Using Constraint Logic Programming for Software Validation*

Angelo E. M. Ciarlini and Thom Frühwirth

The Laboratory of Formal Methods, Departamento de Informática
Pontifícia Universidade Católica do R.J.
Rua Marquês de S.Vicente 225, 22.453-900 - Rio de Janeiro, Brazil
angelo@inf.puc-rio.br
and
Computer Science Institute
University of Munich
Oettingenstr. 67, D-80538 München, Germany
fruehwir@informatik.uni-muenchen.de

Abstract. Validation and testing are expensive tasks in the software development cycle. In order to save time while validating and testing a new system, we propose using logic programming together with constraint solving for the symbolic execution of formal specifications (like hybrid automata or statecharts) of concurrent systems. In order to validate a system, desired properties can be specified in a fragment of first order temporal logic relating variable values at different times. As a result, all possible paths, that satisfy the constraints given initially, are obtained together with the corresponding necessary and sufficient constraints. Using this approach, one can also compute backwards. One can, for example, derive possible ranges of the input variables, be they continuous or discrete, that produce specific, given ranges (i.e. constraints) on the output variables. Since we can think about ranges instead of specific values, it is easier to verify properties and choose good values to test the final code. We implemented these ideas in a prototype in the DExVal project, which is part of the ARTS software development environment.

1 Introduction

Software verification and validation aims at determining whether the software requirements are implemented correctly and completely. Validation and testing are concerned with answering the question "Are we building the right thing?". The challenge in answering this question is how to come up with experiments that make sense and can reveal a maximum number of potential software bugs. Dynamic analysis techniques such as testing involve execution, or simulation, of software to detect errors by analyzing the response to input.

Model checking is a technique for analyzing finite state spaces to determine whether a given property holds or not. It has been applied successfully to hardware, and also, more recently, to software, see e.g. [2]. Since software is usually modelled having an infinite number of states and continuous variables, one tries to approximate it by finite state machines where continuous variables are abstracted into discrete (boolean) ones. However, model checking may have the so-called state explosion problem, because every state and every variable value has to be tried, resulting in an exponential blow-up in the number of states.

* This work is supported by the Brazilian-German CNPQ/GMD project DExVal (Derivation of meaningful Experiments for Validation).

The model (a state machine) represents the important features of a software under consideration. In model checking, desired properties of the software are usually expressed in a variation of first order predicate logic, typically branching time temporal logics such as CTL (Computation Tree Logic) [6]. In such logics, the truth values of predicates are time-dependent. The model is "checked" to see if the desired properties hold in the model. The result is either a claim that the property holds or a counterexample.

Our goal is to symbolically execute and validate formal specifications given as concurrent transition systems (state machines, automata [11]), e.g. hybrid automata [1, 10] or statecharts [9], even in the presence of continuous variables and non-linear expressions. The idea is to make these specifications executable by translating them into a *constraint logic programming* (CLP) language.

It is straightforward to express the transition system in a CLP language, however actions (variable assignment) and concurrency (between different automata) need more thinking. Logical rules describe transitions, where both preconditions (guards) and actions (variable assignments) of the transition are mapped into constraints. Constraints are basically first order predicates for which efficient solvers are available. Constraints enable us to represent possible infinite relations, e.g. $X \leq 5$, finitely.

As opposed to standard finite model checking, the use of constraint solving allows one to work with both discrete and continuous variables directly, and more general, without giving specific values to variables at all. Instead, arbitrary conditions (constraints) on any variables in any state describing properties that should be obeyed during the execution can be given as constraints by the user. We use a first order temporal logic to express user constraints on the variables and its values occurring in different states.

Such constraints are solved and simplified together with new constraints introduced during transitions. The computation follows any path in the transition system, provided that the constraints accumulated in each transition of the path, together with user given constraints, are satisfiable. All possible valid runs are therefore taken into account. The result of a run consists of a path and some time-dependent constraints that must be satisfied by the variables of the automaton at different points in time. The user can scan the result in order to understand all the assignments (to the variables) which enable the following of its path.

In short, instead of checking that a temporal formula (property) given by the user is implied by the system, we compute one or - on request - all paths that are consistent with the user's formula. The implication problem for a property can be expressed by checking that there are no paths that are consistent with the negation of the property.

Since constraints can be imposed on the (variables of the) final state of the computation, one can enumerate all runs that lead to the same final state. The disjunction of the constraints for all possible runs logically describes the conditions under which the final state can be reached. From these conditions, tests can be derived. More concretely, if the user specifies, among other initial constraints, some constraints on the ranges of output values, he or she can get the corresponding ranges of input values that allow paths from an initial state to a final state. In this way, our approach forms a basis for testing and validation, where the output values, or ranges of values, must be known.

In HYTECH [1, 10], an implementation of linear hybrid automata in Mathematica, a procedural language is used to formulate desired properties to be checked (queries). The underlying temporal logic, ICTL is inspired by the duration calculus [5] and CTL [6]. It has clocks and integrators (special counters that increment only if a certain property is true). However, the HYTECH system cannot be easily employed to express all kinds of queries relating variables at different times.

In our tool prototype DExVal [4], expressing properties in a declarative way as constraints, including constraints relating any variables at any (different) times, is straightforward.

Any constraint that can be handled by the constraint solvers, embedded within the CLP language, can be used in DExVal. In particular, non-linear constraints can be handled by some available constraint solvers. In HYTECH, non-linear functions can be approximated by piecewise linear functions, at the cost however of introducing more transitions.

A main characteristic and source of efficiency of CLP languages is that constraint solving and search are interleaved, performed on-the-fly, i.e both constraint solving and search are incremental. This is unlike standard approaches of model checking where all the constraints (propositions) and choices have to be handled at once resulting in huge amounts of data.

In the next section we briefly introduce hybrid automata and constraint logic programming. We then describe the DexVal project and tool that aims to derive meaningful experiments for software validation from symbolic executions in a constraint logic programming language. Section 4 gives the algorithm and implementation used for symbolic execution. We then introduce the example of a bathroom boiler scenario. We end with conclusions and an outlook for future work.

2 Preliminaries

2.1 Hybrid Automata

Hybrid automata [1, 10] have been introduced for modelling mixed discrete - continuous systems. A hybrid automaton is a transition system whose states contain descriptions of continuous activities and whose transitions are discrete and labeled with guarded actions. The state of the automaton changes either instantaneously through a discrete transition associated with system actions or, while time elapses, through a continuous environment activity. The treatment of continuous activities inside states and the absence of hierarchical abstraction into superstates, are the main main features which distinguish it from related approaches such as statecharts [9].

A variety of terminology is used in different papers on hybrid automata [1, 10]. For ease of reference, we have employed the more widely used terminology of transition systems, giving alternative synonymous in brackets.

A *hybrid automaton* consists of a finite number of each of the following components:

- (Data) variables. Typically real-valued, in our case also integer or boolean are possible.
- States (also called: control locations, control modes). There is one initial state and one or more final (terminal) states. States consist of three components:
  - Name.
  - Invariant *inv* (invariant conditions, location invariants)- a constraint on variables. The automaton may reside in the state as long as the invariant holds.
  - Iteration *iter* (continuous activities, flow conditions)- activities (assignments) that specify the new values of the variables based on their values at the last clock. The values of the variables change in this way while the automaton resides in this state. In our case, continuous activities are modelled as variable assignments instead of giving derivatives of the assigned variable.
- Transitions (control switches). They consist of four components:
  - Source state.

- Target state (destination state).
- Guarded actions (discrete actions, jump conditions, guarded assignments, guarded commands). If the guard (guarding condition) holds, the transition can take place and may change values of variables by executing the specified action (assignments). There may also be transitions from final states, typically in non-terminating systems, where final states serve as check-marks.
- Events (synchronization labels)- used to synchronize concurrent automata. We do not use them in this paper, if needed they could be modelled as boolean variables as e.g. in [2].

In a state the values of variables conceptually change continuously, however the iterations actually happen step by step when the clock increases. Iterations takes time, while transitions are instantaneous.

In a *timed hybrid automata* all the automata are synchronized by a machine clock that causes iterations and transitions of states. A clock is a variable that always increases with the rate of one time unit in each iteration and all transitions leave its value unchanged (or reset it to zero). The system modifies the values of the variables according to the state of the automaton at the last clock. Variables that do not occur in activities or actions remain unchanged.

A hybrid system typically consists of several interacting concurrent hybrid automata that coordinate through shared variables and events. All the automata make their modifications simultaneously, i.e. a maximal set of mutually consistent continuous activities and enabled transitions is performed at the same time. We assume that a variable can be modified by only one automaton. Otherwise two concurrent transitions could modify the same variable.

A *state transition diagram* is a pictorial representation of the operation of some given automaton. It is a directed graph where the vertices represent states, which are inscribed with invariants and continuous activities, and the arcs represent state transitions, which are labelled with guarded actions (see the upcoming Figures).

In this paper, we use concurrent timed hybrid automata, in which we allow arbitrary (non-linear) invariants and continuous activities instead of restricting ourselves to linear ones.

2.2 Constraint Logic Programming

Constraint logic programming (CLP) [12, 7, 13] combines the advantages of logic programming (e.g. Prolog) and constraint solving. In logic programming, problems are stated in a declarative way using rules to define relations (predicates of first order logic). Rules describe the conclusions that can be reached given certain premises. A logic programming system searches for all solutions by systematically trying all possible rules using chronological backtracking. In constraint solving, efficient special-purpose algorithms are employed to solve sub-problems involving distinguished relations referred to as constraints.

The key aspect of CLP is the tight integration between a deterministic process, constraint evaluation, and a nondeterministic process, search. During program execution, the logic program incrementally sends constraints to the constraint solver (CS). The CS tries to solve the constraints. The results from the CS cause a priori pruning of branches in the search tree spawned by applying rules in the program. Unsatisfiability of the constraints means failure of the current branch, and thus reduces the number of possible branches, i.e. choices, to be explored via backtracking.

For instance, if one has already accumulated the constraints $X + Y < 5$, $X > 0$ and $Y > 0$ and an inequality constraint solver is being applied, when the variable X is bound to 6, the execution does not need to continue in this branch. Instead, the system will backtrack, undo the binding for X, and explore the next branch.

The syntax of CLP is reminiscent of first order predicate logic, however predicate symbols start with lower case letters and variable names with upper case letters, ':-' stand for implication $\leftarrow$ and ',' for conjunction. For example,

```
fac(N,1) :- N<=1.
fac(N,N*F) :- N>1, fac(N-1,F).
```

is a recursive definition of the factorial function, i.e. `fac(N,M)` holds if $M = N!$. `N<=1` and `N>1` are constraints. A goal (query) `fac(2,B)` will unfold with the second rule into the conjunction `2=N, B=N*F, N>1, fac(N-1,F)`. The constraints are sent to the constraint solver, where they are simplified into `N=2, B=2*F`. `fac(N-1,F)` is recursively unfolded with the first rule yielding `N=2, B=2*F, N-1<=1, F=1`, which simplifies into `B=2` (omitting intermediary variables). Since no more predicates are left to be unfolded, the computation terminates with the answer (solution) `B=2`. If another answer is requested, the system backtracks to the state `N=2, B=2*F, fac(N-1,F)`. It then uses the second rule to unfold `fac`, but this time the resulting constraints are inconsistent, so the answer is `no`.

3 The DExVal Approach

The DExVal (Derivation of meaningful Experiments for Validation) project at the LMF-DI department of PUC-Rio (Pontifícia Universidade Católica do Rio de Janeiro) together with the software engineering department of LMU (Ludwig-Maximilians Universität) [4] is situated within the ARTS project. ARTS is a large scale effort to put industrial software development on a solid formal basis funded in part by Siemens Telecomunicacoes, Brazil. DExVal aims to contribute towards this goal by providing a tool that helps the validation and testing tasks of software derived from formal specifications.

In the ARTS project, one generates the final code of a system directly from object oriented specifications that are translated to a formal level. When an application is being developed inside ARTS, the programmer defines automata to describe the behaviour of the objects of the application.

The DExVal approach basically consists of two phases (in this paper we concentrate on the first phase). In the first phase, means are provided to symbolically execute formal specification given as a transition system (hybrid automata or statecharts). In the second phase of the tool one derives test cases for experiments validating the final code that should implement the automaton specification.

3.1 The DExVal Tool

The DExVal tool (short: DExVal) consults the definitions of the automaton to check whether from a certain initial state it is possible to reach a final state without violating any properties given by the user checking the application.

The user can specify properties as contraints that should hold for all states (universal quantification) and contraints that should hold for at least one state (existential quantification). Constraints include:

- values or ranges of input variables;
- values or ranges of output variables;
- values or ranges of variables at intermediate states;

In general, arbitrary constraints (e.g. linear and non-linear equalities and inequalities) among different variables at different times can be specified. This is done by

using the temporal modalities "since", "until", "always in the future", "always in the past", "sometime in the future" and "sometime in the past".

Therefore, the user can express complex properties such as:

- "since X>Y, Z=1":
  `all(T,since(T,T1,x:T1>y:T1,z:T1=1)`
- "for all states, X has a higher value than its value in the previous state":
  `all(T,x:T>x:(T-1))`
- "X is never higher than Y and less than Z at the same time":
  `not(exists(T,(x:T>y:T, x:T<z:T)))`
- "if, at some time, X>Y, then at most 5 seconds later Z=1":
  `all(T,implies(x:T>y:T,`
  `exists(T1,(systime:T1-systime:T<5, systime:T1>systime:T,z:T1=1))))`

DExVal computes all paths that do not violate the user constraints. Moreover, DExVal informs the user of constraints on certain variables (specified by the user) that must hold to allow the corresponding path to be followed. For example, the user can specify that an output variable, say `X`, is higher than 5 and request that DExVal find all paths and corresponding values for an input variable `Y`. DExVal might inform the user, for instance, that `Y` is less than 10 for a certain path.

3.2 Validation and Testing

Figure 1 shows how we perceive that DExVal can help validation and testing tasks. The first use of DExVal is to obtain, given some assertions (as constraints) on transition paths and timing, ranges in which the values of relevant input variables at design level should be. Since the user has all the ranges, he or she can check whether these ranges are as expected. If at least a part of a range is not as expected, then DExVal can be activated again, receiving as input the detected wrong ranges at the initial state. The user can analize the resulting transition paths and intermediate values of variables in order to discover where the error occurs. After the automata have been corrected, the user can restart the validation process.

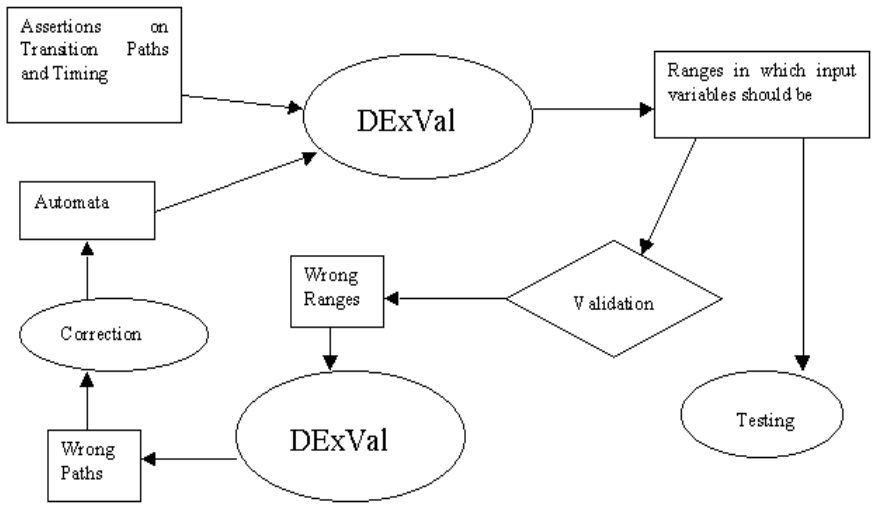

Fig.1. Validation and Testing in DExVal

If the user wants to prove that a certain property does not hold, he or she can execute DExVal providing a user constraint corresponding to the negation of this property. If DExVal finds a path, it is a counterexample for that property. If there is no counterexample, the property holds for all paths.

Since DExVal can provide final constraints on input variables, it is also a useful tool for testing. Suppose, for instance, that DExVal computes that an input variable `X` is either in between 10 and 20 and the output variable `Y` is higher than 100 or `X` is less than 10 or higher than 20 and `Y` is less or equal 100. This result indicates that it is probably better to test the code for `X`=1, 10, 15, 20 and 30 than to test for `X`=12, 13, 14, 15 and 16.

4 Implementation - Symbolic Execution using CLP

In a previous experiment, we coded a small statechart from a real life application at the car-manufacturer BMW, Munich, into the constraint logic programming language Eclipse using its CHR library [8]. The statechart is about an automatic shift gear and its computations are sequential only. The code for the transition system is about 2 pages, the code implementing the example is about 3 pages.

Based on this experience, we implemented the DExVal tool in the CLP language Sicstus Prolog [3]. DExVal implements a concurrent execution of a kind of hybrid automata.

4.1 Preparation Stage - Translating Properties

A computation in DExVal proceeds as follows: At the preparation stage for the symbolic execution, the properties given by the user are translated to sets of simple constraints (sets of conjunctions and/or disjunctions). The references to variables at specific clocks are replaced by the corresponding logic variables. The constraints are inserted in the store of constraints to be incrementally solved by the constraint solver.The user should express constraints according to the following BNF-syntax and informal semantics:

```
Formula - F,F1,F2
F::= all(CV,F)               for all clocks CV, F is true
   | exists(CV,F)            there exists a clock CV when F is true
   | not(F)                  F is not true
   | implies(F1,F2)          if F1 is true then F2 is true
   | since(C,CV,F1,F2)       F1 is true for C'<CV<C and F2 is true for CV=C'
   | until(C,CV,F1,F2)       F1 is true for C<CV<C' and F2 is true for CV=C'
   | all_future(C,CV,F)      F is true for all CV>C
   | all_past(C,CV,F)        F is true for all CV<C
   | sometime_future(C,CV,F) F is true for at least one CV>C
   | sometime_past(C,CV,F)   F is true for at least one CV<C
   | (F1,F2)                 F1 and F2 are true
   | (F1;F2)                 F1 or F2 are true
   | EXP REL EXP             arithmetic equalities or inequalities

Clock variable - CV
CV::= PROLOG_VARIABLE        any string beginning with an uppercase letter

Clock - C
C::= CV | INTEGER_NUMBER     a variable (already quantified) or a number

Relation - R
R::= > | < | >= | =< | =
```

```
Expression  - E              arithmetic expression
EXP::= TERM | TERM + TERM | TERM - TERM | - TERM
TERM::= ELEM | ELEM * TERM | ELEM / TERM
ELEM::= (EXP) | VAR_VALUE | NUMBER

Data variable value - VAR_VALUE
VAR_VALUE::= VAR_NAME:C |  Value at C
             VAR_NAME:i |  Value at the initial state
             VAR_NAME:f    Value at the final state

Variable name - VAR_NAME
VAR_NAME::= PROLOG_ATOM      any string starting with a lowercase letter
```

DExVal translates user constraints like `all(X,v:X>20)` to expressions like `v:1>20`, `v:2>20`, `v:3>20`, ..., where `v` is the name of the variable followed by the clock.

4.2 Main Stage - Symbolic Execution

The symbolic execution of the automata is a search for paths for each concurrent automaton. Each path is a sequence of transitions or continuous activities between the initial state and the final state of the corresponding automaton. For each transition or continuous activity, the constraint solver is informed of new constraints stemming from guarded actions, invariants or iterations. If the store of constraints becomes inconsistent, the current branch fails and DExVal backtracks to try another branch of the search tree. The symbolic execution is performed according to the following algorithm.

Input

`NC` - number of clocks
`NA` - number of automata
`INIT_STATE` - array with the names of the initial state of each automaton
`FINAL_STATE` - array with the names of the initial state of each automaton
`USER_CONSTRAINTS` - properties defined by the user
`VAR_PROJECT` - variables which final constraints should be projected on

Internal data structures

`S[NC].AUTOMATON[NA]` - array with the states of the automaton and Prolog variables at each clock
`CS` - constraint store
`I, J` - counters
`CONST1, CONST2` - lists of constraints

Algorithm

```
Preparation Stage:
1. Create array S[NC].AUTOMATON[NA].
2. Translate properties in USER_CONSTRAINTS to constraints on Prolog
   variables defined in S and add them to CS.

Main Stage:
3. For I:=1 to NA do
     3.1. S[1].AUTOMATON[I].STATE:= INIT_STATE[I].
     3.2. S[NC].AUTOMATON[I].STATE:= FINAL_STATE[I].
4. For I:=1 to NC-1 do
     4.1. For J:=1 to NA do
```

```
        Do iteration corresponding to state
           S[I].AUTOMATON[J].STATE and add constraints to CS.
        If it is impossible to do an iteration of any automata,
        then fail and backtrack.
    4.2. For J:=1 to NA do decide next state
        If invariant (S[I].AUTOMATON[J].STATE) is true
           remain at the same state and add constraints to CS.
        Else choose one of the possible transitions from
           S[I].AUTOMATON[J].STATE to S[I+1].AUTOMATON[J].STATE
           and add constraints to CS.
        If it is impossible to reach the next state of any
        automata, then fail and backtrack.

Final Stage:
5. Collect remaining constraints in CONST1
6. Project CONST1 on variables in VAR_PROJECT, giving CONST2.
7. Translate constraints in CONST2 and path in S and print them.
8. If the user requests more answers, then fail, backtrack to 4.2.
```

4.3 Final Stage - Producing an Answer

When the final state of each automaton is reached, DExVal simplifies the accumulated constraints and projects them on the variables of interest as specified by the user. The resulting constraints typically specify, among other things, the ranges in which variables should be in order to cause the execution of that path. The user may be interested in expressing such constraints in terms of certain variables only. Therefore, DExVal projects the constraints in terms of these variables, according to the user specification. For instance, assume that the user is interested only in the values of variable `X` and the set of constraints is the following `X>2*Y+1 and Y>Z and Z>0`. After the projection, we have only the following constraint `X>1`.

The output shows a sequence of clocks and the corresponding states for each automaton. In addition to the states, the constraints remaining after the projection are also presented.

4.4 CLP Implementation

Most of the translation of automata into CLP is straightforward: States correspond to CLP predicates, transitions to CLP rules:

```
source_state(Clock,Variables) :- true.  % a final state

source_state(Clock,Variables) :-        % a transition
       guard as constraint on Variables,
       action as equality constraints on Variables and NewVariables,
       target_state(Clock,NewVariables).

source_state(Clock,Variables) :-        % a continuous activity
       invariant as constraint on Variables,
       iteration as equality constraints on Variables and NewVariables,
       source_state(Clock+1,NewVariables).
```

Invariants and guards in the automaton can readily be expressed as constraints. The complication was implementing variable assignments occurring in actions and activities in a declarative, logical manner, using equality constraints. To this end, we represent a single variable of the automaton as a sequence of logical variables so that each state comes with a fresh set of variables. The logical variables in subsequent states representing the same data variable are related to each other by constraints:

If the variable is written, i.e. assigned a new value, the constraint on the current variable is the assignment statement seen as an equality constraint, otherwise the logical variables of the subsequent states are equated to each other, their value stays the same (whatever it may be).

The above implementation scheme does not maintain the logical variables associated with a state. In order to be able to constrain them, they are stored in an array (see description above). A straightforward way to do this in CLP is to use a list:

```
source_state(Clock,cons(Vars,nil)) :- true.  % a final state

source_state(Clock,cons(Vars,cons(NewVars,Rest))) :- % a transition
       guard as constraint on Vars,
       action as equality constraints on Vars and NewVars,
       target_state(Clock,cons(NewVars,Rest)).

% Analogously for the rule for continuous activities
```

Concurrency (parallel composition) for automata is achieved by simply executing the conjunction of the involved automata (sharing the clock and the variables):

```
initial_state1(Clock,Vars), initial_state2(Clock,Vars).
```

4.5 Nondeterminism in the Prototype

While constraints already overcome the problem of choosing a value for a variable by working directly with ranges of possible values, there are still two sources of nondeterminism remaining, just as in other tools:

- the choice of the number of clocks for each run, since automata may express nonterminating processes;
- the choice of the next transition (since variables do not have unique values, typically several transitions from a given state are possible)

The nondeterminism is handled in CLP by exploring all choices, one after the other. This chronological backtracking can cause a combinatorial explosion in the number of possibilities that have to be explored.

Note that interleaving is not a source of nondeterminism as in related approaches, because in hybrid automata at each clock all automata do an iteration (and enabled transitions) and all the iterations take only one clock.

5 The Bathroom Boiler Scenario

There are already application examples of the use of the DExVal tool prototype in the domain of industrial processes involving production cells, but we identified the need for another class of examples that clearly illustrate the advantages of the approach taken in the DExVal project. In particular, as opposed to e.g. standard finite model checking, continuous variables can be used without problems in the constraint-based approach, since infinite ranges of values can still be represented by and reasoned with constraints. We therefore chose an example involving physical processes, inspired by the steam boiler problem [10]. The scenario (Fig. 2) involves a warm water boiler of a bathroom, with a automatic water pump, a heater and the possibility of someone taking a shower. Physical units are the amount of water (pumps add water, taking a shower reduces the water level) and the temperature of the water (that depends on the income and outflow of water and on the functioning of the heater).

Consequently, there are five variables: `heater`, `pump`, `shower`, `water_volume` and `temperature`. The boolean variables `heater`, `pump` and `shower` have value 1 if they are on and 0 if they are off. The initial values of each variable and all values of variable `shower` during the execution are input values. `water_volume` and `temperature` are continuous floating point variables representing the current volume of the water in the boiler and its temperature, respectively.

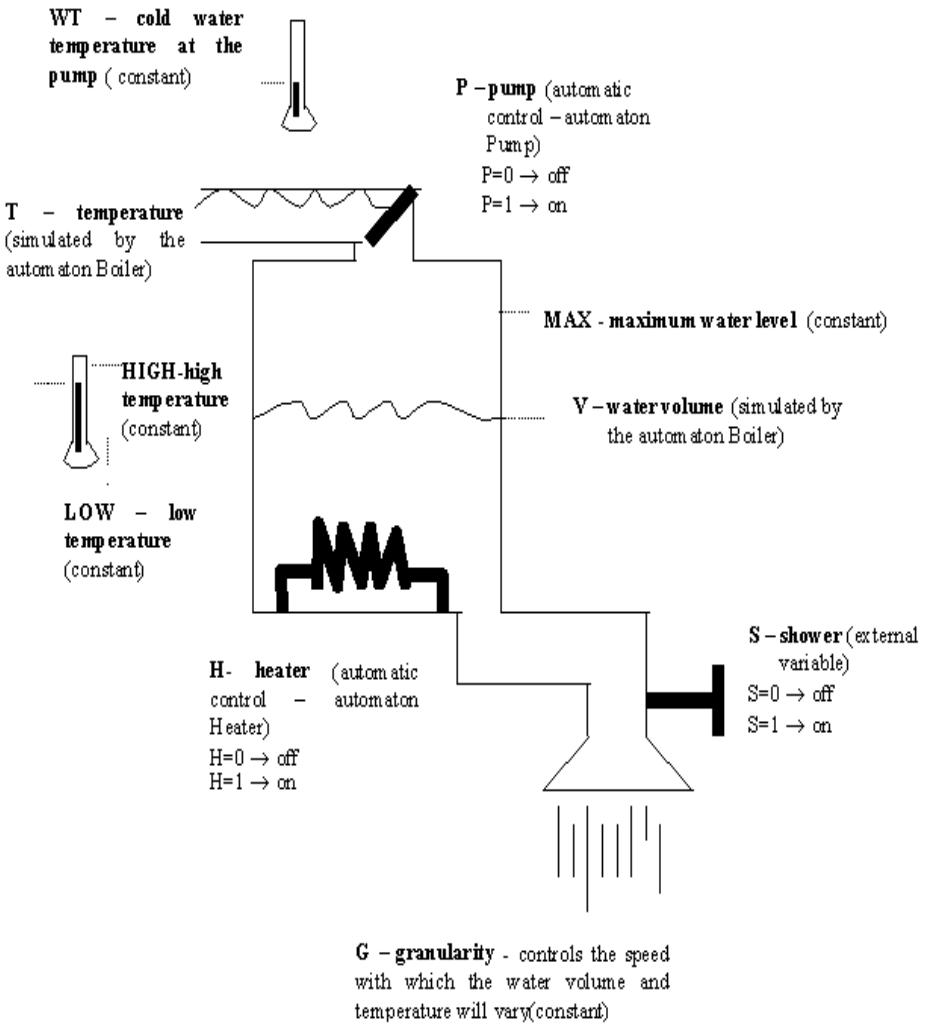

Fig.2. The Bathroom Boiler Scenario

We designed three concurrent automata to model our sample: `Heater` (Figure 3), `Boiler` (Figure 4) and `Pump` (Figure 5). `Heater` checks the temperature and decides if the heater should be on or off. `Pump` checks the water level and decides if the pump should be on or off. `Boiler` represents the physical process itself. It checks whether the heater, the pump and the shower are on or off and updates the water level and the temperature accordingly.

In the automaton `Boiler` there are three states: `empty`, `normal_heater_on` and `normal_heater_off`. The first one treats the special case of there being no warm water. The other states model the variation of the temperature according to the value of the variables `heater` and `pump`. All states control the water volume according to the values of variables `pump` and `shower`. The automaton `Heater` has also three states: `maintain`, `turning_on` and `turning_off`. If the temperature of the water is

in between the lower and upper limits, the automaton remains in state `maintain` and the variable `heater` is not modified. If the temperature is less (higher) than the lower (upper) limit, the variable `heater` is set to 1 (0) and the automaton spends one clock at state `turning_on` (`turning_off`) before returning to state `maintain`. The automaton `Pump` has just two states: `on` and `off`. If the water volume becomes less than a certain limit, the variable `pump` is set to 1 and the automaton goes to state `on`. When the volume becomes higher than the limit, the variable `pump` is set to 0 and the automaton goes to state `off`.

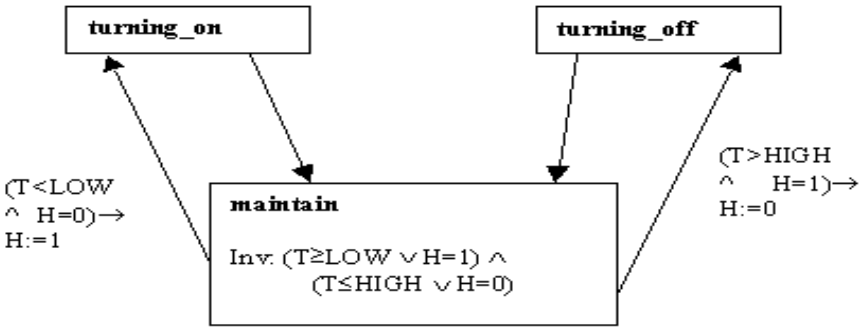

Fig.3. The Heater

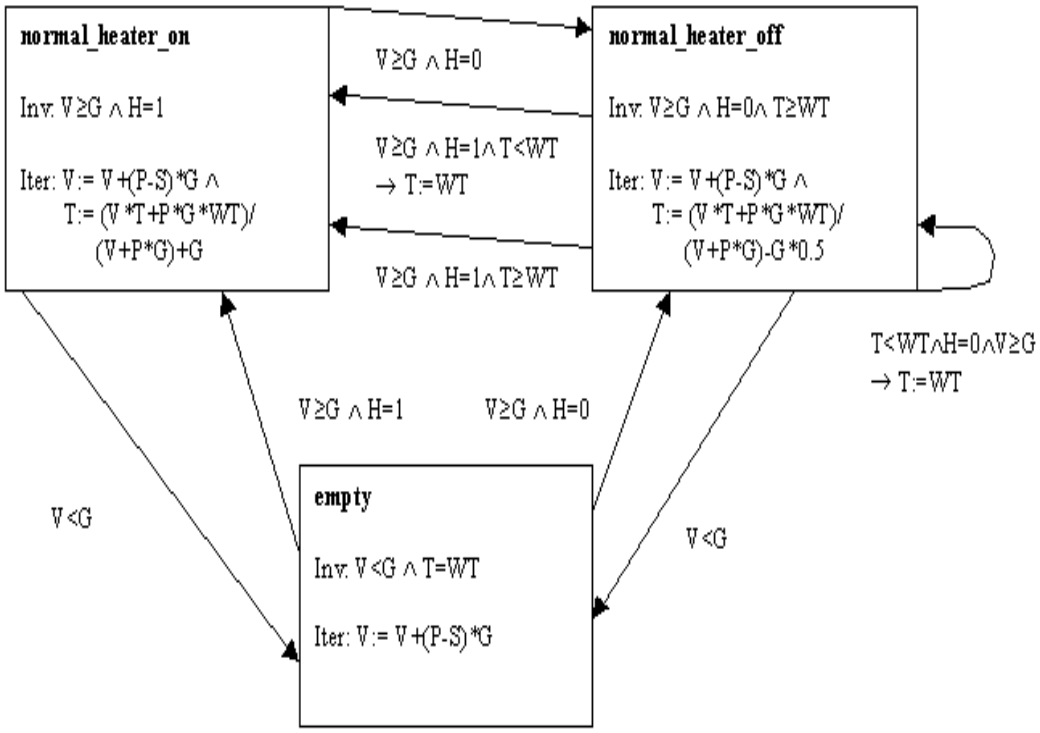

Fig.4. The Boiler

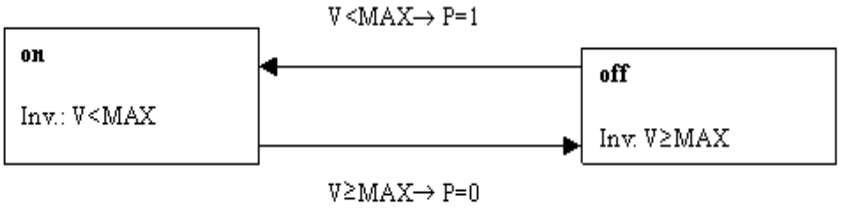

Fig.5. The Pump

With this simple example DExVal can verify interesting properties, such as:

– Minimal initial temperature for taking a warm shower without turning on the heater.

```
INPUT:
CONSTRAINTS:    heater:i=0.0, pump:i=1.0, water_volume:i=10.0,
                shower:i=1.0, all(X,shower:X=1.0),
                all(X,heater:X=0.0)
INITIAL STATES: Pump:on, Heater:maintain,
                Boiler:normal_heater_off
FINAL STATES:   not specified
CLOCKS:         5
VAR_PROJECT:    temperature:i (initial temperature)

OUTPUT:
Clock   Pump       Heater          Boiler
    1     on       maintain        normal_heater_off
    2     on       maintain        normal_heater_off
    3     on       maintain        normal_heater_off
    4     on       maintain        normal_heater_off
    5     on       maintain        normal_heater_off
temperature:1 > 47.18
```

There is only one path, `temperature:1 > 47.18` is a sufficient and necessary constraint on the initial temperature.

– Behaviour of the shower to increase continuously the water volume.

```
INPUT
CONSTRAINTS:    heater:i=0.0, pump:i=1.0, temperature:i=30.0,
                water_volume:i=6.0,
                all(X,water_volume:(X+1)>water_volume:X)
INITIAL STATES: Pump:on, Heater:maintain,
                Boiler:normal_heater_off
FINAL STATES:   not specified
CLOCKS:         5
VAR_PROJECT:    shower and water volume (all clocks)

OUTPUT:
Clock   Pump       Heater          Boiler
    1     on       maintain        normal_heater_off
```

```
    2     on       turning_on      normal_heater_on
    3     on       maintain        normal_heater_on
    4     on       maintain        normal_heater_on
    5     on       maintain        normal_heater_on
shower:[1..4]=0.0; shower:5=FreeVariable;
water_volume:1=6.0; water_volume:2=ρ.0; water_volume:3=10.0;
water_volume:4=12.0; water_volume:5=14.0
```

There is only one path, the shower must be off between clocks 1 and 4 (the constraint `shower:[1..4]=0.0`), at clock 5 it is unconstrained, and the water volume raises.

– Behaviour of the shower to maintain the water volume between the initial and final volumes (which are not specified).

```
INPUT
CONSTRAINTS:    heater:i=0.0, pump:i=1.0, water_volume:i=6.0,
                temperature:i=30.0,
                all(X,(water_volume:X>=water_volume:i,
                       water_volume:X=<water_volume:f))
INITIAL STATES: Pump:on, Heater:maintain,
                Boiler:normal_heater_off
FINAL STATES:   not specified
CLOCKS:         5
VAR_PROJECT:    shower and water volume (all clocks)
```

The Output consists of eight different paths and the corresponding sequences of values for the variables. One of them is exactly the path in the previous answer, the other paths are omitted for space reasons.

6 Conclusions

We proposed using logic programming together with constraint solving for the symbolic execution of formal specifications (such as hybrid automata or statecharts) of concurrent systems. For validating a system, desired properties can be specified as queries in a fragment of first order temporal logic relating variable values at different times. As a result, we obtain all possible paths that satisfy the constraints given initially, together with the corresponding necessary and sufficient constraints. We implemented these ideas in a prototype in the DExVal project, which is part of the ARTS software development environment.

Our preliminary results indicate the potential importance of constraint-based methods for software validation and verification. The advantage of using CLP is that it is easier and quicker to modify the prototype. Constraints enable us to represent possible infinite relations finitely. Any relation is possible, as long as there is a constraint solver for it. With the appropriate constraint solver, we can also solve certain non-linear constraints (e.g. the one computing the new temperature in our bathroom boiler scenario, see Figure 3). However, the limitation in current CLP languages is that constraint solvers are restricted to solving existentially quantified conjunctions of atomic constraints.

The encoding of a hybrid automaton or statechart is performed by hand at the moment. Since the automata are generated in an object-oriented environment (ARTS), it is common for them to rely on message-based communication. So far, only ad hoc ways of translating messages to constraint logic have been produced. To enable fully automatic translation of automata, a generic translation for messages is needed.

An important issue to be explored in the DExVal project is how to generate the actual test cases comprising the meaningful experiments from the results of the abstract run. At the moment, we only have an intuition of what could be called a meaningful experiment. It would be useful if we could define a methodology to discover them using DExVal. Another goal for future work is the integration of DExVal and a debugger. We could compare expected and real values and detect exactly which part of the code generated an error.

Acknowledgements. We would like to thank Christine Mayr and Sebastian Voges, two students from LMU that helped to implement the DExVal tool prototype while visiting LMF-DI. We are also grateful to Thomas Maibaum, Armando Haeberer, José Fiadeiro and Martin Wirsing for stimulating discussions.

References

1. R. Alur, T. A. Henzinger and P.-H. Ho. Automatic Symbolic Verification of Embedded Systems. IEEE Transactions on Software Engineering 22:181-201. 1996.
2. R. J. Anderson et al.. Model Checking Large Software Specifications. ACM SIGSOFT Symposium on the Foundations of Software Engineering. pp. 156-166. October 1996.
3. M. Carlsson and J. Widen. Sicstus Prolog Users Manual, Release 3#0. Swedish Institute of Computer Science. SICS/R-88/88007C. 1995.
4. S. Carvalho et al. Formal Derivation of Meaningful Validation Experiments in the ARTS Environment. 4th DLR/CNPq German-Brazilian Workshop on Information Technology (St. Jähnichen, ed.). Porto Alegre, Brazil. 1997.
5. Z. Chaochen, C.A.R. Hoare and A.P. Ravn. A Calculus of Durations. Information Processing Letters. 40:269-276. 1991.
6. E. M. Clarke, E. A. Emerson and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. ACM Transactions on Programming Languages and Systems. vol 8(2):244-263. 1986.
7. T. Frühwirth and S. Abdennadher. Constraint-Programmierung (in German). Springer Verlag, Heidelberg, Germany. September 1997.
8. T. Frühwirth and P. Brisset. The CHR Library, Version 2, of Eclipse 3.5.3. European Computer-Industry Research Centre, Munich, Germany. January 1996.
9. D. Harel. Automata: A visual formalism for complex systems. Science of Computer Programming 8(3):231 - 274. June 1987.
10. T. A. Henzinger and H. Wong-Toi. Using HYTECH to Synthesize Control Parameters for a Steam Boiler. Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control (J.-R. Abrial, E. Börger and H. Langmaack, eds.). Lecture Notes in Computer Science 1165. Springer-Verlag. pp. 265-282. 1996.
11. Hopcroft and Ullman. Introduction to Automata Theory, Languages and Computations. Addison-Wesley. 1979.
12. J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey. Journal of Logic Programming 19,20:503-581. 1994.
13. K. Marriott and P. J. Stuckey. Programming with Constraints. MIT Press, USA. March 1998.