land c = t_1.src \land c = t_2.trg
\]

Together with the latter constraint this implies that any path of connected transition segments leads from an atomic control state to another one.

**Constraint 3.15** (Unique Input Pattern Variables). Variable identifiers in \( DSV \) are unique and are not used in any transition segment’s input pattern.

**Constraint 3.16** (Type-correctness of Initial Values). Initial values of data state variables are type-correct:

\[
\forall d \in DSV : type^{val}_{AF_3}(init^{dsv}(d)) = type^{dsv}(d)
\]

**Constraint 3.17** (Well-definedness of Used Variables). Expressions in transition segment’s preconditions, output expressions and postcondition assignments refer only to variables defined in \( DSV \) or which are bound by some input pattern of this transition.

### 3.2.3 Automaton Execution

**Definition 3.18** (Atomic Component State). Let \( C \in C_{atm} \) be an atomic component, \( if_{C}^{syn} = (Ic, OC, type^{port}, init^{port}) \) its syntactic interface, and \( Atm_{C} = (S, CP, TS, DSV, type^{dsv}, s, init^{dsv}) \) be its automaton specification. Then the component state is a 3-tuple \( \sigma_{C} = (\lambda_{O}, cs, v) \) with
• $\lambda_O : O_C \rightarrow MSG$ being the current output port valuation mapping.
• $cs \in S$ being the current control state.
• $\nu \in \Upsilon_{AF3}^{var}$ being the current data state variable valuation mapping.

The initial component state $\sigma_C^{init} = (\lambda_O^{init}, cs^{init}, \nu^{init})$ is defined as follows:

$$\forall o \in O_C : \lambda_O^{init}(o) = \text{initport}(o)$$
$$cs^{init} = s$$
$$\forall d \in DSV : \nu^{init}(d) = \text{initdsv}(d)$$

$\Sigma_C$ denotes the set of all component states of $C$. The state of an atomic component consists of its control state, its data state variable values and the messages of its output interface. The state of the input interface is not included here, but is given as input to the step function of this atomic component.

**Definition 3.19** (Atomic Component Step Function). Let $C \in C_{atm}$ be an atomic component and $\lambda_I : I_C \rightarrow MSG$ be a total mapping of input messages and $\Lambda_I$ be the set of all such mappings. Then the component step function is a mapping

$$\delta_{step} : \Sigma_C \times \Lambda_I \rightarrow \Sigma_C$$

The step function maps a given component state and a given input message valuation to a successor component state. The definition of the component step function implies that each component has a deterministic behavior. Although an automaton behavior can be interpreted as non-deterministic, we use the object identifiers of transition segments to obtain a deterministic behavior. I.e. transition segments are implicitly prioritized by their object identifiers and thus non-determinism is resolved in an unfair way.

We can now define the operational semantics of an atomic component using its automaton specification. We begin with a single transition segment and evaluate if it is atomically fireable, i.e. if the current valuation of inputs and data state variables fulfills its specifications.

**Definition 3.20** (Atomic Fireable Transition Segment). Let $C \in C_{atm}$ be an atomic component, $\lambda_I \in \Lambda_I$ an input message valuation, $\sigma_C = (\lambda_O, cs, \nu)$ the current state of the component, and $t = (src, trg, I, G, O, A)$ be an arbitrary transition segment of the automaton specification $Atm_C$. Then the predicate $\text{fireable}_{atomic}$ is defined as follows:

$$\text{fireable}_{atomic}(\sigma_C, \lambda_I, t)$$

$$\forall (\text{inport}, \text{pat}) \in I : \text{msgmatch}(\text{pat}, \lambda_I(\text{inport}), \nu) \land$$
$$\forall \text{pre} \in G : \text{eval}_{AF3}(\text{pre}, \nu') = \text{true}$$

with $\nu' = \nu \cup \bigcup_{(\text{inport}, \text{pat}) \in I} \text{msgbind}(\text{pat}, \lambda_I(\text{inport}), \nu)$ being the variable environment containing data state variable values and locally bound input pattern variables.
Note that from the operational point of view the precondition evaluation may only be executed if the pattern matching succeeded and thus all locally bound variables have been assigned a valid value.

Next, we define the recursive predicate \textit{fireable}, which is true if there is a valid transition starting from the current state and consisting of atomically fireable segments.

\textbf{Definition 3.21 (Fireable Transition).} Let all definitions as before and let \(S\) and \(TS\) be the set of atomic states and transition segments of \(Atm_C\), respectively. Let \(t = \langle src, trg, I, G, O, A \rangle\) be an arbitrary transition segment, and \(N = \{ \langle src', trg', I', G', O', A' \rangle \in TS \mid src' = trg \land src' \notin S \}\) be the set of subsequent transition segments. Then the predicate \(\text{fireable} \subseteq \Sigma_C \times \Lambda_C \times TS\) is defined as follows:

\[
\text{fireable}(\sigma_C, \lambda_I, t) \equiv \\
\text{fireable}_{\text{atomic}}(\sigma_C, \lambda_I, t) \land \\
(N = \emptyset \lor \exists n \in N : \text{fireable}(\sigma_C, \lambda_I, n))
\]

Since transition segments form only acyclic finite sequences starting and ending in atomic states, the predicate is well-defined for all models that obey these restrictions.

If a transition fires its effects, as defined by the output expressions and postcondition assignments, are reflected by the corresponding changes in the component state. We required that any valid transition has disjunct areas of effect (e.g. disjunct output ports and disjunct data state variables in the segments). Thus, the order of application of the effects of such a firing sequence of segments does not matter.

\textbf{Constraint 3.18 (Disjunct Output Port Expression).} On every path of transition segments from atomic state to atomic state no two output port expressions refer to the same output port.

\textbf{Constraint 3.19 (Disjunct Data Variable Assignments).} On every path of transition segments from atomic state to atomic state no two postcondition expressions refer to the same data state variables on the left-hand side of the assignment equation.

The graph of transition segments is searched in a depth-first manner to find a sequence of fireable segments. At each local connector point of that search subsequent segments are considered in order of their unique object identifier. This is captured by the following selection function.

\textbf{Definition 3.22 (Next Segment Selection Function).} Let \(t \in TS\) be a transition segment ending in a local connector point, and \(< \subseteq TS \times TS\) be the total order of transition segments induced by the unique object identifiers. Then the next segment selection function of an automaton \(Atm_C\) is a partial mapping.
Given a fireable transition segment that is not connected to an atomic state, the next segment selection function returns the smallest, subsequent segment, which is also fireable. Note that it must exist, since the transition \( t \) is fireable.

**Definition 3.23 (Transition Segment Fire Rule).** Let \( \sigma_C = (\lambda_O, cs, v) \) be a component state, \( \lambda_I \) be an input port valuation, and \( t = (src, trg, I, G, O, A) \) a transition segment. Then the segment step function \( \delta_{seg} : \Sigma_C \times \Lambda_I \times TS \rightarrow \Sigma_C \) is defined according to the following rules:

- **Terminating segment:**

  \[
  \begin{align*}
  \text{fireable}(\sigma_C, \lambda_I, t) \\
  \text{trg} \in S \\
  v^* = v \cup \bigcup_{(ip, pat) \in I} \text{msgbind}(pat, \lambda_I(ip), v) \\
  \lambda'_O(a) = \begin{cases} 
  \text{eval}_{AF3}(oe, v^*) , & \text{if } \exists (op, oe) \in O : op = a \land oe \notin \epsilon \\
  \epsilon , & \text{otherwise}
  \end{cases} \\
  v'(dsv) = \begin{cases} 
  \text{eval}_{AF3}(ae, v^*) , & \text{if } \exists (av, ae) \in A : av = dsv \\
  v(dsv) , & \text{otherwise}
  \end{cases} \\
  \delta_{seg}(\sigma_C, \lambda_I, t) = (\lambda'_O, \text{trg}, v')
  \end{align*}
  \]

- **Intermediate segment:**

  \[
  \begin{align*}
  \text{fireable}(\sigma_C, \lambda_I, t) \\
  \text{trg} \notin S \\
  \delta_{seg}(\sigma_C, \lambda_I, select_{next}(\sigma_C, \lambda_I, t)) = (\lambda''_O, cs'', v'') \\
  v'' = v \cup \bigcup_{(ip, pat) \in I} \text{msgbind}(pat, \lambda_I(ip), v) \\
  \lambda''_O(a) = \begin{cases} 
  \text{eval}_{AF3}(oe, v^*) , & \text{if } \exists (op, oe) \in O : op = a \land oe \notin \epsilon \\
  \epsilon , & \text{if } \exists (op, oe) \in O : op = a \land oe \notin \epsilon \\
  \lambda''_O(a) , & \text{otherwise}
  \end{cases} \\
  v''(dsv) = \begin{cases} 
  \text{eval}_{AF3}(ae, v^*) , & \text{if } \exists (av, ae) \in A : av = dsv \\
  v''(dsv) , & \text{otherwise}
  \end{cases} \\
  \delta_{seg}(\sigma_C, \lambda_I, t) = (\lambda''_O, cs'', v'')
  \end{align*}
  \]

If the given transition segment is leading to a state (instead of a local connector), then the new current state is set and the output port and data state
variable values are updated according to the output expressions and postcondition assignments. If there is no expression specified for some output port or the specification is $\epsilon$ then the output port value is reset. Data state variables remain unchanged if there is no postcondition assignment specified for some data state variable. Expressions are evaluated using the variable environment of the current component state updated with the variable bindings from the current input patterns.

If the given transition is leading to local connector, then we first execute the subsequent segment and apply the effects of the current segment afterwards. If there is no explicit specification for some output port, then we forward the value we received from the subsequent segment (thus $\epsilon$ is forwarded if no segment of the complete transition has a specification). Data state variables remain unchanged if there is no postcondition assignment specified for some data state variable. Expressions are evaluated using the variable environment of the current component state updated with the variable bindings from the current input patterns (and not the updated one due to preservation of data state variable values for expression evaluation). The result state is the one provided by subsequent segments.

**Definition 3.24** (First Segment Selection Function). Let $t \in TS$ be a transition segment starting in the current state, $\sigma_C = (\lambda_O, cs, v)$ be a component state, and $\prec \subseteq TS \times TS$ be the total order of transition segments induced by the unique object identifiers. Then the first segment selection function of an automaton $Atm_C$ is a partial mapping $select_{first} : \Sigma_C \times \Lambda_C \rightarrow TS$

\[
N = \{ f \in TS \mid f.src = cs \land fireable(f) \}
\]

$t' \in N$

$\forall t'' \in N \setminus \{ t' \} : t' < t''$

$\therefore select_{first}(\sigma_C, \lambda_I) = t'$

The first segment selection function returns the smallest transition segment that starts in the current state and is fireable. It is undefined otherwise.

We can now characterize the component step function of an atomic component. The operational semantics of firing a transition consisting of one or more segments, implies that the output values are updated according to the output expressions and the data state variables are updated according to the postcondition assignments.

**Definition 3.25** (Automaton Component Step Function). Let $\sigma_C = (\lambda_O, cs, v)$ be a component state. Then the component step function $\delta_{step} : \Sigma_C \times \Lambda_I \rightarrow \Sigma_C$ is characterized by the following rules:

- Non-idle transition:

\[
select_{first}(\sigma_C, \lambda_I) = t
\]

\[
\delta_{seg}(\sigma_C, \lambda_I, t) = \sigma_C'
\]

\[
\delta_{step}(\sigma_C, \lambda_I) = \sigma_C
\]
• Idle transition:

\[
\begin{align*}
\text{dom.select}_{\text{first}}(\sigma_C, \lambda_I) &= \emptyset \\
\forall o \in O_C : \lambda_O(o) &= \epsilon \\
\forall dsv \in DSV : v'(dsv) &= v(dsv)
\end{align*}
\]

\[\delta_{\text{step}}(\sigma_C, \lambda_I) = (\lambda_O, cs, v')\]

3.3 System Structure Semantics

The behavior semantics defined in the previous section give the operational semantics of a single atomic component with an automaton specification. Next, we compose components to build larger systems from the atomic building blocks. Therefore, we introduce the set of channels, which consists of three disjunct subsets: the environment input channels, the environment output channels, and the local channels, respectively.

We use this distinction because the operational small-step semantics of a network of components connected with each other and the environment give constraints on when messages are transferred along channels. This depends on the causality specification of the atomic components.

Apart from the environment component, we do not consider hierarchic components in the model, but view them as syntactic sugar to structure large models for more better readability. From the semantic point of view the network of atomic components is relevant.

**Definition 3.26 (Atomic Port Sets).** Let \( C_{atm} \subset C \) be the set of atomic components and \( \text{ifc}_{syn}\text{_{atm}} = (I_{C_{atm}}, O_{C_{atm}}, \text{type}_{C_{atm}}^\text{port}, \text{init}_{C_{atm}}^\text{port}) \) be the syntactic interface of some component \( C_{atm} \in C_{atm} \). Then

- \( I_{atm} = \bigcup_{C_{atm} \in C_{atm}} I_{C_{atm}} \) is the set of all atomic input ports,
- \( O_{atm} = \bigcup_{C_{atm} \in C_{atm}} O_{C_{atm}} \) is the set of all atomic output ports.

**Definition 3.27 (Channels).** Let \( C_{env} \in C \setminus C_{atm} \) be the environment component and \( \text{ifc}_{C_{env}}^{\text{syn}} = (I_{C_{env}}, O_{C_{env}}, \text{type}_{C_{env}}^\text{port}, \text{init}_{C_{env}}^\text{port}) \) be the syntactic environment interface. Then

- \( CH_{in} \subseteq I_{C_{env}} \times I_{atm} \) is the set of environment input channels,
- \( CH_{out} \subseteq O_{atm} \times O_{C_{env}} \) is the set of environment output channels,
- \( CH_{local} \subseteq O_{atm} \times I_{atm} \) is the set of local channels.

Every port in \( I_{atm} \) is assumed to appear at most once in the tuples in \( CH_{in} \) and \( CH_{local} \). Likewise, the ports in \( O_{C_{env}} \) appear at most once in \( CH_{out} \). In other words a port can only have one source channel.

\[33\]
Channels define sender/receiver relations between the ports of atomic component and the environment. Clearly, each receiving port (i.e. atomic component input ports and environment output ports) can only be associated with a single source of information. Again, we assume type-correctness, i.e. ports connected via a channel must have the same type.

**Definition 3.28** (Causal Dependency Relation). Let \( C_{atm} \subset \mathcal{C} \) be the set of atomic components and \( C_{weak} \subseteq C_{atm} \) the subset of weakly causal components as specified in the model. Then the causal dependency relation is relation \( \rightsquigarrow \subset C_{atm} \times C_{atm} \) defined as follows:

\[
C_{weak} \rightsquigarrow C_{atm} \iff C_{weak} \in C_{weak} \land \exists (s, t) \in \mathcal{H}_{local} : s \in O_{C_{weak}} \land t \in I_{C_{atm}}
\]

A component is causal dependent on some component if the latter is weakly causal and there exists a local channel from an output port of the latter to an input port of the former.

**Constraint 3.20** (No Weakly Causal Cycles). To avoid weakly causal cycles we require that in the transitive closure \( \rightsquigarrow^* \) of the causal dependency relation no weakly causal component depends on itself.

\[
\forall C_{weak} \in C_{weak} : C_{weak} \not\rightsquigarrow^* C_{weak}
\]

**Definition 3.29** (Network State). Let \( C_{atm} \) be the set of atomic components. Then the network state is a tuple

\[
\sigma_{net} = ( (\sigma_C)_{C \in C_{atm}}, \lambda_{O_{env}} )
\]

The initial network state is given as follows

\[
\sigma_{net}^{init} = ( (\sigma_C^{init})_{C \in C_{atm}}, \lambda_{O_{env}} )
\]

The state of the network consists of the tuple of atomic component states and the valuation of the environment output ports. We denote with \( \Sigma_{net} \) the set of all network states for a given set of atomic components. \( \sigma_{net}, \sigma_C \) extracts the atomic component state from the network state for a given component \( C \) and \( \sigma_{net}, \lambda_{O_{env}} \) extracts the environment output port valuation.

The initial network state is constructed by using the initial component states as defined in Sec. 3.2.3. The output values of the environment ports need not be initialized because they all will be updated with current values after execution of the network’s components’ first step.

**Definition 3.30** (Current Input Port Value). Let \( L \) be a list of atomic components, \( \sigma_{net}^{orig}, \sigma_{net}^{cur} \) be the original and the current (intermediate) network state
and $\lambda_{env}$ be the current environment input. Then the current value of a port $p \in I_C$ of a component $C \in \mathcal{C}_{atm}$ is defined by

$$
\lambda_{cur}^C(L, p, \sigma_{net}^{orig}, \sigma_{net}^{cur}, \lambda_{env}) =
\begin{cases}
\sigma_{net}^{orig} \cdot \sigma_{C} \cdot \lambda_{O}(s), & \text{if } \exists (s, p) \in \mathcal{CH}_{local} \land s \in O_C \land C \in \mathcal{C}_{atm} \setminus \mathcal{C}_{weak} \\
\sigma_{net}^{cur} \cdot \sigma_{C} \cdot \lambda_{O}(s), & \text{if } \exists (s, p) \in \mathcal{CH}_{local} \land s \in O_C \land C \in L \cap \mathcal{C}_{weak} \\
\lambda_{env}(s), & \text{if } \exists (s, p) \in \mathcal{CH}_{in} \\
\epsilon, & \text{otherwise}
\end{cases}
$$

Since operational semantics describes network execution as a sequence of component executions, we must preserve the output of strongly causal components. This is achieved by using the two network states $\sigma_{net}^{orig}$ and $\sigma_{net}^{cur}$. The first is network state at the beginning of a network step (big-step semantics state), while the second refers to the network state incrementally updated during the component execution (small-step semantics state). Clearly, ports connected to strongly causal components receive the output value from the preceding step, while ports connected to weak causal receive the output from the current network state (note that the sending component will already be executed as required by the causal dependency relation. This is resembled by the list $L$, which contains the weakly causal components that have already been executed, i.e. their result is already present in $\sigma_{net}^{cur}$). Ports connected to environment inputs use the value from the current input valuation while unconnected ports are assumed to be empty.

Computing the values of environment outputs is done in a similar manner.

**Definition 3.31** (Current Environment Output Port Value). Let $L$ be a list of atomic components, $\sigma_{net}^{orig}$, $\sigma_{net}^{cur}$ be the original and the current network state. Then the current value of an environment output port $p \in O_{C_{env}}$ of the environment component $C_{env}$ is defined by

$$
\lambda_{cur}^{env}(L, p, \sigma_{net}^{orig}, \sigma_{net}^{cur}) =
\begin{cases}
\sigma_{net}^{orig} \cdot \sigma_{C} \cdot \lambda_{O}(s), & \text{if } \exists (s, p) \in \mathcal{CH}_{out} \land s \in O_C \land C \in \mathcal{C}_{atm} \setminus \mathcal{C}_{weak} \\
\sigma_{net}^{cur} \cdot \sigma_{C} \cdot \lambda_{O}(s), & \text{if } \exists (s, p) \in \mathcal{CH}_{out} \land s \in O_C \land C \in L \cap \mathcal{C}_{weak} \\
\epsilon, & \text{otherwise}
\end{cases}
$$

Depending on the causality of the sending component the current environment output value is taken from the current network state (i.e. weakly causal) or the original network state (i.e. strongly causal). Again, unconnected ports do not carry a value.

We can now define the small-step semantics rules for executing the list of atomic components.

**Definition 3.32** (Atomic Components Execution Rule). Let $\sigma_{net}$ be a network state and $\lambda_{env}$ be the current environment input valuation. Furthermore, let
L \neq [] be the list of atomic components from Catm sorted in inverse order of the causal dependency order. Then the following rules give the semantics of executing this network components:

\[
\text{exec}_{\text{net}}([], \sigma_{\text{net}}, \lambda_{\text{env}}) = \sigma_{\text{net}}
\]

\[
L.\text{first} = C
\]

\[
\text{exec}_{\text{net}}(L.\text{rest}, \sigma_{\text{net}}, \lambda_{\text{env}}) = \sigma_{\text{net}}'
\]

\[
\forall \text{in} \in I_C : \lambda_{I}(\text{in}) = \lambda_{I}^{\text{cur}}(L.\text{rest}, \text{in}, \sigma_{\text{net}}, \sigma_{\text{net}}', \lambda_{\text{env}})
\]

\[
\delta_{\text{step}}(\sigma_{\text{net}}, \sigma_C, \lambda_I) = \sigma_{C}'
\]

\[
\forall C' \in \text{Catm} : \sigma_{\text{net}}, \sigma_{C'} = \begin{cases} 
\sigma_C', & \text{if } C' = C \\
\sigma_{\text{net}}, \sigma_{C'}, & \text{if } C' \in L.\text{rest} \\
\sigma_{\text{net}}, \sigma_{C'}, & \text{otherwise}
\end{cases}
\]

\[
\forall p \in O_{\text{env}} : \sigma_{\text{net}}, \lambda_{O_{\text{env}}}(p) = \lambda_{O_{\text{env}}}^{\text{cur}}(L, p, \sigma_{\text{net}}, \sigma_{\text{net}}')
\]

\[
\text{exec}_{\text{net}}(L, \sigma_{\text{net}}, \lambda_{\text{env}}) = \sigma_{\text{net}}'
\]

If the list of (remaining) components to be executed is empty the resulting component states and the environment output port valuations remain unchanged.

Otherwise, we take the first component from the list and execute the rest of the network, i.e. all preceding components according to the causal dependency relation. After that we can compute the current input port values of the selected component and execute a single step using the \(\delta_{\text{step}}\) rule from the behavior semantics to obtain an updated component state. Finally, we compose the new network state by taking the updated component states of the selected component and its predecessors and the unchanged components state of all other, not yet executed components. Likewise, the environment output port valuations are updated.

We can finally define the big-step semantics execution rule of the network step function.

**Definition 3.33 (Network Execution Rule).** Let \(\sigma_{\text{net}}\) be a network state, \(\lambda_{\text{env}}\) be the current environment input, and \(L\) the complete list of atomic components sorted as before. The network execution function \(\delta_{\text{net}} : \Sigma_{\text{net}} \times \Lambda_{\text{env}} \rightarrow \Sigma_{\text{net}}\) is defined according to the following rule:

\[
\text{exec}_{\text{net}}(L, \sigma_{\text{net}}, \lambda_{\text{env}}) = \sigma_{\text{net}}'
\]

\[
\delta_{\text{net}}(\sigma_{\text{net}}, \lambda_{\text{env}}) = \sigma_{\text{net}}'
\]

The big-step semantics of network execution is defined by executing every component exactly once and using the updated network state as the successor state.
4 The Target Language C0

For transforming AutoFocus 3 models into C0 programs, we need the following language features of C0, which are a subset of the C0 language features [6, 5]:

- Side-effect free expressions
- Global and local variables
- Non-recursive procedures

4.1 Syntax

This section defines the syntax of the C0 language subset.

4.1.1 Types

We use the following C0 types:

- **bool** - representing the boolean logic values \{true, false\}.
- **int** - representing the 32-bit integer values \{-2^{31}, \ldots, 2^{32} - 1\}.
- **struct** - representing C structures for building composite types. Structures are defined by associating a unique name of the struct with a list of identifier / type pairs. We require identifiers to be unique within one struct definition and we require that no structure is (directly or indirectly) recursive. However, structures may contain members with other struct types. e.g. struct Point { int x; int y; } declares a type Point consisting of two members x and y of type int and struct Rect { struct Point ul; struct Point lr; } declares a type Rect consisting of two struct Points.

Note that there is no void type in C0. Thus, functions always must return a value, which then of course can be ignored by the caller.

4.1.2 Expressions

**Literals.** The literals of bool and int are expressions: true, false, \(-2^{31}, \ldots, 2^{32} - 1\). Furthermore, the generated code will sometimes contain alphanumeric character constants, which are replaced with integer literals by a pre-processing step. These constants always begin with a capital alpha letter to distinguish them from variables, e.g. PRE_PROC_VAL.

\(^4\)We use the term procedure in the context of the C0 language to distinguish it from the term function, which we use in the context of the AutoFocus 3 language. Later, functions will be mapped onto procedures. However, procedures will also be used for other purposes like component step execution or transition segment evaluation.
Table 4: Unary C0 Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Operand Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$!e_0$</td>
<td>negation</td>
<td>bool</td>
<td>bool</td>
</tr>
<tr>
<td>$-e_0$</td>
<td>unary minus</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>$\sim e_0$</td>
<td>bit-wise negation</td>
<td>int</td>
<td>int</td>
</tr>
</tbody>
</table>

**Variables.** Variables are denoted by alphanumeric character sequences, which may also contain the underscore `_` character. Variable identifiers always begin with a lower-case letter to distinguish them from pre-processor constants, e.g. `current_state`.

**Member Access.** If $e_0$ is an expression of some `struct` type $S$ and $m$ is a member identifier of $S$ of type $T_m$ then $e_0.m$ is an expression of type $T_m$ returning the member value of the instance $e_0$ of $S$.

**Unary Operators.** If $e_0$ is an expression with the matching operand type, then the unary operator applications given in Tab. 4 are also expressions with the given result type.

**Binary Operators.** If $e_1, e_2$ are expressions with equal types matching the operand type, then the binary operator applications given in Tab. 5 are also expressions with the given result type.

### 4.1.3 Statements

Our C0 language subset allows the following simple statements (let $e_0, e_1, \ldots$ be expressions and $v$ a variable identifier). $STM_T$ will denote the set of all C0 statements including sequentially composed statements.

**Assignment.** $v = e_0$ assigns the value of the expression $e_0$ to variable $v_0$.

**Structure Instantiation.** $v_0 = S(e_0, \ldots, e_n)$ assigns to $v_0$ the instance of the `struct` type $S$ using $e_0$ to $e_n$ as the structure members.

**Procedure Call.** $v_0 = p(e_0, \ldots, e_n)$ assigns the result of the procedure $p$ called with arguments $e_0, \ldots, e_n$ to the variable $v_0$. When constructing programs later, we require the call graph of the program to be free of cycles, i.e., we require procedures to be non-recursive. This not only guarantees termination of procedure calls, but also allows to compute the execution time and stack memory consumption of a call at compile time.
### Table 5: Binary C0 Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Operand Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e_1 + e_2$</td>
<td>addition</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>$e_1 - e_2$</td>
<td>subtraction</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>$e_1 * e_2$</td>
<td>multiplication</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>$e_1 / e_2$</td>
<td>division</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>$e_1 % e_2$</td>
<td>modulo</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>$e_1 &amp; e_2$</td>
<td>bit-wise and</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>$e_1</td>
<td>e_2$</td>
<td>bit-wise or</td>
<td>int</td>
</tr>
<tr>
<td>$e_1 ^ e_2$</td>
<td>bit-wise xor</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>$e_1 &lt;&lt; e_2$</td>
<td>shift left</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>$e_1 &gt;&gt; e_2$</td>
<td>shift right</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>$e_1 &gt;&gt;&gt; e_2$</td>
<td>unsigned shift right</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>$e_1 &lt; e_2$</td>
<td>less than</td>
<td>int</td>
<td>bool</td>
</tr>
<tr>
<td>$e_1 &lt;= e_2$</td>
<td>less or equal</td>
<td>int</td>
<td>bool</td>
</tr>
<tr>
<td>$e_1 &gt;= e_2$</td>
<td>greater or equal</td>
<td>int</td>
<td>bool</td>
</tr>
<tr>
<td>$e_1 &gt; e_2$</td>
<td>greater than</td>
<td>int</td>
<td>bool</td>
</tr>
<tr>
<td>$e_1 == e_2$</td>
<td>equal</td>
<td>int, bool</td>
<td>bool</td>
</tr>
<tr>
<td>$e_1 != e_2$</td>
<td>not equal</td>
<td>int, bool</td>
<td>bool</td>
</tr>
<tr>
<td>$e_1 &amp;&amp; e_2$</td>
<td>non-strict and</td>
<td>bool</td>
<td>bool</td>
</tr>
<tr>
<td>$e_1</td>
<td></td>
<td>e_2$</td>
<td>non-strict or</td>
</tr>
</tbody>
</table>

**Procedure Return.** \( \text{return } e_0 \) returns from the currently executing procedure with \( e_0 \) being the procedure result.

Note that procedure calls are not part of C0 expressions, while AutoFocus 3 allows expressions containing function calls. The C0 restriction results in the fact that C0 expressions are free of side-effects. In contrast, AutoFocus 3 functions are by definition free of side-effects. Thus, when mapping AutoFocus 3 expressions to C0 expressions, we need to sequentialize nested functions calls by introducing new local variables, each storing the result of a single C0 procedure call. Afterwards, we can use these variables as arguments in successive procedure calls. This function call expressions from AutoFocus 3 are extracted into a semantically equivalent series of C0 procedure calls.

Note also, that any C0 procedure is required to have exactly one return statement as its last statement. Again, a local variable can be used to store the procedure result and return this value at the end of the procedure.

Now, let \( s_0, s_1, \ldots \) be statements, \( b_0 \) an expression of type \( \text{bool} \), and \( c \) an unused variable. Then the following statements are also C0 statements:
If-Conditional. if ( b0 ) { s0 } executes s0 only if the condition b0 evaluates to true and does nothing otherwise.

If-Else-Conditional. if ( b0 ) { s0 } else { s1 } executes s0 only if the condition b0 evaluates to true and executes s1 otherwise.

Sequential Execution. s0; s1 executes s0 and afterwards executes s1. We require that s0 does not contain or end with a return statement. Sequential execution is guaranteed to terminate, since all other C0 statements (as introduced here) terminate and the sequence of statements, i.e. the C0 program, is finite.

4.1.4 Programs

Having introduced the basic syntactic elements of the C0 language, we now define the syntactic structure of C0 programs. C0 programs consist of five parts: type definitions, type declarations, variable declarations, procedure declarations, and procedure definitions.

Note that, here, we do not cover the topic of compilation units in full detail. We assume that there is a single header file containing all the declarations and a single implementation file containing all the definitions. However, the code generator will use different compilation units for different components of the AutoFocus 3 model.

Type Definitions. A type definition introduces a user-defined struct type as explained above.

Type Declarations. Type declarations associate a new alias with a defined type. The syntax of type declarations is typedef <type> <identifier>, e.g. typedef struct Point MyPoint; creates the alias MyPoint for the defined type Point.

Variable Declarations. Variable declarations associates a variable identifier with its type. Its syntax is <type> <variable identifier>, e.g. MyPoint myPointVar; creates a new variable named myPointVar of type MyPoint.

Procedure Declarations. Procedure declarations declare a signature for a part of the C0 program. The syntax of procedure declarations is <type> <procedure name> (<parameter list>), where <parameter list> is an optional list of comma-separated variable declarations declaring the procedure’s parameters.

Procedure Definitions. Procedure definitions associate a procedure signature with an implementation, i.e. a finite sequence of C0 statements. The last statement of this sequence is a return statement. The syntax of procedure definitions is <type> <procedure name> (<parameter list>) { <statement list> }.
C0 Programs. A C0 program is composed of a list of type definitions, a list of type declarations, a list of variable declarations, a list of procedure declarations, and a list of procedure definitions. We assume that there is a parameter-less procedure int main() with return type int. We define this as the starting procedure of the C0 program.

4.2 Semantics

This section defines the semantics of C0 programs.

We use the same denotations as in Sec. 3. However, ID now does not include identifiers starting with underscore as used for variables in AUTOFOCUS 3. Later, we will map AUTOFOCUS 3 variables to C0 variables by prepending a suitable prefix.

We denote by STMT the set of all C0 statement instances (including all composite statement instances).

4.2.1 Type Environment

Definition 4.1 (C0 Type Environment). Let \( \text{TYPE}_{C0} = \{\text{bool, int}\} \cup \{\text{struct}(i, m) | i \in \text{ID}, m \in \text{list ID} \times \text{TYPE}_{C0}\} \) be the set of pre-defined and user-defined types. Then the C0 type environment is a mapping \( \upsilon_{\text{type}}^{C0} : \text{ID} \rightarrow \text{TYPE}_{C0} \), which maps type identifiers to types.

The type environment mapping is statically defined by the type definition and type declaration parts of the C0 program. Each type definition becomes a member of \( \{\text{struct}(i, m) | i \in \text{ID}, m \in \text{list ID} \times \text{TYPE}_{C0}\} \) and \( \upsilon_{\text{type}}^{C0}(\text{struct} i) \) maps to this member identified by \( i \). Furthermore, every type declaration also becomes a mapping in \( \upsilon_{\text{type}}^{C0} \). The set of C0 values will be denoted by \( \text{VALUE}_{C0} \).

Variable Type

Definition 4.2 (Variable Type). Let \( V \subset \text{ID} \) be the set of variables declared by a given model. Then \( \upsilon_{\text{var}}^{C0} : V \rightarrow \text{TYPE}_{C0} \) is a mapping, which maps each variable to its specified type.

Atomic components may have data state variables and local variables defined in transition segment input patterns. Each variable has a fixed typed as given by the mapping \( \upsilon_{\text{var}}^{C0} \).

Note that the code generator does not generate the main procedure. Since we are interested in the behavioral equivalence between AUTOFOCUS 3 models and the generated C0 program, the topic of embedding the model in an execution environment is out of scope of this report. This development phase is usually called deployment and consists of embedding the C0 program into a task structure and, in case of AUTOFOCUS 3 connecting inputs and outputs of the model to inputs and outputs of the execution environment (like sensors and actors of an embedded system).
4.2.2 Variable Environment

Definition 4.3 (Variable Environment). A variable environment is a mapping $v_{\text{C}0}^{\text{var}} : \text{ID} \rightarrow \text{VALUE}_{\text{C}0}$, which maps variable identifiers to their currently stored value.

$\Upsilon_{\text{C}0}^{\text{var}}$ denotes the set of all variable environments. Clearly, a variable environment is assumed to be type safe:

$$\forall v \in \text{dom.} v_{\text{C}0}^{\text{var}} : v_{\text{C}0}^{\text{var}}(v) \in \text{type}_{\text{C}0}^{\text{var}}(v)$$

Variable environments represent the current state of a set of variables (i.e. those variables visible at some point of the execution of a C0 program). Thus, variable environments can be changed by updating variables. Formally, this is captured by the following function updating the variable’s current value.

Definition 4.4 (C0 Value Update Function). Let $v_{\text{C}0}^{\text{var}} \in \Upsilon_{\text{C}0}^{\text{var}}$ be a variable environment, $t \in \text{TYPE}_{\text{C}0}$ be some type, $x \in \text{VALUE}_{\text{C}0}$ be an arbitrary value of type $t$ and $u \in \text{ID}$, $v \in \text{ID}$, with $\text{type}_{\text{C}0}^{\text{var}}(u) = t$, arbitrary variables of type $t$. Then the value update function is a mapping $\text{update} : \Upsilon_{\text{C}0}^{\text{var}} \times \text{ID} \times \text{VALUE}_{\text{C}0} \rightarrow \Upsilon_{\text{C}0}^{\text{var}}$ defined as follows:

$$\text{update}(v_{\text{C}0}^{\text{var}}, u, x)(v) = \begin{cases} x & \text{if } u = v \\ v_{\text{C}0}^{\text{var}}(v) & \text{if } u \neq v \end{cases}$$

4.2.3 Procedure Table

Definition 4.5 (Procedure Definition). A procedure definition $\pi$ is a 4-tuple consisting of the following components:

- $\pi.\text{body} \in \text{STMT}$ defines the body of the procedure with the last statement being a return statement.
- $\pi.\text{rtype} \in \text{TYPE}_{\text{C}0}$ specifies the return type.
- $\pi.\text{params} \in \text{list ID} \times \text{TYPE}_{\text{C}0}$ is the finite list of parameter name and type tuples defining the list of procedure parameters.
- $\pi.\text{locvars} \in \text{list ID} \times \text{TYPE}_{\text{C}0}$ is the finite list of local variable name and type tuples, which does not include the procedure parameters.

$\Pi$ denotes the set of all procedure definitions. If $v_0$ is a variable name, we denote by $v_0 \in \pi.\text{locvars}$ that $v_0$ represents a local variable of the procedure definition $\pi$. We do analogously for $\pi.\text{params}$. Furthermore, $\pi.\text{params.0}, \ldots, \pi.\text{params.n}$ denote the first, $\ldots$, the $n$th entry of the parameter list of the procedure, respectively. We assume that the identifiers of local variables and
local parameters are distinct and that any variable identifier contained neither in the list of local variables nor the list of local parameters is assumed to be a global variable. The code generator uses distinct prefixes for local variables, local parameters and global variables to fulfill this assumption syntactically.

**Definition 4.6 (Procedure Table).** The procedure table is a mapping \( ptable : ID \rightarrow \Pi \), which maps procedure names to procedure definitions.

For reasons of simplicity we assume unique procedure names, i.e. the procedure name is sufficient to bind a procedure call to a procedure definition. Usually the complete procedure signature is used for this binding process. The code generator will take care of generating unique procedure names later.

The procedure table is statically defined by the procedure declaration and procedure definition parts of the C0 program.

### 4.2.4 Program Configuration

From the semantic point of view a C0 program in execution is a tuple consisting of the current state of the global variables, the current stack of unfinished procedure call contexts and the list of remaining statements to be executed (called the program rest). This is captured by the notion of a program configuration.

**Definition 4.7 (Program Configuration).** A program configuration \( \sigma \) is a 2-tuple consisting of the following components:

- \( \sigma.genv \in \Upsilon_{\text{var}}^{C0} \) is the global variable environment.
- \( \sigma.pctxs \in \text{list} \Pi \times \Upsilon_{\text{var}}^{C0} \times ID \) is the stack of unfinished procedure call contexts each defined by a tuple of the definition of the called procedure (its current value accessed via \( \sigma.pctxs.first.pdef \)), the local variable environment (its current value accessed via \( \sigma.pctxs.first.lenv \)), and return value storage identifier (its current value accessed via \( \sigma.pctxs.first.rid \)).

Including the return value storage location in each procedure call context simplifies the handling of procedure calls and return statements.

The initial program configuration \( \sigma_{\text{init}} \) is defined as follows:

- \( \sigma_{\text{init}}.genv = \emptyset \); all global variables values are uninitialized.
- \( \sigma_{\text{init}}.pctxs = [] \); the procedure call context stack is empty.

### 4.2.5 Expression Evaluation

**Definition 4.8 (C0 Expression Evaluation).** Let \( \sigma \) be a program configuration and \( e \) be an expression. Then the function \( \text{eval} \) evaluates the expression as follows:
• Let $e = l$ be a literal:
  \[ \text{eval}(l, \sigma) = l \]
  i.e. literals evaluate to their respective value independent of $\sigma$.

• Let $e = <uo> e_0$ be an unary operator expression from Tab. 4:
  \[ \text{eval}(<uo> e_0, \sigma) = <uo> \text{eval}(e_0, \sigma) \]
  with the usual semantic interpretation of the unary operators.

• Let $e = e_0 <bo> e_1$ be a binary operator expression from Tab. 5:
  \[ \text{eval}(e_0 <bo> e_1, \sigma) = \text{eval}(e_0, \sigma) <bo> \text{eval}(e_1, \sigma) \]
  with the usual semantic interpretation of the binary operators.

• The member access $\text{eval}(e_0.m, \sigma)$ evaluates to the value of the member $m$
  of the struct value $\text{eval}(e_0, \sigma)$.

• Variables are evaluated using the program configuration, let $v \in ID$ be a
  variable identifier:
  \[ \text{eval}(v, \sigma) = \begin{cases} 
  \sigma.pctxs.lenv(v) & \text{if } v \in \text{dom}.(\sigma.pctxs.lenv) \\
  \sigma.genv(v) & \text{otherwise}
  \end{cases} \]

\section*{4.2.6 If-Then Execution}

\textbf{Definition 4.9} (If-Then True Evaluation).

\[ \text{eval}(b_0, \sigma) = \text{true} \]
\[ \langle \sigma, s_0 \rangle \mapsto \sigma' \]
\[ \langle \sigma, \text{if } (b_0) \{ s_0 \} \rangle \mapsto \sigma' \]
\[ \hfill \triangleright \]

If the guard expression of the if-then evaluates to true, the program execution
is continued with the sub-program of the then branch.

\textbf{Definition 4.10} (If-Then False Evaluation).

\[ \text{eval}(b_0, \sigma) = \text{false} \]
\[ \langle \sigma, \text{if } (b_0) \{ s_0 \} \rangle \mapsto \sigma \]
\[ \hfill \triangleright \]

If the guard expression of the if-then evaluates to false, the program state
remains unchanged and the program is continued as specified by the program
state’s program rest.
4.2.7 If-Then-Else Execution

**Definition 4.11 (If-Then-Else True Evaluation).**
\[
eval(b_0, \sigma) = \text{true} \\
\langle \sigma, s_0 \rangle \mapsto \sigma'
\]

If the guard expression of the if-then-else evaluates to true, the program execution is continued with the sub-program of the then branch.

**Definition 4.12 (If-Then-Else False Evaluation).**
\[
eval(b_0, \sigma) = \text{false} \\
\langle \sigma, s_1 \rangle \mapsto \sigma'
\]

If the guard expression of the if-then-else evaluates to false, the program execution is continued with the sub-program of the else branch.

4.2.8 Assignment Execution

**Definition 4.13 (Local Variable Assignment).**
\[
v_0 \in \sigma.pctxs.first.pde.locvars \\
eval(e_0, \sigma) = e_0 \\
\sigma'.pctxs.first.lenv = \text{update}(\sigma.pctxs.first.lenv, v_0, e_0) \\
\langle \sigma, v_0 = e_0 \rangle \mapsto \sigma'
\]

If the left-hand side variable of an assignment refers to a local variable, the local variable environment of the current procedure context is updated with the evaluated right-hand side expression, accordingly.

**Definition 4.14 (Local Parameter Assignment).**
\[
v_0 \in \sigma.pctxs.first.pde.params \\
eval(e_0, \sigma) = e_0 \\
\sigma'.pctxs.first.lenv = \text{update}(\sigma.pctxs.first.lenv, v_0, e_0) \\
\langle \sigma, v_0 = e_0 \rangle \mapsto \sigma'
\]

If the left-hand side variable of an assignment refers to a local parameter, the local variable environment of the current procedure context is updated with the evaluated right-hand side expression, accordingly. Note that this will not happen for C0 programs generated from AutoFocus 3 models. Parameters are never assigned new values.
Definition 4.15 (Global Variable Assignment).

\[
\begin{align*}
    \forall v_0 \in \sigma.pctxs.first.pdef.locvars \\
    \forall v_0 \in \sigma.pctxs.first.pdef.params \\
    \text{eval}(e_0, \sigma) = e_0 \\
    \sigma'.genv = \text{update}(\sigma.genv, v_0, e_0) \\
    \langle \sigma, v_0 = e_0 \rangle \mapsto \sigma' \\
\end{align*}
\]

If the left-hand side variable of an assignment refers to neither a local variable nor a local parameter, the variable is assumed to be a global one. Thus the global variable environment of the current program state is updated with the evaluated right-hand side expression, accordingly.

4.2.9 Structure Instantiation Execution

Definition 4.16 (Structure Instantiation Execution).

\[
\begin{align*}
    \text{eval}(e_0, \sigma) = e_0 \\
    \vdots \\
    \text{eval}(e_n, \sigma) = e_n \\
    s_0 = S(e_0, \ldots, e_n) \in \text{VALUE}_C \\
    \langle \sigma, v_0 = s_0 \rangle \mapsto \sigma' \\
    \langle \sigma, v_0 = S(e_0, \ldots, e_n) \rangle \mapsto \sigma' \\
\end{align*}
\]

Structure instantiation creates the new structure instance by evaluation of the argument expressions and assigns the resulting value to the given variable.

4.2.10 Procedure Call Execution

Definition 4.17 (Procedure Call Execution).

\[
\begin{align*}
    \pi = \text{ptable}(p) \\
    \text{eval}(e_0, \sigma) = e_0 \\
    \vdots \\
    \text{eval}(e_n, \sigma) = e_n \\
    \text{lenv}' = \text{update}(\ldots, \text{update}(\emptyset, \pi.params.0.name, e_0), \ldots, \pi.params.n.name, e_n) \\
    \sigma''.pctxs.first = (\pi, \text{lenv}'', v_0) \\
    \sigma''.pctxs.rest = \sigma.pctxs \\
    \langle \sigma'', \pi.body \rangle \mapsto \sigma' \\
    \langle \sigma, v_0 = p(e_0, \ldots, e_n) \rangle \mapsto \sigma' \\
\end{align*}
\]

\[\triangleright\]
Using the procedure definition stored in the procedure table a procedure call is executed by evaluating all the arguments and assigning these values to the formal parameters of the procedure in a new local variable environment. After that the body of the procedure is executed using the new variable environment as part of the new procedure call context.

4.2.11 Procedure Return Execution

Definition 4.18 (Procedure Return Execution).

\[ \sigma.pctxs \neq [] \]
\[ \text{eval}(e_0, \sigma) = e_0 \]
\[ \sigma'.pctxs.rest = \sigma.pctxs.rest.rest \]
\[ \sigma'.pctxs.first.lenv = \text{update}(\sigma.pctxs.rest.first.lenv, \sigma.pctxs.first.rid, e_0) \]
\[ (\sigma, \text{return } e_0) \rightarrow \sigma' \]

A procedure return is executed by evaluating the return expression and storing this value in the local variable environment of the surrounding procedure context. Furthermore, the current procedure context is dropped and the execution continues with the program rest as given by the program state.

5 Mapping AutoFocus 3 Model to C0 Program

This section describes the mapping from an AutoFocus 3 model to a C0 program. This mapping is the specification of the transformation executed by the code generator. For each mapping step we give the proof of simulation equivalence between the AutoFocus 3 semantics as given in Sec. 3 and the C0 semantics as given in Sec. 4.2. We use the running example to clarify the generators result where needed. Note that we will not treat syntactic sugar in detail. The generator will avoid name clashes (by extending identifiers found in the AutoFocus 3 model in a suitable way as needed) and produce human readable code.

5.1 General Proceeding

First, we map the data definitions, e.g. types and terms (pre-defined or defined in the data dictionary), onto the types and expressions of the C0 language. Furthermore, we show how the pattern matching is implemented in C0.

We then map the environment component and its atomic sub-components of the model into the state space of the C0 program by mapping their interface ports onto global C0 variables. Next, we transform the behavior of atomic components into C0 procedures. Automaton specifications’ dynamic data, e.g. the current control state and the data state variables will be mapped into the state space of the C0 program as well. The transition function defined by the automaton is transformed into a set of C0 procedures that alter this automaton.
state and the interface state. We use an optimized implementation, basically a depth-first traversing of the graph formed by the transition segments, thus taking advantage structuring given in the model.

Finally, we map the environment component into the C0 program by generating a C0 procedure that executes the network of atomic components according to the AutoFocus 3 semantics. The C0 program will not be complete at that point, but rather provide a set of C0 procedures that must be called in a fixed sequence. We summarize this programming interface at the end of this section. The integration of the C0 program into an executable environment, usually called a deployment, is out of scope this report. Instead, we only indicate the general procedure necessary.

Throughout this section we will use the polymorphic function $\text{map}^{AF3}_{C0} : A \rightarrow C$ where $A$ and $C$ depend on the context. For example, if $A$ is the set of all user-defined AutoFocus 3 types then $C$ is the set of all C0 type definitions. Similarly, if $A$ is the set of AutoFocus 3 expression then $C$ is the set of C0 expressions and C0 statements, since some expressions must be mapped to a behaviorally equivalent sequence of C0 statements using local variables as we will see shortly.

5.2 Data Types and Terms

The code generator generates a header file called `data_dictionary.h` and an implementation file `data_dictionary.c`. The header file contains the type and procedure declarations for user-defined types and functions. The implementation file contains the implementations of constructor, discriminator, selector, and user-defined functions.

5.2.1 Mapping Types and Values

**Pre-defined Types** The pre-defined types `boolean` and `int` are mapped to their C0 counterparts `bool` and `int`, respectively. The semantic equivalence follows directly from the definitions.

**Variant Types** User-defined variant types from \{variant($i, v | i \in ID, v \in list ID$)\} are mapped to a subset of the `int` type values. For each variant constructor $v$ a unique integer value is used. No constructor functions are generated since the generator uses the integer values directly. Discriminator functions are implemented by corresponding procedures based on integer equality. However, in order to keep the C0 program readable this mapping is done by using preprocessor macros. The generator introduces substituting macros for every variant constructor $v$ and a `typedef` for the type identifier $i$.

Discriminator function calls are mapped to `int` value equality checks.

Example:

```c
typedef int TYPE_StatusType;
```
Tuple Types  User-defined tuple types are mapped to C0 struct types. For an AUTOFOCUS 3 tuple type \{tuple(i, m) \mid i \in \text{ID}, m \in \text{list \ ID} \times \text{TYPE}_{AF3}\} a corresponding struct i is generated. Each selector m of the single constructor becomes a member of the structure with the correspondingly mapped type.

Constructor function calls are mapped to structure instantiation and subsequent member assignments while selector function calls are mapped to the member access expressions.

Example:

```c
struct ResultType_Result {
  int  digit0;
  int  digit1;
  int  digit2;
  int  digit3;
  bool overflow;
};
typedef struct ResultType_Result TYPE_ResultType;
```

**Definition 5.1 (Type Mapping).** The type mapping \( \text{map}_{AF3}^{C0} : \text{TYPE}_{AF3} \rightarrow \text{TYPE}_{C0} \) is defined as follows:

\[
\begin{align*}
\text{map}_{AF3}^{C0}(\text{boolean}) & = \text{bool} \\
\text{map}_{AF3}^{C0}(\text{int}) & = \text{int} \\
\text{map}_{AF3}^{C0}(\text{variant}(i, v)) & = \{0, \ldots, |v| - 1\} \subseteq \text{int} \\
\text{map}_{AF3}^{C0}(\text{tuple}(i, m)) & = \text{struct}(i, m) \quad \text{with map}_{AF3}^{C0} \text{ applied point-wise} \\
& \quad \text{to the selector types of } m
\end{align*}
\]

**Definition 5.2 (Value Mapping).** The value mapping \( \text{map}_{AF3}^{C0} : \text{VALUE} \rightarrow \text{VALUE}_{C0} \) is defined as follows:

\[
\begin{align*}
\text{map}_{AF3}^{C0}(c) & = c & \text{if } c \in \text{boolean} \lor c \in \text{int} \\
\text{map}_{AF3}^{C0}(v) & = i & \text{if } v \text{ is the } i\text{th variant constructor} \\
\text{map}_{AF3}^{C0}(T(e_0, \ldots, e_n)) & = T'(\text{map}_{AF3}^{C0}(e_0), \ldots, \text{map}_{C0}^{AF3}(e_n)) & \text{is the corresponding struct instance.}
\end{align*}
\]

These two mappings induce the equivalence relation of types and values. The following definition captures this prerequisite for the proofs in the remainder of this report.
Definition 5.3 (Semantic Equivalence of Values). Let \( \text{map}^{AF3}_{C0} \) be defined as above. Then the equivalence relation \( \sim \subset \text{VALUE} \times \text{VALUE}_{C0} \) is defined as follows:

\[
\text{val}_{AF3} \sim \text{val}_{C0} \iff \text{map}^{AF3}_{C0}(\text{val}_{AF3}) = \text{val}_{C0}
\]

Note that the strong typing of AutoFocus 3 expressions ensures that variant type values are not mixed with integer values although both are represented by C0 \texttt{int} values. E.g. \texttt{Busy + 5} is a type-correct expression in C0 since \texttt{Busy} is syntactic sugar for an \texttt{int} value. However, the corresponding AutoFocus 3 expression is not type-correct, i.e. the model is not consistent and the generator does not accept it as input.

5.2.2 Object-dependent Mapping of Variables

TODO: rework this

In AutoFocus 3 variables appear in different contexts or scopes, e.g. data state variables of an automaton or variables bound by input pattern matching of some transition segment. In particular, variable names are unique only with respect to the corresponding context. However, variables of different scopes (e.g. from different components) will be mapped into the global variable environment of the C0 program. To avoid name collisions the code generator uses a context depending variable name mapping.

Definition 5.4 (Variable Name Prefix Mapping). Let \( \text{OBJS} \) be the set of all objects in the AutoFocus 3 model and \( o \in \text{OBJS} \). Then the variable name prefix mapping \( \text{map}^{AF3}_{C0} : \text{OBJS} \times \text{ID} \to \text{ID} \) is an injective mapping of variable names with respect to the context object, i.e.

\[
\forall o \in \text{OBJS} : \forall o' \in \text{OBJS} : \\
o \neq o' \Rightarrow \forall i \in \text{ID} : \text{map}^{AF3}_{C0}(o, i) \neq \text{map}^{AF3}_{C0}(o', i)
\]

To keep the formulas in the remainder of this report more readable, we omit the variable name mapping, i.e. instead of \( \text{map}^{AF3}_{C0}(o, \text{var}) \) for C0, we simply write \text{var} to represent both the original and the mapped variable identifier. The code generator keeps track of the context it is working on and uses suitable prefix strings (e.g. the component name and/or unique object identifiers) to obtain unique C0 variable names.

We derive the equivalence relation of variable environments and program configurations, as defined in Def. 3.3 and Def. 4.7, from the mappings and equivalences introduced so far.

Definition 5.5 (Variable Environment Program Configuration Compatibility). Let \( \nu^{var}_{AF3} \) be a variable environment, \( \sigma \) be a C0 program state. Then

\[
\nu^{var}_{AF3} \sim \sigma \iff \forall v \in \text{dom.} \nu^{var}_{AF3} : \nu^{var}_{AF3}(v) \sim \sigma(v)
\]
This equivalence relation requires only that the value of all variables of the AutoFocus 3 environment is equivalent to the value of the corresponding C0 program configuration. However, the latter may contain additional variable valuations, in particular variables from other contexts or temporary variables used for local computation.

The type of the variables can be deduced from the model object, e.g. a component port is mapped to a variable with a suitable identifier and the type corresponding to the type given in the model.

5.2.3 Terms

As defined in Sec. 3.1.4 and Sec. 4.2.5 the expression languages of AutoFocus 3 and C0 are quite similar. In particular, we assume semantic equivalence between the operators given in Tabs. 2 and 4, and 3 and 5.

Sometimes AutoFocus 3 expressions must be unfolded and transformed into its linearized form because of the restrictions in C0 expressions (e.g. forbidden nested procedure calls). This transformation is given by the following schemas. In these schemas unfold expands to the code needed for unfolding the expression. unfoldres returns the C0 variable introduced to store the result of the corresponding unfold code, i.e. in subsequent expressions unfoldres(e) can be used to obtain the evaluation result of e. The type of the newly introduced variable can always be derived from the source expression due to the strong typing of AutoFocus 3 expressions. Note that we do not consider more intelligent generation mechanisms that might apply unfolding only when necessary.

Definition 5.6 (Unfolding Operation). Let e ∈ EXP be an expression from an AutoFocus 3 model. Then unfold : EXP → STM and unfoldres : EXP → ID map expressions to their unfolded C0 code and the result storage variable identifier, respectively, according to the following rules:

- Atom values
  \[
  \begin{align*}
  e &\equiv \text{atom} \\
  \text{unfoldres}(e) &= r \\
  \text{unfold}(e) &= r = \text{atom};
  \end{align*}
  \]

- Variables (object dependent name omitted)
  \[
  \begin{align*}
  e &\equiv \text{var} \\
  \text{unfoldres}(e) &= r \\
  \text{unfold}(e) &= r = \text{var};
  \end{align*}
  \]

- Unary operators
  \[
  \begin{align*}
  e &\equiv <uo> e_0 \\
  \text{unfoldres}(e) &= r \\
  \text{unfold}(e) &= \text{unfold}(e_0); \\
  r &= <uo> \text{unfoldres}(e_0);
  \end{align*}
  \]
• Strict binary operators

\[ e \equiv e_0 <bo> e_1 \]
\[ \text{unfoldres}(e) = r \]
\[ \text{unfold}(e) = \]
\[ \text{unfold}(e_0); \]
\[ \text{unfold}(e_1); \]
\[ r = \text{unfoldres}(e_0) <bo> \text{unfoldres}(e_1); \]

• Non-strict binary operators \&\& and ||

\[ e \equiv e_0 \&\& e_1 \]
\[ \text{unfoldres}(e) = r \]
\[ \text{unfold}(e) = \]
\[ \text{unfold}(e_0); \]
\[ \text{if} \ (\text{unfoldres}(e_0)) \{ \]
\[ \text{unfold}(e_1); \]
\[ r = \text{unfoldres}(e_1); \]
\[ \} \text{ else } \{ \]
\[ r = \text{false}; \]
\[ \} \]

\[ e \equiv e_0 || e_1 \]
\[ \text{unfoldres}(e) = r \]
\[ \text{unfold}(e) = \]
\[ \text{unfold}(e_0); \]
\[ \text{if} \ (\text{unfoldres}(e_0)) \{ \]
\[ r = \text{true}; \]
\[ \} \text{ else } \{ \]
\[ \text{unfold}(e_1); \]
\[ r = \text{unfoldres}(e_1); \]
\[ \} \]

• If-then-else expression

\[ e \equiv \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \text{ fi} \]
\[ \text{unfoldres}(e) = r \]
\[ \text{unfold}(e) = \]
\[ \text{unfold}(e_0); \]
\[ \text{if} \ (\text{unfoldres}(e_0)) \{ \]
\[ \text{unfold}(e_1); \]
\[ r = \text{unfoldres}(e_1); \]
\[ \} \text{ else } \{ \]
\[ \text{unfold}(e_2); \]
\[ r = \text{unfoldres}(e_2); \]
\[ \} \]
• Discriminator function calls

\[
e \equiv \text{in}_\text{CON}(e_0)
\]

\[
\text{unfoldres}(e) = r
\]

\[
\text{unfold}(e) =
\]

\[
\text{unfold}(e_0);
\]

\[
r = (\text{unfoldres}(e_0) \equiv \text{CON};)
\]

• Constructor function calls (sel0, ..., seln denote the selector slots)

\[
e \equiv \text{CON}(e_0, \ldots, e_n)
\]

\[
\text{unfoldres}(e) = r
\]

\[
\text{unfold}(e) =
\]

\[
\text{unfold}(e_0);
\]

\[
\vdots
\]

\[
\text{unfold}(e_n);
\]

\[
r.\text{sel0} = \text{unfoldres}(e_0);
\]

\[
\vdots
\]

\[
r.\text{seln} = \text{unfoldres}(e_n);
\]

• Selector function calls

\[
e \equiv \text{sel}(e_0)
\]

\[
\text{unfoldres}(e) = r
\]

\[
\text{unfold}(e) =
\]

\[
\text{unfold}(e_0);
\]

\[
r = \text{unfoldres}(e_0).\text{sel};
\]

• Function calls (fun denotes a user-defined function and the corresponding C0 procedure; this mapping is defined later)

\[
e \equiv \text{fun}(e_0, \ldots, e_n)
\]

\[
\text{unfoldres}(e) = r
\]

\[
\text{unfold}(e) =
\]

\[
\text{unfold}(e_0);
\]

\[
\vdots
\]

\[
\text{unfold}(e_n);
\]

\[
r = \text{fun}(\text{unfoldres}(e_0), \ldots, \text{unfoldres}(e_n));
\]

\[\blacktriangledown\]

Expressions are mapped using the unfolding operation.

**Definition 5.7** (Expression Mapping). The expression mapping \(\text{map}^{AF3}_{C0} : \text{EXP} \rightarrow \text{STM} \times \text{ID}\) is defined as follows:

\[
\text{map}^{AF3}_{C0}(e) = \langle \text{unfold}(e), \text{unfoldres}(e) \rangle
\]

\[\blacktriangledown\]
Next, we prove the equivalence of expressions. In the following lemma we assume that user-defined functions are mapped to behaviorally equivalent C0 procedures. That equivalence will be shown after we define the mapping of pattern matchings to C0 code.

**Lemma 5.1 (Expression Equivalence).** Given a variable environment $v_{AF,3}^{var} \in \Upsilon_{AF,3}$, a type-correct expression $e \in \text{EXP}$, and a C0 program configuration $\sigma \in \Sigma$ with

\[
\begin{align*}
v_{AF,3}^{var} &\sim \sigma \\
\text{map}_{C0}^{AF,3}(e) &\equiv (uc, ur) \\
\langle \sigma, uc; \rangle &\rightarrow \sigma'
\end{align*}
\]

Then the value of the expression is equal to the value stored in $ur$:

\[
\text{eval}_{AF,3}(e, v_{AF,3}^{var}) \sim \sigma'(ur)
\]

**Proof.** The proof is done by induction on the structure of the expression.

- **Base case: atom values $e \equiv \text{atom}$**
  
  **Proof.**
  
  \[
  \text{map}_{C0}^{AF,3}(e) = \langle r = \text{atom}, r \rangle \Rightarrow \text{eval}_{AF,3}(e, v_{AF,3}^{var}) = \text{atom} \sim \\
  \text{atom} = \text{update}(\sigma, r, \text{atom})(r) = \sigma'(r)
  \]

- **Base case: variables $e \equiv \text{var}$**
  
  **Proof.**
  
  \[
  \text{map}_{C0}^{AF,3}(e) = \langle r = \text{var}, r \rangle \Rightarrow \text{eval}_{AF,3}(e, v_{AF,3}^{var}) = v_{AF,3}^{var}(\text{var}) \sim \\
  \sigma(\text{var}) = \text{update}(\sigma, r, \sigma(\text{var}))(r) = \sigma'(r)
  \]

- **Induction Hypothesis**
  
  \[
  \forall i : \text{eval}_{AF,3}(e_i, v_{AF,3}^{var}) \sim \sigma'_i(r_i)
  \]

- **Induction case: unary operators $e \equiv \text{uo} \equiv e_0$**
  
  **Proof.** From the definition of $\text{unfold}$ follows
  
  \[
  \langle \sigma, \text{unfold}(e_0); r = \text{uo}\text{unfoldres}(e_0) \rangle \rightarrow \sigma'
  \]
  
  From execution semantics follows
  
  \[
  \langle \sigma^*, r = \text{uo}\text{unfoldres}(e_0); \rangle \rightarrow \sigma'
  \]
and
\[ (\sigma^*, r_0 = \text{unfoldres}(e_0); \ r) \mapsto \sigma'_0 \]

Then we obtain
\[
\begin{align*}
\text{eval}_{AF3}(e, \upsilon_{\text{var}}) &= \langle \sigma_0 \rangle \text{eval}_{AF3}(e_0, \upsilon_{\text{var}}) \\
\langle \sigma_0 \rangle \text{unfoldres}(e_0) &= \sigma^* (\text{unfoldres}(e_0)) = \sigma^* (\text{unfoldres}(e_0)) = \text{update} (\sigma^*, r, \sigma^* (\text{unfoldres}(e_0))) (r) = \sigma'(r)
\end{align*}
\]

\[\blacksquare\]

• Induction case: strict binary operators

**Proof.** From the definition of unfold follows
\[
\begin{align*}
\langle \sigma, \text{unfold}(e_0); \text{unfold}(e_1); \\
r = \text{unfoldres}(e_0) < \text{bo> unfoldres}(e_1); \rangle &\mapsto \sigma'
\end{align*}
\]

From execution semantics follows
\[
\begin{align*}
\langle \sigma, \text{unfold}(e_0); \text{unfold}(e_1); \\
r = \text{unfoldres}(e_0) < \text{bo> unfoldres}(e_1); \rangle &\mapsto \\
\langle \sigma^*, \text{unfold}(e_1); r = \text{unfoldres}(e_0) < \text{bo> unfoldres}(e_1); \rangle &\mapsto \\
\langle \sigma^{**}, r = \text{unfoldres}(e_0) < \text{bo> unfoldres}(e_1); \rangle &\mapsto \sigma'
\end{align*}
\]

And since
\[
\begin{align*}
\sigma^*(\text{unfoldres}(e_0)) &= \sigma^{**}(\text{unfoldres}(e_0)) = \text{eval}_{AF3}(e_0, \upsilon_{\text{var}}) \\
\sigma^{**}(\text{unfoldres}(e_1)) &= \text{eval}_{AF3}(e_1, \upsilon_{\text{var}})
\end{align*}
\]

we obtain
\[
\begin{align*}
\text{eval}_{AF3}(e, \upsilon_{\text{var}}) &= \text{eval}_{AF3}(e_0, \upsilon_{\text{var}}) < \text{bo> eval}_{AF3}(e_1, \upsilon_{\text{var}}) \\
\sigma^{**}(\text{unfoldres}(e_0)) &= \sigma^{**}(\text{unfoldres}(e_1)) = \\
\sigma^{**}(\text{unfoldres}(e_0)) &= \text{update}(\sigma^{**}, r, \sigma^{**}(\text{unfoldres}(e_0)) < \text{bo> unfoldres}(e_1))(r) = \sigma'(r)
\end{align*}
\]

\[\blacksquare\]

• Induction case: non-strict operators (&& only, || is similar)

**Proof.** Case eval\(_{AF3}(e_0, \upsilon_{\text{var}}) = \text{true} \sim \text{true} = \sigma'_0(\text{unfoldres}(e_0))

From the definition of unfold and execution semantics follows
\[
\begin{align*}
\langle \sigma, \text{unfold}(e_0); \text{if}(\text{unfoldres}(e_0))\{ \\
\text{unfold}(e_1); r = \text{unfoldres}(e_1); \} \rangle &\mapsto \\
\langle \sigma'_0, \text{if} \ldots \rangle &\mapsto \langle \sigma'_0, \text{unfold}(e_1); r = \text{unfoldres}(e_1); \rangle &\mapsto \\
\langle \sigma^*, r = \text{unfoldres}(e_1); \rangle &\mapsto \sigma'
\end{align*}
\]

55
and with \( \sigma'(\text{unfoldres}(e_1)) = \sigma^*(\text{unfoldres}(e_1)) \) we obtain

\[
eval_{\text{AF}3}(e, v^\var_{\text{AF}3}) = \eval_{\text{AF}3}(e, v^\var_{\text{AF}3}) \sim \sigma'(\text{unfoldres}(e_1)) = \\
\sigma^*(\text{unfoldres}(e_1)) = \update(\sigma^*, r, \sigma^*(\text{unfoldres}(e_1))(r) = \sigma'(r)
\]

**Case** \( \eval_{\text{AF}3}(e_0, v^\var_{\text{AF}3}) = \text{false} \sim \text{false} = \sigma'_0(\text{unfoldres}(e_0)) \)

From the definition of \( \text{unfold} \) and execution semantics follows

\[
\langle \sigma, \text{unfold}(e_0); \text{if}(\text{unfoldres}(e_0)) \rangle \\
\langle \text{unfold}(e_1); r = \text{unfoldres}(e_1); \rangle \\
\langle \text{else}\{r = \text{false};\} \rangle \\
\langle \sigma'_0, \text{if}. . . \rangle \mapsto \langle \sigma'_0, r = \text{false}; \rangle \mapsto \sigma'
\]

and we obtain

\[
\eval_{\text{AF}3}(e, v^\var_{\text{AF}3}) = \text{false} \sim \sigma(\text{false}) = \\
\sigma'_0(\text{false}) = \update(\sigma'_0, r, \sigma'_0(\text{false})(r) = \sigma'(r)
\]

- **Induction case:** if then else (\textbf{true} case only, \textbf{false} is similar)

**Proof.** **Case** \( \eval_{\text{AF}3}(e_0, v^\var_{\text{AF}3}) = \text{true} \sim \text{true} = \sigma'_0(\text{unfoldres}(e_0)) \)

From the definition of \( \text{unfold} \) and execution semantics follows

\[
\langle \sigma, \text{unfold}(e_0); \text{if}(\text{unfoldres}(e_0)) \rangle \\
\langle \text{unfold}(e_1); r = \text{unfoldres}(e_1); \rangle \\
\langle \text{else}\{r = \text{false};\} \rangle \\
\langle \sigma'_0, \text{if}. . . \rangle \mapsto \langle \sigma'_0, r = \text{false}; \rangle \mapsto \sigma'
\]

and with \( \sigma'(\text{unfoldres}(e_1)) = \sigma^*(\text{unfoldres}(e_1)) \) we obtain

\[
\eval_{\text{AF}3}(e, v^\var_{\text{AF}3}) = \eval_{\text{AF}3}(e, v^\var_{\text{AF}3}) \sim \sigma'(\text{unfoldres}(e_1)) = \\
\sigma^*(\text{unfoldres}(e_1)) = \update(\sigma^*, r, \sigma^*(\text{unfoldres}(e_1))(r) = \sigma'(r)
\]

- **Induction case:** discriminator call (\textbf{CON} is the constructor identifier)

**Proof.** From the definition of \( \text{unfold} \) and execution semantics follows

\[
\langle \sigma, \text{unfold}(e_0); r = (\text{unfoldres}(e_0) == \text{CON}); \rangle \mapsto \\
\langle \sigma'_0, r = (\text{unfoldres}(e_0) == \text{CON}); \rangle \mapsto \sigma'
\]

and we obtain

\[
\eval_{\text{AF}3}(e, v^\var_{\text{AF}3}) = (\eval_{\text{AF}3}(e, v^\var_{\text{AF}3}) == \text{CON}) \sim \\
(\sigma'_0(\text{unfoldres}(e_0))) == \text{CON} = (\sigma'_0(\text{unfoldres}(e_0))) == \text{CON} = \\
\update(\sigma'_0, r, (\sigma'_0(\text{unfoldres}(e_0))) == \text{CON})(r) = \sigma'(r)
\]

\[56\]
• Induction case: constructor call

Proof. From the definition of unfold and execution semantics follows

\[
\begin{align*}
\sigma, \text{unfold}(e_0); \ldots; \text{unfold}(e_n); \\
\ r.\text{sel}_0 = \text{unfoldres}(e_0); \ldots; r.\text{sel}_n = \text{unfoldres}(e_n); \} \mapsto \\
\sigma^*, r.\text{sel}_0 = \text{unfoldres}(e_0); \ldots; r.\text{sel}_n = \text{unfoldres}(e_n); \} \mapsto \sigma'
\end{align*}
\]

and with the hypothesis

\[
\forall i: \sigma^*(\text{unfoldres}(e_i)) = \sigma'_i(\text{unfoldres}(e_i)) \sim \text{eval}_{AF^3}(e_i, v_{AF^3})
\]

and with the definition of member assignment semantics we obtain

\[
\forall i: \sigma'(r.\text{sel}_i) = \sigma'(\text{unfoldres}(e_i))
\]

finally with the definition of member access

\[
\forall i: \text{eval}_{AF^3}(e, v_{AF^3}).\text{sel}_i = \text{eval}_{AF^3}(e_0, v_{AF^3}).\text{sel}_i
\]

we can conclude

\[
\forall i: \text{eval}_{AF^3}(e, v_{AF^3}).\text{sel}_i \sim \sigma'(r.\text{sel}_i)
\]

and thus

\[
\text{eval}_{AF^3}(e, v_{AF^3}) \sim \sigma'(r)
\]

• Induction case: selector call (\textit{sel} denotes the selector identifier)

Proof. From the definition of unfold and execution semantics follows

\[
\begin{align*}
\sigma, \text{unfold}(e_0); r = \text{unfoldres}(e_0).\text{sel}; \} \mapsto \\
\sigma'_0, r = \text{unfoldres}(e_0).\text{sel}; \} \mapsto \sigma'
\end{align*}
\]

and we obtain

\[
\text{eval}_{AF^3}(e, v_{AF^3}) = \text{eval}_{AF^3}(e_0, v_{AF^3}).\text{sel} \sim \sigma'_0(\text{unfoldres}(e_0)).\text{sel} = \\
\text{update}(\sigma'_0, r, \sigma'_0(\text{unfoldres}(e_0)).\text{sel})(r) = \sigma'(r)
\]

• Induction case: function call
Proof. Here, we have to assume that the function and procedure execution is equivalent, i.e. for any set of atomic values \( v_0, \ldots, v_n \) we have \( \text{eval}_{\text{AF3}}(f(v_0, \ldots, v_n), v_{\text{var}}) = \text{eval}_{\text{AF3}}(\phi, v_{\text{var}}) \sim \sigma^\pi(\rho_{\pi}) \) with \( (\sigma, \rho_{\pi} = f(v_0, \ldots, v_n)) \) \( \mapsto \sigma^\pi \) with \( \phi \sim \pi \) are semantically equivalent. This assumption holds, as we shall proof later, due to the fact that functions are non-recursive and pattern matching is equivalent in AUTOFOCUS 3 and C0.

From the definition of \( \text{unfold} \) and execution semantics follows

\[
(\sigma, \text{unfold}(e_0); \ldots; \text{unfold}(e_n); \quad r = f(\text{unfoldres}(e_0); \ldots, \text{unfoldres}(e_n));) \mapsto \\
(\sigma^*, r = f(\text{unfoldres}(e_0); \ldots, \text{unfoldres}(e_n));) \mapsto \sigma'
\]

and with the hypothesis

\[
\forall i : \sigma^*(\text{unfoldres}(e_i)) = \sigma'_i(\text{unfoldres}(e_i)) \sim \text{eval}_{\text{AF3}}(e_i, v_{\text{var}})
\]

and with the equivalence assumption of function calls we obtain

\[
\text{eval}_{\text{AF3}}(e, v_{\text{var}}) = \text{eval}_{\text{AF3}}(\phi, v_{\text{var}}) \sim \sigma^\pi(\rho_{\pi}) = \\
\text{update}(\sigma, \rho_{\pi}, f(\sigma^*(\text{unfoldres}(e_0)), \ldots, \sigma^*(\text{unfoldres}(e_n))))(\rho_{\pi}) = \\
\text{update}(\sigma, r, f(\sigma^*(\text{unfoldres}(e_0)), \ldots, \sigma^*(\text{unfoldres}(e_n))))(r) = \sigma'(r)
\]

That completes the proof of the lemma.

\[\square\]

5.2.4 Pattern Matching

To complete the mapping of the data definition part of the AUTOFOCUS 3 model, we have to construct behaviorally equivalent C0 procedures for user-defined AUTOFOCUS 3 functions. Hereby, we must implement the pattern matching algorithm using C0 statements. We start by giving the mapping of the pattern matching independently, since we will need it again later when we use pattern matching for input port values.

AUTOFOCUS 3 patterns are transformed quite similar to expressions. Given a pattern and a C0 expression, which refers to the value to be matched, the following schemas apply the transformation. Here, \( \text{patcode}(p, cexp) \) expands to the code needed testing the expression. \( \text{patguard}(p, cexp) \) returns the C0 variable introduced to store the result of the pattern matching, i.e. \( \text{patguard}(p, cexp) \) can be used to guard subsequent code using an if-statement.

**Definition 5.8** (Pattern Guard Operation). Let \( p \in \text{PTN} \) be a pattern from an AUTOFOCUS 3 model, and \( cexp \in \text{EXP}_{\text{C0}} \) a C0 expression. Then \( \text{patcode} : \text{PTN} \times \text{EXP}_{\text{C0}} \rightarrow \text{STM} \) and \( \text{patguard} : \text{PTN} \times \text{EXP}_{\text{C0}} \rightarrow \text{ID} \) map a pattern and a value access C0 expression to their guarding C0 code and the guard result storage variable identifier, respectively, according to the following rules (the type of the newly introduced variable \( r \) is \text{bool}):

\[
\text{patcode}(p, cexp) = \quad \text{patguard}(p, cexp) = \\
\text{patcode}(p, cexp) = \quad \text{patguard}(p, cexp) = \\
\text{patcode}(p, cexp) = \quad \text{patguard}(p, cexp) =
\]

58
• Universal matching

\[
\begin{align*}
    p \equiv & \quad \\
    \text{patguard}(p, cexp) &= r \\
    \text{patcode}(p, cexp) &= r = \text{true};
\end{align*}
\]

• Atomic matching

\[
\begin{align*}
    p \equiv & \atom \\
    \text{patguard}(p, cexp) &= r \\
    \text{patcode}(p, cexp) &= r = (\text{atom} == cexp);
\end{align*}
\]

• Variable matching

\[
\begin{align*}
    p \equiv & \var \\
    \text{patguard}(p, cexp) &= r \\
    \text{patcode}(p, cexp) &= v = cexp; \\
    r &= \text{true};
\end{align*}
\]

• Tuple matching (\(sel_0, \ldots, sel_n\) denote the selector slot accesses)

\[
\begin{align*}
    p \equiv & \CON(p_0, \ldots, p_n) \\
    \text{patguard}(p, cexp) &= r \\
    \text{patcode}(p, cexp) &= \text{patcode}(p_0, cexp\.sel_0); \\
    \quad \vdots \\
    \quad \text{patcode}(p_n, cexp\.sel_n); \\
    r &= (\text{patguard}(p_0, cexpr\.sel_0) && \ldots && \text{patguard}(p_0, cexpr\.sel_1));
\end{align*}
\]

The universal matching always succeeds, thus \(r\) becomes \text{true}. Atomic pattern matches are transformed into equality checks. Variables matched against a value result in the assignment of the value to the variable. Finally, matching against a tuple constructor is done by matching all the selector values against their corresponding sub-pattern.

**Definition 5.9 (Pattern Matching Guard).** Let \(p \in PTN\) be the pattern, and \(cexp \in EXP_{C0}\) be the C0 expression to access the value to be matched against the pattern. Then the pattern matching guard \(guard^{AF3}_{C0} : PTN \times EXP_{C0} \rightarrow STM T \times ID\) is defined as follows:

\[
guard^{AF3}_{C0}(p, e) = \langle \text{patcode}(p, e), \text{patguard}(p, e) \rangle
\]
For the transformation of user-defined functions and transition segments it is convenient to have the possibility to guard code by evaluating code derived from a pattern matching. The following lemma proves that the guard code (generated by \texttt{guard}$_{\text{AF3}}$) is equivalent to the AUTOFOCUS 3 pattern matching process. Note that during the operational execution of the guard code some pattern variables might be assigned although the matching fails later (e.g. imagine matching \texttt{C(true, false)} against \texttt{C(A, true)}, which operationally binds \texttt{A} before the matching fails.). However, since we assume that variables bound in patterns are unique, this case is properly treated by the equivalence given in Def. 5.5.

\textbf{Lemma 5.2} (Pattern Matching Guard Execution Equivalence). Given a pattern $p \in $ PTN, a value $v \in $ VALUE, a variable environment $v^\text{var}_{\text{AF3}} \in $ $\Upsilon^\text{var}_{\text{AF3}}$, a C0 program configuration $\sigma \in $ $\Sigma$, and a C0 expression cexp $\in $ EXP$_{C0}$ with

\begin{align*}
  v^\text{var}_{\text{AF3}} &\sim \sigma \\
  \text{guard}$_{C0}$$(p, cexp) = (pc, pg) \\
  v &\sim \sigma(cexp) \\
  (\sigma, pc; ) &\mapsto \sigma'
\end{align*}

Then execution of the pattern guard leads to an equivalent state and the pattern guard variable value is equal to the matching result:

\begin{align*}
  \text{bind}(p, v, v^\text{var}_{\text{AF3}}) &\sim \sigma' \land \\
  \text{match}(p, v, v^\text{var}_{\text{AF3}}) &\sim \sigma'(pg)
\end{align*}

\textbf{Proof.} The proof is done by induction on the structure of the pattern.

- Base case: universal matching $p \equiv \_$

  \textit{Proof.} From definition of \texttt{guard}$_{C0}$ and execution semantics follows

  $$(\sigma, r = \text{true}; ) \mapsto \sigma'$$

  and we obtain

  \begin{align*}
  \text{match}(p, v, v^\text{var}_{\text{AF3}}) &= \text{true} \sim \\
  \text{true} &= \sigma(\text{true}) = \\
  \text{update}(\sigma, r, \sigma(\text{true}))(r) &= \\
  \sigma'(r)
  \end{align*}

  and since $r$ is a fresh variable

  $$\text{bind}(p, v, v^\text{var}_{\text{AF3}}) = v^\text{var}_{\text{AF3}} \sim \text{update}(\sigma, r, \text{true}) = \sigma'$$

- Base case: atomic matching $p \equiv \texttt{atom}$
Proof. From definition of guard$^3$ and execution semantics follows

$$\langle \sigma, r = (cexp == atom); \rangle \mapsto \sigma'$$

and we obtain

$$match(p, v, v^{var}_{AF3}) = (v == atom) \sim
\sigma(cexp) == atom =
\sigma(cexp) == \sigma(\text{atom}) =
\sigma(cexp == atom) =
update(\sigma, r, \sigma(cexp == atom))(r) =
\sigma'(r)$$

and since $r$ is a fresh variable

$$bind(p, v, v^{var}_{AF3}) = v^{var}_{AF3} \sim update(\sigma, r, \text{true}) = \sigma'$$

• Base case: variable matching $p \equiv \text{var}$

Proof. From definition of guard$^3$ and execution semantics follows

$$\langle \sigma, \text{var} = cexp; r = \text{true}; \rangle \mapsto \langle \sigma^*, r = \text{true}; \rangle \mapsto \sigma'$$

and we obtain

$$match(p, v, v^{var}_{AF3}) = \text{true} \sim
\text{true} = \sigma^*(\text{true}) =
update(\sigma^*, r, \sigma(\text{true}))(r) =
\sigma'(r)$$

and since $r$ is a fresh variable and

$$update(\sigma, \text{var}, \sigma(cexp)) = \sigma^*(\text{var}) = \sigma'(\text{var})$$

we conclude with the definition of $\sim$ and $update$

$$bind(\text{var}, v, v^{var}_{AF3}) = update(\text{var}, v, v^{var}_{AF3}) \sim \sigma'$$

• Induction case: Tuple matching

Proof. From definition of guard$^3$ and execution semantics follows

$$\langle \sigma, \text{patcode}(p_0, cexp.sel_0); \ldots; \text{patcode}(p_n, cexp.sel_n); \rangle \mapsto \langle \sigma^*, r = (\text{patguard}(p_0, cexp.sel_0); \& \ldots \& \text{patguard}(p_n, cexp.sel_n)); \rangle \mapsto \sigma'$$
From the induction hypothesis follows

\[ \forall i : \text{match}(p_i, v, \nu_{\text{AF3}}) \sim \sigma^*(\text{patguard}(p_i, cexp.sel_i)) = \sigma'_i(\text{patguard}(p_i, cexp.sel_i)) \]

\[ \forall i : \text{bind}(p_i, v, \nu_{\text{AF3}}) \sim \sigma'_i \]

and we obtain

\[
\begin{align*}
\text{match}(p, v, \nu_{\text{AF3}}) &= \\
\text{match}(p_0, v, \nu_{\text{AF3}}) \&\& \text{match}(p_n, v, \nu_{\text{AF3}}) &\sim \sigma^*(\text{patguard}(p_0, cexp.sel_0)) \&\& \sigma^*(\text{patguard}(p_n, cexp.sel_n)) = \\
\sigma^*(\text{patguard}(p_0, cexp.sel_0) \&\& \text{patguard}(p_n, cexp.sel_n)) &= \\
\text{update}(\sigma^*, r, \sigma^*(\text{patguard}(p_0, cexp.sel_0) \&\& \ldots ))(r) &= \\
\sigma'(r)
\end{align*}
\]

and since pattern variables are disjunct

\[ \text{bind}(p_i, v, \nu_{\text{AF3}}) \sim \sigma^* \]

we conclude with \( r \) is a fresh variable

\[ \text{bind}(p, v, \nu_{\text{AF3}}) \sim \sigma' \]

That completes the proof of the lemma.

5.2.5 User-defined Functions

Since we have transferred the pattern matching evaluation from \textsc{AutoFocus 3} to C0, we are now ready to map user-defined functions.

We first consider a single definition pair. We use two fresh variables in these mappings. \texttt{result} stores the result of the evaluated expression and \texttt{success} is flag indicating whether the pattern matching succeeded.

\textbf{Definition 5.10} (Definition Pair Mapping). The mapping of a single definition pair \( \text{map}_{\text{C0}}^{\text{AF3}} : \text{list (PTN)} \times \text{EXP} \times \text{list ID} \rightarrow \text{STM} \) is defined according to the following rules.

- If the list of patterns in a single definition pair is empty (i.e. in case of no remaining patterns or the universal matching pattern), the expression is unfolded, its result is assigned to the variable \texttt{result} and the \texttt{success} flag is set:

\[
\begin{align*}
\text{map}_{\text{C0}}^{\text{AF3}}(\text{exp}) &= \langle \text{uc}, \text{ur} \rangle \\
\text{map}_{\text{C0}}^{\text{AF3}}([], \text{exp}, \text{paramlist}) &= \text{uc}; \\
\text{result} &= \text{ur}; \\
\text{success} &= \text{true};
\end{align*}
\]
• If the list of patterns contains another pattern, the pattern guarding generation is applied to the resulting code of the transformation of the remaining lists. The pattern guard uses the corresponding parameter variable to match the pattern against.

\[
\text{patlist} \neq [] \land \text{paramlist} \neq [] \land |\text{patlist}| = |\text{paramlist}|
\]

\[
\text{guard}_{AF^3}(\text{patlist.first, paramlist.first}) = \langle \text{pc}, \text{pg} \rangle
\]

\[
\text{map}_{C0}(\text{patlist, exp, paramlist}) = \text{pc}:
\]

\[
\text{if} \ (\text{pg}) \ \{ \\
\text{map}_{AF^3}(\text{patlist.rest, exp, paramlist.rest})
\}
\]

\[
\text{\textless}\textless
\]

The mapping of definition pairs incrementally constructs an if-statement cascade, such that the evaluation of the expression is effectively guarded by the pattern matching code. success is guaranteed to be set to true if and only if all pattern matches succeeded. Correspondingly, result carries then the result of the evaluated expression.

The following lemma proves the equivalence of the AutoFocus 3 pattern matching and the C0 implementation.

**Lemma 5.3 (Function Definition Pair Evaluation Equivalence).** Given a single function definition \(d \in \text{list } \text{PTN} \times \text{EXP}\), a list of argument values \(al \in \text{list } \text{VALUE}\), a variable environment \(v_{AF^3} \in \text{V}_{AF^3}\), a C0 program configuration \(\sigma \in \Sigma\), and a list of parameter identifiers \(pl \in \text{list } \text{ID}\) with

\[
v_{AF^3} \sim \sigma
\]

\[
d.\text{pattern} = [] \lor |d.\text{pattern}| = |al|
\]

\[
|al| = |pl|
\]

\[
\forall i : \text{al.i} \sim \sigma(\text{pl.i})
\]

\[
\sigma(\text{success}) = \text{false}
\]

\[
\text{code} = \text{map}_{AF^3}(d.\text{pattern}, d.\text{exp}, pl)
\]

\[
\langle \sigma, \text{code} \rangle \rightarrow \sigma'
\]

Then, given the pattern matching succeeds, success is true and the evaluation results of the expressions are equal

\[
\text{match}(d.\text{pattern}, al, v_{AF^3}) \Rightarrow
\]

\[
\sigma'(\text{success}) = \text{true} \land
\]

\[
\text{eval}_{AF^3}(d.\text{exp}, \text{bind}(d.\text{pattern}, al, v_{AF^3})) \sim \sigma'(\text{result})
\]

and, given the matching fails, success is false and the state of non-pattern variables is unchanged

\[
\text{match}(d.\text{pattern}, al, v_{AF^3}) \Rightarrow
\]

\[
\sigma'(\text{success}) = \text{false} \land
\]

\[
\text{bind}(d.\text{pattern}, al, v_{AF^3}) \sim \sigma'
\]
Proof. The proof is done by induction on the list of patterns.

• Base case: \texttt{d.pattern} = []

Proof. From definition of \texttt{map^{AF3}} and execution semantics follows

\begin{align*}
\text{map}^{AF3}(d.exp) &= \langle uc, ur \rangle \\
\text{and} \quad \langle \sigma, uc; result = ur; success = true; \rangle &\rightarrow \\
\langle \sigma^*, result = ur; success = true; \rangle &\rightarrow \\
\langle \sigma^{**}, success = true; \rangle &\rightarrow \\
\sigma' 
\end{align*}

We have \texttt{match(d.pattern, al, \upsilon_{AF3}^{val})} = true, which directly proves the second conclusion of the lemma. Furthermore, we have \(\sigma'(success) = update(\sigma^{**}, success, true)(success) = true\), which proves the first half of the implication of the first conclusion of the lemma. For the second half, we use lemma 5.1 and the definition of \texttt{bind} and obtain

\begin{align*}
\text{eval}^{AF3}(d.exp, bind(d.pattern, al, \upsilon_{AF3}^{val})) &= \\
\text{eval}^{AF3}(d.exp, \upsilon_{AF3}^{val}) &\sim \\
\sigma^*(ur) &= update(\sigma^*, result, \sigma^*(ur))(result) = \\
\sigma^{**}(result) &= update(\sigma^{**}, success, true)(result) = \\
\sigma'(result) 
\end{align*}

• Induction Hypothesis:

\begin{align*}
\upsilon_{rest} &= bind(d.pattern.first, al.first, \upsilon_{AF3}^{val}) \\
\text{match}(d.pattern.rest, al.rest, \upsilon_{rest}) &\Rightarrow \\
\sigma'_{rest}(success) &= true \land \\
\text{eval}^{AF3}(d.exp, bind(d.pattern.rest, al.rest, \upsilon_{rest})) &\sim \sigma'_{rest}(result) \\
\neg \text{match}(d.pattern.rest, al.rest, \upsilon_{rest}) &\Rightarrow \\
\sigma'_{rest}(success) &= false \land \\
\text{bind}(d.pattern.rest, al.rest, \upsilon_{rest}) &\sim \sigma'_{rest} 
\end{align*}

• Induction case: \texttt{d.pattern} \neq []

Proof. We now have

\begin{align*}
\text{guard}^{AF3}(d.pattern.first, pl.first) &= \langle pc, pg \rangle \\
\text{restcode} &= \text{map}^{AF3}(d.pattern.rest, d.exp, pl.rest) \\
\langle \sigma, pc; if(pr)\{\text{restcode}; \} \rangle &\rightarrow \\
\langle \sigma^*, if(pr)\{\text{restcode}; \} \rangle &\rightarrow \\
\sigma' 
\end{align*}
and with lemma 5.2

\[
\text{match}(d.\text{pattern}.\text{first}, a.\text{first}, v_{AF3}^\text{var}) \sim \sigma^*(pg) \\
\text{bind}(d.\text{pattern}.\text{first}, a.\text{first}, v_{AF3}^\text{var}) \sim \sigma^*
\]

**Case** \(\sigma^*(pg) = \text{false} \sim \text{false} = \text{match}(d.\text{pattern}.\text{first}, a.\text{first}, v_{AF3}^\text{var})\)

The first conclusion follows directly. For the second conclusion, we have \(\sigma' = \sigma^\ast\) from the semantics of the if-statement execution and \(\sigma(\text{success}) = \text{false} = \sigma^\ast(\text{success})\), since \(pc\) does not alter \(\text{success}\), which proves that \(\sigma'(\text{success}) = \text{false}\). With \(\text{bind}(d.\text{pattern}.\text{first}, a.\text{first}, v_{AF3}^\text{var}) = v_{AF3}^\text{var} \sim \sigma^\ast = \sigma'\) the second part of the implication also holds.

**Case** \(\sigma^*(pg) = \text{true} \sim \text{true} = \text{match}(d.\text{pattern}.\text{first}, a.\text{first}, v_{AF3}^\text{var})\)

From the lemma precondition we have

\[
\langle \sigma^\ast, \langle \text{if}(pg)\{\text{restcode;} \} \rangle \rangle \mapsto \langle \sigma^\ast, \text{restcode;} \rangle \mapsto \sigma'
\]

and from the induction hypothesis we have

\[
\langle \sigma^\ast, \text{restcode;} \rangle \mapsto \sigma'_{\text{rest}}
\]

and from lemma 5.2 follows

\[
\text{bind}(d.\text{pattern}.\text{first}, a.\text{first}, v_{AF3}^\text{var}) = v_{\text{rest}} \sim \sigma^\ast
\]

Now, we assume the first case of the induction hypothesis holds, i.e. the matching succeeds completely. We have to consider only the first conclusion of the lemma. Since \(\sigma' = \sigma'_{\text{rest}}\), both implications follow directly from the first case of the induction hypothesis.

Now, we assume the second case of the induction hypothesis holds, i.e. the matching fails somewhere later. We have to consider only the second conclusion of the lemma. Since \(\sigma' = \sigma'_{\text{rest}}\), both implications follow directly from the second case of the induction hypothesis.

That completes the proof of the lemma.

We now consider a list of definition pairs. We use the *success* flag to control that only the first completely matching function definition pair affects *result*. *success* will be initialized to *false*. Each pair in the list is transformed using the above mapping and the resulting block of code is surrounded with the negated *success* flag guard. The guarded blocks are composed sequentially. Finally, *result* is returned.

**Definition 5.11** (Function Definitions Mapping). The mapping of a function definitions \(\text{map}_{C0}^{AF3} : \text{list (list (PTN) \times EXP) \times list ID} \to \text{STM}\) is defined according to the following rules.
• If the list of definition pairs is empty, no further code is generated:

\[
map_{C_0}^{AF_3}([], \text{paramlist}) = []
\]

• Otherwise the definition pair is transformed and guarded, and the remaining list is transformed correspondingly:

\[
deflist \neq [] \quad \Rightarrow \quad map_{C_0}^{AF_3}(deflist, \text{paramlist}) = \\
\text{if } (!\text{success}) \quad \{ \\
\quad map_{C_0}^{AF_3}(deflist.first.pattern, deflist.first.exp, \text{paramlist}) \\
\} \\
\quad map_{C_0}^{AF_3}(deflist.rest, \text{paramlist})
\]

The following lemma proves that if the value of `success` is initially `false` then the first and only the first matching definition is executed. Afterwards the variable `result` contains the result equivalent to the result of executing `eval_{AF_3}` with the equivalent values.

**Lemma 5.4 (Function Definition List Execution Equivalence).** Given a list of function definition \( dl \in \text{list} (\text{list PTN} \times \text{EXP}) \), with \( |dl| > 0 \), a list of argument values \( al \in \text{list VALUE} \), a variable environment \( \nu_{AF_3}^{var} \in \Upsilon_{AF_3}^{var} \), a C0 program configuration \( \sigma \in \Sigma \), a list of parameter identifiers \( pl \in \text{list ID} \), and the unique variables \( \text{result} \) and \( \text{success} \) with

\[
\nu_{AF_3}^{var} \sim \sigma \\
\forall d \in dl : d.pattern = [] \lor |d.pattern| = |al| \\
|al| = |pl| \\
\forall i : al.i \sim \sigma(pl.i) \\
\sigma(\text{success}) = \text{false} \\
code = map_{C_0}^{AF_3}(dl, pl) \\
\langle \sigma, code \rangle \mapsto \sigma'
\]

Then the execution of the code is equivalent to the execution of the function definition

\[
\sigma'(\text{success}) = \text{true} \\
\text{eval}_{AF_3}^{fun}(dl, al, \nu_{AF_3}^{var}) \sim \sigma'(\text{result})
\]

**Proof.** The proof is done by induction on the list of definitions.

• Basis \( dl = [dl_0] \): from Constraint 3.11 \( dl.pattern = [] \) follows. Then with Lemma 5.3 the conclusion follows, since the minimal index of the function definition used in `eval_{AF_3}^{fun}` can only point to the single pattern available.
• Induction: Since the value `success` is still `false`, the current definition list head is candidate to change this. We have to distinguish two case: the current pattern matches or it does not.

  – Case current pattern matches:
    We conclude from Lemma 5.3 with \((\sigma, \text{if}(!\text{success})\{\ldots\}; \text{rest}) \mapsto (\sigma'', \text{rest})\) that \(\sigma''(\text{success}) = \text{true}\) and \(\text{eval}_{\text{AF3}}^{\text{fun}}(dl, al, (\text{var})_{\text{AF3}}) \sim \sigma''\), since the current head of the definition list is the minimal indexed definition used in \(\text{eval}_{\text{AF3}}^{\text{fun}}\). \(\sigma'' = \sigma'\) follows directly from \(\sigma''(\text{success}) = \text{true}\) and the execution semantics of the generated code, since every definition code changing `result` is guarded by `if(!success)`.

  – Case current pattern does not match:
    In this case we can conclude from Lemma 5.3 that the value of `success` remains unchanged. Thus, the conclusion follows from the induction hypothesis directly, since the minimal index cannot point to the current definition, since the pattern does not match.

Finally, we can map user-defined functions to C0 procedures by using the above mappings. The signature of the procedure corresponds to the signature of the function, thus procedure names are unique, since function names are. Any C0 procedure introduced for other purposes later will have distinct procedure names.

**Definition 5.12** (User-defined Function Mapping). Let \(\phi\) be a function implementation and \(\pi\) be a procedure definition. Then \(\text{map}_{\text{C0}}^{\text{AF3}}: \Phi \rightarrow \Pi\) maps functions to procedures according to the following schema:

  • \(\pi.rtype = \text{map}_{\text{C0}}^{\text{AF3}}(\phi.rtype)\), the return type is mapped directly.
  
  • \(\pi.params = \text{map}_{\text{C0}}^{\text{AF3}}(\phi.params)\), with \(\text{map}_{\text{C0}}^{\text{AF3}}\) being applied point-wise to the parameter identifier and its type of every member of the \(\phi.params\).
  
  • \(\pi.locvars\) is derived during the transformation of the function body to the procedure body. However, we assume the two unique local variables `result` of type \(\pi.rtype\) and `success` of type `bool` are contained herein.

  • The list of function definition pairs \(\phi.defs\) is transformed to procedure body \(\pi.body\) according to the following schematic rule of \(\text{map}_{\text{C0}}^{\text{AF3}}\).

\[
\pi.body =
\begin{align*}
\text{success} &= \text{false}; \\
\text{map}_{\text{C0}}^{\text{AF3}}(\phi.defs, \pi.params) \\
\text{return result;}
\end{align*}
\]

\[\triangleleft\]
Again Constraint 3.11 is useful to conclude the result of any function (and its corresponding procedure) is well-defined.

**Definition 5.13** (Function Table Mapping). Let $ftable : ID \to \Phi$ be the function table of some model. The $map^{AF3}_C$ maps the function table to its corresponding procedure table $ptable : ID \to \Pi$ with

$$\forall \text{funid} \in \text{dom.ftable} : ptable(\text{funid}) = map^{AF3}_C(\text{ftable}(\text{funid}))$$

\[\triangledown\]

**Lemma 5.5** (User-defined Function Call Equivalence). Let $ftable : ID \to \Phi$ be the function table, $f \in \text{dom.ftable}$ a function identifier, $\nu var_{AF3}$ a variable environment, $ptable : ID \to \Pi$ a procedure table, $\sigma, \sigma'$ program configurations, $e_0, \ldots, e_n$ be expressions, and $pcres$ a C0 variable with

$$\nu var_{AF3} \sim \sigma$$

$$\forall i : \nu var_{AF3}(\nu i) \sim \sigma(\nu i)$$

$$ptable = map^{AF3}_C(\text{ftable})$$

$$(\sigma,pcres = f(e_0, \ldots, e_n);) \mapsto \sigma'$$

Then the procedure call is equivalent to the function call.

$$eval^{AF3}(f(e_0, \ldots, e_n), \nu var_{AF3}) \sim \sigma'(pcres)$$

**Proof.** The function call is executed using the $eval^{fun}_{AF3}$ on its definitions, which corresponds to executing the procedure body according to Lemma 5.4. Furthermore, we need to apply the execution semantics of C0 procedure calls and return statements. Then the proof follows directly from the code generated as given by the definitions of $map^{AF3}_C$.

\[\square\]

**Theorem 5.6** (Expression Equivalence). Given a variable environment $\nu var_{AF3}$, an expression $exp \in expr$, and a C0 program configuration $\sigma$ with

$$\nu var_{AF3} \sim \sigma$$

$$map^{AF3}_C(\text{exp}) = (uc, ur)$$

$$(\sigma, [uc; \nu r = ur;]) \mapsto \sigma'$$

Then the value of the expression is equal to the value stored in $r$ and the remaining state is unchanged:

$$\nu var_{AF3} \sim \sigma'eval^{AF3}(e, \nu var_{AF3}) \sim \sigma'(\nu r)$$

**Proof.** Equivalence of the variable environment and the C0 program state after executing follows from the fact that during expression evaluation only unique local variables are assigned and function calls use separate temporary variable environment or procedure contexts. With Lemma 5.1 [Expression Equivalence] and 5.5 [User-defined Function Call Equivalence] the proof of the value equivalence follows.

\[\square\]
5.3 Components

The following definitions give the mapping for generating C0 variables that represent a component’s state. Again, we omitted object dependent identifier generation and variable types.

An atomic component has both an interface state and an automaton state, while the environment component needs only an interface state. Again, we do not consider hierarchic components in between, but only the flattened network of atomic components and the environment component.

**Definition 5.14** (Component Interface State Mapping). Let $C \in (C_{atm} \cup \{C_{env}\})$ be a component, and $ife_{syn} = \langle \text{I}_C, \text{O}_C, \text{type}_{port}, \text{init}_{port} \rangle$ its syntactic interface. Then $map_{C_0}^{AF3} : \text{I}_C \cup \text{O}_C \rightarrow ID$ maps the input and output ports to unique C0 variables used to store the port value. Furthermore, $flag_{C_0}^{AF3} : \text{I}_C \cup \text{O}_C \rightarrow ID$ maps to unique C0 variables of type $\text{bool}$ used to flag the empty message.

We will write $\sigma(p)$ to refer to the message stored in port $p$ and its encoding with the port value and flag variables. We define an equivalence relation for messages to handle $\epsilon$ in a more convenient way.

**Definition 5.15** (Message Equivalence). Let $m \in MSG$ be a message, $\sigma$ be a C0 program state, and $p \in \text{I}_C \cup \text{O}_C$ a port of a component $C$. Then

$$m \sim \sigma(p) \iff (m \neq \epsilon \Rightarrow m \sim \sigma(map_{C_0}^{AF3}(p)) \land \sigma(flag_{C_0}^{AF3}(p)) = \text{true}) \lor (m = \epsilon \Rightarrow \sigma(flag_{C_0}^{AF3}(p)) = \text{false})$$

Message equivalence relies on the port flag variable to signal whether the port value variable holds a valid value. If the flag variable is false, the port value variable may be arbitrary.

The following two definitions extend the equivalence of messages to port valuations.

**Definition 5.16** (Input Port Valuation Equivalence). Let $\lambda_I : \text{I}_C \rightarrow MSG$ be an input port valuation of some component $C$, and $\sigma$ be a C0 program state. Then

$$\lambda_I \sim \sigma \iff \forall ip \in \text{dom.} \lambda_I : \lambda_I(ip) \sim \sigma(ip)$$

**Definition 5.17** (Output Port Valuation Equivalence). Let $\lambda_O : \text{O}_C \rightarrow MSG$ be an output port valuation of some component $C$, and $\sigma$ be a C0 program state. Then

$$\lambda_O \sim \sigma \iff \forall op \in \text{dom.} \lambda_O : \lambda_O(op) \sim \sigma(op)$$

69
Atomic components have additional variables to describe their current state.

**Definition 5.18 (Automaton State Mapping).** Let $C \in C_{atm}$ be an atomic component, $Atm_C = (S, CP, TS, DSV, type_{dsv}, s, init_{dsv})$ be its automaton specification, and $cstate \notin DSV$ be the current state variable. Then $map_{C0}^{AF3} : DSV \cup \{curstate\} \rightarrow ID$ maps data state variables and the current state variable to unique C0 variables. Furthermore, $map_{C0}^{AF3} : S \rightarrow \text{int}$ uniquely assigns an integer value to each control state. Finally, $scratchpaper : DSV \rightarrow ID$ maps the data state variables to another set of C0 variables.

The *scratchpaper* variables will be used to store updated data state variable values during the transition segment execution because the value of data state variables must remain constant until all new output values and their own new values have been evaluated.

We now define the equivalence between the state of an AutoFocus 3 component and its C0 representation.

**Definition 5.19 (Atomic Component State Equivalence).** Let $C \in C_{atm}$ be an atomic component, $\sigma_C = \langle \lambda_O, cs, v \rangle$ its state, and $\sigma \in \Sigma$ be a C0 program state. Then

$$\sigma_C \sim \sigma \iff \lambda_O \sim \sigma \land \text{map}_{C0}^{AF3}(cs) = \sigma(cstate) \land v \sim \sigma$$

An atomic component’s state is equivalent if the output values, the current state and the data state variable values are equivalent. Again, $\sigma$ may contain additional variables used for temporary storage or in different contexts.

The following definition specifies the code generation of the initialization code of atomic components.

**Definition 5.20 (Atomic Component Initialization).** Let $C \in C_{atm}$ be an atomic component, $ifc_C^{syn} = (I_C, O_C, type_{port}, init_{port})$ its syntactic interface, and $Atm_C = (S, CP, TS, DSV, type_{dsv}, s, init_{dsv})$ be its automaton specification. Then the following rules generate the code that initializes the corresponding C0 variables

$$\begin{align*}
o &\in O_C \\
\text{init}_{port}(o) &\neq \epsilon \\
flag_{C0}^{AF3}(o) &= \text{false}; \\
\text{map}_{C0}^{AF3}(o) &= \text{map}_{C0}^{AF3}(\text{init}_{port}(o)); \\
\text{flag}_{C0}^{AF3}(o) &= \text{true}; \\
cstate &= \text{map}_{C0}^{AF3}(s)
\end{align*}$$
$$d \in \mathcal{DSV}$$
$$\text{map}_{C_0}^{AF_3}(d) = \text{map}_{C_0}^{AF_3}(\text{init}_{dsv}(d))$$;

The given rules are applied to all \( o \in \mathcal{O}_C, s \), and all \( d \in \mathcal{DSV} \) and the resulting code snippets are concatenated. Thus, the mapping \( \text{init}_{C_0}^{AF_3} : \mathcal{C}_{atm} \rightarrow \mathcal{STM} \) defines the generation of the initialization code.

Lemma 5.7 (Initialization Equivalence). Let \( \sigma_{C_0}^{\text{init}} \) be the initial state of the component \( C \in \mathcal{C}_{atm} \) and \( \sigma, \sigma' \in \Sigma \) be a C0 program configurations with

$$\langle \sigma, \text{init}_{C_0}^{AF_3}(C) \rangle \mapsto \sigma' \quad \sigma_{C_0}^{\text{init}} \sim \sigma'$$

Proof. The proof is straightforward using the definitions, the generation rules and the assignment execution semantics of C0.

The initialization code of each atomic component must be run once at the beginning of the system execution. The code generator packs this code into a suitable procedure to ease the integration.

5.4 Atomic Behavior

Up to now we generated state variables and initialization code for atomic components. We now transform automaton specifications into behaviorally equivalent C0 procedures.

Therefore, we define C0 procedure for each transition segment. The purpose of this procedure is to match the input patterns of the segment against the current input port values, evaluate the preconditions, call subsequent transition segment procedures in order of the transition segment total order induced by the unique object identifiers, and apply its effect if one of those fired. According to the AutoFocus 3 operational semantics only the first subsequent segment may be fired. This is achieved by using local variables to guard call and effect code.

Note that it is vital to have the constraints ??, ?? and ?? together with the total order on segments, these guarantee atomicity of transition segments (we will see that some precaution is needed with the data state variable values).

Again, we assume that name collisions between components are resolved automatically.

Definition 5.21 (Transition Segment Fire Procedure). Let \( C \in \mathcal{C}_{atm} \) be an atomic component, and \( \text{Atm}_C = (S, CP, TS, \mathcal{DSV}, \text{type}_{dsv}, s, \text{init}_{dsv}) \) be its automaton specification. Then \( \text{segmentfireproc} : TS \rightarrow ID \) is a mapping returning a unique procedure name for each transition segment. The return type of this procedure is \text{bool}.

\( \triangleright \)
5.4.1 Input Pattern Mapping

**Definition 5.22 (Input Pattern Mapping)**. Let \( inp \in (I_C \times PTN \cup \{\epsilon\}) \) be an input pattern of the component \( C \). Then \( inputcode : (I_C \times PTN \cup \{\epsilon\}) \rightarrow STM T \) and \( inputres : (I_C \times PTN \cup \{\epsilon\}) \rightarrow ID \) map the input pattern to its C0 code and result storage variable, respectively, according to the following rules

\[
\begin{align*}
inp &= \{\text{inport, pattern}\} \\
\text{pattern} &\neq \epsilon \\
cexp &= \text{map}^{AF_3}_C(\text{inport}) \\
\text{inputres}(inp) &= \text{patguard}(\text{pattern}, \text{cexp}) \\
\text{inputcode}(inp) &= \text{patcode}(\text{pattern}, \text{cexp})
\end{align*}
\]

\[
\begin{align*}
inp &= \{\text{inport}, \epsilon\} \\
\text{inputres}(inp) &= r \\
\text{inputcode}(inp) &= \text{flag}^{AF_3}_C(\text{inport});
\end{align*}
\]

\( \triangleright \)

**Lemma 5.8 (Input Pattern Equivalence)**. Given a component \( C \), an input pattern \( inp \in (I_C \times PTN \cup \{\epsilon\}) \), an input port valuation \( \lambda_I : I_C \rightarrow MSG \), a component state \( \sigma_C = (\lambda_O, cs, v) \), and program states \( \sigma, \sigma' \) with

\[
\begin{align*}
\sigma_C &\sim \sigma \\
\lambda_I &\sim \sigma \\
inp &= \{\text{inport, pattern}\} \\
(\sigma, \text{inputcode}(inp)) &\rightarrow \sigma'
\end{align*}
\]

Then

\[
\begin{align*}
\sigma_C &\sim \sigma' \\
\lambda_I &\sim \sigma' \\
\text{msgmatch}(\text{pattern}, \lambda_I(\text{inport}), v) &\sim \sigma'(\text{inputres}(inp)) \\
\text{msgbind}(\text{pattern}, \lambda_I(\text{inport}), v) &\sim \sigma'
\end{align*}
\]

**Proof**. The proof for the first mapping rule follows directly from the definition of message pattern matching and lemma 5.2.

For the proof of the second mapping rule we distinguish two cases.

If \( \lambda_I(\text{inport}) = \epsilon \) then \( \sigma(\text{flag}^{AF_3}_C(\text{inport})) = \text{false} \) by definition and thus \( \text{msgmatch}(\text{pattern}, \lambda_I(\text{inport}), v) = \text{true} \sim \text{true} = \sigma'(\text{inputres}(inp)) \) by negation and assignment semantics. Furthermore,

\[
\text{msgbind}(\text{pattern}, \lambda_I(\text{inport}), v) = v \sim \sigma'
\]

since the assignment affects a variable without a corresponding one in \( v \).

If \( \lambda_I(\text{inport}) \neq \epsilon \) then \( \sigma(\text{flag}^{AF_3}_C(\text{inport})) = \text{true} \) by definition and thus \( \text{msgmatch}(\text{pattern}, \lambda_I(\text{inport}), v) = \text{false} \sim \text{false} = \sigma'(\text{inputres}(inp)) \) by negation and assignment semantics. Furthermore,

\[
\text{msgbind}(\text{pattern}, \lambda_I(\text{inport}), v) = v \sim \sigma'
\]

72
since the assignment affects a variable without a corresponding one in $v$.

All other values remain unchanged.

5.4.2 Precondition Mapping

This mapping is done using Def. 5.7 [Expression Mapping] and equivalence follows from Theorem 5.6 [Expression Equivalence].

5.4.3 Output Expression Mapping

**Definition 5.23** (Output Expression Mapping). Let $outp \in (O_C \times EXP \cup \{\epsilon\})$ be an output expression of the component $C$. Then $outputcode : (O_C \times EXP \cup \{\epsilon\}) \rightarrow STM$ maps the output expression to its C0 code according to the following rules

\[
\begin{align*}
\text{outp} &= \langle \text{outport}, \text{expr} \rangle \\
\text{expr} &\neq \epsilon
\end{align*}
\]

\[
\begin{align*}
\text{outputcode}(\text{outp}) &= \text{map}_{C0}^{AF3}(\text{outp}) = \text{map}_{C0}^{AF3}(\text{expr}) \\
\text{flag}_{C0}^{AF3}(\text{outp}) &= \text{true}.
\end{align*}
\]

\[
\begin{align*}
\text{outp} &= \langle \text{outport}, \text{expr} \rangle \\
\text{expr} &= \epsilon
\end{align*}
\]

\[
\begin{align*}
\text{outputcode}(\text{outp}) &= \text{flag}_{C0}^{AF3}(\text{outp}) = \text{false}.
\end{align*}
\]

\[\square\]

**Lemma 5.9** (Output Expression Equivalence). Given a component $C$, an output expression $outp \in (O_C \times EXP \cup \{\epsilon\})$, a component state $\sigma_C = (\lambda_O, c_{\text{state}}, v)$, and program states $\sigma, \sigma'$ with

\[
\begin{align*}
\sigma_C &\sim \sigma \\
outp &= \langle \text{outport}, \text{expr} \rangle \\
\lambda'_O(\text{outport}) &= \begin{cases} 
\text{eval}_{AF3}(\text{expr}, v) &\text{if } \text{expr} \neq \epsilon \\
\epsilon &\text{otherwise} 
\end{cases} \\
(\sigma, \text{outputcode}(\text{outp})) &\rightarrow \sigma'
\end{align*}
\]

Then

\[
\begin{align*}
\text{map}_{C0}^{AF3}(c_{\text{state}}) &= \sigma'(c_{\text{state}}) \\
\lambda'_O &\sim \sigma' \\
v &\sim \sigma'
\end{align*}
\]

**Proof.** The proof follows directly from the assignment semantics and Theorem 5.6 [Expression Equivalence]. Neither the current state nor any data state variable is changed by the output assignment. \[\square\]
5.4.4 Postcondition Assignment Mapping

**Definition 5.24 (Postcondition Assignment Mapping).** Let \( post \in DSV \times EXP \) be a postcondition assignment of the component \( C \). Then \( postcode : (DSV \times EXP) \rightarrow STM \) maps the postcondition assignment to its C0 code according to the following rules

\[
\begin{align*}
\text{post} &= \langle dsv, expr \rangle \\
\text{postcode}(\text{post}) &= \text{scratchpaper}(dsv) \circ \text{map}^{AF^3}_{C0}(expr);
\end{align*}
\]

\[ \triangleright \]

**Lemma 5.10 (Postcondition Assignment Equivalence).** Given a component \( C \), a postcondition assignment \( post \in DSV \times EXP \), a component state \( \sigma_C = \langle \lambda_O, cs, v \rangle \), and program states \( \sigma, \sigma' \) with

\[
\begin{align*}
\sigma_C &\sim \sigma \\
\text{post} &= \langle dsv, expr \rangle \\
\langle \sigma, \text{postcode}(\text{post}) \rangle &\rightarrow \sigma'
\end{align*}
\]

Then

\[
\begin{align*}
\sigma_C &\sim \sigma' \\
\text{eval}_{AF^3}(expr, v) &\sim \sigma'(\text{scratchpaper}(dsv))
\end{align*}
\]

**Proof.** The proof follows directly from the assignment semantics and Theorem 5.6 [Expression Equivalence]. In particular, neither the current state nor any data state variable is changed by the postcondition assignment. \[ \rlap{\triangleright} \]

5.4.5 Transition Segment Input Guard Mapping

**Definition 5.25 (Transition Segment Input Guard Mapping).** Let \( C \in Catm \) be an atomic component, \( t = \langle src, trg, [i_0, \ldots, i_n], G, O, A \rangle \) be an arbitrary transition segment of its automaton specification \( Atm_C \). Furthermore, let \( guard \) be a unique variable. Then \( inputguard : TS \rightarrow STM \) maps the transition segment to its guard code as follows

\[
\begin{align*}
\text{inputguard}(t) &= \text{inputcode}(i_0) \\
&\quad \vdots \\
&\quad \text{inputcode}(i_n); \\
\text{guard} &= \text{inputres}(i_0) \& \ldots \& \text{inputres}(i_n);
\end{align*}
\]

\[ \triangleright \]

**Lemma 5.11 (Transition Segment Input Guard Equivalence).** Given a component \( C \), a transition segment \( t = \langle src, trg, I, G, O, A \rangle \), a component state
\[ \sigma_C = (\lambda_O, cs, v) \], an input message valuation \( \lambda_I \in \Lambda_I \), and program states \( \sigma, \sigma' \) with

\[
\begin{align*}
\sigma_C & \sim \sigma \\
\lambda_I & \sim \sigma \\
\sigma(\text{guard}) & = \text{true} \\
(\sigma, \text{inputguard}(t)) & \mapsto \sigma'
\end{align*}
\]

Then the guard variable value is equivalent to the message matching result and the message pattern binding is equivalent. Other variables remain untouched.

\[
\begin{align*}
\sigma_C & \sim \sigma' \\
\lambda_I & \sim \sigma' \\
(\forall (\text{inport}, \text{pat}) \in I : \text{msgmatch}(\text{pat}, \lambda_I(\text{inport}), v)) & \Leftrightarrow \sigma'(\text{guard}) = \text{true} \\
v \cup \bigcup_{(\text{inport}, \text{pat}) \in I} \text{msgbind}(\text{pat}, \lambda_I(\text{inport}), v) & \sim \sigma'
\end{align*}
\]

**Proof.** The proof follows directly from constraint ??, the semantics of \&\& and lemma 5.8 [Input Pattern Equivalence] by induction on the list of input patterns.

\[ \square \]

### 5.4.6 Transition Segment Precondition Guard Mapping

**Definition 5.26** (Transition Segment Precondition Guard Mapping). Let \( C \in \mathcal{C}_{atm} \) be an atomic component, \( t = (src, trg, I, G, O, A) \) be an arbitrary transition segment of its automaton specification \( \text{Atm}_C \). Furthermore, let \( \text{guard} \) be the same variable as in the last definition. Then \( \text{preguard} : \mathcal{T} \rightarrow \mathcal{STM}_T \) maps the transition segment to its guard code as follows

\[
\text{preguard}(t) = \\
\text{inputguard}(t); \\
\text{if (guard)} { \\
\text{unfold}(g_0); \\
\vdots \\
\text{unfold}(g_m); \\
\text{guard} = \text{unfoldres}(g_0) \&\& \ldots \&\& \text{unfoldres}(g_m); \\
}\}
\]

**Lemma 5.12** (Transition Segment Precondition Guard Equivalence). Given a component \( C \), a transition segment \( t = (src, trg, I, G, O, A) \), a component state \( \sigma_C = (\lambda_O, cs, v) \), an input message valuation \( \lambda_I \in \Lambda_I \), and program states \( \sigma, \sigma' \) with

\[
\begin{align*}
\sigma_C & \sim \sigma \\
\lambda_I & \sim \sigma \\
\sigma(\text{guard}) & = \text{true} \\
(\sigma, \text{preguard}(t)) & \mapsto \sigma' \\
v' & = v \cup \bigcup_{(\text{inport}, \text{pat}) \in I} \text{msgbind}(\text{pat}, \lambda_I(\text{inport}), v)
\end{align*}
\]

\[ \leftarrow \]
Then the guard variable value is equivalent to the precondition evaluation result and the message pattern binding is equivalent, while the component state remains untouched.

\[
\sigma_C \sim \sigma' \\
\lambda_I \sim \sigma' \\
(\forall \text{pre} \in G : \text{eval}_{AF3}(\text{pre}, \nu') = \text{true}) \iff \sigma'(\text{guard}) = \text{true} \\
(\lambda_O, cs, \nu') \sim \sigma' 
\]

**Proof.** Let \(\langle \sigma, \text{inputguard}(t) \rangle \mapsto \sigma''\), then we conclude \(\langle \lambda_O, cs, \nu' \rangle \sim \sigma''\) from lemma 5.11 and \(\langle \lambda_O, cs, \nu' \rangle \sim \sigma'\) from Theorem 5.6, since the precondition code does not alter any variables except expression local ones. The rest follows from the semantics of \text{if then else} and \(\&\&\). \qed

**Lemma 5.13** (Fireable Segment Equivalence). Given a component \(C\), a transition segment \(t = (\text{src}, \text{trg}, I, G, O, A)\), a component state \(\sigma_C = (\lambda_O, cs, \nu)\), an input message valuation \(\lambda_I \in \Lambda_I\), and program states \(\sigma, \sigma'\) with

\[
\sigma_C \sim \sigma \\
\lambda_I \sim \sigma \\
\sigma(\text{guard}) = \text{true} \\
\langle \sigma, \text{preguard}(t) \rangle \mapsto \sigma' 
\]

Then the guard variable value is equivalent to the \text{fireable}_{atomic} predicate and the component state remains unchanged

\[
\sigma_C \sim \sigma' \\
\lambda_I \sim \sigma' \\
\text{fireable}_{atomic}(\sigma_C, \lambda_I, t) \iff \sigma'(\text{guard}) = \text{true} 
\]

**Proof.** Let \(\langle \sigma, \text{inputguard}(t) \rangle \mapsto \sigma''\), then we get from lemma 5.11 the first part of the predicate definition 3.20. The second part follows from lemma 5.11. \qed

### 5.4.7 Transition Segment Sequence Guard Mapping

**Definition 5.27** (Transition Segment Sequence Guard Mapping). Let \(C \in C_{atm}\) be an atomic component, \(t = (\text{src}, \text{trg}, I, G, O, A)\) be an arbitrary transition segment of its automaton specification \(\text{Atm}_C\), \(N = \langle (\text{src}', \text{trg}', I', G', O', A') \in \text{TS} | \text{src'} = \text{trg} \land \text{src'} \notin S \rangle\) be the set of subsequent transition segments connected via a local interface point, and \(< \subset \text{TS} \times \text{TS}\) be the total order of transition segments induced by the unique object identifiers. Furthermore, let \text{guard} be the same variable as in the last definition, \text{ghost} a unique \text{int} variable and \text{fired} a unique \text{bool} variable. Then \text{seqguard} : \text{TS} \rightarrow \text{STM}\ maps the transition segment to its guard code as follows

- Terminating segment: \(N = \emptyset\)

\[
\text{seqguard}(t) = \text{preguard}(t); 
\]
• Intermediary segment: \( N = \{t_0, \ldots, t_n\} \) with \( \forall i, j : i < j \Leftrightarrow t_i < t_j \)

\[
\begin{align*}
\text{segguard}(t) &= \\text{preguard}(t); \\
&\quad \text{if (guard)} \{ \\
&\quad\quad \text{fired} = \text{false}; \\
&\quad\quad \text{if (!fired)} \{ \\
&\quad\quad\quad \text{fired} = \text{segmentfireproc}(t_0)(); \\
&\quad\quad\quad \text{ghost} = 0; \\
&\quad\quad\} \\
&\quad \quad \vdots \\
&\quad\quad \text{if (!fired)} \{ \\
&\quad\quad\quad \text{fired} = \text{segmentfireproc}(t_n)(); \\
&\quad\quad\quad \text{ghost} = n; \\
&\quad\quad\} \\
&\quad \text{guard} = \text{fired}; \\
&\} \\
\end{align*}
\]

the \textit{ghost} variable is not needed for semantic equivalence, but it eases the proof greatly, since its value denotes the execution of a specific if-block.

\textbf{Lemma 5.14} (Transition Segment Sequence Guard Equivalence). Given a component \( C \), a transition segment \( t = \langle \text{src}, \text{trg}, I, G, O, A \rangle \), a component state \( \sigma_C = \langle \lambda_O, cs, v \rangle \), an input message valuation \( \lambda_I \in \Lambda_I \), and program states \( \sigma, \sigma' \) with

\[
\begin{align*}
\sigma_C &\sim \sigma \\
\lambda_I &\sim \sigma \\
\sigma(\text{guard}) &\equiv \text{true} \\
\langle \sigma, \text{segguard}(t) \rangle &\rightarrow \sigma'
\end{align*}
\]

Then the guard variable value is equivalent to the \textit{fireable} predicate

\[
\text{fireable}(\sigma_C, \lambda_I, t) \Leftrightarrow \sigma'(\text{guard}) = \text{true}
\]

\textit{Proof.} Consider \( N = \{ \langle \text{src}', \text{trg}', I', G', O', A' \rangle \in TS \mid \text{src}' = \text{trg} \land \text{src}' \notin S \} \) the set of subsequent transition segments. The proof is done by induction on the acyclic transition segment graph.

- Terminating segment: \( N = \emptyset \)
  From the definition follows \( \text{fireable}(\sigma_C, \lambda_I, t) = \text{fireable}_{\text{atomic}}(\sigma_C, \lambda_I, t) \) and \( \text{segguard}(t) = \text{preguard}(t) \). With lemma 5.13 [Fireable Segment Equivalence] the proof follows .

- Intermediary segment: \( N = \{t_0, \ldots, t_n\} \) with \( \forall i, j : i < j \Leftrightarrow t_i < t_j \)
Consider $\forall i : \text{fireable}(t_i) = false$. From the induction hypothesis we get $\forall i : \text{segmentfireproc}(t_i)(t) = false$. Then the proof follows by applying the definitions of if then else and assignment, since \text{fired} remains false all the time.

Consider $\exists i : \text{fireable}(t_i) = true$. From the induction hypothesis we get $\exists i : \text{segmentfireproc}(t_i)(t) = true$. Then the proof follows by applying the definitions of if then else and assignment, since \text{fired} becomes true in the $i$-th if-block.

Lemma 5.15 (Transition Selection Equivalence). Given a transition segment $t \in TS$ with $\text{fireable}(t)$, and let $N = \{t_0, \ldots, t_n\}$ with $\forall i, j : i < j \Rightarrow t_i < t_j$ be the ordered set of subsequent transition segments with at least one segment ($n \geq 0$). Let a component state $\sigma_C = \langle \lambda_O, cs, v \rangle$, an input message valuation $\lambda_I \in \Lambda_I$, and program states $\sigma, \sigma'$ with

- $\sigma_C \sim \sigma$
- $\lambda_I \sim \sigma$
- $\sigma(\text{guard}) = true$
- $\langle \sigma, \text{segguard}(t) \rangle \mapsto \sigma'$

Let $fcode(t_k) = if (!\text{fired}) \{ \text{fired} = \text{segmentfireproc}(t_k); \}$, then

$$
\begin{align*}
\text{dom.selectnext}(\sigma_C, \lambda_I, t) \neq \emptyset & \Rightarrow \\
\exists t_i \in N : t_i = \text{selectnext}(\sigma_C, \lambda_I, t) & \land \\
\sigma'(\text{guard}) = true & \land \\
\sigma'(\text{ghost}) = i & 
\end{align*}
$$

Proof. The proof is done by induction on the ordered elements in $N$ using lemma 5.14. All segments before $t_i$ are not fireable, thus \text{fired} remains false. Then $t_i$ is fired, i.e. \text{fired} becomes true and \text{ghost} becomes $i$. After that both variables remain so, since the later segments are guarded by $!\text{fired}$. Finally, \text{guard} is updated with the value \text{fired}.

5.4.8 Transition Segment Effect Mapping

Definition 5.28 (Transition Segment Effect Mapping). Let $C \in C_{atm}$ be an atomic component, $t = (src, trg, I, G, [o_0, \ldots, o_n], [a_0, \ldots, a_m])$ be an arbitrary transition segment of its automaton specification $Atm_C$. Furthermore, let \text{guard} be a unique variable. Then $\text{transcode} : TS \rightarrow STM$ maps the transition
segment to its effect code as follows

\[
\begin{align*}
\text{transcode}(t) = \\
&\text{seqguard}(t); \\
&\text{if} (\text{guard}) \{ \\
&\quad \text{outputcode}(o_0); \\
&\quad \ldots \\
&\quad \text{outputcode}(o_n); \\
&\quad \text{postcode}(a_0); \\
&\quad \ldots \\
&\quad \text{postcode}(a_m); \\
&\quad \text{statecode}(\text{trg}); \\
&\} \\
\end{align*}
\]

with

\[
\text{statecode}(\text{trg}) = \begin{cases} [] & \text{if } \text{trg} \notin S \\
\text{cstate} = \text{map}^{\text{AF3}}_{\text{C0}}(\text{trg}) & \text{otherwise} \end{cases}
\]

The complete transition code consists of the guard code including subsequent segment calls and the effect code, which is executed only if the guard holds (i.e. subsequent segments fired already). The effect code updates the output and data state variables values in the component state and if the segment is a terminating segment, it updates the current state variable value.

The guard code searches a path of consecutive fireable transition segments (storing procedure calls on the C0 stack) and once this path is found the effects of all stacked segments are applied (in reverse order). Due to constraint \textit{??}, \textit{??} and \textit{??} the order of effect application does not matter as long as all are atomically applied, i.e. data state variables are written to the scratch paper variables.

The following lemma proves the equivalence between execution semantics with the difference that the resulting C0 program state is equivalent if the scratch paper values are considered instead of the current data state variables. The considered segment must be fireable.

**Lemma 5.16 (Fireable Transition Segment Effect Equivalence).** Given a component \( C \), a transition segment \( t = < \text{src, trg, I, G, O, A} > \), a component state \( \sigma_C = < \lambda_O, cs, v > \), an input message valuation \( \lambda_I \in \Lambda_I \), and program states \( \sigma, \sigma' \) with

\[
\begin{align*}
\sigma_C \sim \sigma \\
\lambda_I \sim \sigma \\
\sigma(\text{guard}) &= \text{true} \\
\forall d \in DSV : \sigma(\text{scratchpaper}(d)) &= \sigma(d) \\
(\sigma, \text{transcode}(t)) &\rightarrow \sigma'
\end{align*}
\]

Then, given the transition is fireable, the resulting state is equivalent (if scratch paper state is considered instead of data state), but data state variable values

79
remain unchanged (let \( \delta_{\text{seg}}(\sigma_C, \lambda_I, t) = (\lambda'_O, cs', v') \)).

\[
\text{fireable}(t) \Rightarrow \\
\sigma'(\text{guard}) = \text{true} \land \\
\lambda'_O \sim \sigma' \land \\
\text{map}_{AF}^{3}(cs') = \sigma'(\text{cstate}) \land \\
\forall d \in DSV : \sigma(d) = \sigma'(d) \land \\
\forall d \in DSV : v'(d) \sim \sigma'(\text{scratchpaper}(d))
\]

Otherwise the component state (including scratch paper variables) remains unchanged and the guard is false

\[
\neg\text{fireable}(t) \Rightarrow \\
\sigma_C \sim \sigma' \land \\
\lambda_I \sim \sigma' \land \\
\sigma'(\text{guard}) = \text{false} \land \\
\forall d \in DSV : \sigma(\text{scratchpaper}(d)) = \sigma'(\text{scratchpaper}(d))
\]

Proof. Consider \( N = \{(\text{src}, \text{trg}', I', G', O', A') \in TS | \text{src} = \text{trg} \land \text{src}' \notin S \} \) the set of subsequent transition segments. The proof is done by induction on the acyclic transition segment graph.

- Terminating segment: \( N = \emptyset \)

Let \( \langle \sigma, \text{seqguard}(t) \rangle \mapsto \sigma'' \). Then \( \text{fireable}(t) \iff \sigma''(\text{guard}) = \text{true} \) with lemma 5.14. With lemma 5.12 we have \( \sigma_C \sim \sigma'' \) and \( \lambda_I \sim \sigma'' \). If the guard is true, the proof follows from the assignment semantics and lemma 5.9 and 5.10 applied step-wise. If the guard is false, the proof follows from the if then else semantics.

- Intermediary segment: \( N = \{t_0, \ldots, t_n\} \) with \( \forall i, j : i < j \iff t_i < t_j \)

- Case \( \text{fireable}(t) = \text{true} \):

Let \( \langle \sigma, \text{seqguard}(t) \rangle \mapsto \sigma'' \). Then \( \sigma''(\text{guard}) = \text{true} \) by lemma 5.14. Let \( \delta_{\text{seg}}(\sigma_C, \lambda_I, \text{select}_{\text{next}}(\sigma_C, \lambda_I, t)) = (\lambda''_O, cs'', v'') \). We get from the induction hypothesis and Def. 3.23 that

\[
\sigma''(\text{guard}) = \text{true} \\
\lambda''_O \sim \sigma'' \\
\text{map}_{AF}^{3}(cs'') = \sigma''(\text{cstate}) \\
\forall d \in DSV : \sigma(d) = \sigma''(d) \\
\forall d \in DSV : v''(d) \sim \sigma''(\text{scratchpaper}(d))
\]

The proof follows from the if then else and assignment semantics and lemma 5.9 and 5.10 applied step-wise.
– Case fireable(t) = false:
Let \(\langle \sigma, seqguard(t) \rangle \mapsto \sigma''\). Then \(\sigma''(guard) = false\) by lemma 5.14.
We get from the induction hypothesis that
\[
\begin{align*}
\sigma_C & \sim \sigma'' \\
\lambda_I & \sim \sigma'' \\
\sigma''(guard) & = false \\
\forall d \in DSV : \sigma(\text{scratchpaper}(d)) & = \sigma''(\text{scratchpaper}(d))
\end{align*}
\]

The proof follows from the if then else semantics.

\[\square\]

5.4.9 Transition Segment Procedure

**Definition 5.29** (Transition Segment Procedure). Let \(C \in C_{atm}\) be an atomic component, \(t = \langle src, trg, I, G, O, A \rangle\) be an arbitrary transition segment of its automaton specification \(Atm_C\). Furthermore, let \(guard\) be a unique variable. Then \(segmentfirecode : TS \rightarrow STM\) maps the transition segment to the body of the corresponding segment fire procedure as follows
\[
\text{segmentfirecode}(t) = \\
\text{transcode}(t); \\
\text{return guard;}
\]
Furthermore, we define a procedure \(\pi_t\) in the C0 procedure table for each transition segment \(t\) with
- \(\pi_t.body = segmentfirecode(t)\);
- \(\pi_t.rtype = \text{bool}\)
- \(\pi_t.params = []\)
- \(\pi_t.locvars\) is computed from the procedure body.

We assume that \(segmentfireproc\) from Def. 5.21 maps each transition segment to a unique procedure identifier, which points to the corresponding procedure definition in the procedure table, i.e. \(ptable(segmentfireproc(t)) = \pi_t\).

**Lemma 5.17** (Transition Segment Equivalence). Given a component \(C\), a transition segment \(t = \langle src, trg, I, G, O, A \rangle\), a component state \(\sigma_C = \langle \lambda_O, cs, v \rangle\), an input message valuation \(\lambda_I \in \Lambda_I\), result a C0 variable, and program states \(\sigma, \sigma'\) with
\[
\begin{align*}
\sigma_C & \sim \sigma \\
\lambda_I & \sim \sigma \\
\forall d \in DSV : \sigma(\text{scratchpaper}(d)) & = \sigma(d) \\
\langle \sigma, \text{result} = segmentfireproc(t) \rangle & \mapsto \sigma'
\end{align*}
\]
Then the resulting state is equivalent (if scratch paper state is considered instead of data state)

\[ \text{fireable}(t) \Rightarrow \]
\[ \sigma'(\text{result}) = \text{true} \land \]
\[ \lambda'_O \sim \sigma' \land \]
\[ \text{map}_{C_0}^{AF^3}(cs') = \sigma'(c\text{state}) \land \]
\[ \forall d \in DSV : \sigma(d) = \sigma'(d) \land \]
\[ \forall d \in DSV : \nu'(d) \sim \sigma'(scratchpaper(d)) \]

Otherwise the component state (including scratch paper variables) remains unchanged and the guard is false

\[ \neg \text{fireable}(t) \Rightarrow \]
\[ \sigma_C \sim \sigma' \land \]
\[ \lambda_I \sim \sigma' \land \]
\[ \sigma'(\text{result}) = \text{false} \land \]
\[ \forall d \in DSV : \sigma(\text{scratchpaper}(d)) = \sigma'(\text{scratchpaper}(d)) \]

**Proof.** The proof follows directly from lemma 5.16 and the procedure call semantics. \qed

### 5.4.10 Component Step Procedure

**Definition 5.30** (Component Step Procedure). Let \( C \in \mathcal{C}_{atm} \) be an atomic component, \( Atm_C = \{s_1, \ldots, s_n\}, CP, TS, \{d_1, \ldots, d_m\}, \text{type}_{dsv}, s, \text{init}_{dsv} \) its automaton specification, \( O_C = \{o_0, \ldots, o_l\} \) be its output port set, and let \( \text{guard} \) be a unique variable. Then \( \text{stepcode} : \mathcal{C}_{atm} \rightarrow \text{STM} \) maps the component’s automaton specification to the step procedure code as follows

\[ \text{stepcode}(C) = \]
\[ \text{scratchpaper}(d_0) = \text{map}_{C_0}^{AF^3}(d_0); \]
\[ \vdots \]
\[ \text{scratchpaper}(d_m) = \text{map}_{C_0}^{AF^3}(d_m); \]
\[ \text{flag}_{C_0}^{AF^3}(o_0) = \text{false}; \]
\[ \vdots \]
\[ \text{flag}_{C_0}^{AF^3}(o_l) = \text{false}; \]
\[ \text{guard} = \text{false}; \]
\[ \text{statetestcode}(s_1); \]
\[ \vdots \]
\[ \text{statetestcode}(s_n); \]
\[ \text{commitcode}(); \]
with \( N = \{ t_0, \ldots, t_k \mid \forall i : t_i \in TS \land t_i/src = s \} \) with \( \forall i, j : i < j \implies t_i < t_j \) be the ordered set of transition segments starting in \( s \)

\[
\text{statetestcode}(s) = \\
\begin{aligned}
&\text{if} \ (\neg \text{guard} \ \& \ \& \ \text{cstate} == \map{AF3}{C_0}(s)) \ {\{} \\
&\quad \text{if} \ (\neg \text{guard}) \ {\{} \\
&\quad \quad \text{guard} = \text{segmentfireproc}(t_0)(); \\
&\quad \quad \ldots \\
&\quad \quad \text{if} \ (\neg \text{guard}) \ {\{} \\
&\quad \quad \quad \text{guard} = \text{segmentfireproc}(t_k)(); \\
&\quad \ {\}} \\
&\ {\}} \\
\end{aligned}
\]

and

\[
\text{commitcode}() = \\
\begin{aligned}
&\text{if} \ (\text{guard}) \ {\{} \\
&\quad \map{AF3}{C_0}(d_0) = \text{scratchpaper}(d_0); \\
&\quad \ldots \\
&\quad \map{AF3}{C_0}(d_m) = \text{scratchpaper}(d_m); \\
&\ {\}} \\
\end{aligned}
\]

The component step procedure works as follows. First the scratch paper variables are initialized to the current data state variable values, the output port values are reset, and the guard is set to false. Then the current state is selected, which also takes into account the guard value, since as soon as the guard becomes true (by some transition execution) the current state will have changed. Thus, operational C0 semantics forces us to work with the guard here. After the current state is selected, each outgoing transition is tried until the first succeeds (the guard is now true and no other segments will be tried). Note that the code generator uses the total order of the unique object identifiers here. Finally, the scratch paper values are written to the data state variables if some segment (sequence) was executed. Otherwise, the data state variables carry their old value and all the output port values are reset.

**Theorem 5.18** (Component Step Equivalence). Given a component \( C \), a component state \( \sigma_C = (\lambda_O, cs, v) \), an input message valuation \( \lambda_I \in \Lambda_I \), and program states \( \sigma, \sigma' \) with

\[
\begin{aligned}
\sigma_C &\sim \sigma \\
\lambda_I &\sim \sigma \\
(\sigma, \text{stepcode}(C)) &\mapsto \sigma'
\end{aligned}
\]

Then the resulting state is equivalent

\[
\delta_{\text{step}}(\sigma_C, \lambda_I) \sim \sigma'
\]
Proof. Let $T = \{ t \in TS | t.src \in S \}$ and $\sigma''$ be the C0 program state after executing $\text{guard} = \text{false}$. Then $\forall d \in DSV : \sigma''(\text{scratchpaper}(d)) = \sigma(d)$, $\sigma''(\text{guard}) = \text{false}$, $\sigma''(\text{cstate}) = \sigma(\text{cstate})$, and $\forall o \in O_C : \epsilon \sim \sigma''(o)$.

We consider two cases:

- Idle transition: $\forall t \in T : t.src = \sigma_C.cs \Rightarrow \text{fireable}(t)$
  We know $\text{dom.select}_\text{first}(\sigma_C, \lambda_I) = \emptyset$ and we must show from Def. 3.25

  \[
  \forall o \in O_C : \lambda'_I(o) = \epsilon \\
  \forall dsv \in DSV : v'(dsv) = v(dsv)
  \]

  We consider only the statecode($cs$), since all other state code sections are guarded by the state check. From lemma 5.17 we conclude that guard stays false and the component state remains unchanged. Therefore, the commit code is not (executed) and the proof follows.

- Specified transition: $\exists t \in T : t.src = \sigma_C.cs \land \text{fireable}(t)$
  We consider only the statecode($cs$), since all other preceding state code sections are guarded by the state check. With lemma 5.15 we know that the first transition segment is executed and from lemma 5.17 that the resulting state is equivalent considering the scratch paper variables. Furthermore, we get that guard is true and that the current state might have changed. It follows that all other transition segments are not considered. All remaining state code sections are also not considered. Finally, the commit code is executed and we conclude from assignment semantics that now the component states are equivalent, since scratch paper values were equivalent.

\[\square\]

5.5 Composite Behavior

Up to now we have proven the semantic equivalence for atomic components. Based on theorem 5.1 we can now proof the semantic equivalence for networks of components. The code generator produces two C0 procedures: an initialization procedure, which must be called once at the beginning, and a network step procedure, which must be called once for each logical step of the system.

We first define the equivalence relation for the network state and the corresponding C0 program. The network consists of the set of atomic components and the environment component. Note that the equivalence relation does not consider the environment input port values. Environment inputs must be filled before the network step procedure is called, thus these inputs are only read by the system. Filling the input port buffers with the current values is up to the deployment structures such as a task executing the C0 program. Usually, the C0 code produced from the model is embedded into a piece of hull code that connects it to the sensors and actuators.
Definition 5.31 (Network State Equivalence). Let $C_{env}$ be the environment component, $ifc_{syn}^{env} = (I_{C_{env}}, O_{C_{env}}, type_{C_{env}}^{port}, init_{C_{env}}^{port})$ the environment interface, $C_{atm}$ be the set of atomic components, and let $\sigma_{net} = \langle (\sigma_C)_{C \in C_{atm}}, \lambda_{O_{env}} \rangle$ be the network state. Furthermore, let $\sigma$ be a C0 program state. Then

\[ \sigma_{net} \sim \sigma \iff \forall C \in C_{atm} : \sigma_C \sim \sigma \land \forall o \in O_{C_{env}} : \lambda_{O_{env}}(o) \sim \sigma(o) \]

The network state is equivalent if the atomic component states and the environment output port valuations are equivalent. It is important to notice that the code generator takes care of producing disjunctive variables for the component interface state and the automaton state of the involved components. During the network step procedure data is transferred between the interface state variables according to the connections induced by the specified channels.

Definition 5.32 (Network Initialization). Let $L = []$ be the list of atomic components from $C_{atm}$ in arbitrary order and let $init_{C0}^{AF3} : C_{atm} \to STM$ be code generation function, which produces the initialization code for the atomic components. Furthermore, let $ifc_{syn}^{env} = (I_{C_{env}}, [o_0, \ldots, o_n], type_{C_{env}}^{port}, init_{C_{env}}^{port})$ be the environment interface. Then $init_{C0}^{AF3} : list (C_{atm}) \to STM$ is defined according to the following rules:

\[
\begin{align*}
L \neq [] & \quad \Rightarrow \\
init_{C0}^{AF3}(L) = & \\
init_{C0}^{AF3}(L.first); \\
init_{C0}^{AF3}(L.rest); \\

init_{C0}^{AF3}([]) = & \\
outputcode(o_0, init_{C_{env}}^{port}(o_0)); \\
& \vdots \\
outputcode(o_n, init_{C_{env}}^{port}(o_n)); \\
\end{align*}
\]

Theorem 5.19 (Network Initialization Equivalence). Let $L \neq []$ be the list of atomic components from $C_{atm}$ in arbitrary order, $\sigma_{net}^{init} = \langle (\sigma_C^{init})_{C \in C_{atm}}, \lambda_{O_{env}} \rangle$ be the initial network state, and $\sigma, \sigma'$ C0 program states with $\langle \sigma, init_{C0}^{AF3}(L) \rangle \mapsto \sigma'$

Then the initial network state is equivalent to the initialized C0 program state:

\[ \sigma_{net}^{init} \sim \sigma' \]
Proof. The equivalence of the atomic components' state follows by applying Lemma 5.7 and the fact that the state variables for different atomic components are disjunct. The equivalence of the environment output port valuations follows from the Definition 5.23, the C0 assignment semantics and the Theorem 5.6.

Definition 5.33 (Channel Transmission). Let \( \mathcal{CH} = \mathcal{CH}_{in} \cup \mathcal{CH}_{out} \cup \mathcal{CH}_{local} \) be the set of channels of the network, and \( \langle \text{src}, \text{dst} \rangle \in \mathcal{CH} \) be a particular channel. Then \( \text{transmit}^{AF3}_{C0} : \mathcal{CH} \rightarrow \text{STM}T \) is a mapping that produces for each channel its transmission code.

\[
\text{transmit}^{AF3}_{C0}(\langle \text{src}, \text{dst} \rangle) =
\begin{align*}
\text{map}^{AF3}_{C0}(\text{dst}) &= \text{map}^{AF3}_{C0}(\text{src}); \\
\text{flag}^{AF3}_{C0}(\text{dst}) &= \text{flag}^{AF3}_{C0}(\text{src});
\end{align*}
\]

Lemma 5.20 (Channel Transmission). Let \( ch = \langle \text{src}, \text{dst} \rangle \in \mathcal{CH} \) be a channel and \( \sigma, \sigma' \) be C0 program states with \( \langle \sigma, \text{transmit}^{AF3}_{C0}(ch) \rangle \vdash \sigma' \) Then

\[
\sigma'(\text{dst}) = \sigma(\text{src})
\]

Proof. The proof follows directly from the definition of \( \text{map}^{AF3}_{C0} \) and \( \text{flag}^{AF3}_{C0} \) and the C0 assignment semantics.

Before giving the formal part of the network step procedure we discuss the idea of the C0 implementation.

The causal dependency relation from Definition 3.28 induces an order on the set of atomic components. Atomic components are executed one after the other according to this order. Channel transmission is intertwined with the execution of atomic components depending on the causality of the component. In order to execute a single step of the network the following steps are executed by the network step procedure assuming that the environment input port valuations have been updated before the procedure is called:

1. Transmit the environment input port values along all channels \( ch \in \mathcal{CH}_{in} \).
2. Transmit the output port values along all channels \( ch \in \mathcal{CH}_{out} \cup \mathcal{CH}_{local} \), if the source port of \( ch \) is an output port of a strongly causal component.
3. Execute all weakly causal components according to the causal dependency order and forward the output port values of each such component directly after its step procedure was called.
4. Finally, execute all step procedures of strong causal components (in arbitrary order). This loads the interface state buffers of these components with the results of the computation step. The result is then forwarded to the receivers during the second step of the next network computation cycle, i.e. the results reach the receivers delayed by one tick as required by strong causality.

86
Note that we use the list of atomic ordered in inverse order as we did in Definition 3.32. This eases the proof of equivalence.

**Definition 5.34 (Network Step Procedure).** Let \( L = [cmp_k, \ldots, cmp_0] \in \text{list}(\mathcal{C}_{\text{atm}}) \) be the list of atomic components sorted inverse to the causal dependency order. We assume that \( L_{\text{strong}} = [cmp_k, \ldots, cmp_j], k + 1 \geq j \geq 0 \) is the (sub-)list of strongly causal components and \( L_{\text{weak}} = [cmp_{j-1}, \ldots, cmp_0] \) is the list of weakly causal components.

Let \( [ch_0^{\text{in}}, \ldots, ch_n^{\text{in}}] \in \text{list}(\mathcal{CH}_{\text{in}}) \) be the list of environment input channels, \( [ch_0^{\text{wc}}, \ldots, ch_m^{\text{wc}}] \in \text{list}(\mathcal{CH}_{\text{out}} \cup \mathcal{CH}_{\text{local}}) \) be the list of channels leaving the weakly causal component \( wc \in L_{\text{weak}} \), and \( [ch_0^{\text{sc}}, \ldots, ch_m^{\text{sc}}] \in \text{list}(\mathcal{CH}_{\text{out}} \cup \mathcal{CH}_{\text{local}}) \) be the list of channels leaving some strongly causal component.

Then the network step procedure is generated as follows:

```plaintext
networkstepcode =
  // envinputcode
  transmit_{C_0}^{AF3}(ch_0^{\text{in}});
  \ldots;
  transmit_{C_0}^{AF3}(ch_n^{\text{in}});

  // strongchannelcode
  transmit_{C_0}^{AF3}(ch_0^{\text{sc}});
  \ldots;
  transmit_{C_0}^{AF3}(ch_m^{\text{sc}});

  // componentexeccode
  stepcode(cmp_0);
  transmit_{C_0}^{AF3}(ch_0^{\text{cmp}_0});
  \ldots;
  transmit_{C_0}^{AF3}(ch_m^{\text{cmp}_0});
  \ldots;
  stepcode(cmp_{j-1});
  transmit_{C_0}^{AF3}(ch_0^{\text{cmp}_{j-1}});
  \ldots;
  transmit_{C_0}^{AF3}(ch_m^{\text{cmp}_{j-1}});
  stepcode(cmp_{j});
  \ldots;
  stepcode(cmp_k);
```

**Theorem 5.21 (Network Step Equivalence).** Let \( \sigma_{\text{net}} \) be a network state, \( \lambda_{L_{\text{env}}} \) be the current environment input, and program states \( \sigma, \sigma' \) with

\[
\sigma_{\text{net}} \sim \sigma \\
\lambda_{L_{\text{env}}} \sim \sigma \\
(\sigma, \text{networkstepcode}) \mapsto \sigma'
\]
Then the resulting state is equivalent

\[ \delta_{\text{net}}(\sigma_{\text{net}}, \lambda_{\text{env}}) \sim \sigma' \]

Proof. We use the following partial executions of the of the network step code:

- \( \langle \sigma, \text{envinpcode} \rangle \mapsto \sigma^1 \) for the environment channel transmission code.
- \( \langle \sigma^1, \text{strongcchanelcode} \rangle \mapsto \sigma^2 \) for the strongly causal component output transmission.
- \( \langle \sigma^2, \text{componentexeccode} \rangle \mapsto \sigma' \) for the computation and transmission code of the weakly causal components.

From Definition 3.33 we have to show the equivalence by induction on the list of components \( L \):

- Base case: \( L = [\] \)
  
  For this case the network step code is empty and thus the equivalence follows trivially.

- Induction Hypothesis:
  
  \( \text{exec}_{\text{net}}(L, \sigma_{\text{net}}, \lambda_{\text{env}}) \sim \sigma' \)

- Induction case: \( L = L_{\text{strong}} \@ L_{\text{weak}} \neq [\]

1. Let us consider \( \sigma^1 \). Let \( \sigma^1_{\text{net}} \sim \sigma^1 \) as in Definition 5.31. With Lemma 5.20 we conclude that

\[
\forall C \in L : \forall p \in I_C : \\
\begin{align*}
\lambda^\text{cur}_{IC}(L, p, \sigma_{\text{net}}, \sigma^1_{\text{net}}, \lambda_{\text{env}}) , & \text{ if } \exists (s, p) \in CH_{\text{in}} \\
\sigma_{\text{net}}(p) , & \text{ otherwise}
\end{align*}
\]

When the first intermediate state is reached the input port values of all ports connected to some environment input have been updated with the respective value.

2. Let us consider \( \sigma^2 \). Let \( \sigma^2_{\text{net}} \sim \sigma^2 \) as in Definition 5.31. With Lemma 5.20 we conclude for the port valuations that

\[
\forall C \in L : \forall p \in I_C : \\
\begin{align*}
\lambda^\text{cur}_{IC}(L, p, \sigma_{\text{net}}, \sigma^2_{\text{net}}, \lambda_{\text{env}}) , & \text{ if } \exists (s, p) \in CH_{\text{in}} \\
\lambda^\text{cur}_{IC}(L, p, \sigma_{\text{net}}, \sigma^2_{\text{net}}, \lambda_{\text{env}}) , & \text{ if } \exists (s, p) \in CH_{\text{local}} \land \\
\sigma_{\text{net}}(p) , & \text{ otherwise}
\end{align*}
\]

Furthermore, for the environment output port valuations, we get

\[
\forall p \in O_{\text{en}} : \\
\begin{align*}
\lambda^\text{cur}_{C_{\text{en}}}(L, p, \sigma_{\text{net}}, \sigma^2_{\text{net}}) , & \text{ if } \exists (s, p) \in CH_{\text{out}} \land \\
\sigma_{\text{net}}(p) , & \text{ otherwise}
\end{align*}
\]
When the second intermediate state is reached the input port values of all ports connected to some environment input port or strongly causal component output ports have been updated with the respective value. Furthermore, all environment output ports connected to some strongly causal component output ports have been updated with the respective value.

3. Let us consider \( \sigma' \). Let \( C_{cur} = L.first \) be the current component and \( L' = L.rest \) be the list of already computed components (in particular the components the current component causally depends on). Let \( \sigma^3_{net} \sim \sigma^3 \) be the network state and program state before executing \( C_{cur} \). We know from the induction hypothesis that

\[
\forall p \in I_{C_{cur}} : \lambda^{cur}_{I_{C_{cur}}}(L', p, \sigma_{net}, \sigma^3_{net}, \lambda_{I_{env}}) \sim \sigma^3(p)
\]

and

\[
\forall C_{pre} \in L' : \sigma^3_{net} \cdot \sigma_{C_{pre}} \sim \sigma^3
\]

and

\[
\forall p \in O_{C_{env}} : \lambda^{cur}_{O_{C_{env}}}(L', p, \sigma_{net}, \sigma^3_{net}) \sim \sigma^3(p)
\]

We now distinguish two cases

a) \( C_{cur} \in L_{weak} \)

With

\[
\text{curcode} \equiv \text{stepcode}(C_{cur});
\]

\[
\text{transmit}^{AF3}_{C_{cur}}(ch^{C_{cur}}_{0});
\]

\[
\vdots
\]

\[
\text{transmit}^{AF3}_{C_{cur}}(ch^{C_{cur}}_{m_{C_{cur}}});
\]

we have \( (\sigma^3, \text{curcode}) \mapsto \sigma' \). We can now apply Theorem 5.18 and Lemma 5.20 and \( \text{exec}_{net}(L, \sigma_{net}, \lambda_{I_{env}}) \sim \sigma' \) follows. In other words the theorem adds the current component’s updated state as required by Definition 3.33, while the lemma application completes the environment output valuations as required by the same definition.

b) \( C_{cur} \in L_{strong} \)

For strongly causal components \( \text{exec}_{net}(L, \sigma_{net}, \lambda_{I_{env}}) \sim \sigma' \) follows by application of Theorem 5.18. It adds the current component’s updated state as required by Definition 3.33, while the environment output was already correct at the intermediary state \( \sigma^2 \).

- The theorem follows by considering the complete list of all atomic components ordered in reverse order of the causal dependency relation as required by Definition 3.33.
6 Resource Consumption

The C0 program generated from AutoFocus 3 models exposes several computable properties: statically bounded memory consumption and deterministic worst-case execution time. Note that both the compiler and the execution environment must also be considered and verified that these properties are preserved. Both memory consumption and execution time provided here form upper bounds that are required from a logical point of view. Compiler and scheduler technology influence these properties on real hardware. These lower layers have been addressed by the Verisoft project. We do not go into details in this report.

Memory Consumption For each variable of given C0 type we need the following (logical) amount of memory cell (assume that one cell is 8 Bit large). Each bool variable needs 1 memory cell. Each int variable needs 4 memory cells. Each struct variable needs the sum of the needs of its constituents, e.g. a tuple of two integer values amounts to a total of 8 memory cells.

Note that the compiler might apply optimization techniques to reduce the memory footprint. A typical example is the compactization of bool values. Instead of wasting 7 bits with each boolean variable, the compiler could use a single byte to store 8 bool values and apply special processor commands to access the value when needed.

Furthermore, the code generator could be optimized to reduce the number of local variables, which were introduced by the unfolding process. However, such optimizations must be verified that they do not destroy the semantic equivalence.

Worst-case Execution Time (WCET) We have restricted the data definitions of the AutoFocus 3 model by disallowing recursive data types and user-defined functions. Furthermore, the transition segments are required to be acyclic. Thus, the generated procedures in the C0 program are also free of recursion. This leads to the possibility of computing execution times statically. However, we can only compute reference values or logical execution times, since the real execution times for a specific processor highly depends on processor technologies (e.g. cache levels, operation pipelining, etc.).

We have only shown that the application software provides computable WCETs. Preservation of this times must be verified for the specific execution environment.

7 Applying verification techniques to AutoFocus 3 models

In this report we have shown the semantic equivalence of AutoFocus 3 models and the generated C0 code. In order to verify the correctness of applications modeled in AutoFocus 3 we need to address the problem of integer overflow,
division by zero and non-deterministic behavior. Our methodology proposes to address this problems at the level of the AutoFocus 3 model using suitable verification mechanisms, in particular model-checking. We have applied these techniques in an industrial case study provided in [3].

**Integer Overflow and Division by Zero** Integer overflow and division by zero can occur during integer computations. From the semantic equivalence of the basic operations in AutoFocus 3 and C0, we know that these problems already arise at the model level. In order to tackle the division we advise that “defensive modeling”, i.e. checking input values before using them in computations, is used carefully in order to avoid overflows and division-by-zero problems. Since the data types of AutoFocus 3 are bounded, we can use model-checking techniques verify critical parts of the system.

**Non-Determinism** The automaton behavior of an AutoFocus 3 component can be non-deterministic, i.e. for a given control state, a data state and a given input valuation there can be more than one transition that is fireable. However, this non-determinism will not be present in the C0 implementation, since the C0 program executes sequentially. Thus a particular transition (e.g. the one with lowest object identifier) will be preferred and the non-determinism will be resolved in an unfair way. In general, we get unreachable C0 code segments.

However, we can verify the AutoFocus 3 application model for non-deterministic behavior using model-checking techniques, since the value space of inputs and the control and data state space is finite.