Proc. (Constraint) Logic Programming and Software Engineering (LPSE), London, July 2000

AutoFocus on 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

The CASE tool AutoFocus allows for modeling and validating concurrent reactive systems on the basis of a simple, clearly defined formal semantics. It is shown how executable code for common Constraint Logic Programming languages can be generated automatically. Applications include simulation for rapid prototyping and debugging as well as the semi-automatic generation of test sequences.
Keywords. CASE, Constraint Handling Rules, Code Generation, Rapid Prototyping, Reactive systems, Simulation, Test Case Generation.

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. Model checking on-the-fly, for instance, composes an automaton describing a system with a property (e.g., in LTL) encoded as an automaton. However, it yet suffers from inefficiency due to a lack of efficient representations such as BDDs.

When *testing* an implementation, the properties to be checked are usually quite specialized. They describe, for instance, partial I/O traces or transition sequences. Such properties may constrain the set of possible system runs (traces) in a significant way, and if these properties are sufficiently specific, they can be used in building a model of manageable size which may help in alleviating the problem of state space explosion.

Testing usually focuses on finite (or even short) system traces, and it usually is inherently incomplete. As formal methods yet do not scale to real size applications, this deficit has to be accepted but borne in mind. The popular remark that testing can only reveal the presence but never the absence of errors also applies to formal methods: One can only check properties that have been formulated by a human. This process, however, obviously is also necessarily incomplete.

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

Simulation and Testing. In this paper, we lay the foundations for testing executable system specifications (system models) as well as implementations. In our view, this turns out to be a generalization of the simulation of reactive systems: If simulation is seen as simply executing a system model with given input traces, then a test case specification, i.e., the specification of a desired system's behavior, consists of stating exactly these input traces [18]. If, on the other hand, simulation is considered to comprise a 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, or by imposing other constraints on these traces. A system that supports this kind of development will then find one or many or all system traces, according to the technique used for test case generation. This point of view also allows one to see simulation/testing as a debugging aid for rapid or evolutionary prototyping: While the model is specified, the developer may ask for certain states to be reached and thus for a set of I/O traces which may be helpful in detecting flaws in the specification [22].

Code Generation. In order to simulate a system it is necessary to have an executable model. We show 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. There are two advantages of a CLP based code generation in contrast to C or JAVA code generators. First, writing correct code generators for Java and C turned out to be a thorny and error-prone task. Since the translation proposed in this paper is short and close to the model, the generated code is easier to validate. Second, the possibility of inverting functions as provided by Logic Programming languages makes these languages a good candidate for building programs for test case generation. We believe that CLP with its mechanisms of a priori pruning the search tree is particularly suited for effectively and efficiently generating test cases. Note, however, that the focus of this article is on code generation. Nonetheless, the underlying design decisions are heavily influenced by the goal of creating a system for the generation of test cases [18].

Overview. The remainder of this paper is organized as follows. After briefly discussing related work, Section 2 presents the CASE tool AutoFocus into which we integrated our CLP based simulation system. The modeling concepts are illustrated along the lines of a simple system consisting of a bulb that blinks according to a timer that may be set by the user. Section 3 starts with a a brief overview over Constraint, Logic, and Constraint Logic Programming. It is then shown how system models in AutoFocus can automatically be translated into CLP by maintaining full compositionality. In section 4 we demonstrate how our approach can be applied to user defined domains via Constraint Handling Rules (CHRs). We also consider the problems of types in CLP. Finally, we draw some conclusions and present current as well as future work in Section 5.

Related work. Code Generation from reactive systems on the basis of CLP has been discussed recently. Delzanno and Podelski [4] focus on general Labeled Transition Systems. The motivation for their translation is model checking. Ciarlini and Frühwirth [3] focus on Hybrid Automata [1] that allow for continuous activities within discrete states. So does Urbina [23]; 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 [10], 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 [19]. Fribourg and Veloso-Peixoto [6] 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 [5] reviews previous work on the relationship between CLP and Model Checking. Finally, there is a wealth of literature on test case generation; [24, 18] contain some relevant references.

2 Modeling with AutoFocus

AutoFocus [12, 13, 22] is a tool for graphically specifying embedded systems. It supports different views on the system model: structure, behavior, interaction, and data type view.

Structural view: SSDs. In AutoFocus, a distributed system is a network of components, possibly connected one to the other, communicating via so-called *channels*. The partners of all interactions are components that are specified in *System Structure Diagrams* (SSD). Figure 1 (a) shows a typical 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 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). Figures 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 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

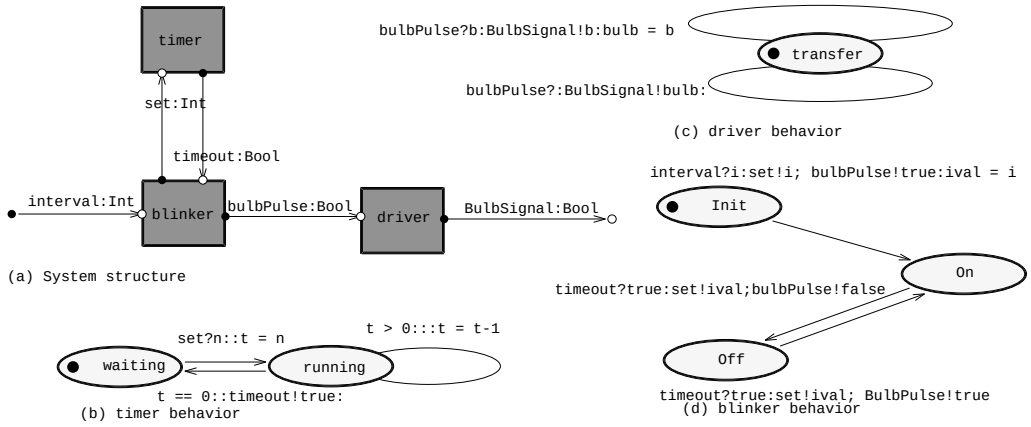

Figure 1: Timer/blinker example

input ports of connected components and the cycle is repeated. This results in a *time synchronous* communication scheme with buffer size 1.

The use of Sequence Charts for visualizing component interaction (traces) is also supported in AutoFocus. Sequence Charts can be used for specifying systems as well as test cases [24]. This issue is omitted here for the sake of brevity but will be referred to again in the conclusion.

Datatype view: DTDs. For the specification of user defined data types and functions AutoFocus provides DTDs. The definitions in DTDs are written in Gofer like functional 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 consistency checks that work on a syntactical basis [14].

Example. Figure 1 shows the structure as well as the behavior of a simple timer example: a system consisting of three components, a timer, a blinker, and a driver. Figure 1 (a) depicts the system's 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: Whenever 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 channel *set*. Part (c) exhibits 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 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 5 shows parts of the automatically created CLP code for this example (which has been modified for this presentation). 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.

Adequacy. Experiences with students as well as industrial partners have shown that the semantics of AutoFocus is quite easy to grasp. In fact, we believe that this is one of the distinguishing features of AutoFocus. Shortcomings of this approach are, nonetheless, problems with asynchronous communication schemes that often occur in telecommunication systems. The semantics of AutoFocus is a stream-based one that has been formally defined in [14]. This feature makes it particularly suitable to code generators that mainly operate on lists which is the case for Prolog and CLP languages.

3 Translation into CLP($\mathcal{FD}$)

In this section, we present a translation of AutoFocus automata into CLP that is (a) fully compositional and (b) can be performed fully automatically. For the sake of clarity, we restrict ourselves to finite integer domains; the handling of other domains is the subject of section 4.

3.1 Logic and Constraint Programming

Our initial goal is the task of simulation and, closely related, automatically deriving test cases. In order to fulfill it, we chose the following programming paradigms for the following reasons: They at least partly allow for automatic function inversion which is a crucial point of this task. A second important issue is the reduction of the set of meaningful test cases, and Constraint Programming seems a natural candidate for resolving this problem. Furthermore, it is possible to work on infinite domains (e.g., real numbers) with finite representations (intervals).

Logic Programming. One of the roots of Logic Programming (LP) is Kowalski's crucial idea to consider algorithms as composed of logic and control, thus explicitly separating the "What?" from the "How?". The ultimate (but, as experience has shown, yet unrealistic) goal of declarative programming languages is to program in a purely declarative way by simply stating a set of relations that describe a given problem. Common LP languages such as Prolog then interpret these relations or predicates in a procedural manner by relying on a powerful general problem solving mechanism, namely backtracking. LP languages exhibit another important feature, namely the possibility of function inversion: Under certain circumstances, given the result of a function application, one can infer the function's arguments (or a set of them).

Constraint Logic Programming. However, there are some pitfalls in LP. On the one hand, solutions of programs (models of logical formulae) are always based on the same carrier set, a term universe (the so-called minimal Herbrand model). On the other hand, in implementations of LP languages, there is a certain order in which predicates are evaluated (in the procedural sense, see above) which may result in infinite evaluations even though the succeeding predicate could prevent infinite backtracking by imposing constraints that its preceding predicate can only satisfy in a finite number of ways. This led to the idea of merging Constraint Languages with LP into Constraint Logic Programming (CLP) languages [16]. These languages allow for the formulation of constraints that are checked for satisfiability in every step of the evaluation of a set of logical formulae (expansion of a node in the resolution tree), and they hence necessitate mechanisms for delaying subexpressions. This yields the possibility of a priori cutting the evaluation

tree of these formulae; the "generate and test" paradigm of LP languages is modified to "constrain and generate". On the other hand, with CLP, one can calculate in domains other than the Herbrand universe, for instance finite (integer) domains $\mathcal{FD}$, or rational or real numbers $\mathcal{Q}$ and $\mathcal{R}$ (one crucial point in the latter two domains is to calculate on finitely representable intervals.) One can, for instance, formulate Linear Programming Problems with a set of unknown variables, and if the CLP language is equipped with suitable constraint solvers (e.g., Simplex), the desired optimal results can be found by binding variables to the corresponding rational numbers or intervals. LP is an instance of CLP with constraints being equations over terms, or finite trees, respectively.

Constraint Handling Rules. Even though there are many constraint solvers available, it turned out that sometimes people do not want to calculate on one specific domain but rather a mixture of different domains, and that there sometimes is need to create new domains and constraint handlers. This led to the development of Constraint Handling Rules (CHR [7]), a meta language that allows for the definition of new constraint handlers (solvers) that, subsequently, can be translated into the corresponding target language, CLP in our case.

3.2 Translation

In the sequel, we will not distinguish between a channel, its name and its history (or "trace": the sequence of symbols that were observable on the channel so far). Histories may be seen as timed streams, and they are implemented as lists. All terms starting with capital letters are variables, with "_" being the anonymous variable, and if the name of a sequence occurs at some position, then we assume that this name is expanded to the corresponding sequence where all elements start with a capital letter (thus becoming Prolog variables). Set operations naturally become operations on sequences/lists (e.g., union becomes concatenation).

Transitions. Let a sequence $x_1, \ldots, x_n$ or $(x_1, y_1), \ldots, (x_n, y_n)$ be denoted by $\overline{x}^n$ or $\overline{(x,y)}^n$, respectively. Furthermore, let the sequence of lists $[x_1|xs_1], \ldots, [x_n|xs_n]$ be denoted by $\overline{[x|xs]}^n$. Each component of an overlined term thus gets an index ranging from 1 to the index of the overlined term, n in this case. For each transition we now introduce a rule

$step(Src, \overline{(L, L')}^n), \overline{I}^k), \overline{O}^\ell), tname(W), Dst) \Leftarrow$
$\quad \Phi(\overline{L}^n, \overline{L'}^n, \overline{I}^k), \overline{O}^\ell)) \wedge \Psi(\overline{L}^n, \overline{L'}^n, \overline{I}^k), \overline{O}^\ell)).$

with Src being the source and Dst being the destination state, and where primed variants of L_i represent the new values of component-local variables L_i after the transition has fired, i.e., after a time tick. In addition, W is a sequence of local variables (or input channels) the respective transition is, for tracing purposes, to be annotated with. The I_i (O_i) consist of pairs $(\{msg, no\_msg\}, value)$, and they represent the state of input (output) channel I_i (O_i): If a value is present at this channel, the first component is set to msg and to no_msg otherwise. Φ and Ψ are the result of translating the pre- and postcondition φ and ψ into constraints (see below). Postconditions may modify the primed L_i, thus resulting in a new data state.

Pre- and Postconditions. Pre- and postconditions are directly translated into a conjunction of constraints by replacing operations on input values and the values of local variables by their respective operation in the constrained domain (e.g., $\leq$ becomes $\#\leq$, see section 4 for other domains than integers). Preconditions involve only unprimed variables whereas postconditions may involve primed variables. If a condition requires that

a value of some channel be present, then the corresponding pair in the head of the *step* predicate must be *msg* in its first component; otherwise, this first component is *no_msg* or even _ if this information is not necessary in order to evaluate the conditions (e.g., if a transition does not involve the state of some channel). Note that *msg* can denote both a concrete value, or, in the case of finite integer domains, an interval (a finite domain variable). Note also that postconditions in AUTOFOCUS always hold and that it is hence not possible to "hide" preconditions within the postconditions.

Idle Transitions. In AUTOFOCUS, if no transition can fire, the system remains in the current state. The translation so far needs to be complemented: For k input channels $I_1, \ldots, I_k$ and m transitions $T_1, \ldots, T_m$ leaving some state s we have $(1 \leq j \leq m)$

$$T_j \equiv step\big(s, \ldots, \big(\overline{(ind_j, I)}^k\big), \ldots, s'\big) \Leftarrow \bigwedge_{\substack{i=1 \\ ind_{ji}=msg}}^{k} match_{ji}(I_i) \quad \wedge pre_j(\overline{L}^n, \overline{I}^k) \wedge action_j(\overline{L}^n, \overline{L'}^n, \overline{I}^k). \tag{1}$$

where $ind_{ij} \in \{\_, msg, no\_msg\}$, $match_{ji}(I_i)$ are the pattern match conditions of transition j involving input channels, pre_j contains the preconditions involving local variables and channel inputs, and $action_j$ is the conjunction of the transition's actions (i.e. assignments to local variables, or postconditions). We do not need to consider outputs here. Formula 1 can be rewritten as

$$\begin{aligned} T_j \equiv\ & step\big(s, \ldots, \big(\overline{(X, I)}^k\big), \ldots, s'\big) \Leftarrow \\ & pre_j(\overline{L}^n, \overline{I}^k) \wedge action_j(\overline{L}^n, \overline{L'}^n, \overline{I}^k) \wedge \\ & \bigwedge_{\substack{i=1 \\ ind_{ji}=msg}}^{k} \big(X_i = msg \wedge match_{ji}(I_i)\big) \wedge \bigwedge_{\substack{i=1 \\ ind_{ji}=no\_msg}}^{k} X_i = no\_msg, \end{aligned} \tag{2}$$

where new variables X_i are introduced, indicating whether a message is sent over the appropriate channel and $=$ is Prolog's unification equality. From formula 2 the body of the negated enabling condition for transition T_j can be derived:

$$\tilde{T}_j \equiv \neg pre_j(\overline{L}^n, \overline{I}^k) \vee \bigvee_{\substack{i=1 \\ ind_{ji}=msg}}^{k} \big(X_i = no\_msg \vee \neg match_{ji}(I_i)\big) \vee \bigvee_{\substack{i=1 \\ ind_{ji}=no\_msg}}^{k} X_i = msg,$$

where $\neg pre_j$ is the negation of the constraints contained in pre_j. Note that the action predicates are left out because they are not influencing the enabledness of transitions.

We hence need decidable complementation operators in the constrained domain. This is closely related to the problem of negation in LP [2]. The CHRs we consider are of the form *name* @ *head* $\Leftrightarrow$ *guard*|*body* with the intuitive meaning that if *head* is in the constraint store and the optional *guard* is evaluated to true, then we might replace *head* by *body* in the constraint store (simplification rules). For a most common type of constraints, namely equality and inequality on finite enumeration types, we define the CHRs $a =^c b \Leftrightarrow a = b|true$ and $a \neq^c b \Leftrightarrow a = b|fail$ with no check of guard bindings. For integers, they are predefined in the $\mathcal{FD}$ library: $\#=$ and $\#\backslash =$.

For each state s, this results in an idle transition predicate

$step\big(s, \ldots, \big(\overline{(X, I)}^k\big), \ldots, s\big) \Leftarrow \tilde{T}_1, \ldots, \tilde{T}_m, \Phi, !.$ where Φ maps the internal variables to their primed (i.e., after the tick associated with the transition) counterparts, i.e., $L'\# = L$. The cut is inserted for efficiency reasons.

$$\mathbf{doStep}\big(\mathbf{Src}, (\overline{[\mathbf{L}|\mathbf{Ls}]}^{\mathbf{n}}), (\overline{\mathbf{I}}^{\mathbf{k}}), (\overline{\mathbf{O}}^{\ell}), \mathbf{TrHist}, \mathbf{Clock}, \mathbf{Clock_{max}}, \{\mathbf{Results}\}\big) \Leftarrow$$
$$Clock\# \leq Clock_{max},$$
$$\%\ \text{do the transition}$$
$$step\big(Src, (\overline{(L, L')}^{n}), (\overline{CurI}^{k}), (\overline{O'}^{\ell}), Trans,\ Dest\big),$$
$$\%\ \text{and restart from the new state}$$
$$doStep\big(Dest, (\overline{[L', L|Ls]}^{n}), (\overline{[CurI|I]}^{k}), (\overline{[O'|O]}^{\ell}),$$
$$[Trans|TrHist], Clock + 1, Clock_{max}, \{Results\}\big).$$

Figure 2: Running one single automaton with *doStep*

Control. Automata are controlled by a predicate *doStep* (Fig. 2). For testing purposes, we declare every state as a final state. This is done by adding a predicate *doStep* in which all histories are copied into the result variables. For test purposes, it is a good idea to find the shortest transition sequence first; this is why this predicate should be added *in front* of the other ones. In this definition, the first argument, *Src*, represents the state out of which some transition is to be taken. The second argument represents the histories of all local variables; the third and fourth arguments represent the histories of the in- and output channels. If for an automaton $\mathcal{A}$ the input before tick t_j is denoted by $i^{\mathcal{A}}(t_j^-)$ and its output after this tick by $o^{\mathcal{A}}(t_j^+)$, then $(\overline{I}^m)$ corresponds to the set of all $i^{\mathcal{A}}(t_j^-)$, and $(\overline{O}^{\ell})$ to the set of all $o^{\mathcal{A}}(t_j^+)$. In normal operation mode (inputs are known at run time), $(\overline{I}^m)$ are instantiated input and $(\overline{O}^{\ell})$ are free output variables. *TrHist* records the transitions, *Clock* has to be smaller than $Clock_{max}$ (finiteness!), and *Results* is a set of arguments into which, at the end, histories of interest will be copied.

Obviously, the first call to *doStep* requires the involved lists to be nonempty. For channels, pairs of the form (*no_msg*,_) should be the elements of the argument lists, and local variables may have arbitrary values. This first call models initial states.

Parallel Composition. Given two automata $\mathcal{A}$ and $\mathcal{B}$, $X \in \{\mathcal{A}, \mathcal{B}\}$, let $in(X)$ ($out(X)$) denote the input (output) ports of automaton X. The composition of two automata requires that some output ports of $\mathcal{A}$ be attached to some input ports of $\mathcal{B}$ via connecting channels and vice versa. We thus introduce a set $\mathcal{Z} \subseteq \big(in(\mathcal{A}) \otimes out(\mathcal{B})\big) \cup \big(in(\mathcal{B}) \otimes out(\mathcal{A})\big)$ which represents the channels to be connected. Remember that the channels are considered sequences (lists) rather than sets; since there are always two ports that are connected, $\otimes$ simply denotes the pairing operator on lists $L_i = [L_i(1), L_i(2), \ldots, L_i(n)]$ for $i \in \{1, 2\}$: $L_1 \otimes L_2 = [(L_1(1), L_2(1)), \ldots, (L_1(n), L_2(n))]$.

We will need to distinguish between internal communication channels and external ones. A first step consists of defining two sets $IBound = \{i | \exists o.(i, o) \in \mathcal{Z}\}$ and $OBound = \{o | \exists i.(i, o) \in \mathcal{Z}\}$, respectively, that contain the set of "connected", or bound, input resp. output ports. Call an input port (or its value) *unbound*, if it is not connected to an output port of another component. Furthermore, let $v(t_j^-)$ denote a variable v's value before tick t_j, and let $v(t_j^+)$ denote v's value after this tick.

The justification for the counterintuitive construction of $\mathcal{Z}$ by making the input channel the *first* component and the output channel the *second* is as follows:

For the composed system $\mathcal{C}$, we have a set of unbound inputs to $\mathcal{A}$ and $\mathcal{B}$, $\mathcal{I}^u = \{i^{\mathcal{C}} | i^{\mathcal{C}} \in (in(\mathcal{A}) \cup in(\mathcal{B})) - IBound\}$ at time t_i^-. The respective set of unbound outputs is $\mathcal{O}^u = \{i^{\mathcal{C}} | i^{\mathcal{C}} \in (out(\mathcal{A}) \cup out(\mathcal{B})) - OBound\}$ at time t_i^+. The composed system's local variables include the internal variables of all components, $\bigcup_{j=1}^{\ell} \big(L_j(t_i^-), L_j(t_i^+)\big)$.

Here, $L_j(t_i^-)$ and $L_j(t_i^+)$ correspond to L_j and L'_j as explained above. We assume the name spaces of the original components to be disjunct. The first component of each pair denotes the variable's value at before tick t_i, and the second denotes its value after tick t_i.

Now, consider $\mathcal{A}$'s output $o^{\mathcal{A}}$ that is connected to $\mathcal{B}$'s input $i^{\mathcal{B}}$, i.e., $(i^{\mathcal{B}}, o^{\mathcal{A}}) \in \mathcal{Z}$. Obviously, we would like the equation $o^{\mathcal{A}}(t_j^+) = i^{\mathcal{B}}(t_{j+1}^-)$ to hold for all j. Internal channels, i.e., channels that connect one component to another, may be regarded as (new) variables of the composed system: Between two ticks, t_j and t_{j+1}, the value of this variable is $o^{\mathcal{A}}(t_j^+) = o^{\mathcal{A}}(t_{j+1}^-)$. Before tick t_j, its value is $i^{\mathcal{B}}(t_j^-)$, for it actually is input to $\mathcal{B}$. After this tick, its value is $o^{\mathcal{A}}(t_j^+)$, for it is output by $\mathcal{A}$. Identifying $o^{\mathcal{A}}(t_j^+)$ with $i^{\mathcal{B}}(t_{j+1}^-)$ yields the variable's evolution from $i^{\mathcal{B}}(t_j^-) = o^{\mathcal{A}}(t_j^-)$ to $i^{\mathcal{B}}(t_j^+) = i^{\mathcal{B}}(t_{j+1}^-) = o^{\mathcal{A}}(t_j^+)$. It is usual to interpret composition of channels as information hiding or existential quantification, respectively, e.g., [14].

This schema is identical to the above schema for internal variables of all components (pairs $L_j(t_i^-), L_j(t_i^+)$). All we need to do in order to model composition of the two components is thus to add variables that evolve from $i^{\mathcal{B}}(t_j^-)$ to $o^{\mathcal{A}}(t_j^+)$. For k connections from $\mathcal{A}$ to $\mathcal{B}$, let $o_p^{\mathcal{A}}$ be connected to $i_p^{\mathcal{B}}$ for $1 \leq p \leq k$. We then define a new set of pairs of variables, namely $\bigcup_{p=1}^{k}(\overline{i_{\mathcal{B},p}}(t_j^-), \overline{o_{\mathcal{A},p}}(t_j^+))$. Note that if this set is complemented by adding the inverse connections from $\mathcal{B}$ to $\mathcal{A}$, we exactly obtain set $\mathcal{Z}$ (modulo variable renaming)! The composed system is executed by interleaving steps from $\mathcal{A}$ with $\mathcal{B}$. The

$$
\begin{array}{l}
\mathbf{step}^{\mathcal{A}\|\mathcal{B}}\big((\mathbf{Src}^{\mathcal{A}}, \mathbf{Src}^{\mathcal{B}}), (\overline{(\mathbf{L}^{\mathcal{A}}, \mathbf{L}^{\mathcal{A}\prime})}^{\mathbf{n}}, \overline{(\mathbf{L}^{\mathcal{B}}, \mathbf{L}^{\mathcal{B}\prime})}^{\mathbf{m}}, (\mathcal{Z})), (\mathcal{I}^{\mathbf{u}}), (\mathcal{O}^{\mathbf{u}}), \\
\qquad (\mathbf{Tr}^{\mathcal{A}}, \mathbf{Tr}^{\mathcal{B}}), (\mathbf{Dest}^{\mathcal{A}}, \mathbf{Dest}^{\mathcal{B}})\big) \Leftarrow \quad \% \ \mathcal{Z} \text{ is new!} \\
\qquad step^{\mathcal{A}}(Src^{\mathcal{A}}, (\overline{(L^{\mathcal{A}}, L^{\mathcal{A}\prime})}^{n}), (in(\mathcal{A})), (out(\mathcal{A})), Tr^{\mathcal{A}}, Dest^{\mathcal{A}}), \\
\qquad step^{\mathcal{B}}(Src^{\mathcal{B}}, (\overline{(L^{\mathcal{B}}, L^{\mathcal{B}\prime})}^{m}), (in(\mathcal{B})), (out(\mathcal{B})), Tr^{\mathcal{B}}, Dest^{\mathcal{B}}).
\end{array}
$$

Figure 3: Interleaving two automata: $\mathbf{step}^{\mathcal{A}\|\mathcal{B}}$

new variables $\overline{i_{\mathcal{B},p}}$ and $\overline{o_{\mathcal{B},p}}$ are used as input (or output, respectively) variables when component $\mathcal{A}$ or $\mathcal{B}$ is executed (Fig. 3).

One minor difference between the newly introduced variables with not composed local variables is that their values are tagged with *msg* or *no_msg* (since channels may carry no signal at all). Except for this, the connected channels indeed exhibit the same characteristics as local variables, and we use this insight for the implementation. Surprisingly at first, this simple approach is everything we need in order to model internal communication.

Note again that our encoding works on a *time synchronous* communication scheme with buffer size 1, even though the appearance of predicate $step^{\mathcal{A}\|\mathcal{B}}$ suggests an asynchronous communication scheme. Furthermore, the underlying interleaving model of concurrency is directly reflected in the CLP rules.

Now, control is implemented by the corresponding predicate $doStep^{\mathcal{A}\|\mathcal{B}}$ (Fig. 4), where $L^{\mathcal{Z}}$ denotes new variable names for the elements of $\mathcal{Z}$, and $|\cdot|$ is the cardinality function on sets (or a length function on sequences, respectively). As in the case of a single component it seems reasonable to add a predicate that copies all histories of interests in the $\{Results\}$ set, thus modeling accepting states. Again, for test purposes, the shortest transition sequence should be found first; this is why this predicate should be added *in front* of the other ones.

Composition of more than two automata. The above translation scheme is fully compositional. There are basically two ways of composing three automata $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$,

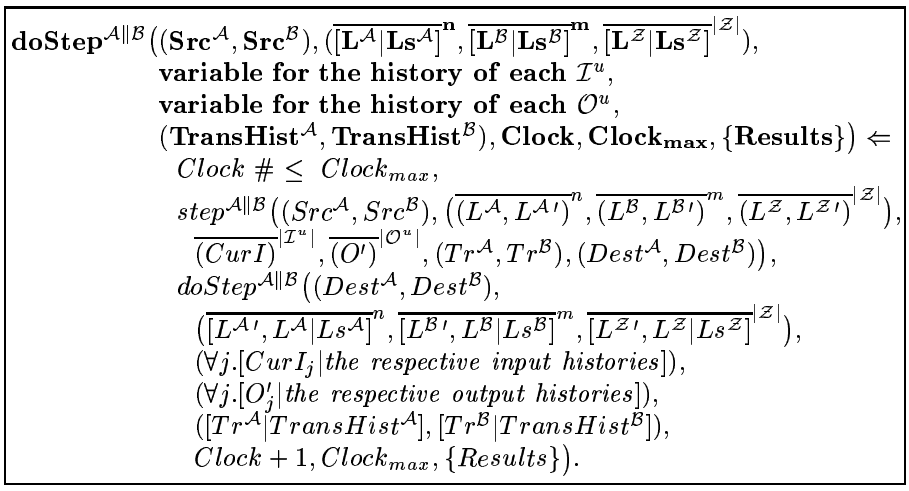

Figure 4: Running two automata concurrently with $\mathbf{doStep}^{\mathcal{A}\|\mathcal{B}}$

depending on the system structure. If the system designer wishes to compose them in a "flat" way, we define $\mathbf{doStep}^{\mathcal{A}\|\mathcal{B}\|\mathcal{C}}$ as above with the difference of having three instead of two source states (three instead of two histories, etc.). Accordingly, $\mathbf{step}^{\mathcal{A}\|\mathcal{B}\|\mathcal{C}}$ then interleaves three instead of two transitions.

If, on the other hand, the designer wishes to compose first $\mathcal{A}$ with $\mathcal{B}$ and then $\mathcal{A}\parallel\mathcal{B}$ with $\mathcal{C}$, he may first compose $\mathcal{A}$ and $\mathcal{B}$ according to our translation scheme. This yields a parallel step predicate $\mathbf{step}^{\mathcal{A}\|\mathcal{B}}$ which can, iteratively, be combined with $\mathbf{step}^{\mathcal{C}}$, yielding the desired parallel composition $\mathbf{step}^{(\mathcal{A}\|\mathcal{B})\|\mathcal{C}}$. The construction of $\mathbf{doStep}^{(\mathcal{A}\|\mathcal{B})\|\mathcal{C}}$ is then analogous to $\mathbf{doStep}^{\mathcal{A}\|\mathcal{B}}$ where, for instance, the first source state is invocated with a pair of source states (for $\mathcal{A}$ and $\mathcal{B}$ or $\mathbf{step}^{\mathcal{A}\|\mathcal{B}}$, respectively). The second source state then corresponds to the one in automaton $\mathcal{C}$. In terms of the respective traces, both translations yield the same semantics. It is evident that this scheme is directly applicable to more than three automata. Furthermore, it is consistent with component oriented system design. Actually, the code generator produces a parallel step predicate $\mathbf{step}^{\mathcal{C}_1\|\mathcal{C}_2\|\cdots\|\mathcal{C}_n}$ for each system structure diagram with n subcomponents $\mathcal{C}_1\parallel\mathcal{C}_2\parallel\cdots\parallel\mathcal{C}_n$ and a $\mathbf{step}^{\mathcal{C}_i}$ predicate for each subcomponent. Thus, the hierarchy in the structure of AUTOFOCUS models is preserved and directly reflected in the generated CLP code.

Aggregation and dangling ports. Imagine we want to aggregate two components into a new one, where some input ports of the inner components are directly connected to the respective input ports of the new component, and some of the inner input ports are dangling (this happens if components are reused and the total functionality is not needed). We hence want to identify a subset of the internal ports with the external ports, which can easily be achieved by unification of the corresponding history variables. The dangling ports always get no signal (which actually is a signal, namely $(no\_msg,\_)$). These remarks also hold if we exchange input with output channels. Finally, signals at dangling output channels are ignored.

Furthermore, in our system, we implemented channel ramification, i.e., copying of channels, and identification mechanisms. The translation is, again, straightforwardly achieved by adequately identifying the corresponding variables.

Fig. 5 shows the generated code for our example (which has been modified for presentational purposes).

$step^b(init^b, (L_i, L_i'), ((\top, TLi), \_), ((\top, true), (\top, TLi)), start^b, on^b) \Leftarrow$
$\quad L_i'\# = TLi.$
$step^b(on^b, (L_i, L_i'), (\_, (\top, true)), ((\top, false), (\top, L_i)), switchOff^b, off^b) \Leftarrow$
$\quad L_i'\# = L_i.$
$step^b(off^b, (L_i, L_i'), (\_, (\top, true)), ((\top, true), (\top, L_i)), switchOn^b, on^b) \Leftarrow$
$\quad L_i'\# = L_i.$
% plus idle transitions ...
$step^t(wait^t, (L_t, L_t'), (\top, TLn), (\bot, \_), set^t(TLn), run^t) \Leftarrow L_t'\# = TLn.$
$step^t(run^t, (L_t, L_t'), \_, (\top, true), timeout^t, wait^t) \Leftarrow L_t\# = 0, L_t'\# = L_t.$
$step^t(run^t, (L_t, L_t'), \_, (\bot, \_), tick^t, run^t) \Leftarrow L_t\# > 0, L_t'\# = L_t - 1.$
% plus idle transitions and code for the driver (step^d)...

$stepStructure((S_1, S_2, S_3), ((L_i, L_i'), (L_t, L_t'), (L_b, L_b'), (C_{bp}^-, C_{bp}),$
$\quad (C_{set}^-, C_{set}), (C_t^-, C_t)), C_i, C_{bs}, (T_1, T_2, T_3), (D_1, D_2, D_3)) \Leftarrow$
$\quad step^b(S_1, (L_i, L_i'), (C_i, C_t^-), (C_{bp}, C_{set}), T_1, D_1),$
$\quad step^t(S_2, (L_t, L_t'), (C_{set}^-), C_t, T_2, D_2),$
$\quad step^d(S_3, (L_b, L_b'), (C_{bp}^-), C_{bs}, T_3, D_3).$
$\%copy$
$doStepStructure(S, (AL_i, AL_t, AL_b, AL_{bp}, AL_{set}, AL_t), AI_i, AO_{bs}, AHst_{trans},$
$\quad Clock, Clock_{max}, S, (AL_i, AL_t, AL_b, AL_{bp}, AL_{set}, AL_t), AI_i, AO_{bs}, AHst_{trans}) \Leftarrow$
$\quad Clock\# \leq Clock_{max}.$
$\%go$
$doStepStructure(S, ([L_i|Ls_i], [L_t|Ls_t], [L_b|Ls_b], [L_{bp}|Ls_{bp}], [L_{set}|Ls_{set}], [L_t|Ls_t]),$
$\quad I_i, O_{bs}, Hst_{trans}, Clock, Clock_{max}, S,$
$\quad (AL_i, AL_t, AL_b, AL_{bp}, AL_{set}, AL_t), AI_i, AO_{bs}, AHst_{trans}) \Leftarrow$
$\quad Clock\# \leq Clock_{max},$
$\quad stepStructure(S, ((L_i, L_i'), (L_t, L_t'), (L_b, L_b'), (L_{bp}, L_{bp}'), (L_{set}, L_{set}'), (L_t, L_t')),$
$\quad CI_i, CO_{bs}, Trans, D),$
$\quad doStepStructure(D, ([L_i', L_i|Ls_i], [L_t', L)t|Ls_t], [L_b', L_b|Ls_b], [L_{bp}', L_{bp}|Ls_{bp}],$
$\quad [L_{set}', L_{set}|Ls_{set}], [L_t', L_t|Ls_t]), [CI_i|I_i], [CO_{bs}|O_{bs}], [Trans|Hst_{trans}],$
$\quad Clock + 1, Clock_{max}, S,$
$\quad (AL_i, AL_t, AL_b, AL_{bp}, AL_{set}, AL_t), AI_i, AO_{bs}, AHst_{trans}).$

Figure 5: Timer/blinker code. $\bot$ is no_msg, $\top$ is msg.

4 Leaving finite domains via CHR

For the sake of brevity, our translation concentrated on finite (integer) domains. This section shows how arbitrary domains can be involved.

First of all, it is worth noting that the results from the previous section vacuously hold for CLP($\mathcal{Q}$,$\mathcal{R}$) if the underlying constraint solver supports these domains (which is the case for most CLP systems, e.g., Eclipse, Sicstus, GnuProlog[1]). Note also that without appropriate abstraction techniques, verification techniques such as model checking cannot cope with such systems because of a priori infinite state spaces. However, we see testing and such techniques as complementary rather than rivaling approaches. The $\mathcal{FD}$ constraint operators simply have to be replaced by the corresponding $\{\mathcal{Q}, \mathcal{R}\}$ operators. These domains exhibit the advantage of a "free" abstraction into intervals (i.e., the system calculates with intervals rather than points). Obviously, model checkers without similar abstractions are not able to deduce any information about a system involving such data types.

[1] www.ecrc.de/eclipse, www.sics.se/sicstus.html, pauillac.inria.fr/~diaz/gnu-prolog

Recursive types. In addition, during the task of modeling a system, the need for user-defined data types frequently occurs. AUTOFOCUS allows for the definition of polymorphic functional data types which may be recursive. While the focus of [21, 24] is on a general translation of types, we discuss how (recursive) data types can be translated into constraints in the remainder of this section. Domains other than $\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$ require the definition (and automatic generation) of appropriate constraint handlers. Constraint Handling Rules (CHR [7]) allow for the definition of constraint handlers on arbitrary domains. In the context of testing, if at the end of the simulation/generation process the constraint store contains delayed type constraints, the latter are used for the determination of actual test sequences. Since AUTOFOCUS is equipped with a type checker, run-time type checking as described in the next paragraph is not necessary for simulation purposes. However, it is necessary when test cases are to be constructed: Input values of a correct type have to be generated.

Example: Peano terms. The most simple example are finite enumeration types. Since they are comprised in the following example, we omit their explicit discussion. Consider the timer of section 2. We want to maintain its functionality while replacing the finite integer domain by Peano terms (s/0 terms). The obvious idea is to encode these terms by predicates $peano(z).$ and $peano(s(X)) \Leftarrow peano(X)$. However, whenever these rules are evaluated with an unbound variable as argument, the program will not terminate. We thus have to move the handling of data types from the Prolog part into the constraint logic part.

This is easily done by defining CHRs p_ax1 @ $peano(z) \iff true$ and p_ax2 @ $peano(s(X)) \Leftrightarrow peano(X)$. Note that z demonstrates how enumerative types are implemented. For efficiency reasons (and for an easier handling of the remaining constraint store which will be used for, e.g., the generation of test cases), we add a failing constraint: p_fail @ $peano(X) \Leftrightarrow X \neq z, X \neq s(\_)|fail.$ For the timer, we need a predicate leq which corresponds to $\leq$ on integers. The functional definition $z\ leq^f\ \_ \rightarrow\ true,\ s(\_)\ leq^f\ z \rightarrow\ false,\ s(X)\ leq^f\ s(Y) \rightarrow X\ leq^f\ Y$ is automatically compiled (basically by flattening the function definition) into its constraint logic counterpart as shown in Figure 6. In order to add more knowledge to the constraint system,

$$
\begin{array}{c}
leq1\ @\ leq^{cl}(z, X, tt) \Leftrightarrow peano(X)|true. \\
leq2\ @\ leq^{cl}(z, X, Y) \Leftrightarrow Y \neq tt|fail. \\
leq3\ @\ leq^{cl}(s(X), z, ff) \Leftrightarrow true. \\
leq4\ @\ leq^{cl}(s(X), z, Y) \Leftrightarrow Y \neq ff|fail. \\
leq5\ @\ leq^{cl}(s(X), s(Y), Z) \Leftrightarrow peano(X), peano(Y)|leq^{cl}(X, Y, Z). \\
fail\ @\ leq^{cl}(X, Y, Z) \Leftrightarrow X \neq z, X \neq s(\_); Y \neq z, Y \neq s(\_); Z \neq tt, Z \neq ff|fail.
\end{array}
$$

Figure 6: Function $\leq$ on integers becomes CHR leq.

we could add the constraint *fail* and its symmetric counterpart. What remains to do is replace the constraint operators on integers by the newly defined ones and to add a type check in every predicate for the in- and output channels: *peano(I)* or *peano(O)*, respectively. In this context, it is noteworthy that in his decision to design the Temporal Logic of Actions [17] as an untyped language, Lamport argues that "types add a great deal of complexity to a logic" and render reasoning about programs much more difficult.

The example of Figure 5 can easily be modified in order to work on Peano terms: "X+1" becomes s(X), # $\leq$ becomes $leq/3$ with the corresponding arguments. The translation of arbitrary functions is done in exactly the same way as the translation of types by flattening expressions. This is the subject of the following paragraph.

Generalization and Types. We have seen that the timer example of section 2 can easily be transformed into a similar one by replacing constraints on integers by constraints on recursively defined s/0 terms. In this paragraph we describe the integration of arbitrary types into our framework.

Since during design of the translation we always have borne in mind the generation of test cases, we needed to take some care in binding free variables. Consider a transition $X?Z$ where X is an external input channel which checks if there is some value on channel X. We have to make sure Z is of the type of channel X. This motivates the following translation of types into constraint logic. Consider a data type declaration $t(\vec{\alpha}) = c_1(\vec{\beta_1})|\ldots|c_n(\vec{\beta_n})$ where $\vec{\alpha}$ is a sequence of type variables, $\mathcal{V}ar(\vec{\beta_i})$ is a subset of $\vec{\alpha}$, and $\vec{\beta_i}$ are terms composed of constructors, type operators, and type variables. *Type operators* are constants that appear on the left hand sides of data type declarations. *Constructors* are constants that *exclusively* appear on right hand sides (in $list(\alpha) = nil|c(\alpha, list(\alpha))$, for instance, $list$ is a type operator whereas nil and c are constructors). Type operators as well as constructors may be of arbitrary (finite) arity. Aliasing is not allowed (e.g., $data\ nat = int$), but can be simulated with additional constructors.

We now define a constraint predicate $isType/3$ for a representative constructor $c(\vec{\beta})$ where $isType(TName, TV, Const)$ should be read as "$Const$ is of type $TName$ with type variables TV". First of all, we have to flatten recursive types. Call a subterm minimal iff its head symbol is a type operator and no proper subterm contains a type operator. We iteratively replace all minimal subterms in $c(\vec{\beta})$ by fresh variables and maintain a list that associates the fresh variable X_i with its (parameterized) type operator, $\tau_i(\vec{\gamma_i})$ where $\vec{\gamma_i}$ is the sequence of variables that are occurring in the subterm that is to be replaced (with possibly multiple occurrences; for a monomorphic τ_i, $\vec{\gamma_i}$ is the empty sequence). If $\vec{\gamma_i}$ contains constructors, the type declaration is illegal since type operators and type constructors have been mixed. Let $c(\vec{\beta'})$ denote the result of this operation which terminates when no more type operators occur in the term. If the corresponding type operator T is parameterized by type variables $\vec{\alpha}$, we define $\mathbf{isType}(\mathbf{T}, \vec{\alpha}, \mathbf{c}(\vec{\beta'})) \Leftrightarrow isType(\tau_1, \gamma_1, X_1) \wedge \ldots \wedge isType(\tau_k, \gamma_k, X_k)$. If there are ℓ definitions on the right hand side of T's definition, there are ℓ such constraint predicates. What remains to do is add a failing constraint that asserts that the constraint predicate fails if the conditions can not be fulfilled. Figure 7 shows an example for Peano terms as well as lists.

```
peano1 @ isType(peano, _ , z) <=> true.
peano0 @ isType(peano, _ , s(X0)) <=> isType(peano, _, X0).
failpeano @ isType(peano, _, _C) <=>
  (_C\=z, _C \= s(X0)
  ; nonvar(_C), _C = s(X0), \+ isType(peano, _, X0)
  ; nonvar(_C), _C=z, \+ true) | fail.
list1 @ isType(list, A, nil) <=> isType(_, _, A).
list0 @ isType(list, A, c(A, X0)) <=> isType(list, A, X0),
          isType(_, _, A).
faillist @ isType(list, A, _C) <=> (_C\=nil, _C \= c(A, X0)
   ; nonvar(_C), _C = c(A, X0),
     \+ (isType(list, A, X0), isType(_, _, A))
   ; nonvar(_C), _C=nil, \+ isType(_, _, A)) | fail.
```

Figure 7: Types as automatically constructed CHRs

Finally, the existing finite domain on natural numbers is incorporated into this schema by using special CLP predicates: Within Eclipse, for instance, we may define $\mathcal{FD}$ by FD @ $isType(int, \_, X) \Leftrightarrow dvar\_attribute(X, \_)|true$ and $FDfail$ @ $isType(int, \_, X) \Leftrightarrow$

$\backslash + dvar\_attribute(X, \_) | fail$.

For the sake of simplicity, we ignored built-in selector or projection functions, e.g., hd and tl in $list(\alpha) = nil | c(hd : \alpha, tl : list(\alpha))$. Their translation, however, is straightforward (e.g., $hd(c(X, Y), R) \Leftrightarrow R = X | true$). Note that in this case we want to bind variable R.

5 Conclusions and Future work

Simulation and testing are the most widely used validation techniques for arbitrary sytems. 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.

The code generator has been used for the determination of output traces of several case studies, e.g., NASA's Mars Polar Lander [22], a system consisting of event-discrete as well as time-continuous components. Our current work concentrates on the determination of test sequences (I/O traces) on the grounds of partial I/O specifications via Sequence Charts [15, 8, 9]. There is some evidence that by encoding Sequence Charts as constraint systems and using these constraints while executing the model, the search space for test sequences can be significantly reduced (a-priori pruning of the search tree). The existing Sequence Chart editor in AutoFocus is being extended: We currently work on a semantically clear integration of specification concepts for transitions and negation within Sequence Charts. The existing editor allows for the specification of test cases for testing based on Bounded Model Checking [24]. To date, Sequence Charts have to be translated into CLP by hand. The automatic translation is the subject of ongoing work. Furthermore, the CLP code from within AutoFocus is the basis for work on the generation of test cases for hybrid systems that are not discretized [20] in an ad-hoc manner. Future work also includes the automatic evaluation of 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 testing process.

Another important isue is the genral definition of what a "meaningful" test case is. Finally, the use of functional logic languages such as Curry [11] seems to be particularly suited for our purposes. However, there are no efficient non-proprietary implementations yet that support different CLP libraries and Constraint Handling Rules.

Acknowledgment. We would like to thank Oscar Slotosch and Thomas Stauner for fruitful discussions and clarifying comments on an earlier version of 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] K. Apt and R. Bol. Logic Programming and Negation: A Survey. *J. Logic Programming*, 19/20:9–71, 1994.

[3] 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.

[4] 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.

[5] 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.

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

[7] T. Frühwirth. Constraint Handling Rules. In A. Podelski, editor, *Constraint Programming: Basics and Trends (LNCS 910)*, pages 90–107. Springer Verlag, 1995. www.pst.informatik.uni-muenchen.de/personen/fruehwir/chr-intro.html.

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

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

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

[16] J. Jaffar and M. Maher. Constraint Logic Programming: A Survey. *J. Logic Programming*, 20:503–581, 1994.

[17] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[18] 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.

[19] O. Müller and T. Stauner. Modelling and verification using Linear Hybrid Automata. *Mathematical Computer Modeling of Dynamical Systems*, 6(1):71–89, March 2000.

[20] I. Péter, A. Pretschner, and T. Stauner. ROOM for Hybrid Systems: A Formal Grasp, 2000. Submitted to Integrated Formal Methods (IFM'00).

[21] J. Philipps and O. Slotosch. The quest for correct systems: Model checking of diagrams and datatypes. In *Proc. IEEE Asian Pacific Software Engineering Conference (APSEC'99)*, pages 449–458, 1999.

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

[23] 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.

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