2nd workshop Rule-Based Constraint Reasoning and Programming, Singapore, 9/00

Testing Concurrent Reactive Systems with Constraint Logic Programming*

Heiko Lötzbeyer and Alexander Pretschner
Institut für Informatik, Technische Universität München
Arcisstraße 21, 80290 München, Germany
www4.in.tum.de/~{loetzbey,pretschn}

Abstract

In the domain of reactive systems, work on validation has mainly focused on verifying *specifications*. A complementary problem consists of validating an *implementation* w.r.t. its specification (testing). We propose a way to compute test sequences on the basis of system as well as test case specifications that is based on Constraint Logic Programming. Furthermore, we provide a clear terminology for concepts within the testing context.
Keywords. Debugging, Validation, Testing, Automatic Test Case Generation, Simulation, CASE, Constraint Handling Rules, Message Sequence Charts

1 Introduction

Considerable effort has been devoted to the specification and verification of reactive systems. While in terms of specification, results are already used in practice (CASE tools), there are still severe shortcomings of verification techniques. To mention a few, these include problems with state space explosion, usually restriction to finite state spaces, not intuitive and hence impractical formalisms, and they mostly aim at verifying properties of a system specification. However, the latter point may resolve just one part of the overall problem since if the specification is assumed to be "correct" or "consistent", this does not necessarily mean the actual implementation also is.

Model checking enables one to prove usually highly general properties or invariants. Because of the properties' generality, there is not much information available, and model checking is usually done in two steps: After generating the model, specified properties are checked. If, on the other hand, properties are more specialized, it is possible to interleave the model's generation with the verification of these properties. This is the case when one wants to *test* an implementation; the properties in question then describe, for instance, traces or transition sequences: Such properties may constrain possible system traces in a significant way, and if these properties are sufficiently specific, they can be used in building a model of manageable size. Furthermore, testing usually – and also in this paper – focuses on finite (or even short) system traces. (Possibly bounded) Model checking and testing are hence different techniques with different purposes, and it is natural to implement these different techniques by different means.

*This work was in part supported with funds of the Deutsche Forschungsgemeinschaft under reference number Be-1055/7-2 within the priority program KONDISK.

In this paper, we lay the foundations for testing executable system specifications (system models) as well as implementations w.r.t. their specification. In our view, this turns out to be a generalization of the simulation of reactive systems. We sketch earlier work on how system models formulated within the CASE tool AutoFocus can naturally be translated into Constraint Logic Programming (CLP) languages. This translation scheme can be fully automated and is fully compositional with an interleaving model of concurrency. With our approach, one may use the same basis for test case generation and test execution. This paper shows how test cases (i.e., I/O traces) can be computed efficiently.

We consider the heart of the testing process of reactive systems to consist of *test case specifications*, i.e., formal descriptions of specific test purposes. Examples include "some specific output will be written by the implementation" which amounts to black box testing of the implementation, or "some transition sequence will be executed" which amounts to white box testing of the specification (system model). *Test cases* are artifacts that satisfy a given test case specification, and they may be possibly partial traces, constraints over traces, transition sequences, etc. (e.g., remaining delayed constraints from the constraint store). Test cases can then either directly be fed into a simulation, or by using techniques such as interval/limit analyses or other *selection hypotheses* [11], be refined into actual *test sequences* for the implementation. This is done in order to decrease the usually enormous amount of test cases, a yet unsolved problem. We believe that CLP with its mechanisms of a priori pruning the search tree is particularly suited for effectively and efficiently solving it. Test case specifications can be formulated by means of Message Sequence Charts (MSCs [19]), and when implemented in CLP, they considerably reduce the search tree's dimensions, thus alleviating the problems of state space explosion.

Overview. The remainder of this paper is organized as follows. Section 2 presents the CASE tool AutoFocus into which we integrated our CLP based system for simulation and test case generation. It is then sketched in section 3 how system models in AutoFocus can automatically be translated into CLP by maintaining compositionality. This is the basis for our approach to the generation of test cases. Section 4 then demonstrates our approach with a simple timer example. The example is taken from [22]. In section 5 we first give a few definitions of terms related to testing and show how the example of section 4 can be tested. The use of Sequence Charts within the testing context is treated in section 6. Finally, in section 7 we draw some conclusions, discuss advantages and shortcomings of our approach, and we present current as well as future work.

Related work Code Generation from and validation of reactive systems on the basis of CLP has been discussed recently. Delzanno and Podelski [8] focus on general Labeled Transition Systems. The motivation for their translation is model checking. Ciarlini and Frühwirth [4] focus on Hybrid Automata [1] that allow for continuous activities within discrete states. So does Urbina [29]; his work aims at verification techniques on the basis of CLP. In principle, hybrid automata could also be used for modeling purely discrete systems: Gupta and Pontelli [14], for instance, provide a translation of timed automata into CLP. Timed automata form a proper subset of hybrid automata. One of the major drawbacks of both types of automata is, however, that they are not well suited for modular design [24]. Fribourg and Veloso-Peixoto [10] give another approach to the generation of CLP code from concurrent automata similar to Labeled Transition Systems with explicit constraints. All approaches share the commonality of synchronizing components via variables rather than explicit channels (modularity), and of not being supported by tools. Fribourg [9] reviews previous work on the relationship between CLP and

Model Checking.

[30] present a different approach to generating test sequences on the basis of propositional logics. Systems and properties (test cases) are translated into propositional logics and checked for satisfiability; the results are translated back into traces. In terms of its intention, this approach is similar to the one presented in this paper. However, it inherently is restricted to finite system.

Finally, there is a wealth of literature on test case generation; [30, 23] contain some relevant references.

2 Modeling with AutoFocus

AutoFocus [16, 17] is a tool for graphically specifying embedded systems. It supports different views on the system model: structure, behavior, interaction, and data type view. Each view concentrates on a fixed part of the specified model.

Structural view: SSDs. In AutoFocus, a distributed system is a network of components, possibly connected one to the other, and communicating via so-called channels. The partners of all interactions are always components which are specified in *System Structure Diagrams* (SSD). Figure 1 (a) shows a typical SSD. The corresponding system is discussed in section 4. In this static view of the system and its environment, rectangles represent components and directed lines visualize channels between them. Both of them are named with a label. Channels are typed and directed, and they are connected to components at special entry and exit points, so called *ports*. Ports are visualized by filled and empty circles drawn on the outline (the *interface*) of a component. As SSDs can be hierarchically refined, ports may be 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 interface to its environment.

Behavioral view: STDs. The *behavior* of an AutoFocus component is described by a *State Transition Diagram* (STD). Figure 1 (b) through (d) show typical STDs. Initial states are marked with a black dot. An STD consists of a set of *control states*, *transitions* and *local variables*. The set of local variables builds the automaton's *data state*. Hence, the internal state of a component consists of the automaton's control as well as its data state. A transition can be complemented with several annotations: a label, a precondition, input statements, output statements and a postcondition. The precondition is a boolean expression that can refer to local variables and transition variables. Transition variables are bound by input statements, and their life-cycle is restricted to one execution of the transition. Input statements consist of a channel name followed by a question mark and a pattern. An output statement is a channel name and an expression separated by an exclamation mark. The expression on the output statement can refer to both local variables as well as transition variables. A transition can *fire* if the precondition holds and the pattern on the input statements match the values read from the input. After execution of the transition the values in the output statements are copied to the appropriate ports and the local variables are set according to the postcondition. Actually the postcondition consists of a set of actions that assign new values to the local variables, i.e. the assignments set the automaton's new data state.

Communication semantics. AutoFocus components have a common global clock, i.e., they all perform their computations simultaneously. The cycle of a composed system consists of two steps: First each component reads the values on the input ports and computes new values for local variables and output ports.

After the time tick, the new values are copied to the output ports where they can be accessed immediately via the input ports of connected components and the cycle is repeated. This results in a *time synchronous* communication scheme with buffer size 1.

Interaction view: MSCs. Message Sequence Charts (MSCs) are used to describe the interaction of components. In contrast to Message Sequence Charts as defined in [19], AUTOFOCUS MSCs refer 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 MSC describes a message that is sent via a channel from one component to another. This is denoted by a horizontal arrow from the source to the destination component. Internal messages between two components are annotated with the channel and the contents of the message separated by a dot. Annotations on external messages do not refer to channels but rather to external ports of the component. To illustrate this fact, the port name and the message value are delimited by ! (send) or ? (receive) in analogy to transition annotations in STDs.

Note that it does not take any time to transfer messages, 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 and each component always needs a tick for the computation of new output values.

Additional information about the internal state of one or more components can be given by means of *conditions*. Conditions are written in hexagons on the related components' axes and can refer to both the control and the data state of the component. Furthermore, MSCs can be structured with boxes. A box can contain one or several sub-MSCs, or indicate that a no further described actions can occur. Figures 4 and 5 show typical examples of exemplary system runs. The conditions in Figure 4 refer to the control state of the blinker component.

Datatype view: DTDs. For the specification of user defined data types and functions AUTOFOCUS provides DTDs. The definitions in DTDs are written in a Gofer-like syntax.

Even though different views are mainly orthogonal, there is a natural portion of overlapping that may result in inconsistent specifications. For detecting inconsistencies between different views, the tool has some built-in user-definable consistency checks that work on a syntactical basis [18].

3 Translation into CLP

This section shortly sketches how CLP code can be generated automatically from AUTOFOCUS models. With the exception of the `doStep` predicate that actually performs steps in the system, the translation is the same as the translation described in detail in [22].

We first focus on the translation of a single component. Basically, transitions are encoded as predicates. For each component we define a predicate `step_component` which simulates a single step of the component. Two of its arguments hence serve as source and destination states. Furthermore, we associate two arguments with each local variable: they contain its values before and after the transition has fired. Whether or not a transition fires depends on the preconditions that are associated with each transition. Preconditions can easily be translated into constraints for they actually are constraints. Postconditions in AUTOFOCUS, on the other hand,

generally just assign new values to local variables and write new values on channels. Channels connected to a component, or its ports, respectively, are the last missing piece: we associate one variable with each input and one variable with each output port. Variables for input ports contain the values of input channels before the transition fires (if it does), and variables for output ports contain new values on the output channels after the transition has fired.

Function definitions and types. AUTOFOCUS allows for user-defined possibly recursive functions that can be used as transition code. Since the language is a functional one with an eager semantics (lazy *if* included), our current system translates these function definitions into CLP by flattening. It turned out, however, that for the determination of test sequences this is not enough; we thus currently assess the benefits of implementing the language lazily on the basis of another operational semantics, namely Narrowing as used in the functional logic language Curry [15]. Curry also integrates constraint solvers which makes it a particularly promising candidate.

There is room for discussion on how much functionality should be put in transition code (and thus, how intricate defined functions should be). For smaller systems, it seems to be a good idea to encode almost all functionality within the state space structure of the system whereas for larger ones, one should not refrain from putting quite some functionality into the transitions - simply because the system would become even larger without this encoding. A similar question is concerned with the decision of encoding a state by a variable or an actual state (a circle in the AUTOFOCUS behavioral view).

Concerning types, a general translation on the basis of Constraint Handling Rules has been proposed in [22].

Parallel composition. The parallel composition of components turns out to be particularly simple. This is due to the time-synchronicity inherent to the semantics of AUTOFOCUS. The idea is to encode the channel between two ports of two different components by a new local variable - a local variable of the composed system. This simple scheme (see [22] for details) also applies to the composition of more than two automata and defines a step predicate for the composed system. Note that a step predicate for a composed system has the same structure as a step predicate for a single component. Parallel composition is thus fully compositional.

Execution. The step predicate of any component (both single components and composed components) relates one configuration of the system (i.e. actual values of local variables, control state and channels) to the succeeding configuration after performing a single step. For a whole system run, the step predicates for the components in the composed system must be called subsequently. This is done by the `doStep` predicate (Fig. 1). The `doStep` predicate takes a list for each local variable and channel as well as a list to store the fired transitions and the control state. The heads of the lists which contain the values of the local variables, input channels and the control state, build up the actual configuration of the system. The second element is supposed to be the next configuration. The actual parameters are always open lists of the form `[E|_]`; the tail will be unified with another list of the form `[F|_]` (the formal parameter) if computation proceeds, or with `[stop]` if the computation stops. This implementation allows one to implement constraints on a list that grows during execution - without changing its name that is inherently tied to the constraint.

The lists for output channels and the fired transitions are not used for guiding the unconstrained simulation but rather included for observing the simulation. By

calling the step predicate, the next configuration will be computed and the fired transition with its output values determined. Hereafter, the `doStep` predicate will be called with the tails of all lists. When the simulation stops, all lists are terminated with a special "stop" item which denotes the end of the simulation.

The simulation is started by calling the `doStep` predicate with a set of open lists (i.e. lists with unspecified tails) that are only constrained by some initial values (initial values for local variables, first input values and the initial control state). We call these lists the "history lists". Then the `doStep` predicate calls the step predicates and further constraints the history lists. This can be repeated any times until a special rule of the `doStep` predicate inserts the `stop` item, terminates all lists and stops the simulation.

Constrained simulation. In order to obtain system runs with certain properties the simulation has to be restricted in some way. We can do this by explicitly specifying additional constraints over the initially unconstrained history lists on which the simulation runs. At first it is not possible to determine whether the constraints hold or not as no element of the history list is instantiated and the constraints are pushed into the constraint store. In order to start the simulation, the initial values have to be put in the lists. This is done by instantiating the first elements of the lists for local variables, input channels and the control state. By instantiating any element in a list on which a delayed goal exists (i.e. a goal which was pushed into the constraint store), the delayed goal will be woken and thus the CLP-system tries to resolve the delayed constraint. Therefore the assignment of the initial values of the system can already force some constraints to fail so that no conforming system trace is possible. If no constraint fails, simulation can start. In each step of the simulation more variables in the history lists are instantiated, the corresponding delayed goals woken, and, if the goal fails, the system must backtrack.

Constraints. We have defined some example constraints on lists via CHR. Probably the most important and simple constraint is the membership relation. We define

```
cmem(X,L) <=> nbMember(X,L)|true.
cmem(X,L) <=> nbMember(stop,L)| fail. % this rule will only be taken
                                      % when first rule fails
```

where `nbMember` is a member predicate that does not bind any variables. As soon as one element in list `L` is instantiated with the searched element, the goal succeeds and the constraint is eliminated. The occurrence of the stop item in the list denotes the termination of the simulation and the constraint fails as the desired element has not been found yet. In this case the system is forced to backtrack and instantiate the history lists' variables with other values. Otherwise, nothing can be derived, and the goals remain in the constraint store.

Efficient constraint solving. The above definition of the `cmem` constraint has one major drawback. Each time the constraint is woken, all instantiated items of the list are checked against the searched item. Therefore, if the system performs n steps, the checking of the constraint requires $O(n^2)$ comparisons. This can be done more efficiently. The elements of each history list get instantiated from the head of the list towards the end. Variables which are instantiated once will never be changed in the future except when backtracking. Therefore we can check the instantiated prefix of the list and reduce the constraint to the uninstantiated rest. For the `cmem` predicate we define

```
fail_cmem      @ cmem(X,[H|T]) <=> nonvar(X), nonvar(H), H=stop| fail.
transform_cmem @ cmem(X,[H|T]) <=> nonvar(H), nonvar(X), H \= X|cmem(X,T).
succeed_cmem   @ cmem(X,[H|T]) <=> nonvar(X), nonvar(H), X = H|true.
```

If the head of the constrained list is instantiated and not equal to the searched element, Rule (`transform_cmem`) transforms the `cmem` constraint on the whole list to a constraint on the tail of the given list. Rule (`succeed_cmem`) eliminates the constraint whenever the searched element is found and Rule (`fail_cmem`) lets the constraint fail when the stop item is reached. In general we can classify the rules in three categories: rules for the failure of a constraint (e.g. `fail_cmem`), rules for the transformation of a constraint (e.g. `transform_cmem`), and rules for the elimination of a constraint (e.g. `succeed_cmem`). Another example is the `xBeforeY` constraint. `xBeforeY` requires the list to contain an element `Y` which requires the precedence of an element `X`. The definition of an efficient constraint solver is as follows:

```
xBeforeY(X,Y,[H|T]) <=> nonvar(X), nonvar(Y), nonvar(H), H=X| cmem(Y,T).
xBeforeY(X,Y,[H|T]) <=> nonvar(X), nonvar(Y), nonvar(H), H=Y| fail.
xBeforeY(X,Y,[H|T]) <=> nonvar(X), nonvar(Y), nonvar(H), H=stop| fail.
xBeforeY(X,Y,[H|T]) <=> nonvar(X), nonvar(Y), nonvar(H)| xBeforeY(X,Y,T).
```

4 Example

Figure 1 shows the structure as well as the behavior of a simple timer example: a

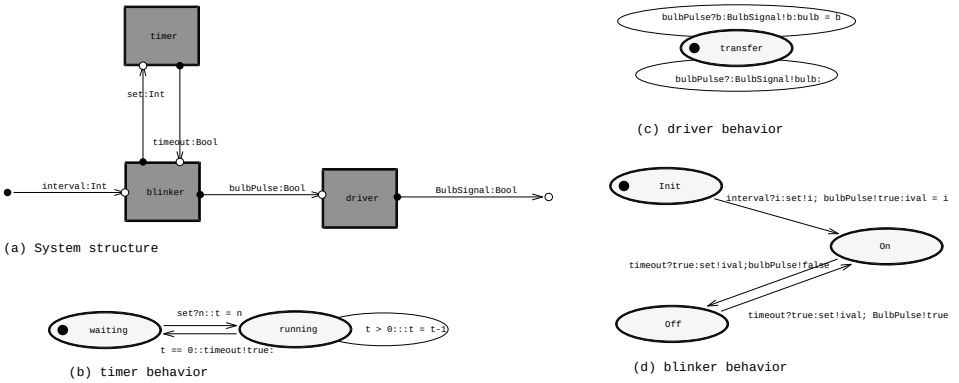

Figure 1: The timer-blinker example

system consisting of three components, a timer, a blinker, and a driver. Figure 1 (a) describes the system structure. It shows how the following channels are connected: The *set* and *timeout* channels of the timer and the blinker, and the driver's as well as the blinker's *bulbPulse* channels. Part (b) of the figure shows the timer's behavior: If the component receives a value on the *set* channel, it sets its internal variable t to this value, and decrements it until zero is reached. The timer then waits for the next value on the *set* channel. Part (c) shows the driver's behavior. At every tick, the driver outputs the last value received on its input channel. Finally, part (d) is a description of the blinker's behavior. Once it receives a value on its input channel *Interval*, the internal variable *ival* is set to this value, and the very same value is written to the timer. Whenever the timer decrements this value to zero, the bulb's status is toggled, and the timer is reset.

In the composed system, the timer is hence initialized with the value at channel *Interval*. The timer then ticks until it reaches a timeout, and whenever the timeout

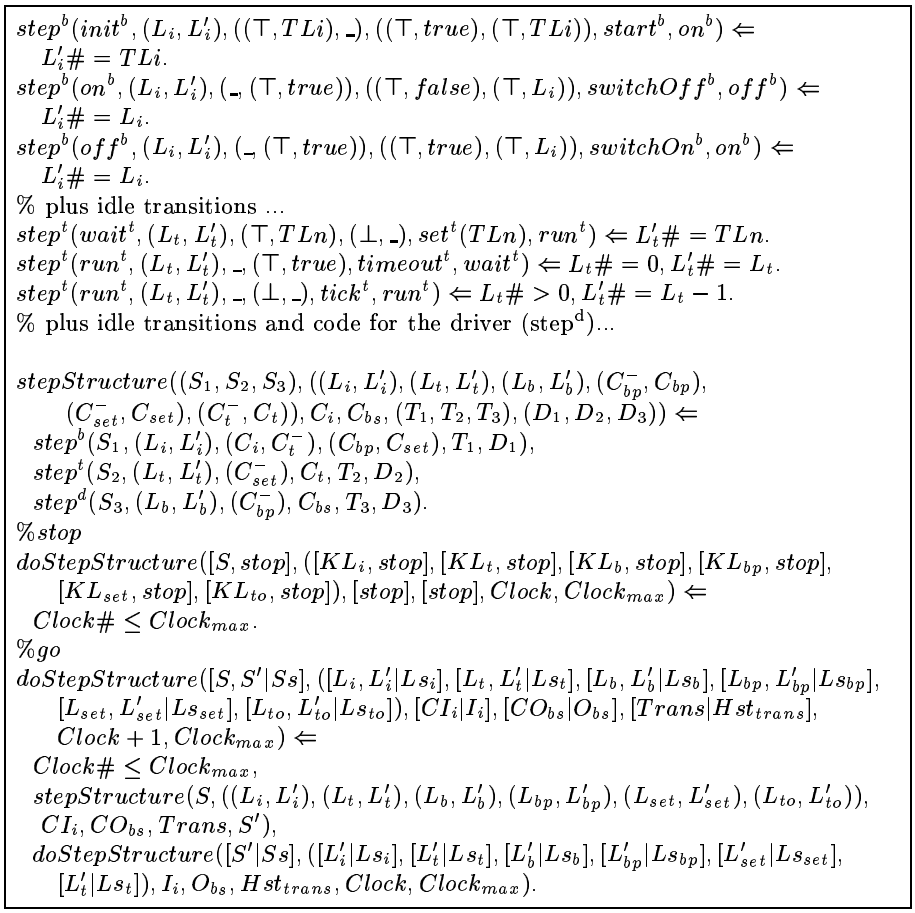

Figure 2: Timer/blinker code. $\bot$ is no_msg, $\top$ is msg.

is reached, the timer is reset, and a signal is sent to the driver. Within each time period, the driver copies its last input value to its output channel.

Figure 2 shows the automatically created CLP code for this example (which has been modified for this presentation). The first argument (tuple) of all predicates denotes the source state, the second argument (tuple) local variables and their updated values, the third the input, the fourth the output channel, the fifth the transition's name, and the sixth the destination state. Note that we need to distinguish between presence and absence of a value on a channel. This is why the variables that are associated with channels are tagged with $\top$ (message present), or $\bot$ (message absent), respectively. We chose to add the driver to our example in order to emphasize the compositionality of our approach. A driver component can be specified and tested separately from the rest of the system.

5 Testing

There is confusion on how to use terms such as "test case", "test sequence", or "test case specifications" (see, e.g., [2, 12, 20]). We first define these and related terms, show how our CLP based approach to test case generation fits into this terminology, and explain why simulation can be viewed as a special case of test case generation. We then relate the definitions to the timer example of section 4 and give a few

examples of test case generation.

Definitions. Figure 3 shows how some of the following terms are related, and, more particularly, gives an overview over the structure of the test case generation technique we propose.

1. *Validation* is the process of determining conformance of a specification with informal requirements.

2. *Verification* is the process of establishing a relation between two objects, where the relation as well as the objects are formalized. The desired results of verification need neither be effectively nor efficiently computable.

3. A *system specification* is, in its most abstract form, a relation between (possibly infinite) input- and output histories. Z [27] or FOCUS [3] are examples for languages to formulate specifications. System specifications are considered to be validated, and they need not necessarily be executable.

4. The *system model* is a somewhat more concrete formulation of the system specification. In AUTOFOCUS, for instance, system models consist, among other things, of extended finite state machines and diagrams that describe the system's architecture. If implementations are to be tested, the system model is supposed to be consistent with the system specification.

5. *Implementations* are executable system models that are usually regarded from a black box point of view (programs in C or Prolog, or hardware). Implementations can and tend to be incorrect w.r.t. their specifications. Testing an implementation w.r.t. its specification is also called *conformance testing*. For a formal relationship between specification and implementation, see [26, 28].

6. *Traces* are concrete pairs of possibly infinite input- and output histories. For test purposes, we will restrict ourselves to finite traces. Traces are one means of specifying a system as well as test cases (see below).

7. The *test object* is either the implementation (black box) or the system model (white box). It is the entity one wants to deduce information about.

8. A *test purpose* is the possibly informal formulation of properties of the test object that, later on, will be verified. This verification process needs neither be effective nor complete. Examples of test purposes include "there exists some transition sequence satisfying some property", or "there is a trace satisfying some property", or "there is an output history satisfying some property", or combinations hereof.

9. A *test case specification* is a formal description of a test purpose (e.g., a message sequence chart). The test case specification can be black box (details of the implementation are unknown, and we use information about the system model to derive "tests") or white box (if the system model is to be tested, which, actually, boils down to (incomplete) verification).

10. *Test cases* are artifacts that satisfy a test case specification, and they cannot exist without their test case specification. Examples include actual input histories, transition tours, traces, or constraints over them. The latter point implies that they need not be executable. Test cases also are the root for further analyses, e.g., interval/limit analysis. The results of these analyses can then be used to generate a set of actual test sequences.

11. *Test sequences*, on the other hand, are necessarily executable, i.e., can be fed into some implementation or executable system model. We distinguish between test cases and test sequences because usually, it is hard work to determine actual sequences from a set of constraints (as formulated by a set of message sequence charts, or some other constraints directly formulated in CLP[1]). Test sequences are rather operational whereas test cases are rather denotational, and test cases as well as test sequences are to be deduced from the test case specification (which, in general, is given by a human). This is why we do not call a "test case" the union of test case specifications and test sequences. [11] points out that the mere existence of a specification does not necessarily imply the possibility of interpreting the outcome of a test. This is due to the fact that specifications tend to be more abstract than implementations and may be formulated in inherently different formalisms; there is a need for *observational equivalence*.

12. A *test method* then consists of determining the test object, the system model and possibly the implementation, a set of test purposes, test case specifications and methods for the determination of test cases or test sequences.

13. Finally, *testing* comprises the activities of determining a test method and feeding the test cases into the implementation or executable system model. The result are transition tours or I/O histories that, w.r.t. a given test case, are consistent or not consistent, which allows for the formulation of (14) *verdicts*.

Remarks. (1) Testing is usually understood as an inherently incomplete process. Incompleteness is a result of incomplete computation processes for test cases or test sequences, respectively, as well as of incomplete test case specifications. (2) Within this terminology, simulation can be seen as a special case of test case generation. If simulation is seen as simply executing a system model with given input traces, then the test case specification consists of stating exactly these input traces. If, on the other hand, simulation is considered to comprise the modeling of the environment, then the environment's as well as the system's behavior can be computed with a test case specification merely stating that traces are not to exceed some finite length. The system will then find one or many or all system traces, according to the technique used for test case generation.

Methodology. Our approach leads hence to two different testing strategies (Fig. 3). *System models* can be tested in a white box manner: Test cases are derived from test case specifications and the system model by executing the model within our CLP simulation. All this is done by constraining the set of possible traces w.r.t. the test case specification. The resulting traces can now (but need not necessarily) be refined in order to generate test sequences, and these sequences can then be fed into the CLP simulation. This includes interval analyses or a partitioning of the possible input traces by hand or by using delayed constraints from the constraint store. *Implementations*, on the other hand, can be tested in a black box manner: Test cases are computed and refined in exactly the same way they are computed for the system model's white box test. The remaining step consists of translating the input traces and the expected output traces for the system model into input (output) traces for the implementation. By means of suitable driver components, tests of the implementation can then be executed, and the actual output traces be compared with the expected ones.

[1] Note that in CLP, a constraint based formulation of a test case may be executable, in fact one of the reasons why we chose CLP for our purposes.

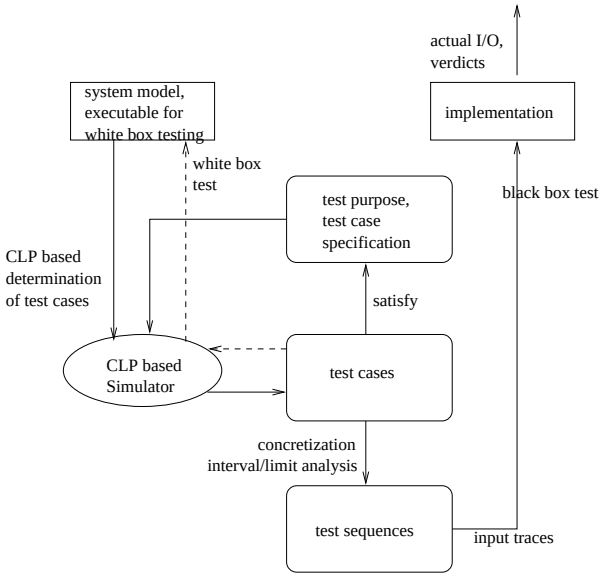

Figure 3: Testing: Overview

In terms of testing, we thus advocate the following development process: (1) build the specification, (2) build the system model, (3) specify test cases (black box and/or white box), (4) in addition to other verification techniques, test the model, (5) implement the system, and (6) test the implementation.

Example. The CLP simulation offers several possibilities for the generation of meaningful test cases. In this context, the CLP simulation has two main advantages over conventional simulation: backtracking (LP) and decision delaying (CLP). On one hand, Prolog's backtracking ability allows the simulation to go backward step by step and look for solutions in other paths of the search tree without restarting. On the other hand, some decisions can be delayed by the use of CLP system. This reduces the search tree significantly and results in an efficient test sequence generation. By now, test sequence generation within CLP is done by querying the Prolog system. The current status of our system offers a fully automatic test sequence generation for the following common kinds of test case specifications:

Arbitrary simulation. The simulation can be fed with any desired input sequence. As the length of the actual simulation is restricted by the maximal number of clock ticks, `ClockMax`, it terminates in any case and produces a test sequence with no more than `ClockMax` steps – as far as one exists. Example: Our blinker system has one input channel of type integer. A possible test purpose "The implementation should be tested with different interval values" can be expressed in a test case specification. This requires partitioning of the interval length in some equivalence classes and at least one test for each class. A given partition of the interval's length (possibly derived via the classification tree method, or by interval analysis on the delayed constraints in the store) yields one test case per equivalence class. We assume the equivalence classes -1, 0, 1, 2-10, and >10. An appropriate test sequence can now be found by querying the CLP system and additionally constraining the input sequence with the desired equivalence class. In the negative case, the resulting traces contain only the same negative timer value T with no timeouts. This is due to a somewhat incomplete specification; the calculation of the idle transitions correctly negates the cases $T\# = 0$ and $T\# > 0$, yielding

$T\# < 0$ for the idle transition. In case of interval=0 or interval=1 the results are straightforward, i.e. the simulation produces test sequence of varying length but with equal input/output traces. The other two equivalence classes produce several different test sequences, some of them with concrete input data and others referring to certain intervals.

Transition sequences. A more sophisticated way of producing test sequences is to propose a desired transition sequence, e.g., by means of MSCs, and compute a matching sequence of input/output values. The proposed sequence need not be complete, i.e. specifying a subsequence is also allowed. Example: We want to test the transition from state `Off` to state `On` in the blinker component. For simplicity we call this transition `OffOn`. A corresponding test purpose could be "the transition `OffOn` should fire and its correct execution, i.e., the fulfillment of certain postconditions, be tested". Figure 4 illustrates the test case specification. By querying the CLP simulation we can find a test sequence that drives the implementation of the blinker component in the `Off` state and then executes the transition. This is done by restricting the last transition of the blinker's transition history to `OffOn`. One of the computed test cases is shown in Figure 5.[2] Note this test sequence does not fulfill the requirements of the given test purpose as the correct execution of the `OffOn` transition is not tested (i.e., we did not specify the postconditions). We would hence need to combine this sequence with one or more other sequences that test the correctness of the execution. Currently, we are working on the integration of MSCs into the process of specifying transition sequences.

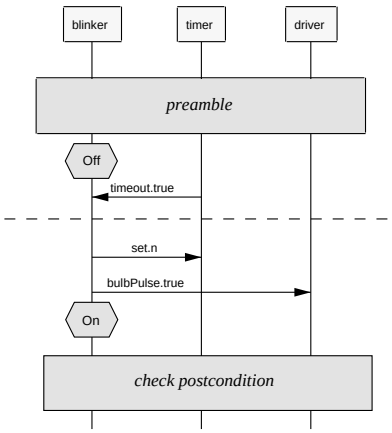

Figure 4: Test case specification

State combinations. A common task in test sequence generation is the determination of an I/O sequence that drives an implementation in a certain state. This can be done by determining the final state in which the simulation should stop. Example: Drive the blinker component into the state `Off`. This is done by restricting the blinker component's final state in the CLP simulation to `Off`. The simulation produces several test sequences starting with an interval of 0 and 5 clock ticks. An additional restriction of the produced test sequences to a certain length leads to a more specific test sequence generation. By constraining the length of the desired test sequence to 7, the CLP simulation produces exactly 9 sequences with intervals

[2] Actually, the system calculated a *test sequence*; we chose the graphical representation (*test case*) for this paper.

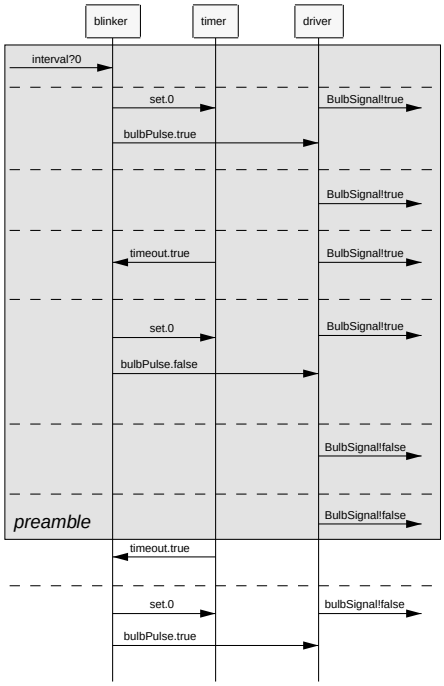

Figure 5: Computed test sequence

ranging from 0 to 3. A longer interval is not possible as the implementation will then need at least 8 clock ticks to get into the state `Off`.

Output sequences. Sometimes an implementation should be tested w.r.t. the ability to produce certain outputs. The determination of appropriate input values that produce the desired output sequence can be a thorny task as functions are usually not reversible. The CLP system is able to reverse some of the functions by the use of constraint handling rules. Otherwise input data can be found with the help of Prolog's backtracking ability and successively instantiating unknown input values. Example: For our blinker example we want a test sequence where the length of the output interval is exactly 5 ticks long. Therefore we require the output sequence to be `...`, `(msg,true)`, `(msg,false)`, `(msg,false)`, `(msg,false)`, `(msg,false)`, `(msg,true)`, `....` We can express this requirement by additionally constraining the output sequence to include the desired subsequence (e.g., again via MSCs). The Prolog system produces test sequences of various length beginning from 13 clock ticks and a constant input interval of 2. Note that an input interval of 2 yields an output interval length of 5 because of internal signal delays.

Coverages. The quality of test cases is often measured by coverage analysis. There exists several metrics for the coverage measurement like statement coverage, branch coverage, or decision coverages. In terms of our models this corresponds to state coverage (i.e. all states have to be visited) or even stronger transition coverage (i.e. all transitions have to be fired at least once). A test case with such a property can easily be obtained with by querying the CLP simulation, as long as one exists. In order to get transition coverage, the history list for the fired transitions has to be constrained in such a way that it contains all specified transitions. For the blinker example the corresponding constraint to obtain transition coverage is

```
cmem((start,_,_),T), cmem((switchOff,_,_),T), cmem((switchOn,_,_),T),
cmem((_,set,_),T), cmem((_,timeout,_),T), cmem((_,tick,_),T),
cmem((_,_,receive),T),cmem((_,_,transf),T)
```

After querying the CLP system we get the following I/O trace of the system with a maximum clock value of 10:

```
I=[(msg, 1), _, _, _, _, _, _, _, _, stop]
O=[(msg, true), (msg, true), (msg, true), (msg, true), (msg, true),
   (msg, false),(msg, false),(msg, false),(msg, false), stop]
T=[(start, idle, transf), (idle, set, receive), (idle, tick, transf),
   (idle, timeout, transf), (switchOff, idle, transf), (idle, set, receive),
   (idle, tick, transf), (idle, timeout, transf), (switchOn, idle, transf),
   stop]
```

Note that the interval value has to be set to a minimum value of one in order to get the timer's tick transition fired.

Combinations. Some of the above discussed techniques for test sequence generation can be successfully combined in order to get more specific test sequences. For example, a predetermined transition should be tested with various input values. Therefore, we can compute one test sequence for each pair consisting of a transition and an input value. We just have to ensure that the input value and the state transition occur at the same time. This can be done by comparing the length of the input sequence and the transition sequence.

The examples show that the CLP system is able to find appropriate test sequences as long as the desired sequence can be constrained effectively. Of course the CLP test case generation does not enforce the use of a special test method. Furthermore, traditional test methods are typically not able to take advantage of all features of the CLP simulation.

6 Languages for Testing

A crucial question in the testing context is the question which specification language should be used. Raw CLP predicates or formulas specified in temporal logics surely are not the best choice.

In terms of representation, automata are a good choice when general properties are to be tested. However, when sequences of transitions, sequences of states or component interactions, i.e., partial I/O traces, form the major part of the property to be tested, they suffer from the fact that they do not contain a *sequential* notion of progressing time. Particularly for the first kind of test case specifications, Sequence Charts seem to be more appropriate. Note that this argumentation is about (intuitive) representation rather than expressivity. This motivates the following discussion on specification languages for test cases.

In contrast to automata, MSCs explicitly visualize the progress of time in a sequential manner. They describe exemplary component interactions, and are usually deployed to specify use cases that show a certain required behavior of the system model or its implementation, respectively. Condition boxes in MSCs can be used to specify system states [21, 13] - a fact that is used in work on their translation into automata [21]. The incorporation of special symbols for transitions into MSCs is the subject of ongoing work. Note that this does not advocate the integration of MSCs and automata into a new kind of language for the specification of systems, but rather for the specification of test cases. In terms of system specification, a clear distinction between behavior and interaction view seems to be reasonable.

The specification of properties also necessitates a construct for negation. If negation is to be used on levels of the property other than the outmost one (i.e., "the following property should *not* hold"), there is a need for describing "partial" negation within MSCs which is the subject of ongoing work. In addition, for the specification of test cases, the specification of iteration is needed within MSCs.

One problem with these extensions is the formal definition of their semantics which, in case of negation, turns out to be far from being trivial. Another complication arises from the fact that there are different interpretations of MSCs. Two subsequent messages (arrows) in a diagram may mean "nothing but these two messages occur within the specified time in the specified order", or they may mean "the partial I/O behavior in question must contain the two specified messages in their specified order". It seems to be reasonable to admit both interpretations [7].

Summarizing, we consider MSCs as an appropriate, intuitive means for test case specifications in cases where I/O behavior is to be tested. This necessitates constructs for iteration. Furthermore, for sequences of states or transitions, MSCs seem to be a good choice, provided that constructs for states (condition boxes) or transitions (special arrows) are supported. However, other test cases may require other specification languages. Furthermore, the presentation of computed test cases as MSCs or in the Combined Tree and Tabular Notation (e.g., [20]) may be a promising approach. The determination of a set of appropriate languages as well as its integration in our CLP system is the subject of ongoing work.

7 Conclusions and Future work

There is a strong and increasing industrial need for tool support in testing and, more particularly, test case generation. To address this problem, we presented a framework for simulating and testing concurrent reactive systems on the grounds of Constraint Logic Programming and Constraint Handling Rules. W.r.t. model checking, we see testing as a complementary technique, with other goals and other implementations. System specifications in AutoFocus were shown to be automatically compilable into CLP/CHR code in a fully compositional way, taking into account recursive functions as well as recursive data types that occur in the system specification. We showed how a system compiled into CLP can be simulated and demonstrated how typical tasks in the generation of test cases can be fulfilled within our approach. Test cases may be generated on arbitrary domains, allowing for potentially infinite ones (e.g., via interval representation). Even though we concentrated on testing, we believe that exactly the same techniques can be used as a debugging aid when specifying a system. We proposed an integration of this scheme with test case specifications based on high level languages such as MSCs that, at present, must be compiled into CLP by hand.

Currently, we are investigating possibilities to derive actual test sequences from test cases by scrutinizing delayed goals in the constraint store. We believe that analyses such as interval/boundary analysis will yield a good class of test sequence representatives for the actual process of testing the implementation. Furthermore, we are focusing on heuristics when to use forward or backward analyses. The presented approach is a forward analysis, but by switching variables and transforming constraints, it can be made a backward analysis. In addition, we consider the augmentation of transitions with counters a promising way to reduce the risk of running into loops (because of Prolog's depth-first search strategy).

Interesting open problems include the problem of negating properties, in MSCs for test case specification as well as in the calculation of idle transitions for automata and the problem of function inversion in CLP. In addition, we are considering the extension of our approach to mixed discrete-continuous, or hybrid, systems [25] (for

test case specifications, we focus on HySCs [13]). For simple differential equations with exponential functions as solutions, it seems possible to implement a specialized solver for this kind of constraints. This means that we do not have to counter-intuitively insert clock ticks into continuous activities (as is, for instance, done in [4]). This is a focus of our current work. Hybrid automata [1] are not well suited for modular design tasks [24], so we concentrate on extended finite state machines with continuous activities. For Hybrid systems in the context of CLP, cf., e.g., [29]; see [5] for the testing of hybrid systems. Last but not least, we are discussing the integration of abstract interpretation techniques [6] into our framework.

Acknowledgment. We would like to thank Oscar Slotosch and Thomas Stauner for fruitful discussions and extensive comments on this paper.

References

[1] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, February 1995.

[2] B. Beizer. *Black Box Testing.* John Wiley and Sons, 1995. ISBN 0-471-12094-4.

[3] M. Broy, F. Dederich, C. Dendorfer, M. Fuchs, T. Gritzner, and R. Weber. The design of distributed systems - an introduction to focus. Technical Report TUM-I9202, Technische Universität München, 1992.

[4] A. Ciarlini and T. Frühwirth. Using Constraint Logic Programming for Software Validation. In *5th workshop on the German-Brazilian Bilateral Programme for Scientific and Technological Cooperation*, Königswinter, Germany, March 1999.

[5] A. Ciarlini and T. Frühwirth. Automatic derivation of meaningful experiments for hybrid systems. In *Proc. ACM SIGSIM Conf. on Artificial Intelligence, Simulation, and Planning (AIS'00)*, Tucson, AZ, March 2000.

[6] P. Cousot and R. Cousot. Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM symp. on Principles of Programming Languages (POPL'77)*, pages 238–252, 1977.

[7] W. Damm and D. Harel. LSC's: Breathing Life into Message Sequence Charts. In *Proc. 3rd Intl. Conf. on Formal Methods for open object-based distributed systems (FMOODS'99)*, 1999.

[8] G. Delzanno and A. Podelski. Model Checking in CLP. In *Proc. Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, pages 223–239, 1999.

[9] L. Fribourg. Constraint logic programming applied to model checking. In *Proc. 9th Int. Workshop on Logic-based Program Synthesis and Transformation (LOPSTR'99)*, LNCS 1817, Venice, 1999. Springer Verlag.

[10] L. Fribourg and M. Veloso-Peixoto. Automates Concurrents á Contraintes. *Technique et Science Informatiques*, 13(6):837–866, 1994.

[11] M. Gaudel. Testing can be formal, too. In P. Mosses, M. Nielsen, and M. Schwartzbach, editors, *Proc. Intl. Conf. on Theory and Practice of Software Development (TAPSOFT'95)*, LNCS 915, pages 82–96, Aarhus, Denmark, May 1995.

[12] J. Grabowski. *Test Case Generation and Test Case Specification with Message Sequence Charts.* PhD thesis, Universität Bern, 1994.

[13] R. Grosu, I. Krüger, and T. Stauner. Hybrid Sequence Charts. In *Proc. 3rd IEEE Intl. Symp. on Object-oriented Real-time distributed Computing (ISORC 2000).* IEEE, 2000.

[14] G. Gupta and E. Pontelli. A Constraint-based Approach to Specification and Verification of Real-time Systems. In *Proc. IEEE Real-time Symposium*, pages 230–239, San Francisco, December 1997.

[15] M. Hanus (ed.). Curry: An Integrated Functional Logic Language. www.informatik.uni-kiel.de/~curry/report.html.

[16] F. Huber, S. Molterer, A. Rausch, B. Schätz, M. Sihling, and O. Slotosch. Tool supported specification and simulation of distributed systems. In B. Krämer, N. Uchihira, P. Croll, and S. Russo, editors, *Proc. Intl. Symp. on Software Engineering for Parallel and Distributed Systems*, pages 155–164. IEEE, 1998.

[17] 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.

[18] F. Huber, B. Schätz, and G. Einert. Consistent Graphical Specification of Distributed Systems. In J. Fitzgerald, C. Jones, and P. Lucas, editors, *Industrial Applications and Strengthened Foundations of Formal Methods (FME'97)*, LNCS 1313, pages 122–141. Springer Verlag, 1997.

[19] ITU. ITU-T Recommendation Z.120: Message Sequence Charts (MSC), November 1999.

[20] K. Knightson. *OSI protocol conformance testing - IS 9646 explained.* McGraw-Hill, 1993. ISBN 0-07-035134-1.

[21] I. Krüger. *Distributed Systems Design with Message Sequence Charts.* PhD thesis, Munich University of Technology, 2000.

[22] H. Lötzbeyer and A. Pretschner. AutoFocus on Constraint Logic Programming. In *Proc. Logic Programming and Software Engineering (LPSE'00)*, London, July 2000.

[23] H. Lötzbeyer and A. Pretschner. Concurrent Reactive Systems and Constraint Logic Programming: A framework for compositional testing and validation, 2000. Internal report, Technische Universität München.

[24] O. Müller and T. Stauner. Modelling and verification using Linear Hybrid Automata. *Mathematical Computer Modeling of Dynamical Systems*, 6(1), March 2000.

[25] A. Pretschner, O. Slotosch, and T. Stauner. Developing Correct Safety Critical, Hybrid, Embedded Systems. In *Proc. New Information Processing Techniques for Military Systems, NATO Research*, Istanbul, 2000. To appear.

[26] S. Sadeghipour. *Testing Cyclic Software Components of Reactive Systems on the Basis of Formal Specifications.* PhD thesis, TU Berlin, 1998.

[27] J. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 2nd edition, 1992.

[28] J. Tretmans and A. Belinfante. Automatic testing with formal methods. In *7th. European Intl. Conf. on Software Testing, Analysis & Review (EuroSTAR'99)*, Barcelona, Spain, November 1999.

[29] L. Urbina. Analysis of Hybrid Systems in CLP(R). In E. C. Freuder, editor, *Proc. 2nd Intl. Conf. on Principles and Practice of Constraint Programming*, LNCS 1118, pages 451–467, Cambridge, Massachusetts, USA, 1996. Springer Verlag.

[30] G. Wimmel, H. Lötzbeyer, A. Pretschner, and O. Slotosch. Specification Based Test Sequence Generation with Propositional Logic. *J. Software Testing, Verification & Reliability (STVR): Special Issue on Specification Based Testing*, 2000. To appear.