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. In contrast to test automation techniques for purely discrete controllers this involves the generation of both discrete and real-valued, potentially time-continuous, input data to the system under test. To this end, the test automation techniques introduced here are allocated in two-layers: The upper layer contains a symbolic test case generator constructing test cases as paths through an abstracted representation of the transition graph specifying 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. This component applies interval analysis techniques identifying the domains from where to pick the appropriate test data. The well known complexity problems of the various paving algorithms used in interval analysis are circumvented by three main concepts: First, sequences of constraints, each element representing a conjunct of a larger global constraint, are processed separately, thereby keeping the dimension of the local constraint problems involved at an acceptable level. Second, interval vectors containing the global solution set are contracted using forward-backward interval constraint propagation. Third, both symbolic test case generator and constraint solver learn to avoid symbolic transition sequences whose prefixes are already known to be infeasible and to avoid interval solutions for local constraints which are known to be in conflict with other local constraints to be satisfied for the same symbolic test case, respectively.

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[1]. 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 [4]. Moreover, they also apply to structural software testing, where the TDIOHS represent the control flow graphs of the software units under test.

*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`).

[1]The methods described here alternatively allow to specify the guaranteed behavior of the operational environment interacting with the controller and to derive tests from this environment specification. In this paper, however, we only focus on SUT specifications, while leaving environment behaviour unspecified.

This is the extended version of an article to appear in the Proceedings of the SOQUA'06, November 6, 2006, Portland, OR, USA.

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 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 subtask: 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 by unions of non-intersecting intervals $I \subset \mathbb{R}^n$, where n is the number of free variables involved in the CSP. These collections of intervals are called *pavings* of the constraint solution set $\mathbb{S} \subseteq \mathbb{R}^n$. This approach is supported by providing modified versions of arithmetic and set operations working on and resulting in intervals or unions thereof.

In many applications of interval analysis – for example, when approximating the volume of an n-dimensional set specified by a CSP – pavings have to be refined until they completely cover (approximation from outside) or fill (approximation from inside) the solution set with sufficient precision. Performing this task with the most basic algorithms of interval analysis – the so-called *regular subpavings* – generally requires exponential time. Therefore additional algorithms with polynomial complexity have been provided (the *contractors*), in order to narrow the discrepancy between solution sets and approximating intervals, so that regular subpavings only have to be applied as "last resort" during the final approximation steps.

For the purpose of test generation, however, CSPs have another quality which leads to further complexity reductions: It generally suffices to select a small number of elements $x \in \mathbb{S}$ (in most cases just one) from the solution set of a CSP. These x are used as inputs to the SUT at certain points in time t during the test execution (t may be a component of the solution vector x). As a consequence, it is possible to stop the paving process as soon as a sufficient number of solutions $x \in \mathbb{S}$ have been found. This approach can be further optimized by noting that for the purpose of test generation, CSPs c are typically constructed from conjuncts, $c \equiv c_1 \wedge \ldots \wedge c_n$, each c_i involving a much smaller number of free variables than c. The c_i correspond, for example, to guard conditions of specific transitions or state invariants/flow conditions to be ensured or violated on purpose by the means of state changes provoked by the test environment. In these situations, interval analysis suggests to apply *finite sub-solvers* ϕ_i for providing interval approximations of each c_i-solution set and find a suitable way of integrating the partial solutions into the global one covering $\mathbb{S}$. Keeping in mind that complete approximations of $\mathbb{S}$ are not required we devise an algorithm which identifies only a limited number of intervals $I(c_i) \subseteq \mathbb{S}(c_i)$ and then propagates the $I(c_i)$ to the other c_j, with the purpose of stopping all further approximations of $\mathbb{S}(c_i)$ if $I(c_i)$ is already compatible with all other c_j. If this is not the case, the algorithm can *learn* which edges of $I(c_i)$ are incompatible with one or more other constraints c_j: It is useless to further refine the paving of $\mathbb{S}(c_i)$ with any intervals $J(c_i)$ sharing edges with $I(c_i)$ which are already known to be in conflict with c_j. If some of the variables x_i are not real-valued but integral numbers, the paving process is restricted in these dimensions i to edges containing at least one integral number. This process is combined with forward-backward interval constraint propagation which turns out to be a very powerful general-purpose[2] contractor.

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.

1.1 Overview

Section 2 introduces the basic notions about testing, time-discrete input-output hybrid systems and interval analysis. Section 3 introduces the formal notion of a symbolic test case and describes several test strategies suitable for pursuing different quality objectives. In Section 4, a framework for test case/test data generation systems is introduced, where all the relevant components cooperating for this purpose, together with basic interfaces and interaction principles have been identified. Sections 5 and 6 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 7 contains the conclusions and describes work in progress beyond the scope of this paper.

1.2 Related Work

Our TDIOHS differ from Henzinger's *hybrid automata (HA)* introduced in [11] mainly by the fact that TDIOHS do not model invariants and flow conditions in an explicit way. The discussion of TDIOHS in Section 2.2 will show that they

[2]Specialized, potentially more powerful, contractors exist for restricted types of constraints, such as linear CSPs.

basically represent time-discrete dynamical control systems where all flow conditions have been handled beforehand, using discretized solutions. Our TDIOHS were motivated by similar ideas as the rectangular hybrid automata introduced in [12].

For practical applications it is useful to apply variants of TDIOHS allowing for hierarchic decomposition of control modes into new "lower-level" TDIOHS refining the respective modes. Examples of hierarchic HA approaches have been given in [2] and [4], the latter integrating them as a profile into the Unified Modeling Language UML2.0. In the context of the present paper it suffices to present the algorithms involved for the flat original version of HA: Their application to hierarchic specifications is practically implemented by introducing a specific strategy in the sense of Section 3 (this is our preferred approach), or, alternatively, by flattening the specification to a non-hierarchic representation [9].

A formal basis for testing has been initially established in the context of finite-state machines [5] and process algebras [10] without time. The main results showed that test strategies existed for establishing semantic equivalence or some refinement relation between the SUT and its specification. These results could be extended to timed I/O automata [18]. In contrast to this, test generation for the purpose of proving equivalence between SUT and its specification is no longer possible if more general HA are involved. This follows from undecidability results given in [12]. As a consequence, it seems both desirable to elaborate test strategies which are promising to uncover at least certain types of failures (Section 3) and, alternatively, to restrict the class of HA which are admissible as controller models, so that more decidability results are available (see, for example, [12]).

The design of the symbolic test case generator described in Section 5 has been influenced by the novel solutions for graph traversal with the objective of implementing specific strategies or, equivalently, coverage criteria suggested in [6].

The definitions and results from interval analysis used in this paper are based on [13, 14]. Interval analysis has been frequently applied in combination with abstract interpretation for various aspects of static software analysis; see, for example, [7]. Several authors have investigated more general geometric forms (polyhedra) than n-dimensional intervals for the purpose of model checking for hybrid systems, see [3] for an example from the field of reachability analysis.

A rather comprehensive description of testing terminology can be found in [17]; the technical terms required for this paper are introduced in Section 2.1.

2. CONCEPTS AND TERMINOLOGY

2.1 Testing Terminology

A typical hybrid systems control scenario is depicted in Figure 1: The controller – this may also consist of a network of cooperating computers – interacts with the physical environment to be controlled (also called *equipment under control (EUC)*) by monitoring observables $(y_1, \ldots, y_k)$ and affecting EUC behavior by means of controls $(u_1, \ldots, u_m)$. The required behavior of the physical system can be changed

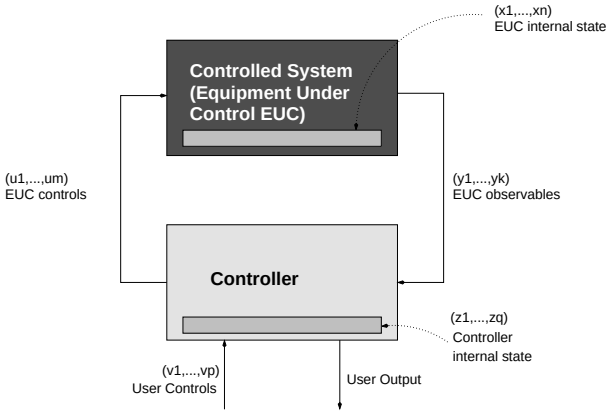

Figure 1: Controller and equipment under control.

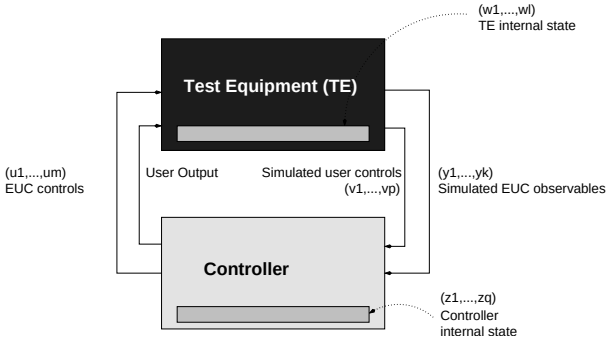

Figure 2: Controller and test equipment replacing real operational environment.

through the user – or another computer interacting with the controller – by changing the user control inputs $(v_1, \ldots, v_p)$. The controller outputs status data to the user. All interface data u_i, y_i, v_i as well as the user output may involve both discrete and analog values, the latter transmitted to the EUC by means of actuators performing digital/analog (D/A) conversion and received from the EUC using sensors and A/D converters.

For testing hybrid control systems in an automated way it is necessary that the *test equipment (TE)* replaces the complete operational environment of the controller which is now denoted as *system under test (SUT)* (Figure 2): The TE simulates user inputs $(v1, \ldots, v_p)$ and EUC behavior $(y_1, \ldots, y_k)$ as far as visible to the SUT and monitors SUT actions on its EUC outputs $(u_1, \ldots, u_m)$ and user output interfaces, for the purpose of checking compliance of the SUT with its specified behavior. For *black box testing* these are the only interfaces which can be accessed by the TE. If a portion of the internal SUT state $(z_1, \ldots, z_q)$ can be observed or manipulated this is called *grey box testing*.

Testing is performed in order to ensure – or, at least, increase the confidence – that a software-based product complies with a set of given *quality criteria*. Software quality criteria are defined for example in ISO/IEC 9126 [1]; typical characteristics – so-called *quality attributes* – are functional correctness, reliability, usability, efficiency, maintainability

and portability. For a particular SUT specific quality objectives apply, this results in a selection and prioritization of the associated attributes. In order to identify tests which are particularly useful to uncover product deficiencies violating the aspired quality goals, *test strategies* are elaborated, defining the *test levels* and the *test case design techniques* to be used in the test campaign. Test levels define the portions of the SUT – for example, the complete system, sub-systems, modules – which are exercised during the test by stimulating its input interfaces and monitoring its outputs. Examples are system integration tests, software integration tests and module tests. Test case design techniques deal with the development of (timed sequences of) input data to the SUT and the specification of the expected SUT reactions to these inputs. A test case may be regarded as a partial specification of the SUT suitable for checking a particular well-defined objective or requirement. Depending on the test case design technique, a test case may already specify the concrete SUT pre-state from where the test objective can be checked, the inputs driving the SUT into this pre-state, the consecutive inputs and expected SUT outputs and the associated timing requirements and admissible post-states. Alternatively, a test case may be represented by more abstract specifications and the determination of concrete *test data* is performed in an additional refinement step. One or more test cases are executed in suitable causal ordering within a *test procedure* which describes the concrete actions for their execution. For automated testing, procedures are executable programs running on the test equipment. In particular, test procedures contain all the mechanisms – for example, driver calls and data conversion routines – to generate the test data and check concrete SUT output against expected results, as specified in the associated test cases.

2.2 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 $I \cap O = \emptyset$,
- $Guard$ is the set of all quantifier–free predicates over V,
- $Init : Loc \to Guard$ is a function mapping locations to predicates,
- $Assign$ is 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 $\vec{t} = (t_1, t_2, \ldots) \in T^{|V-I|}$ in which T is the set of all terms over V,
- $Trans \subseteq Loc \times Guard \times Assign \times Loc$ is the set of *transitions*.
- $Labels = \{\lambda \in Guard \times Assign \mid \exists l_1, l_2 \in Loc : (l_1, \lambda, l_2) \in Trans\}$ is the set of *transition labels*.

Let $val \in \mathrm{dom}^{|V|}$ 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. $\forall i \in \mathbb{N}\ \exists (l, g, (\vec{x}, \vec{t}), l') \in Trans : l = l_i, l' = l_{i+1}$, $val_i(g) = \texttt{true}$, $val_{i+1}(x_1) = val_i(t_1), \ldots, val_{i+1}(x_{|V-I|}) = val_i(t_{|V-I|})$.

The TDIOHS $\mathcal{H}$ is called *deterministic* if in every possible run $\langle (l_1, val_1), (l_2, val_2), \ldots \rangle$ of $\mathcal{H}$ only one transition is enabled at a time:

$$\begin{aligned} \forall i \in \mathbb{N}; (l_i, g, a, l), (l_i, g', a', l') \in Trans : \\ val_i(g) = \texttt{true} \wedge val_i(g') = \texttt{true} \implies \\ (l_i, g, a, l) = (l_i, g', a', l') \end{aligned}$$

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 $\vec{t}$ 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,

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

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:

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

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:

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

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 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:

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

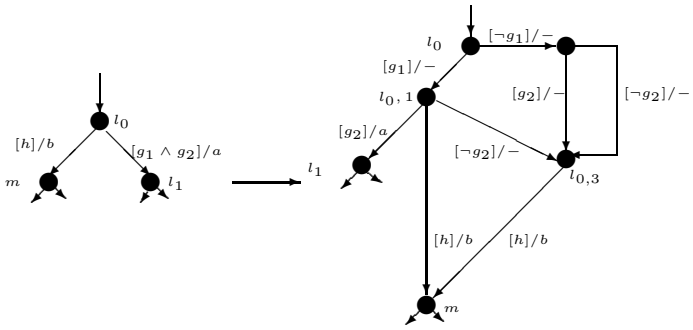

Figure 3: Simplification of conjunctive guards.

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 3.

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. *$Loc_P(\mathcal{H}_2) = Loc_P(\mathcal{H}_1) \cup \{l_{0,1}, l_{0,2}, l_{0,3}\}$, where $l_{0,1}, l_{0,2}, l_{0,3}$ are fresh location identifiers,*
2. *$Trans(\mathcal{H}_2) = (Trans(\mathcal{H}_1) - T_1) \cup T_2$, where*

$$
\begin{aligned}
T_1 &= \{(l, g, b, l') \in Trans(\mathcal{H}_1) \mid l = l_0\} \\
T_2 &= \{(l_0, g_1, \varepsilon, l_{0,1}), (l_{0,1}, g_2, a, l_1), (l_{0,1}, \neg g_2, \varepsilon, l_{0,3}), \\
&\quad (l_0, \neg g_1, \varepsilon, l_{0,2}), (l_{0,2}, g_2, \varepsilon, l_{0,3}), (l_{0,2}, \neg g_2, \varepsilon, l_{0,3})\} \cup \\
&\quad \{(l_{0,1}, g, b, m) \mid (l_0, g, b, m) \in Trans_1 - \{\tau_0\}\} \cup \\
&\quad \{(l_{0,3}, g, b, m) \mid (l_0, g, b, m) \in Trans_1 - \{\tau_0\}\}
\end{aligned}
$$

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.

PROOF. Locations $l_{0,1}, l_{0,2}, l_{0,3}$ are valid processing locations, since the only transitions entering them do not change any variable values. Moreover, since $\mathcal{H}_1$ is I/O safe, guard $g_1 \wedge g_2$ does not refer to input variables, so that the guards $g_1, g_2, \neg g_1, \neg g_2$ used in the new transitions of $\mathcal{H}_2$ have free variables in $V - I$, too.

Let r_1 be a run of $\mathcal{H}_1$. We will construct an I/O-equivalent run of $\mathcal{H}_2$. To this end, we decompose r_1 into segments,

$$r_1 = u_1^1 \frown w_1^1 \frown u_1^2 \frown w_1^2 \frown \dots$$

such that the u_1^i do not visit location l_0, whereas the segments $w_1^j = \langle\ (l_0, v_0^j), (m^j, z^j)\ \rangle$ start in location l_0 and perform transitions into any post-location of l_0. In particular, l_1 is such a possible post-location m, and it is possible that several consecutive $w_1^j, w_1^{j+1}, \dots$ occur in r_1 if the post-location of m^j is again l_0. Our goal is to construct segments w_2^j such that

$$r_2 = u_1^1 \frown w_2^1 \frown u_1^2 \frown w_2^2 \frown \dots$$

is I/O-equivalent to r_1: Since u_1^1 does not visit location l_0, it can be performed by $\mathcal{H}_2$ as well. It remains to show the existence of w_2^1 such that $u_1^1 \frown w_1^1$ leaves $\mathcal{H}_1$ in the same state as $u_1^1 \frown w_2^1$ leaves $\mathcal{H}_2$. This existence is shown independently of the position j of w_2^j, so that the equivalence $r_1 \sim r_2$ follows by induction. In order to construct the w_2^j, distinguish 4 cases regarding the valuation $v_0^j(g_1 \wedge g_2) \equiv v_0^j(g_1) \wedge v_0^j(g_2)$:

Case 1: $v_0^j(g_1) = \texttt{true} = v_0^j(g_2)$. Set

$$w_2^1 = \langle\ (l_0, v_0), (l_{0,1}, v_0), (m^j, z^j)\ \rangle$$

It remains to show that $u_1^1 \frown w_2^1$ is a run of $\mathcal{H}_2$. By construction of T_2 and since $v_0^j(g_1) = \texttt{true}$, $(l_0, g_1, \varepsilon, l_{0,1})$ is the only possible transition from state (l_0, v_0) in $\mathcal{H}_2$. Since assignment ε does not change any local or output variables, v_0 is also a possible valuation in post-location $l_{0,1}$ (which is realized by keeping the inputs stable while the transition is performed). As a consequence, the same guards evaluating to `true` in state (l_0, v_0) remain `true` in state $(l_{0,1}, v_0)$. Therefore g_2 also evaluates to `true` in $l_{0,1}$, so transition $(l_{0,1}, g_2, a, l_1)$ is enabled in state $(l_{0,1}, v_0)$ of $\mathcal{H}_2$. Again by construction of T_2, all other transitions enabled in $\mathcal{H}_1$ in state (l_0, v_0) are enabled in $\mathcal{H}_2$ with new pre-state $(l_{1,0}, v_0)$ but identical assignments and post-locations. Since inputs to $\mathcal{H}_2$ can be freely chosen, (m^j, z^j) is a possible post-state of $(l_{0,1}, v_0)$ in $\mathcal{H}_2$.

The analogous argument applies to Case 2: $v_0^j(g_1) = \texttt{true} = \neg v_0^j(g_2)$ – set

$$w_2^1 = \langle\ (l_0, v_0), (l_{0,1}, v_0), (l_{0,3}, v_0), (m^j, z^j)\ \rangle$$

and Case 3: $v_0^j(g_1) = \texttt{false} = \neg v_0^j(g_2)$ – set

$$w_2^1 = \langle\ (l_0, v_0), (l_{0,2}, v_0), (l_{0,3}, v_0), (m^j, z^j)\ \rangle$$

and Case 4: $v_0^j(g_1) = \texttt{false} = v_0^j(g_2)$ (set w_2^1 as in Case 3).

Conversely, we will now consider runs r_2 of $\mathcal{H}_2$ structured as described above, and construct equivalent runs $r_1 \in Run(\mathcal{H}_1)$. To this end, we consider the possible transitions in $\mathcal{H}_2$ starting from location l_0. The possible cases have been identified already above, and we proceed in the same order.

Case 1: $v_0^j(g_1) = \texttt{true} = v_0^j(g_2)$ and

$$w_2^j = \langle\ (l_0, v_0), (l_{0,1}, v_{0,1}), (m^j, z^j)\ \rangle$$

where m is a location which exists both in $\mathcal{H}_1$ and $\mathcal{H}_2$. Since $(l_0, g_1, \varepsilon, l_{0,1})$ is the only possible transition from state (l_0, v_0) in $\mathcal{H}_2$ and ε does not change local variables and outputs, valuations v_0 and $v_{0,1}$ may only differ with respect to input variables, that is, $v_0|(V - I) = v_{0,1}|(V - I)$. Since guards of I/O-safe TDIOHS never depend on input variable valuations, we get an equivalent run $r_2' \sim r_2$ if w_2^j in r_2 is exchanged by

$$w_2^{j'} = \langle\ (l_0, v_{0,1}), (l_{0,1}, v_{0,1}), (m^j, z^j)\ \rangle$$

Now construct an equivalent run r_1 by setting

$$w_1^j = \langle\ (l_0, v_{0,1}), (m^j, z^j)\ \rangle$$

Again the construction of T_2 implies that any state transition from $(l_{0,1}, v_{0,1})$ to (m^j, z^j) in $\mathcal{H}_2$ has an equivalent $\mathcal{H}_1$-transition from $(l_0, v_{0,1})$, ending in the same target state (m^j, z^j).

The other cases are handled in an analogous way:
For Case 2, $v_0^j(g_1) = \texttt{true} = \neg v_0^j(g_2)$, use $w_2^{j'} = \langle\ (l_0, v'_{0,1}), (l_{0,1}, v_{0,3}), (l_{0,3}, v_{0,3}), (m^j, z^j)\ \rangle$, for Case 3, $v_0^j(g_1) = \texttt{false} = \neg v_0^j(g_2)$ and Case 4, $v_0^j(g_1) = \texttt{false} = v_0^j(g_2)$ use $w_2^{j'} = \langle\ (l_0, v_{0,3}), (l_{0,2}, v_{0,3}), (l_{0,3}, v_{0,3}), (m^j, z^j)\ \rangle$. This completes the proof of the lemma. □

An analogous lemma holds for disjunctive guard composition (see Figure 4):

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*

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

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.

PROOF. In analogy to the proof of Lemma 1. □

Remarks. The following remarks aim at clarifying the type of "real-world" systems which can be suitably modeled as TDIOHS:

1. The variable model for input and output matches well for systems reading from and writing to interfaces with shared variable character. Examples are (a) shared memory interfaces of software processes communicating with other tasks on the same computer, (b) hardware I/O via DMA and (c) hardware I/O via dual-ported RAM interface boards.
2. Racing conditions between environment writing to input variables which are read simultaneously by the system are avoided by introducing ring buffers accessed in shared memory, DMA or dual-ported RAM.
3. TDIOHS with dedicated input and output locations are realized by the widely used "main loop" programming model for sequential reactive controllers: The program body consists of an infinite control loop whose body is structured into the (a) input phase which copies input data from shared memory etc. to local variables, (b) processing phase which only operates on local variables, so that instable inputs during that phase are ignored and outputs remain stable as well, and (c) output phase where values of certain local variables are copied to the associated output variables. Observe that this programming model is also intrinsic to *programmable logic controllers (PLCs)*.

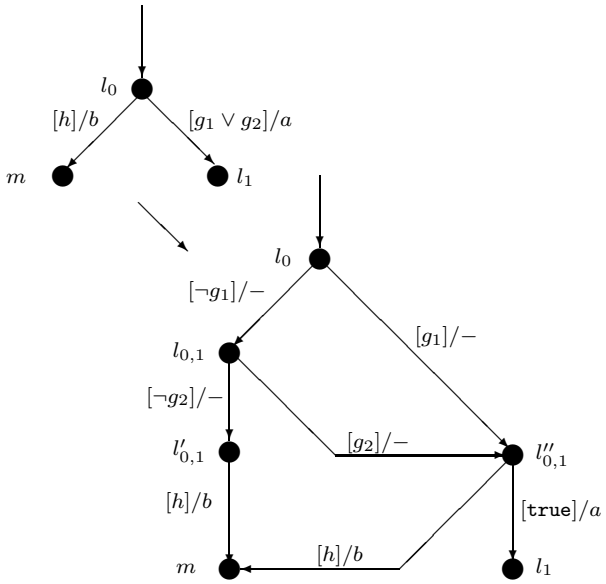

Figure 4: Simplification of disjunctive guards.

4. TDIOHS are also well-suited for modeling sequential functions f or methods or systems thereof ("function call trees"), where the functions operate on both discrete and floating point variables and may call sub-functions which are considered as being part of the testing environment (so-called *function stubs*): When f is called and input parameters are passed on the function stack f is in its initial input location. If f reads from global variables which may by asynchronously manipulated by the environment, this is usually done by copying the global data x_g to local variables x_l , while parallel write access to x_g is blocked, for example, by using semaphores. Obviously, this corresponds to an input location, too. If f returns while passing a return value over the stack and writing to reference parameters or global variables this corresponds to an output location. When f calls a sub-function in a statement like $z = h(y)$; this is represented by an output location passing y to the called function, followed by an input location where the function return of h is written to z.
5. Time information provided by the clock of a computer system is modeled as a (discrete or real-valued) input variable. Since guards are arbitrary Boolean expressions, timeout conditions can be expressed, but the clock values are refreshed only at discrete points in time, namely at input locations.

6. The internal processing parts of I/O safe TDIOHS can be easily decomposed while preserving equivalence: If an assignment $x_1 := x_2 + x_3$ is re-structured into two consecutive ones, say $x_1 := x_2$ and $x_1 := x_1 + x_3$, introducing a new location for this purpose, the new TDIOHS will produce equivalent runs because the x_i are local variables, and therefore their valuation cannot be changed by inputs during internal processing steps.

7. The definition of input and output locations used in this paper differs from the concept of *timed I/O automata* used in [18]. The main reason for this is that we do not consider atomic actions which have to be synchronized in communications between the system and its environment. For the types of systems described above our concept represents a very realistic model: The environment may change inputs to the system at any time without being blocked, since I/O is based on shared variables. The system however, will detect the input change only if it is stable at least until the next input location has been reached. Otherwise the input value is simply ignored.

2.3 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 $I, J, I_i, ...$ 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 $I, J, ...$ 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 [13]. 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 *test* over $\mathbb{R}^n$ is an n-tuple predicate, i.e. a Boolean-valued function from $\mathbb{R}^n$ to $\mathbb{B} = \{0, 1\}$. We say a test is 1 over an interval $[a, b]$, if for all $x \in [a, b]$ the value of the predicate is 1. In other words the predicate holds over $[a, b]$.

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 $I = [a_1, b_1] \times ... \times [a_i, b_i] \times ... \times [a_n, b_n]$ 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.

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,

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

Note that the initialisation 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 [5, 18] – 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:* High-level requirements are identified by application experts as collections of symbolic test cases covering specific locations and transition sequences. The associated test suite ensures that each requirement has been exercised at least once within the suite.
- *Structural coverage:* Test case selection is driven by objectives to cover as many states, transitions or paths of the TDIOHS specification as possible.

- *Absence of specific failure types:* Based on given fault models, test cases with the capability to uncover the failure types of interest are constructed. For example, all implemented transitions with correctly specified guard conditions and assignments, but incorrect target locations, represent a well-defined failure class. A suitable strategy for this class would be based on testing characterization traces as described in [5].

- *Uniform statistical test case distribution:* In this strategy (see [6]) symbolic test cases are constructed using a randomized strategy ensuring that a uniform coverage of all branches is achieved: Since different transitions t_1, t_2 may possess different numbers of subsequent transition paths, just exercising t_1 and t_2 with the same frequency does not ensure a uniform distribution of visited transitions on their respective path subtrees.

To illustrate the test case and test data generation concepts in this paper, we focus on the *Modified Condition / Decision Coverage (MCDC)*. Quoting the standard [16], 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.'* Obviously, MCDC is a structural coverage strategy, but it also covers aspects related to fault models: Tests driven by the MCDC strategy are likely to uncover faults where – due to an erroneous guard implementation – the wrong transition is taken from a given location. MCDC is required by the standard [16] 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 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.

Further observe that MCDC is not the same as *multiple condition coverage* defined in [15]: The latter requires that all possible combinations of conditions should be exercised by a test; these combinations do also include some that do not affect the decision's outcome: For example, if statement `if (A and B)` requires test cases resulting in all four combinations of `A, not(A), B, not(B)`, whereas MCDC only requires combinations (1) `A and B`, (2) `not(A)` and `B` arbitrary, (3) `A and not(B)`: Since C/C++ compilers do not evaluate `B` if `not(A)` holds, `B` does not affect the decision's outcome if `A` is already `false`.

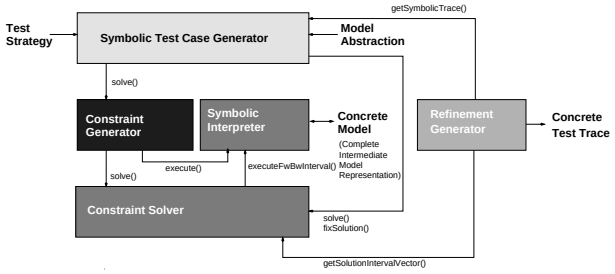

Figure 5: Generic architecture for test automation systems.

4. A FRAMEWORK FOR AUTOMATED TEST GENERATION

For the construction of test cases and associated concrete data which are suitable for testing hybrid systems, numerous techniques and associated algorithms have to be combined. In order to keep the number of associated software components manageable we suggest a basic test automation framework that allows to "allocate" concrete solutions – that is, programmed algorithms – on specific components of the framework and specifies the required interfaces to other components. This framework is shown in Figure 5.

The task of the *symbolic test case generator* is to implement the objectives of a given strategy in an abstract manner. To this end, it associates a coverage goal with the strategy and traverses an abstraction of the specification model in order to identify a collection of symbolic test cases sufficient to yield the required coverage. For TDIOHS the model abstraction consists of the transition graph of the hybrid system, where each guard g and assignment a of a transition (l_0, g, a, l_1) has been replaced by the label λ identifying the guard/assignment pair (g, a).

The lower layer of the framework is represented by the *constraint solver*. Its task is to generate the concrete input data to the SUT turning a given symbolic test case into an executable bounded run, or otherwise to signal infeasibility of this test case. It is advisable to enforce a separation of concerns, so that the solver only knows about constraints, without having to observe their origin as paths through the abstracted transition graph of a TDIOHS. To this end, the *constraint generator* is responsible for collecting the guard conditions to be fulfilled in order to cover (parts of) the traces suggested by the test case generator. For this task the effects of assignments on variable valuations and, consequently, guard conditions has to be taken into account. These effects are calculated by the *symbolic interpreter* performing abstract interpretation of assignment expressions and term-replacements in guards.

In our tool implementation the symbolic interpreter supports three interpretation modes:

1. For a given sequence of concrete input values the interpreter calculates the concrete assignments, guard

evaluations and outputs. This mode is used to validate concrete input/output traces against logic assertions (trace logic or temporal logic) and to determine the path covered by concrete input data sets suggested by the testing specialists.

2. For a given sequence of *intervals* for each input variable the interpretation is performed using the interval arithmetic version of each operator and mathematical function, as sketched in Section 2.3. This mode is used by the solver for contracting interval vectors known to represent supersets of constraint solutions. The mode is offered in both directions along a trace, as is required by the forward-backward constraint propagation used as the main contraction mechanism (Section 6).

3. In the abstract interpretation mode all input sequences on variables $x \in I$ are represented by abstract symbol sequences $x_1, x_2, \ldots$ and the interpreter only performs term replacement as specified by the assignments.

The solution constructed by the constraint solver consists of a sequence of interval vectors $I_0, I_1, \ldots$, associating an interval of possible values for each input variable and input situation of the symbolic test case. The *refinement generator* selects concrete values from these vectors, which results in a sequence of concrete input valuations $val_1(x), val_2(x), \ldots$ for each input variable x.

5. SYMBOLIC TEST CASE GENERATION

As sketched in Section 4, 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 feedback 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 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 6 and 7 show an TDIOHS transition graph and its associated STCT for bounded maximal path length $k = 5$.

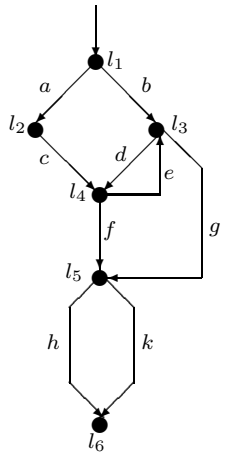

Figure 6: Symbolic TDIOHS representation ($Init(l) =$ `false` for all $l \neq l_1$).

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:

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

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 6 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 7. 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)

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

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*(**inout** $stct : STCT$) : $\mathbb{B}$ **begin**
let $(N, E, L, \phi, \psi, \sigma, \pi, \rho) = stct$ **in begin**
  $retval :=$ `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**
      $retval :=$ `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 $(f \oplus \{x \mapsto y\})(z) =$ **if** $z = x$ **then** y **else** $f(z)$. Expanding the STCT by $k > 0$ steps is simply performed by k-fold invocation of *expandStct()*:

function *expandStctBy*(**inout** $stct : STCT$; **in** $k : \mathbb{N}$) : $\mathbb{B}$ **begin**
  $i := 0$;
  while $(i < k \wedge expandStct(stct))$ **do**
    $i := i + 1$;
  enddo
  expandStctBy := $(0 < i)$;
end

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 *select*(**in** $stct : STCT$; **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 = $head(\phi(t))$;
    endlet
  endif
endlet
end

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 *covered*($H : TDIOHS$; $stct : STCT$; $C : \mathbb{P}(Loc \times \mathbb{N})$) : $\mathbb{B}$ **begin**
  let $(N, E, L, \phi, \psi, \sigma, \pi, \rho) = stct$ **in begin**
  covered := $(Trans = \{\psi(x) \mid x \in C\})$;
endlet
end

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($stct, x'$);
  enddo
endlet
end

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

In this algorithm $\pi^p(x)$ denotes the p-fold application of the parent function π, starting at $\pi^0(x) = x$, so $\pi^p(x)$ is the p^{th} predecessor of x.

A complementary algorithm which 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.

6. 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

$$Init(l_1)[\vec{x}_1/\vec{x}] \wedge \bigwedge_{i=1}^{k-1} g_i[\vec{x}_i/\vec{x}] \wedge \bigwedge_{i=1}^{k-1} (\vec{x}' = \vec{t}_i)[\vec{x}_i/\vec{x}, \vec{x}_{i+1}/\vec{x}'] \ .$$

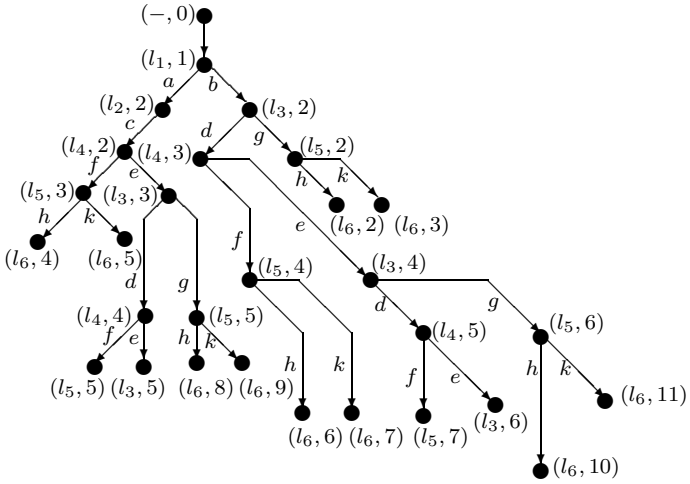

Figure 7: Symbolic test case tree.

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

$$\mathcal{I} \models Init(l_1)[\vec{x}_1/\vec{x}] \quad (1)$$

$$\mathcal{I} \models g_i[\vec{x}_i/\vec{x}] \text{ for each } i < k \quad (2)$$

$$\mathcal{I}(x_{i+1}) \supseteq \overset{\circ}{t}_{i,x}\,[\vec{x}_i/\vec{x}](\mathcal{I}) \quad \text{for each } i < k \text{ and each assignment } (x, t_{i,x}), \quad (3)$$

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[3] 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

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

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 $\mathcal{I}(u) \supseteq \mathcal{I}(v)\ \ \mathring{op}\ \ \mathcal{I}(w)$ or $\mathcal{I}(u) \supseteq\ \mathring{op}\ \mathcal{I}(v)$, 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.[4] 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 $u \supseteq v\ \ \mathring{op}\ \ w$ (or $u \supseteq\ \mathring{op}\ \ v$, 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., [19]): 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 of the bounds in the reason. We use the unique implication point technique [19] 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.

An example of our conflict analysis scheme is depicted in Fig. 8. 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.

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

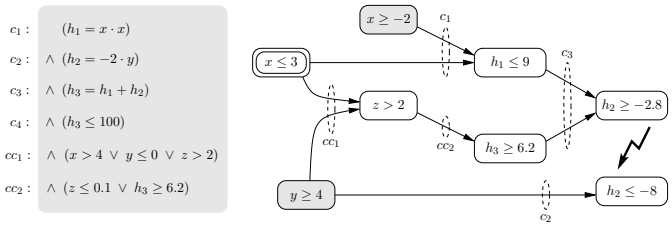

Figure 8: Conflict analysis

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) nogood 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. [8], where the detailed algorithm can be found). The enormous speedups obtained from learning bounds $x \sim c$ rather than nogood intervals $x \notin I$ can be traced back to the stronger pruning of the search space: while a nogood $x \notin I$ 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 [8, Sect. 5].

7. 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). For separation of concerns, the methods applied are structured into two layers: (1) The upper layer consists of the *symbolic test case generator* selecting test cases according to various test coverage strategies. Symbolic test cases are sequences of transitions which have to be checked with respect to feasibility by constructing concrete input data for the system under test. This task is delegated to (2) the *constraint solver* representing the lower layer of this test automation system architecture. The solver constructs input data using techniques from interval analysis. The well-known complexity problems to be encountered when applying regular subpavings to constraint solving problems in a straight-forward way are avoided by

- decomposing the global constraint problem into a sequence of conjunctive local constraints – each conjunct associated with a transition guard of the symbolic test case – and by applying subpaving techniques only to these local constraints usually possessing fewer free variables than the global one,
- using forward-backward interval constraint propagation as a very powerful contractor.

The algorithmic performance on both layers is further optimized by learning to avoid symbolic transition sequences whose prefixes are already known to be infeasible (upper layer) and to avoid interval solutions for local constraints which are known to be in conflict with other local constraints to be satisfied for the same symbolic test case (lower layer).

The "real-world systems" where these test generation techniques can be applied are

- *Discrete dynamical systems.* Reactive control systems with a single-task main loop structure processing both discrete and analogue data on a discrete time basis typically synchronized with main loop cycles of constant duration Δt. For these systems, time-continuous evolutions have to be pre-processed, so that flow conditions specified by differential equations can be calculated in each main loop cycle as discrete Δt integration steps $\vec{x}_{n+1} = F(\Delta t, \vec{x}_n, \vec{y}_n, \vec{u}_n)$ where $\vec{x}_n$ denotes the internal SUT state, $\vec{y}_n$ the feed-back from the equipment under control and $\vec{u}_n$ the user controls, evaluated at main loop cycle n which is performed in time interval $[n \cdot \Delta t, (n+1) \cdot \Delta t], n = 0, 1, \ldots$
- *Sequential modules and libraries* performing control algorithms involving Boolean, integer and real arithmetics.

For SUT comprising several parallel interacting agents it is useful to introduce a parallel operator on TDIOHS. This is performed in analogy to parallel hybrid automata as explained in [11], but without utilization of synchronous events shared between parallel agents. For more complex testing applications it is also useful to model the test equipment as a separate TDIOHS running in parallel and interacting with the SUT. This permits to specify test execution techniques for non-deterministic SUT: If, after having performed a prefix of a pre-planned transition sequence, the SUT reacts with an unexpected output y, the test equipment can "change track" to another test case execution coinciding with the prefix but continuing with y. This technique, however, requires that the test data for different symbolic test cases has been prepared in advance, since constraint solving cannot be performed in hard real-time. Moreover, switching between suitable test case candidates may require an amount of back-tracking which also impairs hard real-time performance. Non-determinism frequently arises in case of SUT consisting of parallel agents, where each agent is locally deterministic but the SUT behavior at its output interface may appear non-deterministic due to scheduling effects.

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. For practical application, the tool needs a collection of other solver components whose description is outside the scope of this paper, but which are also currently implemented or already integrated into the tool, in order to allow for a wider range of test applications. For example,

- specialized solvers are currently implemented for handling linear constraint problems (Gauss elimination for regular linear equations, Simplex method for linear inequations),
- non-linear constraints possessing isolated single-point solutions are solved using optimization techniques,
- input constraints for string variables may be specified using regular expressions; the constraints are internally handled by means of the finite state machine encodings associated with each regular expression.

For several of these solvers interval versions exist (see [14, Chapters 4 and 5]), so that they can be applied within the same interval analysis setting as described in this article.

The symbolic test case generation algorithm *generateStc()* specified in Section 5 requires that the solver returns the maximal length of the test case prefix for which input data could be generated in order to make the associated transition sequence feasible. Currently, this is trivially realized by dropping the constraint associated with the last transition guard of the symbolic test case and trying to solve the reduced constraint, as soon as the full conjunction turns out to be infeasible. In order to make this "roll-back" mechanism more efficient, an *undo-stack* is currently being developed, so that the impact of each local constraint c_n on the restriction of a potential solution interval I and on the choice of a bisections can be undone as soon as the full global constraint $\bigwedge_{i=1}^{n} c_i$ turns out to be infeasible.

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 [6].
- Based on the results from [6], a time-discrete version of the W method is implemented by extending the symbolic test case tree in such a way that the characterization set of each location is still contained in the tree [18]. For systems with small state spaces (for example, PLCs), the location set of the TDIOHS is expanded, thereby encoding state variable valuations directly within the locations. This allows to apply the W method directly on the transition graph and STCT structures, without having to take different variable values into account.
- The interval vector solutions produced by the constraint solver can be used in order to generate several input data sets for the same symbolic test case. This is useful in order to exercise identical paths through the transition graph with different test data, in particular, in order to try out boundary values.

A further aspect not covered in this article concerns hierarchic TDIOHS, where locations may be refined by sub-TDIOHS, following the concept of OR-states in Statecharts. This aspect has been implemented by managing a hierarchy of TDIOHS specifications, linked with each other by means of references from locations to subordinate TDIOHS. The same technique is applied when dealing with higher-level formalisms like HybridUML [4], where transitions are labeled by Boolean methods acting as guards and void methods acting as actions. These methods are compiled into a *control flow graph* representation, where edges correspond to if-else conditions and nodes contain sequences of assignments using 3-address-code[5]. As a consequence, control flow graphs are just special cases of TDIOHS, and the internal model representation labels TDIOHS transitions representing Statecharts with references to TDIOHS representing the Boolean guard method and the associated action, respectively.

8. ACKNOWLEDGMENTS

The authors would like to thank Christian Herde for many fruitful discussions and for his contributions to the solver technology. Furthermore, we would like to thank Serge Fopoussi, Tatiana Kotas, Helge Löding and Xavier Noumbissi Noundou for their excellent support in implementing a major portion of the concepts described in this paper. 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[6]). The author from Bremen has been 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[7]). The tool development has been supported by Siemens Transportation Systems in Braunschweig and Verified Systems International GmbH in Bremen.

9. REFERENCES

[1] *ISO/IEC 9126-1:2001 Software engineering – Product quality – Part 1: Quality model*. International Organization for Standardization, 2001.

[2] R. Alur, T. Dang, J. Esposito, R. Fierro, Y. Hur, F. Ivančić, V. Kumar, I. Lee, P. Mishra, G. Pappas, and O. Sokolsky. Hierarchical hybrid modeling of embedded systems. *Lecture Notes in Computer Science*, 2211:14–31, 2001.

[3] R. Alur, T. Dang, and F. Ivancic. Reachability analysis of hybrid systems via predicate abstraction. In C. Tomlin and M. Greenstreet, editors, *Hybrid Systems: Computation and Control: 5th International Workshop, HSCC 2002, Stanford, CA, USA*, volume 2289 of *Lecture Notes in Computer Science*. Springer, 2002.

[4] 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, April 2006. Special Section on Specification and Validation

[5]For methods programmed in C/C++, these control flow graphs can be generated using the GNU g++ compiler.
[6]`http://www.avacs.org`
[7]`http://www.tzi.de/agbs/projects/hybris`

of Models of Real Time and Embedded Systems with UML.

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

[6] 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.

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

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

[9] S. Gnesi, D. Latella, and M. Massink. Formal test-case generation for uml statecharts. In *Ninth IEEE International Conference on Engineering Complex Computer Systems (ICECCS'04)*, iceccs, pages 75–84, 2004.

[10] M. C. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.

[11] 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.

[12] T. Henzinger and P. Kopke. Discrete-time control for rectangular hybrid automata. In *ICALP 97: Automata, Languages, and Programming*, Lecture Notes in Computer Science 1256, pages 582–593. Springer, 1997.

[13] T. J. Hickey, Q. Ju, and M. H. van Emden. Interval arithmetic: From principles to implementation. *Journal of the ACM*, 48(5):1038–1068, 2001.

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

[15] G. J. Myers. *The Art of Software Testing*. John Wiley& Sons, Inc., 2004.

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

[17] A. Spillner, T. Linz, and H. Schaefer. *Software Testing Foundations*. dpunkt.verlag, Heidelberg, 2006.

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

[19] 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.