AUTOMATIC DERIVATION OF MEANINGFUL EXPERIMENTS FOR HYBRID SYSTEMS*

Angelo E. M. Ciarlini

Departamento de Informática
Pontifícia Universidade Católica do R.J.
Rua Marquês de S.Vicente, 225
Rio de Janeiro, RJ 22.453-900, Brazil
angelo.ciarlini@uol.com.br
.

Thom Frühwirth

Computer Science Institute
University of Munich
Oettingenstr. 67,
D-80538 München, Germany
fruehwir@informatik.uni-muenchen.de
.

Hybrid Systems, Symbolic Execution, Validation, Testing, Constraint Logic Programming

ABSTRACT

Some authors have recently suggested the use of constraint logic programming (CLP) for the verification of hybrid systems. In this paper, we offer evidence that CLP can also be used for the derivation of meaningful test cases. In our approach, formal specifications of concurrent hybrid systems are automatically translated into a format executable by a CLP program. The user can then study the behaviour of his system by specifying conditions under which the execution should be performed. The conditions are specified declaratively in a fragment of first order temporal logic relating variables' values at different times. Such a specification is translated automatically into constraints, which are taken into account during a symbolic execution performed by the CLP program. The symbolic execution generates paths consistent with the user-defined conditions, together with a set of corresponding necessary and sufficient constraints on the input values. We apply an algorithm based on the projection of these constraints to try to extract good input values for testing the final code of the system.

1 INTRODUCTION

In Software Engineering, software 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 most of the potential software bugs. Therefore, the choice of good input values is an essential part of the validation process.

In this paper we present a test data generation approach that uses the DExVal tool (Derivation of Meaningful Experiments for Validation) (Ciarlini and Frühwirth 1999). We have been developing DExVal for the automatic derivation of test data for concurrent hybrid systems, i.e. concurrent systems in which there are both discrete and continuous variables. In order to derive meaningful test cases, we *symbolically execute* formal specifications given as *hybrid automata* (Alur et al. 1996; Henzinger and Wong-Toy 1996). The idea is to make these specifications executable by translating them into a *constraint logic programming* (CLP) language.

Constraints are basically first order predicates for which efficient solvers are available. Constraints enable us to represent possible infinite relations finitely, e.g. $X \leq 5$ represents all values for X that are less than 5. The use of constraint solving enables us to work directly with both discrete and continuous variables, and more abstractly, without giving specific values to variables at all.

In order to define a situation to be tested, the user can specify conditions relating the values of the variables at different times using a fragment of first order temporal logic. Such conditions are translated into constraints, which are solved and simplified together with new constraints introduced during transitions. The computation follows any possible path in the transition system, provided that the constraints accumulated in each transition of the path, together with the user given constraints, can be satisfied. All possible valid runs are therefore taken into account. The result of a run consists of a path and some time-dependent constraints on input variables. In order to cope with termination problems for infinite runs, the symbolic execution performs an *iterative deepening* search. We use an algorithm based on

* This work was partially supported by the CNPq/GMD Brazilian-German Programme on Scientific and Technological Cooperation and the work of the first author was supported by FAPERJ – Fundação de Amparo à Pesquisa do Estado do Rio de Janeiro - Brazil

the projection of the resulting constraints to obtain test data for our automata.

The DExVal tool has been implemented in SICStus Prolog (Carlsson and Widen 1995). Figure 1 shows the architecture of DExVal. Rectangles represent modules, ellipses correspond to data repositories, and arrows to the flow of data.

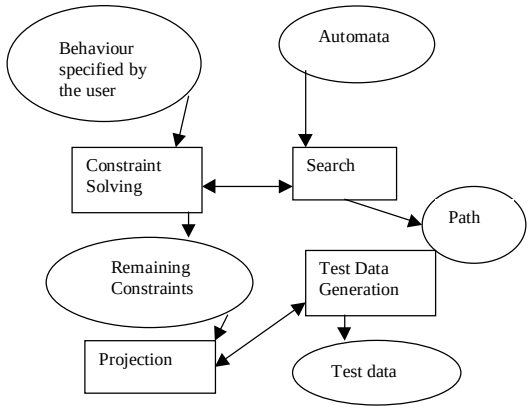

Figure 1: DExVal Architecture

In Section 2, we present concepts that are necessary for the introduction of the context in which we work. Section 3 describes the symbolic execution of hybrid automata implemented in the DExVal tool, while Section 4 describes the generation of test cases from a set of constraints on input variables. In Section 5, we introduce the example of a bathroom boiler scenario. Section 6 contains our concluding remarks.

2 BACKGROUND

In order to introduce the context in which our work is performed we describe hybrid automata and, briefly, constraint logic programming.

2.1 Hybrid Automata

Hybrid automata have been introduced for modelling mixed discrete - continuous systems. A hybrid automaton is a transition system the states of which contain descriptions of continuous activities and the transitions of which are discrete and labelled with guarded actions. The state of the automaton changes either instantaneously through a discrete transition associated with system actions or, as time elapses, through a continuous activity. The treatment of continuous activities inside states and the absence of hierarchical abstraction into superstates are the main features that distinguish it from related approaches such as statecharts (Harel 1987). A variety of terminologies is used in different papers on hybrid automata. For ease of reference, we have employed the more widely used terminology of transition systems, giving alternative synonyms 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 (invariant conditions, location invariants) – a constraint on variables. The automaton may reside in the state as long as the invariant holds.
  - Iteration (continuous activities, flow conditions) – activities (assignments) that specify the new values of the variables based on their current values. 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.

In a *timed hybrid automata* all the automata are synchronized by a machine clock that causes iterations and transitions of states. The system modifies the values of the variables according to the state of the automaton at the last clock tick. In a state, the values of the variables conceptually change continuously, however the iterations actually happen step by step. Iterations take time, while transitions are instantaneous. Variables that do not occur in activities or actions remain unchanged.

A hybrid system typically consists of several interacting *concurrent hybrid automata* that co-ordinate through shared variables and events. We assume that all the automata make their modifications simultaneously, i.e. a set of mutually consistent continuous activities and enabled transitions is performed at the same time. We assume also that a variable can be modified by only one automaton. Otherwise two concurrent transitions could modify the same variable leading to write conflicts. A *state transition diagram* is a pictorial representation of the operation of a 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 Fig. 3).

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) (Jaffar and Maher 1994; Marriot and Stuckey 1998) combines the advantages of logic programming (such as 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 non-deterministic process, search. During program execution, the logic program incrementally sends constraints to the constraint solver, which tries to solve the constraints. The results from the solver cause a priori pruning of branches in the search tree spawned by applying rules in the program. Non-satisfaction 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 $X+Y<5$ 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.

3 SYMBOLIC EXECUTION

In order to validate their systems, the DExVal users create scenarios with instances of classes of automata. The relationship among different instances is defined by the specification of their parameters and the synchronization constraints on the transitions. They can also inform the initial and final states of the automata, the variables they would like to project the remaining constraints onto (in order to derive test data), and the limits to the sizes of the prefixes of the runs that should be taken into account.

Since a scenario has been defined, the users can specify their conditions on the behaviour of the system using a temporal logic. They can specify conditions that should hold for all times (universal quantification), conditions that should hold for at least one time (existential quantification), and conditions that should hold at certain times. A variable x in a certain state at time t can be thus identified by giving it the appropriate timestamp t, written $x:t$. The timestamp "i" represents the initial time (corresponding to the initial values of the variables) and the timestamp "f" represents the final time. Conditions describe values or ranges of input variables, output variables and internal variables. In general, arbitrary constraints (e.g. 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'.

DExVal creates a set of CLP rules according to the specified scenario. The rules representing the states of each automaton have clauses for the description of invariants, iterations and transitions, defined as follows:

```
automaton_name(invariant-ST1,OLD_VARS,CONSTRAINTS)
automaton_name(iteration-ST1,OLD_VARS,
                NEW_VARS,CONSTRAINTS)
automaton_name(transition-ST1-ST2, OLD_VARS,
                NEW_VARS,CONSTRAINTS)
```

In these clauses, ST1 stands for the current state and ST2 for the future state (after a transition). OLD_VARS and NEW_VARS are lists of pairs <VAR_NAME, VAR_VALUE>, where VAR_NAME is the name of a variable and VAR_VALUE is a Prolog variable. OLD_VARS lists variables the current values of which are used in the invariants, iterations and transitions (each VAR_VALUE represents the current value of the corresponding VAR_NAME). NEW_VARS lists variables the values of which are modified by iterations and transitions (each VAR_VALUE represents the value of the corresponding VAR_NAME after the modification). CONSTRAINTS are CLP constraints relating variable values that appear in OLD_VARS and NEW_VARS.

At the start of the symbolic execution, the conditions specified by the user are translated and sent to the DExVal Constraint Solver.

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 Prolog variables contained in the OLD_VARS and NEW_VARS arguments of our rules are bound to the corresponding values in the data structure we use for representing the values of the variables at the different points in time during the execution. After binding the variables, the constraints contained in the argument CONSTRAINTS are sent to the constraint solver. Therefore, the constraint solver receives continuously new constraints stemming from guarded actions, invariants or iterations. If the store of constraints becomes inconsistent, the current branch of the search fails and DExVal backtracks to try another branch of the search tree.

When the final state of each automaton is reached, DExVal simplifies the accumulated constraints and projects them onto the variables of interest as specified by the user. The accumulated constraints specify the allowed ranges for

variables and the dependencies between variables that cause the execution of that path. From such constraints, we derive the test cases.

In order to avoid termination problems, our search uses an iterative deepening procedure. The user defines a minimum length *MIN* for the paths, an increment *INC* and a limit to the number of iterations *LIMIT*. Initially, DExVal performs a deep first search trying to find paths that have a number of clock ticks between *MIN* and *(MIN+INC)*. If no path is found, the upper limit is expanded to *(MIN+2*INC)* and so on, until the limit *(MIN+LIMIT*INC)* is reached.

As we do not know how long a path can be, the treatment of properties resulting of quantification on time tends to be complicated. In order to be able to treat quantification, disjunction and the co-ordination of different constraint solvers we implemented a specialized constraint solver using Constraint Handling Rules (CHR) (Frühwirth 1995). This specialized constraint solver controls and co-ordinates SICStus Prolog's different constraint solvers. For instance, whenever a new clock tick is considered in the path, a constraint $\forall T(v:T>20)$, specifying that the value of variable *v* is greater than 20 at all times, generates a new constraint to be sent to the constraint solver of the Real numbers.

Recently, some authors have suggested the use of constraint logic programming for implementing and reasoning about timed automata and hybrid systems (Pontelli and Gupta 1997; Delzanno and Podelski 1999; Urbina 1996; Fribourg 1998). The main distinguishing novelty of our approach is that we use a CLP-based symbolic execution for the automatic derivation of test data for hybrid systems. Therefore, we try to provide the means for the satisfaction of testing criteria. In particular, the expressiveness of the language for the specification of test situations is essential for this purpose.

4 TEST DATA DERIVATION

The output of the symbolic execution is a path specifying the state of each automaton at each clock and a set of constraints that certain variables (specified by the user) should obey if the hybrid system executes this path and the user-specified conditions are satisfied. If the user asks DExVal to obtain constraints on the input variables, he obtains sufficient and necessary conditions for the execution of the corresponding path.

In order to derive test data from the output of a symbolic execution, we defined an algorithm based on an alternation between the projection of the remaining constraints onto each variable and the assignment of values to the variables. When we project a set of constraints onto only one variable of type Real, we usually obtain its domain. We can then choose any value from this domain. When we bind the value of a variable, the initial set of constraints is evaluated again by the constraint solver. After this new evaluation, we can project the constraints onto another variable and so on, until we obtain values for all input variables. This approach is deterministic, as we are always re-evaluating the set of constraints. We therefore have an efficient procedure, in which we do not need, other than in exceptional cases, to use a backtracking search because of bad choices for the variables' values.

As most of the errors can occur when the values assigned to the variables are close to the limit of their domains, we defined three different criteria for the choice of a value for a variable the domain of which is known:

- mid: in which we choose an intermediate value for testing the typical behaviour of the system;
- max: in which we choose the maximum (or almost maximum) value a variable can assume in the domain; and
- min: in which we choose the minimum (or almost minimum) value a variable can assume in the domain.

The last two criteria are used to find critical situations that can generate an error. When we know that a variable is higher (or lower) than a certain value but not equal to it, we obtain values for our tests by subtracting (or adding) a very small constant. In this case, it might be necessary to use backtracking because the calculated value could be out of the domain. That being so, we can generate an even smaller constant to be added to (or subtracted from) the domain open limit.

As mentioned before, when we assign a value to one of the variables, the domains of the other variables can change. In order to obtain all possible critical combination of values, we can alternate the first variable to have the constraints projected onto it.

Some authors have suggested the analysis of constraints for test data derivation, but they use neither constraint solving nor projection (DeMillo and Offut 1991; Offut et al. 1999). Instead, they apply a dynamic domain reduction procedure, in which arbitrary choices can generate a lot of unnecessary search, using a backtracking mechanism. Moreover, they are not deterministic. Our test data approach presents the following features:

- It can be used for most of the test data derivation criteria, such as *coverage of paths* (Korel 1990) and *mutants* (DeMillo and Offut 1991). Coverage of paths can be tested by means of the specification of the initial and final states of the automata during the symbolic execution. The derivation of data that can *kill a mutant* (i.e. that generates a different behaviour for a slightly modified specification) is achieved by means of the specification of constraints that should hold at specific times.
- We have a deterministic process in which we know that, when a path is generated by the symbolic execution, we can possibly obtain not only one but many test data for a specific situation.
- The user has an expressive language for the specification of situations he would like to test. We are not limited to the specification of initial and final states. Val-

ues or ranges of output and intermediate variables, in particular, can be easily specified by the user.

- We can test concurrent hybrid automata.

Finally, we should stress that our test data approach can be used not only for testing final code but also as a tool for the validation of models that simulate real world processes. For instance, whenever test data is generated from a formal specification of a real world process and the expected behaviour is not confirmed by experience, we know that the formal specification is not good enough and should be modified.

5 THE BATHROOM BOILER SCENARIO

In order to clearly illustrate the advantages of the approach taken in the DExVal project, we chose an example involving physical processes, inspired by the steam boiler problem (Henzinger and Wong-Toi 1996). In particular, as opposed to e.g. standard finite model checking, continuous variables can be used without difficulty in the constraint-based approach, since infinite ranges of values can still be represented by and reasoned with constraints.

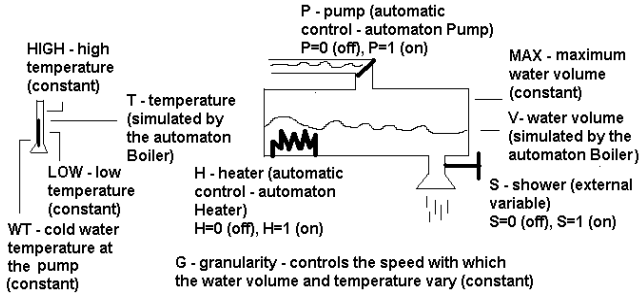

Figure 2: The Bathroom Boiler Scenario

The scenario (Fig. 2) involves a warm water boiler of a bathroom, with an 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 (which depends on the ingress 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.

We designed three concurrent automata to model our sample: *Heater*, *Boiler* and *Pump*. Figure 2 shows automaton *Boiler*, the other automata are simpler and we do not show them for space reasons. *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.

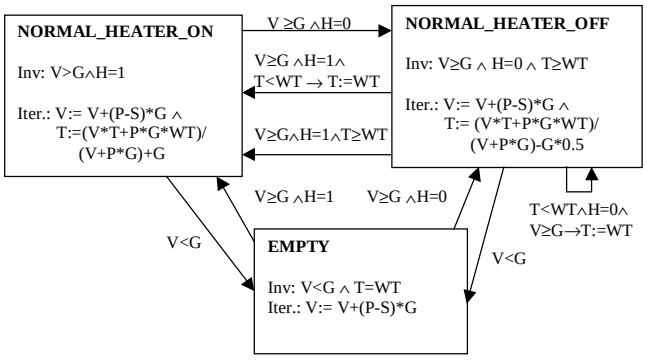

Figure 3: The Automaton Boiler

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 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* also has three states: *MAINTAIN*, *TURNING_ON* and *TURNING_OFF*. If the temperature of the water is between the lower and upper limits, the automaton remains in state *MAINTAIN* and the variable *heater* is not modified. If the temperature is less (greater) 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 greater than the limit, the variable *pump* is set to 0 and the automaton goes to state *OFF*.

Assume that the formal specifications were implemented, generating a simulation of our hybrid system. The control of the temperature by the heater can be tested by asking DExVal to provide "good" values for the variable *shower* and for the initial *temperature*. In order to perform a test, a condition that can be observed should be specified. For instance the user can specify the following :

```
water_volume:i=10.0∧temperature:i<100∧
∀T (heater:T=0)
```

Such a condition is used to inform DExVal that the initial water volume is 10, the heater will remain *off* all the time and the water initial temperature is lower than 100. For the case in which the shower is *on* all the time and the implemented system runs for 5 clocks, DExVal proposes the

following values for the initial temperature: 47.181, 73.59 and 99.999. Using these values as input for the implemented code, the user can observe whether the heater behaves as expected, i.e. it remains *off* all the time.

DExVal can also be used to try to kill mutants. For instance, the transition in automaton *Boiler* from the state *NORMAL_HEATER_ON* to the state *NORMAL_HEATER_OFF* occurs when the value of the variable *heater* switches to *off*. Assume, for instance, that the condition

```
∀T(T≠f→state(boiler):T='NORMAL_HEATER_ON')∧
heater:f=0
```

is specified informing DExVal that the boiler remains at state *NORMAL_HEATER_ON* all the time before the last clock tick, and, at the last clock tick, the heater turns off. DExVal provides then the values for the initial temperature and for the variable *shower* that cause the transition. If these values are used as input for the final code, then the behaviour of the variable *temperature* (after the moment the transition should occur) reveals whether the implemented code is correct or not: If the temperature is not reduced, then the code implemented is a mutant.

6 CONCLUDING REMARKS

Our preliminary results indicate the potential importance of constraint logic programming for the derivation of properties of hybrid systems and also for the generation of test cases. We described a symbolic execution method that enables the user to specify test conditions. The output of our symbolic execution is the input for the test data generation procedure. An algorithm based on the projection of constraints is used for the generation of "good" input values for testing the final code. Our approach seems to be compatible with various test data criteria, including the ability to "kill" mutants.

We are currently working on the integration of the symbolic execution and test data generation procedures. We are also working on the enhancement of our specialized constraint solver. Our test data generation depends on the projection of constraints onto the values of the input variables. Depending on the case, the projection may not be trivial. In particular, it is impossible to develop an algorithm able to solve and project any kind of non-linear constraints. The use of Constraint Handling Rules, however, gives us the opportunity to develop dedicated constraint solvers that are able to deal with specific kinds of non-linear constraints.

REFERENCES

Alur, R.; T. A. Henzinger and P.-H. Ho. 1996. "Automatic Symbolic Verification of Embedded Systems". *IEEE Transactions on Software Engineering*, 22:181-201.

Carlsson, M. and J. Widen. 1995. *Sicstus Prolog. User's Manual, Release 3.0*. Swedish Institute of Computer Science, SICS/R-88/88007C.

Ciarlini, A. and T. Frühwirth. 1999. "Symbolic Execution for the Derivation of Meaningful Properties of Hybrid Systems", (Poster) In *Proc. 16th. International Conference on Logic Programming (ICLP'99)* (Las Cruces, New Mexico, USA).

DeMillo, R.A. and A.J. Offut. 1991. "Constraint-Based Automatic Test Data Generation". *IEEE Transactions on Software Engineering*, 17(9):900-910.

Delzanno,G. and A. Podelski. 1999. "Model Checking in CLP". In *Proc. Second International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'99*. (Rance Cleaveland, ed.), LNCS, Springer-Verlag.

Fribourg, L. 1998. *A Closed-Form Evaluation for Extended Timed Automata*. Technical Report LSV-98-2, Laboratoire Specification et Verification, ENS de Cachan, Cachan, France.

Frühwirth, T. 1995. "Constraint Handling Rules". In *Constraint Programming: Basics and Trend*, A. Podelski (ed.), Springer LNCS 910.

Harel, D. 1987. "Automata: A visual formalism for complex systems". *Science of Computer Programming* 8(3):231 - 274.

Henzinger, T. A. and H. Wong-Toi. 1996. "Using HYTECH to Synthesize Control Parameters for a Steam Boiler". In *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control* (J.-R. Abrial, E. Börger and H. Langmaack, eds.). LNCS 1165. Springer-Verlag. 265-282.

Jaffar, J. and M. J. Maher. 1994. "Constraint Logic Programming: A Survey". *Journal of Logic Programming* 19,20:503-581, 1994.

Korel, B. 1990. "Automated Software Test Data Generation". *IEEE Transactions on Software Engineering*, 16(8):870-879.

Marriott, K. and P. J. Stuckey. 1998. *Programming with Constraints*. MIT Press, USA.

Offut, A.J.; Z. Jin and J. Pan. 1999. "The Domain Reduction Procedure for Test Data Generation". To appear, *Software Practice and Experience*.

Pontelli, E. and G. Gupta. 1997. "A Constraint-Based Approach for Specification and Verification of Real-time Systems". In *Proc. 1997 IEEE Real Time Systems Symposium*, IEEE Computer Society. 230-239.

Urbina, L. 1996. "Analysis of Hybrid Systems in CLP(R)". In *Proc. Principles and Practice of Constraint Programming CP'96*. LNCS 1118. Springer-Verlag, 451-467.