

# Testing Consistency between Behavior and Interaction Descriptions

Joan Rigo

Departament de Matemàtiques et Informàtica  
Universitat des les Illes Balears  
Carretera de Valldemossa, Km 7.5  
E-07071 Palma de Mallorca, Spain  
dmjrv4@ps.uib.es

Oscar Slotosch

Technische Universität  
München  
Institut für Informatik  
Arcisstr. 21  
80290 München, Germany  
slotosch@in.tum.de

Marc Sihling

Technische Universität  
München  
Institut für Informatik  
Arcisstr. 21  
80290 München, Germany  
sihling@in.tum.de

Heiko Lötzbeyer

Technische Universität  
München  
Institut für Informatik  
Arcisstr. 21  
80290 München, Germany  
loetzbey@in.tum.de

*Abstract*— In this paper we present a way to automate consistency tests between two different kinds of software description techniques using PROLOG. Especially, we focus on hierarchical state transition diagrams and interaction sequences both of which specify aspects of a system’s dynamics. In contrast to the similar graphical representations of UML our description techniques come with clear semantics which serve as the basis for all consistency tests.

After a brief discussion of the involved description techniques, we present a specification for an example application. Then, we evolve an PROLOG algorithm to automatically test an actual implementation as given by a state automaton to corresponding requirements as opposed by a given set of interaction sequences.

*Keywords*— Testing, Specification, Behavior, Interaction, Description Technique,

## I. INTRODUCTION

The development of software systems is a difficult and error prone. This is even true for small or medium-sized system which base on complex algorithms, a plethora of data structures, or obfuscated patterns of interaction.

Therefore, within practical software development, graphical description techniques are frequently used to introduce abstraction and thus reduce complexity. However, even worldwide standards like the UML [10] lack a clear understanding of relationships between the different views upon a system’s model (cf. [1]).

For this reason, formal methods allow to define precise semantics for these description techniques. Beyond that, they allow us to define relations between the different views

This work was supported by the German Information Security Agency (BSI) within the project Quest as well as by Siemens within the project FORSOFT

on the system. These relations are the foundation for methods and tools that check the consistency among the views.

There are different kinds of consistency, ranging from simple definedness requirements via conditions within one document to conditions between several documents. These consistency conditions can be divided into static properties like type correctness and dynamic properties concerning the behavior of the components. While static properties can easily be checked by corresponding tools, like, for instance, type checkers, dynamic criteria still need user interaction to be checked.

However, tool support is a decisive argument for the practical usage of a method, especially in the field of formal methods which require proper tools to help the developer to ensure the correctness within the development process. Most industrial developers prefer the formal model checking method, since it offers very good support for the verification of crucial properties.

In this paper, we describe a Prolog program that checks some dynamic consistency conditions automatically. Prolog [6] is a declarative programming language that offers good support for representing arbitrary concepts. The foundation is a database of facts and rules which are used by the Prolog interpreter to find solutions to specific problems. The significant feature of Prolog is its backtracking mechanism which allows to handle different possible ways to a solution. If one alternative doesn’t work out it takes another one next time.

In Section II we demonstrate the description techniques for static as well as dynamic aspects of the system. We then present a general metamodel for these techniques in Section III and show how to translate diagrams to Prolog code. In Section IV, the algorithm is evolved and reasoned

about. Finally, we draw conclusions and talk about related work in the last section.

### A. A Coarse Development Process

In the following, we shortly describe our development process that combines practical software development with formal methods (see [3] for more details).

The first step in the development process is the specification of the system's boundary in some kind of blackbox view. In this phase, the developer also learns at which points the system interacts with its environment, for example to exchange data or to get instructions. These interaction points are modeled by channels which transport messages from one system to another. Corresponding to the direction of this flow of information we distinguish input and output channels. Moreover, channels are typed which restricts the kinds of messages that can be sent over it. Bidirectional channels can be easily modeled using both an input and an output channel.

Next, the view is changed from black-box to glass-box. This allows to identify the components of the system and to model in which way they communicate, again using channels between them. The relation between black-box and glass-box view is called *structural refinement*. Assuming that the contained components itself can be refined the system might be described in a hierarchy of components each of which possibly connected to other ones by channels. This is described in greater detail in the next section.

The third step is to model the interactions between the components and to deduce concrete test cases for the components. The interactions can be represented graphically by using the interaction description diagrams of Section II-C. The interactions are a first, mostly incomplete specification of the requirements of the system. For example, an interaction sequence could be used to specify in which way a person can receive money from an ATM machine although this system has a lot of more features. The essential point is, that these specifications describe only one of many possible interactions.

Next, the design of the system is developed. It should specify the behaviour of system completely. We use state transition diagrams (see Section II-B) to model the system design.

Before the system is developed according to the design-specification it should be ensured (to avoid design-specification errors, the most expensive errors) that the design meets the requirements. Therefore, it is necessary to *check* that the behavior of the components can produce the interactions captured in the requirements. For this purpose we define a Prolog representation of behavior and interactions in Section IV, that allows to verify whether the interactions fit to the behavior.

The rest of the development process depends on specific characteristics of the system. The critical components have to be implemented using formal methods and the correctness of the resulting program has to be proved. One way of doing this is to give an executable model in terms of a concrete state transition diagram and to prove that this be-

havior is a *behavioral refinement* of the design-specification. For small systems this can be proved by model checking techniques.

For components without highest quality requirements it is sufficient to develop the program manually from the design specification. A validation of the system can be used to test the (non-formally developed) components against the interaction descriptions of the requirements, or to generate test sequences from the design-specification, using specification-based test methods. One problem of most of these methods is to find concrete values for the generated interaction descriptions. This can also be done with our Prolog program.

Running the test is the last step in the process. If errors are found only the coding has to be changed, because the design-specification has been verified against the requirements using the Prolog program of Section IV.

This coarse development process (including the testing of non-formally developed components) is summarized in the following list of steps:

1. black-box,
2. structural refinement,
3. interactions between components,
4. concrete runs of single components (test cases),
5. design of behavior,
6. check design,
7. develop components (coding),
8. generate general test cases,
9. find input values, and
10. test.

This development process has to be refined, especially for a development using formal methods. However, this is not in the scope of this paper.

### B. Example: A Fax Machine

An elementary example helps us in the following to demonstrate description techniques and to show the translation to prolog code. Moreover, the corresponding development process as well as ongoing ideas are sketched using this example of a fax machine.

A fax machine is a component which takes a Fax image and a phone number, builds up a connection to the respective receiver, sends the data, disconnects, sends back a notification regarding the success of the transfer, and waits for the next job. There are many possible reasons for an error, like a wrong phone number or a receiving fax machine which is busy or an empty fax image. However, we just consider the busy line to simplify the example.

As it is a fax machine for the international market, it can be configured to use old-fashioned pulse dialing or the modern multi-frequency mode. This information is part of the fax machine's setup and not provided for each job.

## II. DESCRIPTION TECHNIQUES

In this section we present different graphical description techniques for the specification of distributed systems. Mainly, we propose so called *System Structure Diagrams* for the specification of the system's structure and

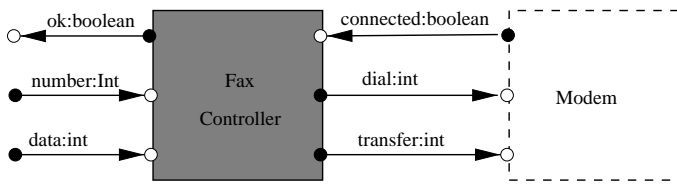


Fig. 1. SSD for the fax machine

*State Transition Diagrams* and *Interaction Description Diagrams* for the corresponding behavior. After a brief discussion about these rather well-known techniques, the next chapter introduces an according metamodel which will then allow us to translate the diagrams to an abstract syntax, and this way to a Prolog specification.

### A. System Structure Diagrams

A distributed system is a network of components, connected to each other in some way and communicating via so called channels. Therefore, the partners of all interactions are always components which are specified in *System Structure Diagrams* (SSD). In this static view of the system and its environment, rectangles represent components and directed lines visualize channels between them. Both of them are annotated with the corresponding label. Channels are typed and connected to components at special entry and exit points, so called *ports*. Exit ports and entry ports are visualized by filled and empty circles drawn on the outline ("the interface") of a component. As SSDs can be hierarchically refined, ports are also connected to the inside of a component. Accordingly, ports which are not related to a component are meant to be part of unspecified components which define the "outside world" and thus the component's border.

Figure 1 shows a SSD for the example of a fax machine. The system has already been refined into two subcomponents, the fax controller and the modem. The interface to the environment of the fax machine are the three channels on the right of the fax controller. They provide the fax image as well as the phone number to be dialed. The output channel is used for a confirmation if the fax was successfully transferred or an error message otherwise.

The other three channels are similar and for the communication with the modem. Again, the type of the channel sending the fax image is chosen to be `Int` for reasons of simplicity. The feedback channel is reflecting the state of the connection. The modem

component is drawn with a dashed line to emphasize the fact that it is of no further concern in this paper. The following sections thus focus on the fax controller and its behavior.

### B. State Transition Diagrams (STD)

The "behavior" of a component mainly affects the ongoing communication on its channels and is reflected in its state. Incoming channels are read, some computation is done possibly changing the component's state and new val-

ues are written on the output channels accordingly. To describe this activity, we propose *State Transition Diagrams* which define a set of *control states* of a component and *transitions* to specify exactly under which circumstances a component switches to another (or the same) state. To ease this kind of modeling, we introduce local variables which can be read and modified during a transition. The set of local variables define the *data state* of the component. Both, control state and data state together form the state of the component. Formally, a transition consists of a precondition, an input pattern, new output values and a postcondition. A transition can *fire* if the precondition holds and the pattern matches the values read from the input channels. If it does, the output values are written to the specified channels and the postcondition holds. Note that it does not have to fire as we allow non-deterministic automata where several transitions can fire in the same state. However, this semantics will come into effect in Prolog programs later on.

Similar to SSDs, state diagrams can be further refined in substates. Therefore, we define so called *interface points* (IP) to specify in which way a transition to a state is continued to one of its substates (or one of the substate's interface points, respectively). An IP belongs to exactly one state. As a state can thus be entered and left through several possible IPs it seems reasonable to define specific entry and exit actions which are executed accordingly.

A transition might be cut into several parts by interface points with each part being defined as mentioned above. Although composed from several parts, the overall transition is treated in the automaton as a single one. By defining entry and exit points we determine the direction in which an interface point can be passed and thus avoid unwanted combinations of transition segments. Each transition starts out from an atomic state which is not further refined, possibly leaves some of its superstates by passing their exit IPs thus executing their exit actions and then enters another hierarchy of states (executing entry actions respectively) until it reaches another atomic state. Note, that it is allowed to split a transition at an interface point which thus subsumizes a group of transitions. Figure 2 shows the STD of the fax controller with the state Connecting further refined.

The semantics of composed transitions can be expressed by a set of elementary transitions. For example, the transition segments `successP`, `successMF`, and `doTransfer` can be expressed by the transitions `successP/doTransfer` and `successMF/doTransfer`.

The semantics of such derived transitions is determined by a combination of the attributes of all participating transition segments and interface points. The pre- resp. postcondition is the conjugation of the pre- resp. postconditions of all participating transition segments. Input and output statements are simply concatenated. The action consists of the sequential composition of the transition segment's actions and the exit and entry actions of the IPs. The semantics of `successP/doTransfer` is equivalent to an elementary transition with input "connected?true" and

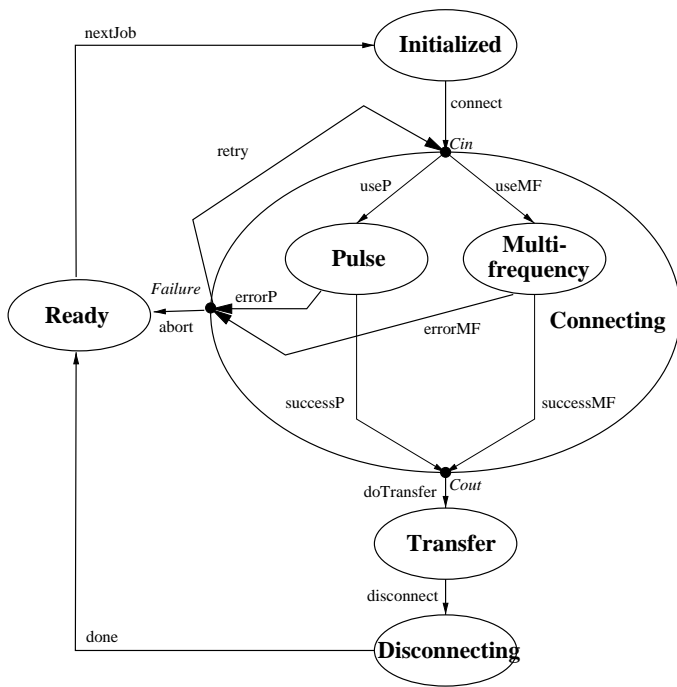


Fig. 2. STD for the fax controller

output “transfer!fax”, precondition “true”, postcondition “true”, and action “skip”.

As can be seen in Figure 2, the fax controller switches between four states one of which is further refined in three substates. The substates of the state *Connecting* work on establishing a connection to the receiving host. After choosing the proper dialing mode which is actually part of the fax controller’s configuration, the number and the dial command are sent to the modem. If an error occurs during this procedure, the controller waits some time and retries once before it states the failure of this job. As definitions of transition segments tend to be rather cumbersome, the segments of Figure 2 are annotated with labels which refer to Table 3.

In general, the behavior of a system is defined by the cooperation between those automata which are associated to the system’s atomic components (subcomponents, which are not further refined). The behavior of the example system is thus defined by the automaton in Figure 2 as well as the behavior of the modem which is ignored for a reduced complexity.

### C. Interaction Description Diagram

Interaction Description Diagrams (IDDs) are used to describe the interaction of components. They are similar to Message Sequence Charts [12]. In contrast to Message Sequence Charts, IDD refers to time-synchronous systems. The progress of time is explicitly modeled by ticks, which are represented by dashed lines. All actions between two successive ticks are considered to occur simultaneously, i.e. the order of these actions is meaningless. An action in an IDD describes a message that is sent via a channel from

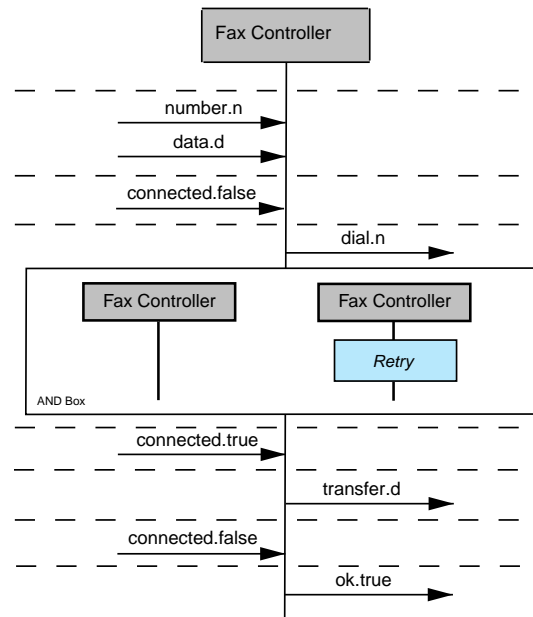


Fig. 4. IDD for successful transmission

one component to another component. This is denoted by a horizontal arrow from the source component to the destination component. The message is annotated with the channel and the contents of the message separated by a dot.

Note that it does not take any time to transfer the message, but time is consumed on the ticks, when the computations of all components are performed synchronously. As a consequence, the output values cannot depend on the input values of the same time slice. The component always needs a tick for the computation of new output values.

In contrast to sequence diagrams which only specify a single interaction, IDDs can describe groups of possible interactions. On one hand, the content of a message might be a global variable restricted by predicates. On the other hand, variants of an IDD can be grouped together in boxes. Thus, *AND-boxes* denote the group of interactions which contain all variants whereas *OR-boxes* demand to choose a single variant. The latter type is useful for underspecification, i.e. to leave some details open. Boxes with only a single sub-IDD are used for hierarchical structuring. For such boxes, it is not necessary to assign them as AND or OR-boxes, since there is no difference.

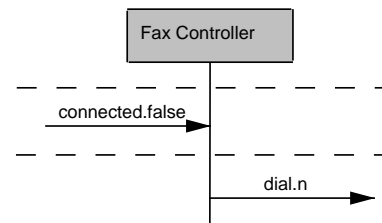


Fig. 5. Retry to establish connection

label	from	to	in	out	pre	post	action
nextJob	<b>Ready</b>	<b>Initialized</b>	number?n,data?d	—	n, d defined	—	number=n fax=d
connect	<b>Initialized</b>	<b>Connect.Cin</b>	connected?false	—	—	—	—
useP	<b>Connect.Cin</b>	<b>Pulse</b>	—	dial!number	mode = p	—	—
useMF	<b>Connect.Cin</b>	<b>Multif.</b>	—	dial!number	mode = mf	—	—
successP	<b>Pulse</b>	<b>Connect.Cout</b>	connected?true	—	—	—	—
successMF	<b>Multifrequency</b>	<b>Connect.Cout</b>	connected?true	—	—	—	—
doTransfer	<b>Connect.Cout</b>	<b>Transfer</b>	—	transfer!fax	—	—	—
disconnect	<b>Transfer</b>	<b>Disconnecting</b>	—	—	—	—	—
done	<b>Disconnecting</b>	<b>Ready</b>	connected?false	ok!true	—	—	—
errorP	<b>Pulse</b>	<b>Connect.Failure</b>	connected?false	—	—	—	—
errorMF	<b>Multifrequency</b>	<b>Connect.Failure</b>	connected?false	—	—	—	—
retry	<b>Connect.Failure</b>	<b>Connect.Cin</b>	—	—	—	—	—
abort	<b>Connect.Failure</b>	<b>Ready</b>	—	ok!false	—	—	—

Fig. 3. Transition Segments

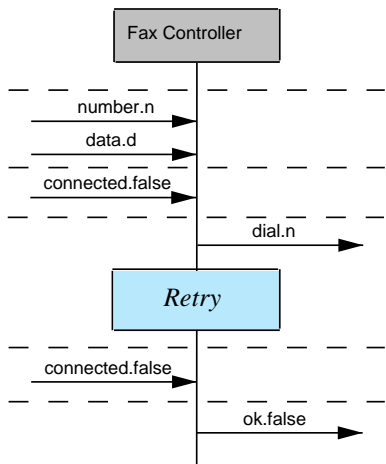


Fig. 6. Unsuccessful transmission

Figures 4 to 6 show some example interaction specifications for the fax controller. A successful fax transmission is described in Figure 4 using the box defined in Figure 5. In this example, we use an AND-box to denote that the system must be able to perform a second dial attempt, if the first attempt fails to connect to the remote fax machine. Figure 6 shows the IDD for an unsuccessful fax transmission. The overall behavior of the Fax machine is described by an AND-box that contains the IDD for a successful and an unsuccessful transmission.

In this paper, we restrict the IDD to one component with its environment and we do not use iterative constructs.

### III. METAMODEL AND REPRESENTATION

The last section introduced the description techniques for the specification of a system's desired behavior (IDDs) as well as an actual implementation (STDs). In order to write a prolog program which checks if the implemented behavior is consistent with the desired one, we present a translation of the description techniques to Prolog based on an appropriate metamodel.

#### A. Metamodel

A couple of simplifications will reduce the complexity of the description techniques and thus enhance readability and understanding of the presented concepts. This way, we focus on testing the consistency between the descriptions of a single test component, which is not further refined. Figure 7 shows the corresponding metamodel using the syntax of UML's class diagrams (see [2]). The test component's interface is determined by its related ports — channels can be omitted.

STDs can be hierarchically refined with each state possibly consisting of a couple of interface points and transition segments, each connecting two interface points. Similarly straightforward is the definition of IDD as a sequence of messages, ticks, and boxes, which themselves contain IDD.

Note that the metamodel contains features not used in the specification of the fax machine, for example, the restriction of a message's variables by corresponding precondition. Also, the metamodel is based on types, values, and terms of Prolog. Actions, predicates, and patterns are also modeled in Prolog and are referred to as Strings (*AssignString*, *PredicateString*, *PatternString*).

There are a lot of consistency criteria for the metamodel, for example the ports referred in the messages belong to the component, the types match, every input and output port is used at most once in the composed transitions, no cyclic transitions, etc. As these checks can be easily automated and thus it is reasonable in this context to assume models consistent, regarding the above criteria before translating them to Prolog code.

#### B. Translation Basics

General translation schemes presented in this chapter form the foundation for the translation of a model in Prolog syntax in the following. Starting out from all diagrams which — using the description techniques presented above — specify the test component as well as the desired behavior, an actual model can be easily obtained from the metamodel. Similar, the translation of this model to Prolog is easy if an appropriate representation of the metamodel in Prolog is given.

The constituents of the metamodel are the main parts of



is one big state transition diagram without substates.

2. **Generation:** In this step a set of *runs* is generated from the sequence charts. This is done by replacing AND- and OR-boxes by the building unions and choices from the sets of the included sequences. Furthermore generation can include the generation of concrete input values (if this is required in the transitions). The result is a (possibly infinite) set of runs that can be tested against the transition machine from the first step.
3. **Execution:** the last step is to test whether the runs can be executed. If all runs can be executed the sequence diagram fits to the state transition diagram, i.e. the specifications are consistent.

The backtracking principle of Prolog allows to interleave the second with the third step, such that not all runs have to be generated. Beside the gained efficiency (compared to conventional testing) this allows for testing of infinitely many runs (for example if a simple identity sequences: input  $x$  of type float and output  $x$  is tested against a transition with the pattern *input?x; output!x*).

Within the next sections we describe the steps more detailed, an example is treated in Section IV-E.

### B. Flattening Hierarchical State Transition Diagrams

The hierarchy within the State Transition Diagrams is only a syntactic means to structure large transition diagrams. The semantics that we use is flat. Therefore in a preprocessing step we flatten the State Transition Diagrams.

States with substates are replaced by substates. Transition segments between states and substates are concatenated to transitions that include all actions from the parts.

In the example of the fax machine the state **Connect** is replaced by the substates **ConnectPulse** and **ConnectMultifrequency**. The transition segments *connect* and *useP* (see table 3) are connected via the interface point *Cin* to a transition (see table 8) The flattening does not require backtracking and is therefore realized during the generation of prolog code for the model. The translation from the transition diagrams to Prolog is rather straightforward according to the guidelines from the last chapter.

### C. Generation of Test Runs

In order to simplify the evaluation of the IDD, we must rewrite the term representing it in a set of logical combinations of *simple* testsequences. We say a testsequence is simple if it does not contain any AND/OR-boxes. This rewriting preserves the sequence of actions, particularly that of messages and ticks.

As mentioned earlier, sequence charts specify a group of possible interactions. As the actual value of a message might be a variable, the described behavior might also apply to a set of values, possibly restricted by additional conditions. A path through a sequence chart with a chosen variant for each box (in which more than one variant is offered) is called a *testsequence*. Moreover, a *run* is a testsequence together with values for all free variables for which

the corresponding conditions hold. A testsequence is called successful if there is at least a single run for which the given state automaton interacts according to this testsequence. If there exist AND-boxes within the IDD we define testsequences *adjacent* to a given testsequence (at a given AND-box) if they are the same in the preceding interaction but differ in the chosen variant. Finally, the *overall test* for the combination of a state transition diagram and a sequence chart is called successful if there is at least one successful testsequence and for all passed AND-boxes all respective adjacent testsequences are also successful.

Types might specify an infinite set of values which might result in an infinite set of runs for a testsequence. If the conditions hold for all messages and the actual values, a valid run for the current testsequence has been found. Otherwise the algorithm will not terminate because it will generate new test runs and then fail to execute them. Therefore we introduced bounds for all types that restrict the generation of the values for that type. If a bound is reached a warning is given out, such that the user know that the executed test has been performed partially.

We chose a different way to translate sequences to Prolog. As the basic idea is that of an ordered list of actions, it seems natural to use infix operators to create a term as a representation of a sequence. The different kinds of actions are reflected in different kinds of infix operators.

### D. Symbolic Execution of Test Runs

In this section we describe how the test runs resulting from the above generation for sequence charts, are matched against the state transition machines, flattened in the first step of our algorithm.

For a simple testsequence the Prolog program simulates the given automaton using different runs of this testsequence until a run is found for which the automaton behaves as specified in the testsequence. This implies that the algorithm finds all possible runs, hence all allowed values for variables in messages. The type of the variable is that of the corresponding channel and in the Prolog program simply seen as an additional condition for this variable.

The simulation starts at the initial state of the automaton and tries to match all actions between two ticks to the input and output pattern of a transition. If the match succeeds the simulation continues in the next state, otherwise backtracking searches other transitions that can be executed. If the test runs is finished testing was successfully, i.e. the program has found a path through the automaton that process the specified interactions (input and output messages).

### E. Example of Consistency Test

In this section we illuminate the dynamic behaviour of the algorithm by means of a small part of our example. The results of the evaluation of the algorithm are described within the following section.

Suppose we executed a test run, such that our state has mode Multifrequency, i.e. the variable *mode* is equal to *mf*.

from	to	in	out	pre	post	action
<b>Initialized</b>	<b>Connect</b>	<i>Cin</i>	connected?false	mode = p	—	—

Fig. 8. Flat Transition

Suppose that we want to execute the first two transitions actions that lead to the sequence diagram in Figure 6.

The program starts from the Ready state, executes the transition `nextJob` and arrives at state `Initialized`. At this stage, the program fires the transition segment `Connect` from `Initialized` to `Connect.Cin`. But the state `Connect` is not a final state, then the program must fire another transition segment, in our case `useP` to reach some atomic state. The program tries to satisfy the transition segment precondition (`mode = p`), but fails because `mode` is equal to `mf`. At this moment, the program backtracks until `Cin` and tries other possible transition segment, `useMF` in our case, that satisfies the precondition.

This is an example of backtracking. It is possible, of course, a case in which the program backtracks deeper, not only one transition segment, but more than one.

## V. CONCLUSION AND RELATED WORK

We used Prolog for the consistency test between the behaviour of a system, specified by hierarchic and non deterministic automata and the interactions describing the requirements for the specification of the test.

The backtracking mechanism of Prolog was used for the following purposes:

- selection of appropriate input data values
- selection of alternatives within the non deterministic design specification
- selection and retry among different test sequences

The precise semantics of the component described by the automata using input and output values allows us to check consistency of the design against the interaction requirements.

AutoFocus [8] uses similar description techniques, however consistency checking possibilities are until now restricted to static aspects of the modeled systems. And the notion of hierarchy in the automata of AUTOFOCUS is only syntactical, whereas in our work the transitions are composed from transition segments between different levels of abstraction This allow us a more adequate modeling of the behaviour of complex systems.

UML [2] as well has different description techniques as presented here. However, since the semantics are not defined within one formal model, the (static and dynamic) relations between the description techniques is not feasible.

Another, more formal, field of related work is model checking. Model Checking [5], [4] is an efficient technique for a complete searching algorithm through the state space of behavioural descriptions, and the model checker can verify properties formulated using temporal logic. Interaction descriptions can be expressed as temporal logic properties [7]. However, model checking can only be applied to systems with a finite and small state space. In our example the

input value is a natural number. Therefore model checking cannot be applied to it.

Since model checkers first construct the reachable state space and then start to check the property, the algorithm is not goal-directed. Our Prolog program uses the IDD as guide through the possible behaviours of the system, and the presented algorithm is therefore better suited for testing those relatively constructive properties.

The consistency between SDL [9] processes and MSCs [12], can be guaranteed by generating the MSC as a trace from the simulation. Some SDL tools can perform a complete search, however, the current tendency is to link model checkers to SDL tools.

Our future work is to integrate the hierarchical STDs and the IDDs into the tool AUTOFOCUS, and to extend the prolog program, in order to handle systems with several components. Another extension is to use a similar program to check whether an abstract and a concrete STD are compliant to each other. This can be used to verify some refinement steps during the development process.

## ACKNOWLEDGMENTS

For helpful comments on previous versions of this paper we thank Bartomeu Serra, Albert Llamós, Klaus Bergner and Franz Huber.

## REFERENCES

- [1] K. Bergner, A. Rausch, and M. Sihling. Using UML for modeling a distributed Java application. Technical Report TUM-19735, Technische Universität München, Institut für Informatik, 1997.
- [2] G. Booch, I. Jacobson, and J. Rumbaugh. *UML Summary*. Rational Software Cooperation, Jan. 1997. Version 1.0.
- [3] M. Broy and O. Slotosch. Enriching the Software Development Process by Formal Methods. In *International Workshop on Current Trends in Applied Formal Methods (FM-TRENDS 98)*, Okt 1998. to appear.
- [4] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking:  $10^{20}$  States and Beyond. *Information and Computation*, 98(2):142–170, June 1992.
- [5] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.
- [6] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer, Heidelberg, 2 edition, 1984.
- [7] M. Haubner. Transformation von MSCs in temporallogische Formeln, 1997.
- [8] F. Huber, S. Molterer, B. Schätz, O. Slotosch, and A. Vilbig. Traffic Lights - An AutoFocus Case Study. In *1998 International Conference on Application of Concurrency to System Design*, pages 282–294. IEEE Computer Society, 1998.
- [9] ITU-T. *Recommendation Z.100, Specification and Description Language (SDL)*. ITU, 1993.
- [10] OMG. *OMG Unified Modeling Language Specification*. Version 1.3, Object Management Group, 1999.
- [11] J. Rigo. A prolog program to test consistency between behavior and interaction, <http://www4.informatik.tu-muenchen.de/~sihling/program.pr>, 1998.
- [12] Z.120. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC)*. ITU-TS, Geneva, 1996.