Test Automation for Hybrid Systems

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

Martin Fränze*
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).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SOQUA ’06, November 6, 2006, Portland, OR, USA.
Copyright 2006 ACM 1-59593-584-3/06/0111 ...$5.00.

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

General Terms
Verification

Keywords
Test case generation, interval analysis, MCDC coverage, hybrid systems

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 \( S \subseteq \mathbb{R}^n \) 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 \( S \subseteq \mathbb{R}^n \) by unions of non-intersecting intervals \( I \subseteq \mathbb{R}^n \) (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. Our TDIOHS were motivated by similar ideas as the rectangular hybrid automata introduced in [8].

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 [10]. 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 \( \mathcal{H} = (\text{Loc}, \text{Init}, V, I, O, \text{Trans}) \) where \( \text{Loc} \) 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 \( \text{Guard} \) denote the set of all quantifier-free predicates over \( V \), \( \text{Init} : \text{Loc} \rightarrow \text{Guard} \) a function mapping locations to predicates, \( \text{Assign} \) the set of all pairs \((x, \bar{f})\) where \( \bar{f} = (x_1, x_2, \ldots) \) is the vector of all variables in \( V \) (a.k.a. controlled variables) and \( \delta = (\ell_1, \ell_2, \ldots) \in T^{\text{Val}} \) is the set of all terms over \( V \). Then \( \text{Trans} \subseteq \text{Loc} \times \text{Guard} \times \text{Assign} \times \text{Loc} \) is the set of transitions. \( \text{Labels} = \{ \lambda \mid \lambda \in \text{Guard} \times \text{Assign} \mid \exists l, l_1 \in \text{Loc} : (l_1, \lambda, l) \in \text{Trans} \} \) denotes the set of transition labels.

Let \( \text{val} \in \text{dom}^{V} \) 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 \( \text{val} \) be denoted by \( \text{val}(v) \) and \( \text{val}(t) \), respectively. A run of a TDIOHS \( \mathcal{H} \) is an infinite sequence of pairs \((\ell_t, \text{val}_t)\), \((\ell_2, \text{val}_3)\), \ldots \) of locations and valuations which satisfies the following properties: (1) \( \text{val}(\text{Init}(l_1)) = \text{true} \), (2) \( \forall i \in \mathbb{N} \exists (l_i, g, (\bar{x}, \bar{f}), \ell') \in \text{Trans} : l = l_i, \ell' = l_{i+1}, \text{val}(g) = \text{true}, \text{val}_{i+1}(x_1) = \text{val}(t_1), \ldots, \text{val}_{i+1}(x_{\text{Val}-1}) = \text{val}(t_{\text{Val}-1}) \). A \( k \)-bounded run is a run of a fixed length \( k \in \mathbb{N} \), i.e. the sequence of location-valuation pairs \((\ell_1, \ldots, \ell_k)\) is finite. The set of all \( k \)-bounded runs of \( \mathcal{H} \) is denoted by \( \text{Run}(\mathcal{H}, k) \).

Let \( \text{Val}(\bar{v}) \) denote the set of variables from \( V \) referenced in term vector \( \bar{t} \) and Stable\((\bar{x}, \bar{f})\) denote the variable components \( x_i \) of vector \( \bar{x} \) whose values are unaffected by the associated assignment term \( t_i \) in any valuation (so \( t_i \equiv x_i \)). An input location \( l_1 \in \text{Loc}_1 \subseteq \text{Loc} \) 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,

\[
\text{Loc}_1 = \{ l_1 \in \text{Loc} \mid \forall (l_0, g, (\bar{x}, \bar{f}), l) \in \text{Trans} : \text{Val}(l) \subseteq I \cup O \subseteq \text{Stable}(\bar{x}, \bar{f}) \}
\]

An output location \( l_1 \in \text{Loc}_O \subseteq \text{Loc} \) is characterized by the requirement that all transitions entering this state perform only assignments from local to output variables:

\[
\text{Loc}_O = \{ l_1 \in \text{Loc} \mid \forall (l_0, g, (\bar{x}, \bar{f}), l_1) \in \text{Trans} : \text{Val}(l) \subseteq V - (I \cup O) \subseteq \text{Stable}(\bar{x}, \bar{f}) \}
\]

An internal processing location \( l_1 \in \text{Loc}_C \subseteq \text{Loc} \) is characterized by the requirement that entry assignments may only read from and write to local variables:

\[
\text{Loc}_P = \{ l_1 \in \text{Loc} \mid \forall (l_0, g, (\bar{x}, \bar{f}), l_1) \in \text{Trans} : \text{Val}(l) \subseteq V - (I \cup O) \subseteq \text{Stable}(\bar{x}, \bar{f}) \}
\]

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 $r = \langle l_i, \text{val}_i \rangle | i \in \mathbb{N} \rangle$ and $r' = \langle l'_i, \text{val}'_i \rangle | i \in \mathbb{N} \rangle$ of I/O safe TDIOSs 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:

$$r \sim r' \equiv \forall l \in M \exists \text{Loc} \cup \text{Loc}_O = r' | \text{Loc}_1 \cup \text{Loc}_O$$

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

I/O safe TDIOSs 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.

**Lemma 1.** Let $\varepsilon = \langle \bar{x}, \bar{x} \rangle$ denote the stable assignment which does not change any local variables or outputs. Given TDIOSs $\mathcal{H}_1$ and transition $\tau_0 = \langle l_0, g, r, g_2, a, l_1 \rangle \in \text{Trans}(\mathcal{H}_1)$, construct a new TDIOS $\mathcal{H}_2$ by setting

1. $\text{Loc}_F(\mathcal{H}_2) = \text{Loc}_F(\mathcal{H}_1) \cup \{l_0, l'_0, l''_0\}$, where $l_0, l'_0, l''_0$ are fresh location identifiers,

2. $\text{Trans}(\mathcal{H}_2) = (\text{Trans}(\mathcal{H}_1) - T_1) \cup T_2$ where

   $$T_1 = \{ \langle l, g, b, l' \rangle \in \text{Trans}(\mathcal{H}_1) | l = l_0 \}$$

   $$T_2 = \{ \langle l_0, g, r, g'_2, a, l_1 \rangle \in \text{Trans}(\mathcal{H}_1) \} \cup \{ \langle l_0, g, r, g'_2, a, l_1 \rangle \} \cup \{ \langle l'_0, g, b, m \rangle | \langle l'_0, g, b, m \rangle \in \text{Trans}_1 - \{ \tau_0 \} \} \cup \{ \langle l''_0, g, b, m \rangle | \langle l''_0, g, b, m \rangle \in \text{Trans}_1 - \{ \tau_0 \} \}$$

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.

An analogous transformation exists for disjunctive guards:

**Lemma 2.** Let $\varepsilon = \langle \bar{x}, \bar{x} \rangle$ denote the stable assignment which does not change any local variables or outputs. Given TDIOSs $\mathcal{H}_1$ and transition $\tau_0 = \langle l_0, g, r, g_2, a, l_1 \rangle \in \text{Trans}(\mathcal{H}_1)$, construct a new TDIOS $\mathcal{H}_2$ by setting

1. $\text{Loc}_F(\mathcal{H}_2) = \text{Loc}_F(\mathcal{H}_1) \cup \{l_0, l'_0, l''_0\}$, where $l_0, l'_0, l''_0$ are fresh location identifiers,

2. $\text{Trans}(\mathcal{H}_2) = (\text{Trans}(\mathcal{H}_1) - T_1) \cup T_2$ where

   $$T_1 = \{ \langle l, g, b, l' \rangle \in \text{Trans}(\mathcal{H}_1) | l = l_0 \}$$

   $$T_2 = \{ \langle l_0, g, r, g'_2, a, l_1 \rangle \in \text{Trans}(\mathcal{H}_1) \} \cup \{ \langle l_0, g, r, g'_2, a, l_1 \rangle \} \cup \{ \langle l'_0, g, b, m \rangle | \langle l'_0, g, b, m \rangle \in \text{Trans}_1 - \{ \tau_0 \} \} \cup \{ \langle l''_0, g, b, m \rangle | \langle l''_0, g, b, m \rangle \in \text{Trans}_1 - \{ \tau_0 \} \}$$

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/agb/projects/hybris.
2.3 Test Cases and Strategies

A symbolic test case for TDIOHS $\mathcal{H}$ is a finite sequence of transitions $(t_1, \ldots, t_k)$ with $t_i = (c_i, g_i, a_i, l'_i) \in \text{Trans}$ satisfying $l'_i = l_{i+1}$ 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,

$$\forall v_1, \ldots, v_k \in \text{dom}^{|V|} : r = ((l_1, v_1), \ldots, (l_k, v_k)) \in \text{Run}(\mathcal{H}, k) \wedge v_{\text{val}}(g_1) = \ldots = v_{\text{val}}(g_k) = \text{true}$$

Note that the initialization condition for runs also implies that $v_{\text{val}}(\text{Init}(l_1)) = \text{true}$. 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 $(v_{\text{val}}|l_i) \mid i = 1, \ldots, k$ is called the test (input) data and sequence $(v_{\text{val}}|O) \mid i = 1, \ldots, k$ 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, 12] – 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 [11], 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 a given location. MCDC is required by the standard [11] 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 feedback about feasibility 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 \( H \)-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 a TDIOHS transition graph and its associated STCT for bounded maximal path length \( k = 5 \).

Given a TDIOHS \( H = (\text{Loc}, \text{Init}, V, I, O, Trans) \), we introduce the associated STCT \( Stcts \) of length \( k \) by means of an algorithm which also shows how to extend \( Stcts \) into some \( Stcts_{k+k} \) if the original tree is insufficient to reach the coverage goals. Let “\( \sim \)” a fresh location symbol not contained in \( \text{Loc} \) and \( \tau_0 \) a fresh transition symbol. Then the components of an STCT are defined for \( k = 0, 1, 2, \ldots \) as

\[
Stcts = (N_k, E_k, L_k, \phi_k, \psi_k, \sigma_k, \pi_k, \rho_k)
\]

which are typed as follows:

\[
N_k \subseteq \{(−, 0)\} \cup (\text{Loc} \times N_k)
E_k \subseteq N_k \times \text{Labels} \times N_k
L_k \subseteq N_k
\phi_k : \{\tau_0\} \cup Trans \rightarrow N_k^*
\psi_k : N_k \rightarrow \{\tau_0\} \cup Trans
\sigma_k : \text{Loc} \rightarrow N
\pi_k : N_k \rightarrow N_k
\rho_k : N_k \rightarrow Trans^*
\]

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 \( \phi_k \) maps transitions \( t \) to the list of nodes \((l, n)\) in \( Stcts \) having \( t \) as target node. For example, transition \((l_5, h, l_6)\) in the TDIOHS of Figure 2 is mapped to

\[
\phi_0(l_5, h, l_6) = \langle \langle l_0, 2 \rangle, \langle l_0, 4 \rangle, \langle l_0, 6 \rangle, \langle l_0, 8 \rangle, \langle l_0, 10 \rangle \rangle
\]

in the STCT of Figure 3. If the test case generator learns about the infeasibility of a path from the root of \( Stcts \) to the node \((l, n)\) then this node is deleted from \( \phi_0(t) \) and the nodes from all continuation paths of \( (l, n) \) are removed from the respective images under \( \phi_k, \psi_k \). Nodes \((l, n)\) of \( Stcts \) to the transition of \( H \) whose corresponding edge in \( Stcts \) ends at \((l, n)\), \( \sigma_k \) keeps track of the counters \( n = \sigma(t) \) to be associated with \( H \)-locations \( t \) when inserting them as nodes \((l, n)\) into the tree. \( \pi_k \) maps nodes to their parent nodes. \( \rho_k \) maps a node \((l, n)\) to the symbolic test case derived from the \( Stcts \) path starting at the root and ending at \( (l, n) \). These data structures are initialized as (recall that \( \varepsilon \) denotes the trivial assignment which does not change anything)

\[
N_0 = \{(-, 0)\} \cup (\text{Loc} \times \{1\})
E_0 = \{((-0), \text{Init}(l'), \varepsilon, (l', 1)) \mid l' \in \text{Loc}\}
L_0 = N_0 \setminus \{(-, 0)\}
\phi_0 = \{\tau_0 \mapsto \{(l, 1) \mid l \in \text{Loc}\}\}
\psi_0 = \{x \mapsto \tau_0 \mid x \in N_0\}
\sigma_0 = \{l \mapsto 2 \mid l \in \text{Loc}\}
\pi_0 = \{(-, 0) \mapsto (-, 0)\} \cup \{(l, 1) \mapsto (-, 0) \mid l \in \text{Loc}\}
\rho_0 = \{x \mapsto \varepsilon \mid x \in N_0\}
\]

Let \( STCT \) denote the type of an STCT as induced by the component types introduced above. Algorithm \( \text{expandStct()} \) inputs an existing STCT and changes it by expanding each leaf for one transition step, if a corresponding transition exists in \( H \) and if the test case associated with the path from

the root to this leaf has not yet been marked as infeasible.

```plaintext
function \text{expandStct}(\text{inout stct} : STCT) : \text{STCT} \begin{cases} \text{begin} & \text{let } (N, E, L, \phi, \psi, \sigma, \pi, \rho) = \text{stct in begin} \quad \text{retval} := \text{false}; \\ & \text{forall} (l, n) \in \{x \in L \mid \phi(\psi(x)) \neq \langle \rangle\} \text{ do} \\ & \text{forall} (\lambda, l') \in \{(a, b) \mid (l, a, b) \in Trans\} \quad \text{retval} := \text{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 \circ \{(l, \lambda, l') \mapsto \phi(\lambda, l') \setminus \langle(l', n')\}\}; \\ & \psi := \psi \circ \{(l', n') \mapsto (l, \lambda, l')\}; \\ & \sigma := \sigma \circ \{l' \mapsto n' + 1\}; \\ & \pi := \pi \circ \{(l', n') \mapsto (l, n)\}; \\ & \rho := \rho \circ \{(l', n') \mapsto \rho(l, n) \setminus \langle(l, \lambda, l')\}\}; \\ & \text{enddo} \\ & \text{expandStct := retval;} \\ & \text{endlet} \\ & \text{end} \end{cases}
```

In this algorithm \( \circ \) denotes the functional overriding operator defined by \( f \circ x := y \circ x \) if \( z = x \) then \( y \) else \( f(z) \). Expanding the STCT by \( k \) steps is simply performed by \( k \)-fold invocation of \( \text{expandStct}() \).

The complete symbolic test case generation algorithm specified in function \( \text{generateStct}() \) below references two generic functions encapsulating the strategy-dependent part of the generation algorithm: \( \text{select}() \) 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, \( \text{select}() \) is instantiated by a function which selects paths in the STCT containing edges \((l, n), \lambda, (l', n')\) whose associated transitions \((l, \lambda, l')\) in \( H \) have not yet been covered at all. Function \( \text{covered}() \) 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 \text{true} if the strategy-specific coverage goals have been reached. For MCDC coverage, \( \text{covered}() \) just checks whether the edges \((l, n), \lambda, (l', n')\) covered so far in the STCT correspond to all transitions \((l, \lambda, l')\) in \( H \).

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

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 \( t_{p+1} \) and its subordinate STCT subtree are marked as infeasible. This task is performed by the – strategy-independent – algorithm \texttt{infeasible()} which inputs the STCT and the target node associated with \( t_{p+1} \). Infeasibility is recorded in the STCT data structure by removing STCT nodes from the image sequences of transitions \( t_{p+1}, \ldots, t_p \) under \( \phi \).

\textbf{procedure} \texttt{infeasible(inout stct : STCT; \in x : N) begin let \((N, E, I, \phi, \psi, \sigma, \pi, \rho) = \texttt{stct in begin t} := \psi(x); \\
\phi := \phi \uplus \{ t \mapsto \phi(t) - x\}; \\
\forall (x, x') \in E \texttt{do infeasible(stct, x')} \texttt{endall} \end{align*}

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 \texttt{generateStc} initializes the STCT \( \texttt{stct} \) and the set \( \mathcal{C} \) 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 \( i < k \) (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 \( \mathcal{C} \). The target node of the first infeasible transition in \( \rho(x) \) (if any) is passed to procedure \texttt{infeasible()} which takes care of removing the infeasible STCT nodes from the range of \( \phi \). When the \texttt{select()} operation returns \((-0)\) this means that either the coverage goal has been reached or the STCT has to be expanded.

\textbf{function} \texttt{generateStc : B begin stct := \((N_0, E_0, L_0, \phi_0, \psi_0, \sigma_0, \pi_0, \rho_0)\); \in i := 0; \mathcal{C} := \emptyset \texttt{while} \neg \texttt{covered}(H, \texttt{stct}, \mathcal{C}) \land i < \texttt{maxExpansions} \land \texttt{expandStctBy(stct, k)} \texttt{do} \in i := i + 1; \x := \texttt{select(stct, C)}; \texttt{while} \x \neq (-0) \texttt{do} \in m := \texttt{solve(\rho(x))}; \n := \#\rho(x); \mathcal{C} := \mathcal{C} \cup \{\lambda^m(\x) \mid p = n - m, n - m + 1, \ldots, n\}; \x := \texttt{select(stct, C)}; \texttt{enddo} \texttt{generateStc := covered(H, stct, C)}; \texttt{end} \end{align*}

In this algorithm \( \lambda^m(\x) \) denotes the \( m \)-fold application of the parent function \( \pi \).

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 \( t_i \) 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 \( t_i \). 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 \( \{t_1, \ldots, t_k\} \) with \( t_i = (l_i, g_i, (\bar{x}_i, \bar{t}_i), l_i') \in \texttt{Trans} \) 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

\[
\text{Init}(l_i)[\bar{x}_i/\bar{E} \land \bigwedge_{i=1}^{k-1} g_i[\bar{x}_i/\bar{E}] \land \bigwedge_{i=1}^{k-1} (\bar{x}_i = \bar{t}_i)[\bar{x}_i/\bar{E}, \bar{x}_{i+1}]/\bar{E}].
\]

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

\[
\mathcal{I} \models \text{Init}(l_1)[\bar{x}_1/\bar{E}] \quad \text{and} \quad \text{for each } i < k \text{ and each assignment } (x, t_{i,x}),
\]

is a necessary and —if point intervals are admitted— sufficient condition for real-valued satisfiability of the above constraint. Here, \( t \) 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. \( \mathcal{I} \models t_1 \leq t_2 \) if \( \sup t_1(\mathcal{I}) \leq \inf t_2(\mathcal{I}) \), etc. Extracting the valuations \( \text{val}_i \in \text{dom}[\mathcal{V}^i] \) 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 value $\text{val}(x) \in \mathcal{I}(x_i)$. The same is done for the initial values of all controlled variables $x \in V - I$, i.e.

we select $\text{val}(x) \in \mathcal{I}(x_0)$ arbitrarily. For $1 \leq i \leq k$ we then calculate the values $\text{val}(x)$ 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 $\langle (l_1, \text{val}(l_1)), \ldots, (l_k, \text{val}(l_k)) \rangle$.

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 interprets 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 such a way that it has to process a conjunction of constraints of the forms

\[
\begin{align*}
\text{bound} & := \var \geq \text{rational} \text{ const} \ | \ | \var \geq \text{rational} \text{ const} \\
\text{triplet} & := \var = \text{var} \ | \text{bop} \ | \text{var} \\
\text{pair} & := \var = \text{var} \ | \text{wop} \ | \text{var} \\
\end{align*}
\]

only, where

\[
\begin{align*}
\text{bop} & := + \ | \ - \ | \ * \ | \ / \ | \ldots \\
\text{wop} & := - \ | \ \sin \ | \ \exp \ | \ldots
\end{align*}
\]

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 $h$:

\[
s > 0 \ | \ h = w - z \ | \ h = s
\]

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

1. Initialization: All bounds $x \sim c$ from the constraint,

$\sim \in \{\geq, \geq, <, \leq\}$, 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 $u = op w$ and pairs $u = op v$ which are not satisfied in the sense of (3), i.e. which violate $\mathcal{I}(u) \supseteq \mathcal{I}(v) \cap \mathcal{I}(w)$ or $\mathcal{I}(u) \supseteq \mathcal{I}(v)$, respectively.\(^2\)

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

\[
\mathcal{I}' = \mathcal{I} \cap [x \sim I(x) \cap [x \in R \ | \ x \sim c]].
\]

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 resheluffings) 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 $u = op w$ or the current pair $u = op v$ within the set $C$ of unresolved constraints, depending on whether it is satisfied in the sense of $u \supseteq v \supseteq w$ or $(u \supseteq v \supseteq w)$, 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 $u = op w$ 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., [13]): 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 [13] 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. 4. Let $x^2 - 2y \leq 100$ be a fragment of a for-
The decomposition of this fragment into triplets, pairs and bounds $c_1, \ldots, c_4$ and already learned conflict clauses $c_{11}, c_{12}$ 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 $(x \geq -2) \lor (x \leq 3) \lor (y \geq 4)$ 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 $x \not\in I$ 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 $x \not\in I$ can be traced back to the stronger pruning of the search space: while a no-good $x \not\in I$ 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 $I(x)$. 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


