Test Automation for Hybrid Systems

Bahareh Badban[*]
Department of Computing Science
C. v. Ossietzky Universität, Oldenburg, Germany
Bahareh.Badban@uni-oldenburg.de

Martin Fränzle[*]
Department of Computing Science
C. v. Ossietzky Universität, Oldenburg, Germany
fraenzle@informatik.uni-oldenburg.de

Jan Peleska[†]
TZI Center for Computer Technologies
University of Bremen, Germany
jp@tzi.de

Tino Teige[*]
Department of Computing Science
C. v. Ossietzky Universität, Oldenburg, Germany
Tino.Teige@informatik.uni-oldenburg.de

ABSTRACT

This article presents novel results on automated test generation for hybrid control systems, which involves the generation of both discrete and real-valued, potentially time-continuous, input data to the system under test. Our generation techniques are allocated in two layers: The upper layer contains a symbolic test case generator constructing test cases as paths through an abstracted representation model of the system under test. Different test strategies designed to pursue various quality objectives lead to different selections of symbolic test cases. Symbolic test cases are transformed into feasible, i. e., executable, test cases by constructing concrete sequences of input data, allowing the execution of the pre-planned transition sequence. The input data construction is performed by the lower layer consisting of a constraint solver which applies interval analysis techniques to identify the domains from where to pick the appropriate test data. This process is made efficient by combining subpaving with forward-backward interval constraint propagation. On both layers learning algorithms are applied in order to avoid the spending of computation time on paths and sub-constraints, respectively, which are already known not to contribute to the solution.

[*]Work of the authors situated at Oldenburg has been partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS, `http://www.avacs.org`).

[†]Partly supported by the Deutsche Forschungsgemeinschaft DFG as part of the priority programme SPP 1064 on *Software Specification – Integration of Software Specification Techniques for Applications in Engineering* (SPP 1064, HYBRIS, `http://www.tzi.de/agbs/projects/hybris`).

SOQUA'06, November 6, 2006, Portland, OR, USA.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.5 [**Software Engineering**]: Testing and Debugging

1. INTRODUCTION

Hybrid systems perform control tasks involving the processing of both discrete and real-valued, potentially time-continuous (analog), data. As a consequence, testing hybrid systems controllers requires the generation and evaluation of discrete and analog I/O data written to and read from interfaces of the system under test (SUT). This article contributes to the problem of *automated specification-based test generation* for hybrid systems: Test data are derived from hybrid systems specifications describing the required behavior of the SUT. As specification formalism we use *time-discrete input-output hybrid systems (TDIOHS)* which are suitable for describing sequential (possibly non-terminating) time-discrete dynamical control systems. Our results, however, only rely on an appropriate internal representation of the specification model, so that different formalisms can be supported via transformation front-ends. In particular, our concepts can be applied to hybrid variants of Statecharts [2]. Moreover, they also apply to structural software testing, where the TDIOHS represent the control flow graphs of the software units under test.

Testing is usually applied with the objective to investigate specific quality objectives in the SUT, such as functional and behavioral correctness with respect to given specifications, stability in boundary situations and robustness against illegal environment behavior. These objectives induce test strategies aiming at exercising specific portions of existing specifications or of additional requirements elaborated by the testing specialists. As a consequence the implementation of strategies in test suites requires the construction of input sequences "driving" the SUT into states corresponding to certain specification locations. For complex hybrid systems this task typically involves (1) the traversal of graphs representing abstracted specification structure like control locations, transitions and abstract labels, (2) the symbolic interpretation of conditions, invariants, flows and actions and (3) the generation and solution of constraints derived from the guard conditions to be fulfilled in order to exercise certain portions of the specification on the SUT. For TDIOHS the sub-task (2) is simplified because flow conditions and invariants do not appear explicitly in the specification.

This article mainly contributes to the first and third sub-task: With respect to (3), we focus on *constraint solving problems (CSPs)* involving real-valued variables. Intuitively speaking, CSPs specify the multi-dimensional sets $\mathbb{S} \subseteq \mathbb{R}^n$ from where input data to the SUT should be selected, in order to stimulate a certain SUT execution path suggested by the selected test strategy. The SAT solving methods for Boolean problems and solvers for integral-valued CSPs are obviously not sufficient when real valued variables and, in particular, non-linear CSPs are involved. Moreover, they do not possess “natural” extensions for solving real-valued CSPs. Therefore we advocate the application of *interval analysis*, where CSPs are solved by approximating the solution sets $\mathbb{S} \subseteq \mathbb{R}^n$ by unions of non-intersecting intervals $I \subset \mathbb{R}^n$ (so-called *subpavings*). To avoid the complexity problems typically arising from direct application of subpaving techniques, forward-backward interval constraint propagation is applied which turns out to be a very powerful contractor for shrinking intervals containing the solution. Moreover, a detailed conflict analysis helps to avoid further subpavings in “directions” already known to be incompatible with the CSP to be solved.

For handling sub-task (1) described above, the constraint solving techniques are combined with a *symbolic test case generator* selecting paths through the transition graph of a given TDIOHS according to a given strategy. This allows to encapsulate all tasks concerning test coverage in a separate layer. This layer also applies learning strategies by avoiding to re-select sequences of symbolic transitions which are infeasible because no inputs making the associated guard evaluations `true` can be constructed.

Overview. Section 2 introduces the basic notions about time-discrete input-output hybrid systems and interval analysis, introduces the formal notion of symbolic test cases and describes several test strategies suitable for pursuing different quality objectives. Sections 3 and 4 contain the main results of this paper, where the symbolic test case generator is specified and our new solvers for typical constraints induced by coverage goals for hybrid specification are described. Section 5 contains the conclusions and describes work in progress beyond the scope of this paper.

Related Work. Our TDIOHS differ from Henzinger’s *hybrid automata (HA)* introduced in [7] mainly by the fact that TDIOHS do not model invariants and flow conditions in an explicit way. TDIOHS basically represent time-discrete dynamical control systems where all flow conditions have been handled beforehand, using discretized solutions.

The design of the symbolic test case generator described in Section 3 has been influenced by the novel solutions for graph traversal with the objective of implementing specific strategies or, equivalently, coverage criteria suggested in [4]. The definitions and results from interval analysis used in this paper are based on [8]. Interval analysis has been frequently applied in combination with abstract interpretation for various aspects of static software analysis; see, for example, [5].

2. CONCEPTS AND TERMINOLOGY

2.1 Hybrid Systems

A *time–discrete input–output hybrid system* (TDIOHS) is a tuple $\mathcal{H} = (Loc, Init, V, I, O, Trans)$ where Loc is a finite set whose elements are called *locations*, V is a finite set of (discrete and continuous) variables, I, $O \subseteq V$ are sets of *input* and *output* variables, respectively, with $I \cap O = \emptyset$. Let $Guard$ denote the set of all quantifier-free predicates over V, $Init : Loc \rightarrow Guard$ a function mapping locations to predicates, $Assign$ the set of all pairs $(\vec{x}, \vec{t})$ where $\vec{x} = (x_1, x_2, \ldots)$ is the vector of all variables in $V - I$ (a.k.a. *controlled* variables) and $\vec{t} = (t_1, t_2, \ldots) \in T^{|V-I|}$, where T is the set of all terms over V. Then $Trans \subseteq Loc \times Guard \times Assign \times Loc$ is the set of *transitions*. $Labels = \{\lambda \in Guard \times Assign \mid \exists l_1, l_2 \in Loc : (l_1, \lambda, l_2) \in Trans\}$ denotes the set of *transition labels*.

Let $val \in \text{dom}^{|V|}$ be a *valuation* of all variables occurring in $\mathcal{H}$, and let the value of $v \in V$ and of the term t over V under val be denoted by $val(v)$ and $val(t)$, respectively. A *run* of a TDIOHS $\mathcal{H}$ is an infinite sequence of pairs $\langle (l_1, val_1), (l_2, val_2), \ldots \rangle$ of locations and valuations which satisfies the following properties: (1) $val_1(Init(l_1)) = \texttt{true}$, (2) $\forall i \in \mathbb{N}\ \exists (l, g, (\vec{x}, \vec{t}), l') \in Trans : l = l_i, l' = l_{i+1}$, $val_i(g) = \texttt{true}$, $val_{i+1}(x_1) = val_i(t_1)$, …, $val_{i+1}(x_{|V-I|}) = val_i(t_{|V-I|})$. A *k–bounded* run is a run of a fixed length $k \in \mathbb{N}$, i.e. the sequence of location-valuation pairs $\langle p_1, \ldots, p_k \rangle$ is finite. The set of all k–bounded runs of $\mathcal{H}$ is denoted by $Run(\mathcal{H}, k)$.

Let $V(\vec{t})$ denote the set of variables from V referenced in term vector $\vec{t}$ and $Stable(\vec{x}, \vec{t})$ denote the variable components x_i of vector $\vec{x}$ whose values are unaffected by the associated assignment term t_i in any valuation (so $t_i \equiv x_i$). An *input location* $l_1 \in Loc_I \subseteq Loc$ is specified by the requirement that every transition entering l_1 only executes assignments where input variables are copied to *local* variables $x_i \in V - (I \cup O)$, that is,

$$Loc_I = \{\ l_1 \in Loc \mid \forall (l_0, g, (\vec{x}, \vec{t}), l_1) \in Trans : \\ V(\vec{t}) \subseteq I \wedge O \subseteq Stable(\vec{x}, \vec{t})\ \}$$

An *output location* $l_1 \in Loc_O \subseteq L$ is characterized by the requirement that all transitions entering this state perform only assignments from local to output variables:

$$Loc_O = \{\ l_1 \in Loc \mid \forall (l_0, g, (\vec{x}, \vec{t}), l_1) \in Trans : \\ V(\vec{t}) \subseteq V - (I \cup O) \subseteq Stable(\vec{x}, \vec{t})\ \}$$

An *internal processing location* $l_1 \in Loc_P \subseteq Loc$ is characterized by the requirement that entry assignments may only read from and write to local variables:

$$Loc_P = \{\ l_1 \in Loc \mid \forall (l_0, g, (\vec{x}, \vec{t}), l_1) \in Trans : \\ V(\vec{t}) \subseteq V - (I \cup O) \wedge O \subseteq Stable(\vec{x}, \vec{t})\ \}$$

A TDIOHS is called *I/O safe* if it possesses no other locations apart from input, processing and output locations, and the free variables of all guards are members of $V - I$. These conditions imply that input changes during processing steps

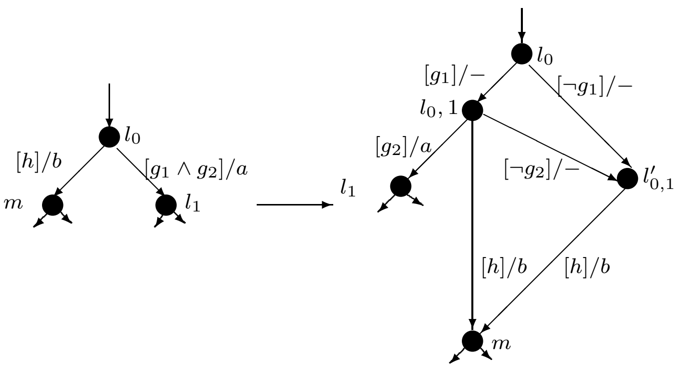

Figure 1: Simplification of conjunctive guards.

are disregarded by the system: Only when a new input location is entered the new input valuations are copied to local variables, and are used by guards and assignment terms.

Two runs $r = \langle (l_i, val_i) | i \in \mathbb{N}\rangle$ and $r' = \langle (l'_i, val'_i) | i \in \mathbb{N}\rangle$ of I/O safe TDIOHS are called *I/O equivalent* if they receive the same sequence of input data in their input locations, produce the same output sequences in their respective output locations and have an identical interleaving of these inputs and outputs:

$$r \sim r' \equiv r|(Loc_I \cup Loc_O) = r'|(Loc_I \cup Loc_O)$$

In this definition $r|M$ with $M \subseteq Loc$ denotes the restriction of run r to the subsequence of all pairs (l, val) with $l \in M$.

I/O safe TDIOHS can be re-structured in several ways preserving the possible runs of original and transformed system up to I/O equivalence. For the purpose of this paper, we illustrate this property by the following lemmas, concerning the simplification of guards by means of additional states and transitions. The transformation characterized by Lemma 1 is illustrated in Figure 1[1].

Lemma 1. *Let $\varepsilon = (\vec{x}, \vec{x})$ denote the stable assignment which does not change any local variables or outputs. Given TDIOHS $\mathcal{H}_1$ and transition $\tau_0 = (l_0, g_1 \wedge g_2, a, l_1) \in Trans(\mathcal{H}_1)$, construct a new TDIOHS $\mathcal{H}_2$ by setting*

1. *$Loc_P(\mathcal{H}_2) = Loc_P(\mathcal{H}_1) \cup \{l_{0,1}, l'_{0,1}\}$, where $l_{0,1}, l'_{0,1}$ are fresh location identifiers,*
2. *$Trans(\mathcal{H}_2) = (Trans(\mathcal{H}_1) - T_1) \cup T_2$, where*

$$\begin{aligned} T_1 &= \{(l, g, b, l') \in Trans(\mathcal{H}_1) \mid l = l_0\} \\ T_2 &= \{(l_0, g_1, \varepsilon, l_{0,1}), (l_0, \neg g_1, \varepsilon, l'_{0,1})\} \cup \\ &\quad \{(l_{0,1}, g_2, a, l_1), (l_{0,1}, \neg g_2, \varepsilon, l'_{0,1})\} \cup \\ &\quad \{(l_{0,1}, g, b, m) \mid (l_0, g, b, m) \in Trans_1 - \{\tau_0\}\} \cup \\ &\quad \{(l'_{0,1}, g, b, m) \mid (l_0, g, b, m) \in Trans_1 - \{\tau_0\}\} \end{aligned}$$

which are the only changes from $\mathcal{H}_1$ to $\mathcal{H}2$. Then $\mathcal{H}_1$ and $\mathcal{H}2$ perform I/O-equivalent runs.

[1]A full proof for this lemma can be found in the extended version of this paper which is available under `http://www.tzi.de/agbs/projects/hybris`.

An analogous transformation exists for disjunctive guards:

Lemma 2. *Let $\varepsilon = (\vec{x}, \vec{x})$ denote the stable assignment which does not change any local variables or outputs. Given TDIOHS $\mathcal{H}_1$ and transition $\tau_0 = (l_0, g_1 \vee g_2, a, l_1) \in Trans(\mathcal{H}_1)$, construct a new TDIOHS $\mathcal{H}_2$ by setting*

1. *$Loc_P(\mathcal{H}_2) = Loc_P(\mathcal{H}_1) \cup \{l_{0,1}, l'_{0,1}, l''_{0,1}\}$, where $l_{0,1}, l'_{0,1}, l''_{0,1}$ are fresh location identifiers,*
2. *$Trans(\mathcal{H}_2) = (Trans(\mathcal{H}_1) - T_1) \cup T_2$, where*

$$\begin{aligned} T_1 &= \{(l, g, b, l') \in Trans(\mathcal{H}_1) \mid l = l_0\} \\ T_2 &= \{(l_0, \neg g_1, \varepsilon, l_{0,1}), (l_0, g_1, \varepsilon, l''_{0,1}), (l_{0,1}, \neg g_2, \varepsilon, l'_{0,1}), \\ &\quad (l_{0,1}, g_2, \varepsilon, l''_{0,1}), (l''_{0,1}, \mathtt{true}, a, l_1)\} \cup \\ &\quad \{(l'_{0,1}, g, b, m) \mid (l_0, g, b, m) \in Trans_1 - \{\tau_0\}\} \cup \\ &\quad \{(l''_{0,1}, g, b, m) \mid (l_0, g, b, m) \in Trans_1 - \{\tau_0\}\} \end{aligned}$$

which are the only changes from $\mathcal{H}_1$ to $\mathcal{H}2$. Then $\mathcal{H}_1$ and $\mathcal{H}2$ perform I/O-equivalent runs.

2.2 Interval Analysis

For any pair of (possibly real) numbers a and b, we identify the interval $[a, b]$ as $\{x \in \mathbb{R} \mid a \leq x \leq b\}$. If either of a or b do not belong to the interval then the corresponding "[" sign or "]" would be replaced by "(" sign or ")", respectively. In the sequel we use the letters $I, J, I_i, \ldots$ to represent an interval in $\mathbb{R}$. A *box* $I^n \subseteq \mathbb{R}^n$ is a Cartesian product of n intervals in $\mathbb{R}$. For simplicity, we might use $I, J, \ldots$ to represent an n-dimensional box as well.

Interval operations. Given two intervals $[a, b]$ and $[c, d]$, and a binary operation op, we define the interval operation $\overset{\circ}{op}$ over these two intervals as $[a, b] \overset{\circ}{op} [c, d] = \{x \; op \; y \mid x \in [a, b], \; y \in [c, d]\}$. Likewise for unary operations we define: $\overset{\circ}{op}([a, b]) = \{op(x) \mid x \in [a, b]\}$. As a result $[a, b] \overset{\circ}{\cdot} [c, d] = [\min S, \max S]$ where $S = \{a \cdot c, a \cdot d, b \cdot c, b \cdot d\}$; also $[a, b] \overset{\circ}{+} [c, d] = [a+c, b+d]$ and $[a, b] \overset{\circ}{-} [c, d] = [a-d, b-c]$, for a proof of these cases see [8]. Defining the interval function like this might also cause partially defined functions, for instance in case of division, if 0 occurs in the divisor interval then the interval operation would no longer be total. In these cases we would exclude the elements which cause incompleteness, form the interval and then compute the interval operation. Given a term t we identify its interval extension $\overset{\circ}{t}$ as a term in which all the operations are replaced by their interval extension.

A *subpaving* of a box I is a union of (some of) its non-intersecting (possibly connected in the borders) non-empty subboxes. A *bisector* of a box $I = [a_1, b_1] \times \ldots \times [a_i, b_i] \times \ldots \times [a_n, b_n]$ is a subbox J of it whose jth interval for some $1 \leq j \leq n$ is either $[(a_j + b_j)/2, b_j]$ or $[a_j, (a_j + b_j)/2]$ and for all $1 \leq k \neq j \leq n$ its kth interval is $[a_k, b_k]$. Now we define the set B_I of bisectors of I, the union of the sets B_i, which are recursively defined as follows: $B_0 = \{I\}$ and for each $i \in \mathbb{N}$: B_{i+1} is the set of bisectors over B_i. A subpaving of I is called *regular* if it is a subset of B_I.

Having a CSP c represented by $c = \bigwedge_{i=1}^{n} c_i$, where each c_i has less free variables than c and has a solution set $\mathbb{S}(c_i)$, a *finite sub-solver* ϕ_i for c_i is a finite algorithm to compute new intervals for some variables in c_i where other variables in c_i are known, in such a way that the resulting subbox is yet a subset of $\mathbb{S}(c_i)$. For example let $c = (x \leq 1 \wedge x = e^y)$, then from the first constraint we can deduce that $x \leq 1$. Now let c_i be $x = e^y$; this results in $x > 0$. Hence from this constraint and the previous one we obtain a new interval for x which is $(0, 1]$.

Given a constraint c and a box I, *contracting* c means replacing I with a smaller subbox J such that the solution set $\mathbb{S}$ is still a subset of J, i.e. $\mathbb{S} \subseteq J \subset I$. A *contractor* for c is any operator that can contract it.

2.3 Test Cases and Strategies

A *symbolic test case* for TDIOHS $\mathcal{H}$ is a finite sequence of transitions $\langle t_1, \ldots t_k \rangle$ with $t_i = (l_i, g_i, a_i, l'_i) \in Trans$ satisfying $l_1 \in Init$ and $l'_i = l_{i+1}$ for $i = 1, \ldots, k-1$. A symbolic test case is *feasible* if valuations can be found, turning the test case into a k-bounded run of $\mathcal{H}$, that is,

$$\begin{aligned} &\exists val_1, \ldots, val_k \in \text{dom}^{|V|} : \\ &\quad r = \langle (l_1, val_1), \ldots, (l_k, val_k) \rangle \in Run(\mathcal{H}, k) \wedge \\ &\quad val_1(g_1) = \ldots = val_k(g_k) = \text{true} \end{aligned}$$

Note that the initialization condition for runs also implies that $val_1(Init(l_1)) = \texttt{true}$. Further observe that for deterministic $\mathcal{H}$ this enforces the execution of transitions t_i while for nondeterministic TDIOHS, this only offers the "chance" for their execution.

In the run r, sequence $\langle (val_i|I) \mid i = 1, \ldots, k \rangle$ is called the *test (input) data* and sequence $\langle (val_i|O) \mid i = 1, \ldots, k \rangle$ is called the *expected result*. In these expressions, $(f|X)$ denotes function domain restriction to elements from X.

A *test strategy* specifies a collection of symbolic test cases. Numerous test strategies aiming at different quality objectives exist. The strategies aiming at behavioral equivalence between the SUT and its specification – most notably, the well-known W method and variants thereof [3, 10] – are of considerable theoretical value, but cannot be completely covered in most practical test campaigns, since the number of test cases required to prove behavioral correctness is extremely high for non-trivial sizes of the SUT state space. Alternative strategies aim at requirements coverage, structural coverage, absence of specific failure types or uniform statistical test case distribution [4].

To illustrate the test case and test data generation concepts in this paper, we focus on the *Modified Condition / Decision Coverage (MCDC)* and related coverage criteria for logical expressions [1]. Quoting the standard [9], MCDC demands that *'Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, and each condition in a decision has been shown to independently affect that decision's outcome. A condition is shown independently to affect a decision's outcome by varying just that condition while holding fixed all other possible conditions.'* Tests driven by the MCDC strategy are likely to uncover faults where – due to an erroneous guard implementation – the wrong transition is taken from

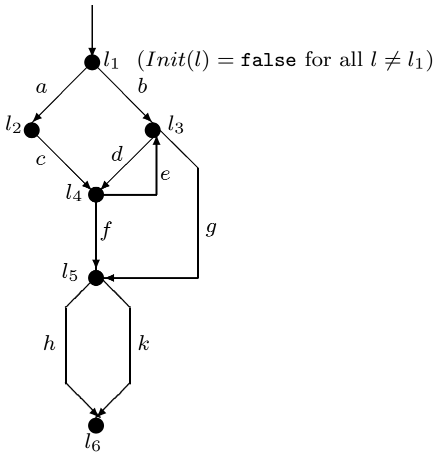

Figure 2: Symbolic TDIOHS representation.

a given location. MCDC is required by the standard [9] to be achieved when testing avionic software with highest criticality level. Given an arbitrary TDIOHS $\mathcal{H}$, the transformations specified in Lemma 1 and 2 can be repeatedly applied until transformation reaches a fixed point. The resulting I/O-equivalent TDIOHS $\mathcal{H}'$ only contains atomic guard conditions, and MCDC coverage is equivalent to covering each transition of $\mathcal{H}'$. Observe that MCDC coverage is a useful coverage goal both for structural software (e. g. module) testing and specification-based testing, and that the concepts described here apply to both testing areas: In the former case, the TDIOHS models the control flow graph of the software to be tested, whereas in the latter case the TDIOHS represents the specification model. In both situations it is advisable to cover different valuations of guard conditions as prescribed by the MCDC coverage goal.

3. SYMBOLIC TEST CASE GENERATION

Test case and test data generation is performed by means of two interacting components operating on different levels of abstraction. In this section, we describe the upper layer, the *symbolic test case generator*, whose task it is to select symbolic test cases according to the underlying strategy. These test cases are delegated to the solver for generation of concrete test data. The solver's feed-back about infeasibility of (suffixes of) an abstract test case is used within the test case generation layer for learning to avoid these infeasible paths in future generations. Since testing always deals with bounded runs, test cases are initially generated with a fixed maximal length k. If the strategic goals cannot be met while observing this bound, k is increased and longer symbolic test cases are generated along "directions" where feasible paths may still exist. These concepts will be illustrated now for the MCDC coverage strategy introduced in Section 2.3. Application of this strategy is prepared by repeated application of the TDIOHS transformations specified in Lemma 1 and 2, so that the resulting TDIOHS $\mathcal{H}$ is still equivalent to the initial one but only possesses atomic (that is, non-conjunctive and non-disjunctive) guard conditions. As a result, achieving MCDC coverage is equivalent to exercising each transition of $\mathcal{H}$.

The central data structure used in the symbolic test case generator is the *symbolic test case tree (STCT)* which captures (feasible and infeasible) bounded-length paths through the transition graph of the TDIOHS $\mathcal{H}$ under consideration. The nodes of an STCT correspond to locations l of $\mathcal{H}$ but are augmented by a number n, so that (l, n) is a unique node identifier in the tree. This is necessary since an $\mathcal{H}$-location may occur several times in the tree if it is reachable by more than one path through the transition graph. Figures 2 and 3 show an TDIOHS transition graph and its associated STCT for bounded maximal path length $k = 5$.

Given a TDIOHS $\mathcal{H} = (Loc, Init, V, I, O, Trans)$, we introduce the associated STCT $Stct_k$ of length k by means of an algorithm which also shows how to extend $Stct_k$ into some $Stct_{k+k'}$ if the original tree is insufficient to reach the coverage goals. Let "$-$" a fresh location symbol not contained in Loc and τ_0 a fresh transition symbol. Then the components of an STCT are defined for $k = 0, 1, 2, \ldots$ as

$$Stct_k = (N_k, E_k, L_k, \phi_k, \psi_k, \sigma_k, \pi_k, \rho_k)$$

which are typed as follows:

$$\begin{aligned}
&N_k \subseteq \{(-,0)\} \cup (Loc \times \mathbb{N})\\
&E_k \subseteq N_k \times Labels \times N_k\\
&L_k \subseteq N_k\\
&\phi_k : \{\tau_0\} \cup Trans \to N_k^*\\
&\psi_k : N_k \to \{\tau_0\} \cup Trans\\
&\sigma_k : Loc \to \mathbb{N}\\
&\pi_k : N_k \to N_k\\
&\rho_k : N_k \to Trans^*
\end{aligned}$$

Components N_k and E_k denote the sets of nodes and edges, respectively, and L_k contains the leaves of the tree. The mappings $\phi_k, \psi_k, \sigma_k, \pi_k, \rho_k$ represent auxiliary data structures used for symbolic test case generation and learning about infeasible paths: Function ϕ_k maps transitions t to the list of nodes (l, n) in $Stct_k$ having t as target node. For example, transition (l_5, h, l_6) in the TDIOHS of Figure 2 is mapped to

$$\phi_5(l_5, h, l_6) = \langle (l_6, 2), (l_6, 4), (l_6, 6), (l_6, 8), (l_6, 10) \rangle$$

in the STCT of Figure 3. If the test case generator learns about the infeasibility of a path from the root of $Stct_k$ to the node (l, n) then this node is deleted from $\phi_k(t)$ and the nodes from all continuation paths of (l, n) are removed from the respective images under ϕ_k. ψ_k maps a node (l, n) of $Stct_k$ to the transition of $\mathcal{H}$ whose corresponding edge in $Stct_k$ ends at (l, n). σ_k keeps track of the counters $n = \sigma(l)$ to be associated with $\mathcal{H}$-locations l when inserting them as nodes (l, n) into the tree. π_k maps nodes to their parent nodes. ρ_k maps a node (l, n) to the symbolic test case derived from the $Stct_k$ path starting at the root and ending at (l, n). These data structures are initialized as (recall that ε denotes the trivial assignment which does not change anything)

$$\begin{aligned}
&N_0 = \{(-,0)\} \cup (Loc \times \{1\})\\
&E_0 = \{((-,0), Init(l'), \varepsilon, (l',1)) \mid l' \in Loc\}\\
&L_0 = N_0 - \{(-,0)\}\\
&\phi_0 = \{\tau_0 \mapsto \langle (l,1) \mid l \in Loc \rangle\}\\
&\psi_0 = \{x \mapsto \tau_0 \mid x \in N_0\}\\
&\sigma_0 = \{l \mapsto 2 \mid l \in Loc\}\\
&\pi_0 = \{(-,0) \mapsto (-,0)\} \cup \{(l,1) \mapsto (-,0) \mid l \in Loc\}\\
&\rho_0 = \{x \mapsto \langle\, \rangle \mid x \in N_0\}
\end{aligned}$$

Let $STCT$ denote the type of an STCT as induced by the component types introduced above. Algorithm *expandStct*() inputs an existing STCT and changes it by expanding each leaf for one transition step, if a corresponding transition exists in $\mathcal{H}$ and if the test case associated with the path from the root to this leaf has not yet been marked as infeasible.

```
**function** $expandStct(\mathbf{inout}\ stct : STCT) : \mathbb{B}$ **begin**
**let** $(N, E, L, \phi, \psi, \sigma, \pi, \rho) = stct$ **in begin**
  $retval := \texttt{false};$
  **forall** $(l, n) \in \{x \in L \mid \phi(\psi(x)) \neq \langle\, \rangle\}$ **do**
    **forall** $(\lambda, l') \in \{(a, b) \mid (l, a, b) \in Trans\}$ **do**
      $retval := \texttt{true};$
      $n' := \sigma(l');$
      $N := N \cup \{(l', n')\};$
      $E := E \cup \{((l, n), \lambda, (l', n'))\};$
      $L := (L - \{(l, n)\}) \cup \{(l', n')\};$
      $\phi := \phi \oplus \{(l, \lambda, l') \mapsto \phi(l, \lambda, l') \frown \langle (l', n') \rangle\};$
      $\psi := \psi \oplus \{(l', n') \mapsto (l, \lambda, l')\};$
      $\sigma := \sigma \oplus \{l' \mapsto n' + 1\};$
      $\pi := \pi \oplus \{(l', n') \mapsto (l, n)\};$
      $\rho := \rho \oplus \{(l', n') \mapsto \rho(l, n) \frown \langle (l, \lambda, l') \rangle\};$
    **enddo**
  **enddo**
  $expandStct := retval;$
**endlet**
**end**
```

In this algorithm $\oplus$ denotes the functional overriding operator defined by $(f \oplus \{x \mapsto y\})(z) = \textbf{if}\ z = x\ \textbf{then}\ y\ \textbf{else} f(z)$. Expanding the STCT by $k > 0$ steps is simply performed by k-fold invocation of *expandStct()*.

The complete symbolic test case generation algorithm specified in function *generateStc()* below references two generic functions encapsulating the strategy-dependent part of the generation algorithm: *select()* inputs the current state of the STCT and the set C of all nodes in the tree which already have been covered by previously generated test cases and returns a "suggestion" for the next STCT node to be covered. If, according to the underlying strategy, no more nodes need to be reached or the paths to the remaining nodes are infeasible, the function returns the root node $(-, 0)$. For the MCDC coverage used in our example strategy, *select()* is instantiated by a function which selects paths in the STCT containing edges $((l, n), \lambda, (l', n'))$ whose associated transitions (l, λ, l') in $\mathcal{H}$ have not yet been covered at all. Function *covered()* is the second generic function referenced by the generation algorithm below: It evaluates the TDIOHS structure, the STCT and the STCT nodes covered so far and returns `true` if the strategy-specific coverage goals have been reached. For MCDC coverage, *covered()* just checks whether the edges $((l, n), \lambda, (l', n'))$ covered so far in the STCT correspond to all transitions (l, λ, l') in $\mathcal{H}$.

```
**function** $select(\mathbf{in}\ stct : STCT;$
                 $\mathbf{in}\ C : \mathbb{P}(Loc \times \mathbb{N})) : (Loc \times \mathbb{N})$ **begin**
**let** $(N, E, L, \phi, \psi, \sigma, \pi, \rho) = stct$ **in begin**
  $T := \{\psi(x) \mid x \in C\};$
  $U := \{u \in Trans - T \mid \phi(u) \neq \langle\, \rangle\};$
  **if** $T = Trans \vee U = \emptyset$ **then**
    $select := (-, 0);$
  **else**
    **let** $t \in U$ **in begin**
      $select = head(\phi(t));$
**endall**
```

As shown below, the constraint solver is invoked by the generator by passing a symbolic test case $tc = \langle t_1, \ldots, t_p \rangle$ as input parameter. The solver returns the length $q \in \{0, \ldots, p\}$ of the test case prefix which was feasible. For $q < p$, the target STCT node corresponding to the first infeasible transition t_{q+1} and its subordinate STCT subtree are marked as infeasible. This task is performed by the – strategy-independent – algorithm *infeasible()* which inputs the STCT and the target node associated with t_{q+1}. Infeasibility is recorded in the STCT data structure by removing STCT nodes from the image sequences of transitions $t_{q+1}, \ldots, t_p$ under ϕ.

```
procedure infeasible(inout stct : STCT;
                     in x : N) begin
let $(N, E, L, \phi, \psi, \sigma, \pi, \rho) = stct$ in begin
  $t := \psi(x)$;
  $\phi := \phi \oplus \{t \mapsto \phi(t) - x\}$;
  forall $(x, \lambda, x') \in E$ do
    $infeasible(stct, x')$;
endall
```

In the algorithm above, $\phi(t) - x$ denotes the operation which removes element x from sequence $\phi(t)$.

Now we are ready to present the complete generation algorithm. Function *generateStc* initializes the STCT *stct* and the set C of covered nodes. The proper generation algorithm is performed within a loop that terminates when the coverage goals have been reached or when no further expansions of the STCT are possible or acceptable. For a given STCT version of maximal depth $i \cdot k$ (i is the number of expansions which have been performed so far) the algorithm proceeds by selecting a new tree node x and generating the associated symbolic test case $\rho(x)$ which is passed to the solver. If at least a prefix of $\rho(x)$ was feasible, the associated nodes are marked as covered by adding them to C. The target node of the first infeasible transition in $\rho(x)$ (if any) is passed to procedure *infeasible()* which takes care of removing the infeasible STCT nodes from the range of ϕ. When the *select*() operation returns $(-, 0)$ this means that either the coverage goal has been reached or the STCT has to be expanded.

```
function generateStc : $\mathbb{B}$ begin
  $stct := (N_0, E_0, L_0, \phi_0, \psi_0, \sigma_0, \pi_0, \rho_0)$;
  $i := 0$;  $C := \emptyset$
  while $\neg covered(H, stct, C) \wedge i < maxExpansions$
        $\wedge\ expandStctBy(stct, k)$ do
    $i := i + 1$;
    $x := select(stct, C)$;
    while $x \neq (-, 0)$ do
      $m := solve(\rho(x))$;
      $n := \#\rho(x)$;
      if $0 < m$ then
        $C := C \cup \{\pi^p(x) \mid p = n - m, n - m + 1, \ldots, n\}$;
      endif
      if $m < n$ then
        $infeasible(stct, \pi^{n-m-1}(x))$;
      endif
      $x := select(stct, C)$;
    enddo
  enddo
  $generateStc := covered(H, stct, C)$;
end
```

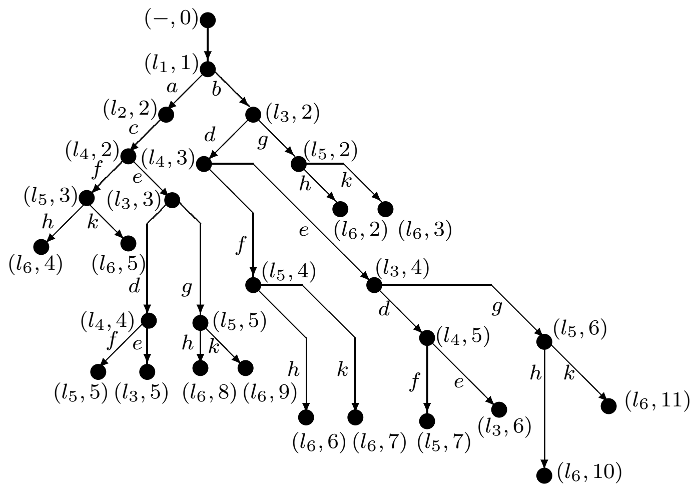

Figure 3: Symbolic test case tree.

In this algorithm $\pi^p(x)$ denotes the p-fold application of the parent function π.

A complementary algorithm which, due to the usual space limitations, is not shown here, is applied after the STCT has been expanded to a pre-defined maximal depth and some transitions t_i still remain to be covered: In this situation, a new tree containing all "reversed" paths from the target location l^* of t_i as root to the initial location of the TDIOHS represented by the leaves of the tree is incrementally constructed by backward breadth-first search, starting at l^*. A transition t_i can be identified as unreachable if this tree cannot be further expanded and each path in the tree contains an infeasible node.

4. SOLVERS FOR HYBRID CONTROL CONSTRAINTS

Aiming at test case generation, i.e. checking feasibility of a symbolic test case $\langle t_1, \ldots t_k \rangle$ with $t_i = (l_i, g_i, (\vec{x}, \vec{t_i}), l'_i) \in Trans$ and, if so, generating appropriate test input data, our constraint solver addresses satisfiability of non-linear arithmetic constraints over real-valued variables plus Boolean variables for encoding the control flow. If the t_i are concretely given (i.e., not symbolically characterized through a predicate), test case generation amounts to finding a satisfying solution to the arithmetic constraint

$$Init(l_1)[\vec{x}_1/\vec{x}] \wedge \bigwedge_{i=1}^{k-1} g_i[\vec{x}_i/\vec{x}] \wedge \bigwedge_{i=1}^{k-1} (\vec{x}' = \vec{t_i})[\vec{x}_i/\vec{x}, \vec{x}_{i+1}/\vec{x}'] \ .$$

Existence of an interval subpaving $\mathcal{I} : V_k \to \mathbb{I}$, where $V_k = \{x_i \mid x \in V, i \in \mathbb{N}_{\leq k}\}$, satisfying this constraint in the sense that

$$\begin{array}{rcll} \mathcal{I} & \models & Init(l_1)[\vec{x}_1/\vec{x}] & (1) \\ \mathcal{I} & \models & g_i[\vec{x}_i/\vec{x}] \text{ for each } i < k & (2) \\ \mathcal{I}(x_{i+1}) & \supseteq & \overset{\circ}{t}_{i,x}\ [\vec{x}_i/\vec{x}](\mathcal{I}) & \text{for each } i < k \text{ and each assignment } (x, t_{i,x}), \quad (3) \end{array}$$

is a necessary and —if point intervals are admitted— sufficient condition for real-valued satisfiability of the above constraint. Here, $\overset{\circ}{t}$ denotes the interval lifting of term t, i.e. t

with all operators lifted to their interval extension, and constraint satisfaction for the guards and the init condition is in the strong sense, i.e. $\mathcal{I} \models t_1 \leq t_2$ iff $\sup \overset{\circ}{t_1}(\mathcal{I}) \leq \inf \overset{\circ}{t_2}(\mathcal{I})$, etc. Extracting the valuations $val_i \in \mathrm{dom}^{|V|}$ for every step i from the computed interval solution $\mathcal{I}$ works as follows. For every $i < k$ and for every input variable $x \in I$ we choose an arbitrary[2] value $val_i(x) \in \mathcal{I}(x_i)$. The same is done for the initial values of all controlled variables $x \in V - I$, i.e. we select $val_0(x) \in \mathcal{I}(x_0)$ arbitrarily. For $1 < i \leq k$ we then calculate the values $val_i(x)$ of the controlled variables $x \in V - I$ from their respective terms $t_{i-1,x}$. Please note that every instance x_i of a controlled variable x is defined by exactly one term $t_{i-1,x}$ and every term $t_{i-1,x}$ contains (already assigned) variables x_{i-1} only. Obviously, combining the respective locations and valuations yields a k-bounded run $\langle (l_1, val_1), \ldots, (l_k, val_k) \rangle$. An interval solution fulfilling conditions (1) to (3) can be established by a split-and-prune algorithm, as described below. Such an algorithm is guaranteed to find a solution provided there is one which interpretes all variables by non-point intervals, and often also succeeds otherwise. The latter is achieved by exploiting the structure of the problem, namely that the values of non-input variables in some step i are functional images (mediated through assignments) of those of the variables in steps $j < i$. Thus, it makes sense to organize the search for a satisfying interval solution as a (non-chronological) backtrack search nesting splits in temporally forward direction of the transition sequence, while applying constraint propagation through contractors in arbitrary sequence and temporal direction.

The algorithm operates on a rewriting of the constraints to a form resembling three-address code, i.e. applies auxiliary variables in a such a way that it has to process a conjunction of constraints of the forms

$$\begin{array}{rcl} \mathit{bound} & ::= & \mathit{var} \geq \mathit{rational\_const} \mid \mathit{var} > \mathit{rational\_const} \\ & \mid & \mathit{var} < \mathit{rational\_const} \mid \mathit{var} \leq \mathit{rational\_const} \\ \mathit{triplet} & ::= & \mathit{var} = \mathit{var}\ \mathit{bop}\ \mathit{var} \\ \mathit{pair} & ::= & \mathit{var} = \mathit{uop}\ \mathit{var} \end{array}$$

only, where

$$\begin{array}{rcl} \mathit{bop} & ::= & + \mid - \mid * \mid / \mid \ldots \\ \mathit{uop} & ::= & - \mid \sin \mid \exp \mid \ldots \end{array}$$

Observe that these syntactic restrictions require the introduction of additional variables and conjuncts if comparisons between variables occur in the original constraint: For $z, w \in V$, a constraint $z < w$ is transformed into three conjuncts, each using three-address code representation with the syntactical restrictions as specified above, by introducing a *slack variable* s and an auxiliary variable h:

$$s > 0 \wedge h = w - z \wedge h = s$$

The algorithm then starts from the initial, unconstrained interval assignment $\mathcal{I}(v_i) = [\min \mathrm{dom}\ v_i, \max \mathrm{dom}\ v_i]$ for each $v_i \in V_k$ and iterates the following steps:

1. Initialization: All bounds $x \sim c$ from the constraint, with $\sim \in \{\geq, >, <, \leq\}$, are pushed onto an initially empty *implication queue*, which is the central data structure mediating the constraint propagation process and permitting learning from failed branches in the search tree. A set C of currently unresolved triplets and pairs is filled with those triplets $u = v\ \ op\ \ w$ and pairs $u =\ op\ \ v$ which are not satisfied in the sense of (3), i.e. which violate $\mathcal{I}(u) \supseteq \mathcal{I}(v)\ \overset{\circ}{op}\ \mathcal{I}(w)$ or $\mathcal{I}(u) \supseteq\ \overset{\circ}{op}\ \mathcal{I}(v)$, respectively.

2. Interval constraint propagation: A bound $x \sim c$ is retrieved from the implication queue and applied to the current interval valuation $\mathcal{I}$ by intersecting $\mathcal{I}$ with the models of the bound, thus replacing $\mathcal{I}$ with

$$\mathcal{I}' = \mathcal{I} \oplus [x \mapsto \mathcal{I}(x) \cap \{x \in \mathbb{R} \mid x \sim c\}] .$$

If $\mathcal{I}'(x) \neq \mathcal{I}(x)$ then the algorithm visits all triplets and pairs containing x. For each such triplet or pair, it applies the corresponding contractors (including those originating from the possible reshufflings) over and over until no further interval narrowing is achieved.[3] The resulting new, i.e. narrowed, bounds are pushed onto the implication queue. If the contractors yield an empty interval for some of the entailed variables then we proceed with *conflict analysis* in step 4. Otherwise, we remove or add the current triplet $u = v\ \ op\ \ w$ or the current pair $u =\ op\ \ v$ within the set C of unresolved constraints, depending on whether it is satisfied in the sense of $u \supseteq v\ \ \overset{\circ}{op}\ \ w$ (or $u \supseteq\ \overset{\circ}{op}\ \ v$, resp.), corresponding to condition (3). We proceed with step 2 iff the implication queue is non-empty. We are done if both the implication queue and C are empty, having constructed a satisfying assignment in the sense of conditions (1) to (3).

3. Splitting: If C is non-empty then we take some triplet $u = v\ \ op\ \ w$ or pair $u =\ op\ \ v$ from C and split the interval assignment, provided that it is not a point–interval, of some of its right-hand variables by pushing a bound tighter than the bounds assigned by $\mathcal{I}$, e.g. a bisecting bound, to the implication queue and proceed at step 2. We do *not* store the converse of that bound as a possible backtracking point, since an appropriate assertion will in case of conflict be generated by the conflict analysis scheme explained in step 4. For the sake of efficiency, we give preference to triplets or pairs containing input variables and to splitting these when selecting the triplet or pair and the variable to be split.

4. Conflict analysis and backjumping: In order to be able to tell reasons for conflicts (i.e., empty interval valuations) encountered, our solver maintains an implication graph akin to that known from propositional SAT solving (e.g., [11]): all asserted bounds are recorded in a stack-like data structure which is unwound upon backtracking when the bounds are retracted. Within the stack, each bound not originating from a split, i.e. each bound a originating from a contraction, comes equipped with pointers to its antecedents. The antecedent of a bound a is a triplet, pair or conflict clause c containing the variable v plus a set of bounds for the other free variables of c which triggered the contraction a. By following the antecedents of a conflicting assignment, a reason for the conflict can be obtained: reasons correspond to cuts in the antecedent graph, and such reasons can be "learned" for pruning the future search space by adding a *conflict clause* containing the disjunction of the negations

[2]Test strategies may refine this choice deterministically, e.g. selecting either the mean value or some border value of $\mathcal{I}(x_i)$.

[3]In practice, one stops as soon as the changes become negligible.

of the bounds in the reason. We use the unique implication point technique [11] to derive a conflict clause which is general in that it contains few bounds and which is asserting upon *backjumping* to the last split level contributing to the conflict, i.e. upon undoing all splits and contractions younger than the chronologically youngest split among the antecedents of the conflict.

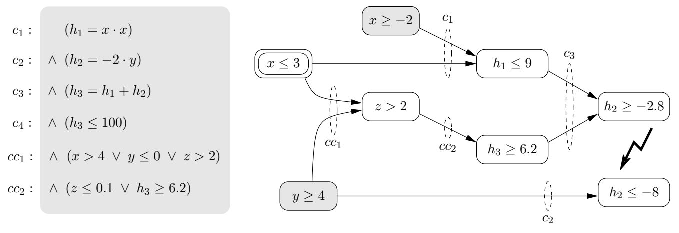
$c_1 :$ $(h_1 = x \cdot x)$
$c_2 :$ $\wedge\ (h_2 = -2 \cdot y)$
$c_3 :$ $\wedge\ (h_3 = h_1 + h_2)$
$c_4 :$ $\wedge\ (h_3 \leq 100)$
$cc_1 :$ $\wedge\ (x > 4\ \vee\ y \leq 0\ \vee\ z > 2)$
$cc_2 :$ $\wedge\ (z \leq 0.1\ \vee\ h_3 \geq 6.2)$
$x \geq -2$
$x \leq 3$
$z > 2$
$y \geq 4$
$h_1 \leq 9$
$h_3 \geq 6.2$
$h_2 \geq -2.8$
$h_2 \leq -8$
c_1
c_2
c_3
cc_1
cc_2

Figure 4: Conflict analysis

An example of our conflict analysis scheme is depicted in Fig. 4. Let $x^2 - 2y \leq 100$ be a fragment of a formula to be solved. The decomposition of this fragment into triplets, pairs and bounds $c_1, \ldots, c_4$ and already learned conflict clauses cc_1, cc_2 are shown on the left. Assume $x \geq -2$ and $y \geq 4$ have been asserted on split levels k_1 and k_2, and we are entering a new split level $k_3 > \max(k_1, k_2)$ by asserting $x \leq 3$. The resulting implication graph, ending in a conflict on h_2, is shown on the right. Edges relate implications to their antecedents, dashed ellipses indicate the propagating clauses. Following the implication chains from the conflict yields the conflict clause $\neg(x \geq -2) \vee \neg(x \leq 3) \vee \neg(y \geq 4)$ which becomes unit after backjumping to split level $\max(k_1, k_2)$, then propagating $x > 3$.

Note that, in contrast to (generalized) no-good learning as known from CSP, we are not confined to learning forbidden combinations of value assignments in the search space, which here would amount to learning disjunctions of interval disequations $x \notin I$ with x being a problem variable and I an interval. Instead, our algorithm may learn arbitrary combinations of bounds over both problem and auxiliary variables, which has proven to be extremely powerful upon benchmarks (cf. [6], where the detailed algorithm can be found). The enormous speedups obtained from learning bounds $x \sim c$ rather than no-good intervals $x \notin I$ can be traced back to the stronger pruning of the search space: while a no-good $x \notin I$ would only prevent a future visit to any subinterval of I, a bound $x \geq c$, for example, blocks visits to any interval whose left endpoint is at least c, no matter how it is otherwise located relative to the current interval valuation $\mathcal{I}(x)$. The number of visits to conflicting interval assignments thus avoided is exponential in the number of variables in the problem, thus providing speedups in the range of up to a million on constraint problems with just some thousands of variables, reflecting a corresponding pruning in the search space [6, Sect. 5].

5. CONCLUSIONS AND ONGOING WORK

We have presented methods and algorithms for automated test case and test data generation of time-discrete input-output hybrid systems (TDIOHS). The techniques presented in this article have been implemented in a test automation tool which is currently applied for testing embedded systems from the avionics and the railway domains (evaluation results regarding the efficiency and scalability of the algorithms described are currently compiled and will be shown during the workshop presentation). For practical application, the tool needs a collection of other solver components whose description is outside the scope of this paper, but which are required in order to allow for a wider range of test applications. For example, specialized solvers are currently implemented for handling linear constraint problems, and input constraints for string variables may be specified using regular expressions. In addition to the MCDC test coverage strategy described in this paper, additional strategies are currently integrated into the tool: After having reached MCDC coverage, additional test cases are constructed in order to reach a uniform statistical distribution of paths through the transition graph representing the system under test. To this end, we follow the approach described in [4].

6. REFERENCES

[1] P. Ammann, J. Offutt, and H. Huang. Coverage criteria for logical expressions. In *Proceedings of the 2003 International Symposium on Software Reliability Engineering (ISSRE '03)*, pages 99–107, Denver, CO, 2003.

[2] K. Berkenkötter, S. Bisanz, U. Hannemann, and J. Peleska. The HybridUML Profile for UML 2.0. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(2):167–176, 2006.

[3] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–186, Mar. 1978.

[4] A. Denise, M.-C. Gaudel, and S.-D. Gouraud. A generic method for statistical testing. In *Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)*, pages 25–34, 2004.

[5] B. J. Ellis. Automation for exception freedom proofs. In *18th IEEE International Conference on Automated Software Engineering, 2003.* IEEE, 2003.

[6] M. Fränzle, C. Herde, S. Ratschan, T. Schubert, and T. Teige. Interval Constraint Solving Using Propositional SAT Solving Techniques. CP 2006 Workshop on the Integration of SAT and CP Techniques, 2006.

[7] T. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, pages 278–292. IEEE Computer Society Press, 1996.

[8] L. Jaulin, M. Kieffer, O. Didrit, and É. Walter. *Applied Interval Analysis*. Springer-Verlag, 2001.

[9] SC-167. *Software Considerations in Airborne Systems and Equipment Certification*. RTCA, 1992.

[10] J. Springintveld, F. Vaandrager, and P. D'Argenio. Testing timed automata. *Theoretical Computer Science*, 254(1-2):225–257, March 2001.

[11] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In *IEEE/ACM International Conference on Computer-Aided Design*, 2001.