Automated Conformance Verification of Hybrid Systems

Harald Brandl, Martin Weiglhofer, and Bernhard K. Aichernig
Institute for Software Technology, Graz University of Technology, Austria
Email: {brandl, weiglhofer, aichernig}@ist.tugraz.at

***Abstract*—Due to the combination of discrete events and continuous behavior the validation of hybrid systems is a challenging task. Nevertheless, as for other systems the correctness of such hybrid systems is a major concern. In this paper we present a new approach for verifying the input-output conformance of two hybrid systems. This approach can be used to generate mutation-based test cases. We specify a hybrid system within the framework of *Qualitative Action Systems*. Here, besides conventional discrete actions, the continuous dynamics of hybrid systems is described with so called *qualitative actions*. This paper then shows how labeled transition systems can be used to describe the trace semantics of Qualitative Action Systems. The labeled transition systems are used to verify the conformance between two Qualitative Action Systems. Finally, we present first experimental results on a water tank system.**

***Index Terms*—model-based testing, mutation testing, action systems, qualitative reasoning, hybrid systems**

I. Introduction

Discrete systems that have interaction with continuous processes are denoted as hybrid systems. Most embedded systems, e.g. a weather station sensing temperature etc. or a fuel injection controller, are of this kind. The combination of discrete and continuous systems usually leads to large state spaces and the integration of these two concepts make manual testing a complex, tedious and time-consuming task. In order to ensure the correctness of complex systems, as hybrid systems are, model checking and model-based testing are applied today. This work presents a novel approach to verify the input-output-conformance (ioco) [1] between two hybrid systems. One possible application of such a conformance verification is the generation of test cases based on mutations of a specification.

Hybrid systems provide a closed-loop view on control programs operating in their environment which allows to draw more conclusions, e.g. stability or long term behavior, than by looking at the controller in isolation. In order to reason about such systems various formalisms, e.g. *Hybrid Automata* [2], Hybrid Process Algebras like χ [3], *Hybrid Action Systems* [4] or *Continuous Action Systems* [5], have been proposed.

The challenge in reasoning about hybrid systems is to deal with infinite state spaces which arise because of continuously evolving real-valued model variables. By applying predicate abstraction techniques this can be countered. A model checker for hybrid systems is *HyperTech* [6], the successor of *HyTech*, which employs interval arithmetic methods to over-approximate solutions of Ordinary Differential Equations (ODEs). This and related techniques compute polyhedral inclusions of trajectories which requires numerical calculations.

Research herein was funded by the EU project ICT-216679, Model-based Generation of Tests for Dependable Embedded Systems (MOGENTES).

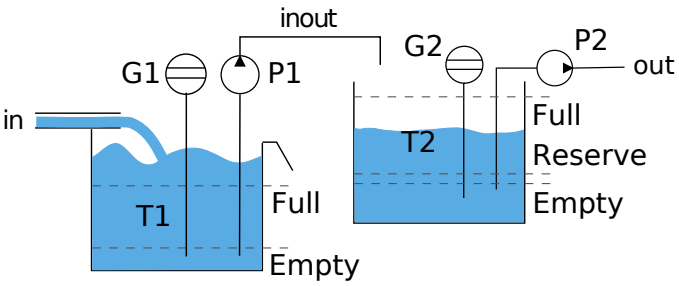


Fig. 1. Two-Tank Pump System.

We propose to use *Qualitative Reasoning (QR)* [7], a technique from Artificial Intelligence for common sense reasoning about physical systems, to describe the continuous behavior of hybrid systems.

In QR the behavior of a continuous system is described with *Qualitative Differential Equations (QDEs)* which are an abstraction of ODEs. Given a specification in the form of a set of QDEs and the definition of an initial state, simulation engines like *QSIM* [7] or *Garp3* [8] infer a transition system which contains all possible behaviors the system may show over time. The behavior inference does not rely on numerical information; all values (except zero) in the qualitative domain are symbolic forming a finite total order.

Our previous work [9] introduces *Qualitative Action Systems (QAS)* as an adaptation of *Hybrid Action Systems (HAS)* [4]. In QAS the differential actions of HAS are replaced by corresponding qualitative actions resulting in a system model with discrete state space. We augment actions with labels (names) that may have parameters. Furthermore, named actions are partitioned into disjoint *input* and *output* actions. This gives an LTS semantics to (qualitative) action systems which we use in turn to decide *ioco* between two systems. The remaining unnamed actions are *internal* and cause τ transitions during system evolution.

We developed a prototype tool written in GNU Prolog which explores two given QAS while checking conformance on-the-fly. For solving the qualitative differential equations we use *ASIM* [10] which is also implemented in GNU Prolog.

Example I.1. Consider the example hybrid system in Figure 1 which serves as our running example. In the two-tank system tank $T1$ is on a lower level than the tank $T2$. $T1$ is being filled with water having some inflow rate in. Both tanks ($T1$, $T2$) are connected by the pump $P1$ which is controlled such that it fulfills the following requirements:

If the water level in $T2$ decreases below a certain *Reserve* mark and $T1$ is full, pump $P1$ starts pumping water until $T2$ is full or $T1$ gets empty. In addition, the controller needs to control the pump $P2$ that is pumping water out of $T2$: $P2$ shall be turned on as long as a button *WaterRequest* is pressed and there is enough water in $T2$ ($T2$ not *Empty*).

The continuous dynamics of the system is expressed by two coupled differential equations:

$$\dot{x}_1 = (in - inout)/A_1 \quad \text{and} \quad \dot{x}_2 = (inout - out)/A_2$$

Here, A_1 and A_2 are the base areas of the two tanks and x_1 and x_2 denote the current level in the tanks. The variables *in*, *inout*, and *out* denote the flow rates into $T1$, between $T1$ and $T2$, and out of $T2$ respectively. Based on this system description we will develop a formal QAS model which is then used for conformance verification in the application of mutation-based test case generation. □

The remainder of this paper is organized as follows. Section II comprises the basic concepts applied in this work, i.e. Qualitative Reasoning. Then Qualitative Action Systems are introduced in Section III. In Section IV we discuss the input-output conformance relation, and present our approach for on-the-fly conformance checking in Section V. We show first experimental results in Section VI, and discuss related work in Section VII. Finally, we conclude in Section VIII.

II. Qualitative Reasoning

Continuous processes are usually described with differential equations or directly via continuous functions. For hybrid systems the continuous behavior is specified with ODEs. However, the solution space of such equations is infinite, i.e. depending on the boundary conditions various real-valued functions (trajectories) over dense time are solutions to the ODE.

In Qualitative Reasoning (QR) continuous behavior corresponds to a sequence of temporally ordered states where each state binds all variables to values. Real-valued variables are abstracted to so-called *quantities* which have finite types (quantity spaces) defined as an ordered set of points (landmarks) and open intervals. The landmarks in the system define the important points where the behavior is changed from a qualitative point of view, e.g. water is frozen below 0°C, liquid between 0 and 100 °C, and changes into steam above 100°C. In addition the value of a quantity consists of an abstract gradient ($\frac{\partial}{\partial t}$), $\delta =_{df} \{-, 0, +\}$ indicating that the quantity is decreasing, steady, or increasing respectively. Thus, a QR model is defined as follows.

Definition 1 (QR model): A QR model M is a tuple $M = (Q, QS, QDE)$ where Q is a set of quantities, $QS = \{QS_{q_0}, \ldots, QS_{q_n}\}$ is a set of quantity spaces such that there is a quantity space for every $q_i \in Q$, and QDE is a set of qualitative differential equations. The type of a quantity $q_i \in Q$, denoted by $type(q_i)$ is given in terms of the quantity space and in terms of the abstracted gradient, i.e. $type(q_i) \in (QS_{q_i} \times \delta)$.

Example II.1. In order to model the continuous dynamics of the flows in the two water tanks with QDEs we define the system quantities with their according quantity spaces. The quantities x_1 and x_2 have the quantity spaces $T1 =_{df} \langle 0, \mathit{Empty}, \mathit{Full}\rangle$ and $T2 =_{df} \langle 0, \mathit{Empty}, \mathit{Reserve}, \mathit{Full}\rangle$ respectively. For the sake of simplicity we omit an additional landmark *Overflow* for tank $T1$ and assume that when the water climbs above *Full* it will overflow. The flow rates have the quantity space $\mathit{FR} =_{df} \langle 0, Max\rangle$. We also need to introduce auxiliary quantities in order to be able to set up the QDEs. The auxiliary quantities, i.e. diff_1 and diff_2, have to link different QDEs, hence they only need a coarse quantity space, $\mathit{NZP} =_{df} \langle \mathit{minf}, 0, \mathit{inf}\rangle$. In the following we use the *QSIM* notation where $\mathit{minf} =_{df} -\infty$ and $\mathit{inf} =_{df} \infty$. Furthermore, $add(x, y, z)$ denotes the addition $x + y = z$ and $\mathit{d/dt}(x, y)$ means that y is the qualitative derivation of x.

The qualitative model of the two-tank system is given by the following QDEs:

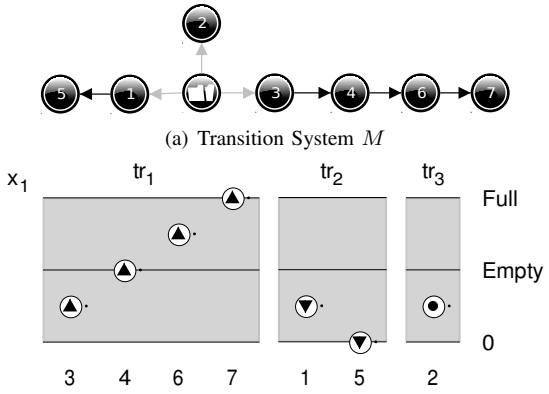
$$\begin{aligned} &\mathit{add}(\mathit{diff}_2, \mathit{out}, \mathit{inout}) \wedge \mathit{add}(\mathit{diff}_1, \mathit{inout}, \mathit{in}) \wedge \\ &\mathit{d/dt}(x_1, \mathit{diff}_1) \wedge \mathit{d/dt}(x_2, \mathit{diff}_2) \end{aligned} \tag{1}$$

□

A benefit of qualitative models is that the values are symbolic. Landmarks (except zero) can stand for any real number; for the process of behavior inference we only need the ordering of landmarks, not their exact numerical values. For instance the quantity *water_temperature* may have a value *(0..boiling_point, +)* meaning that the temperature is increasing somewhere in the liquid phase of water. Here *0..boiling_point* stands for the open interval *(0, boiling_point)*.

In other words, a system model is essentially a constraint system stating relations between quantities. From such a model simulation engines, e.g. *QSIM* [7], generate a transition system by starting from a given initial state and inferring successor states which satisfy all model constraints (QDEs). Then the successors of the successors are computed and so on until no new state is discovered. Since the domain is finite, simulation will always generate a finite number of states. However, depending on the system model the generated transition system can get quite big.

Beside system constraints in the form of QDEs there are also implicit constraints imposed by the underlying theory. For example the *continuity law* [7] requires that the value of a quantity between two successive states changes only continuously, i.e. there is no jump over values in the domain of either QS or δ. For instance a quantity cannot change its value from one landmark to another one without crossing the interval in between. In addition a non-continuous change of direction, e.g. an immediate change from increasing to decreasing without steady in between, is also disallowed.



(a) Transition System M

(b) Traces of the fill level x_1

Fig. 2. Qualitative evolution of the fill level of tank $T1$

We define the simulation result of a QR model M as a transition system.

Definition 2 (Simulation result): Given a QR model $M = (Q, QS, QDE)$ and an initial valuation for every quantity $q \in Q$, then the simulation result is a transition system $TS =_{df} (S, s_0, T, v)$. $S \subset \mathbb{N}_0$ denotes a set of states, $s_0 =_{df} 0$ is the initial state, $T : S \times S$ is the transition relation obtained by simulating the QR model, and $v : S \mapsto (Q \mapsto QS_q \times \delta)$ is a valuation function binding states to value assignments for all quantities $q \in Q$.

A simulation result is basically a graph showing all possible behaviors of the QR model. The trace semantics of this simulation result is given in terms of sequences of state valuations.

Definition 3 (Trace semantics): The trace semantics of the QR transition system $TS = (S, s_0, T, v)$ is defined as follows:

$$\textit{traces}(TS) =_{df} \{\langle v(s_0), v(s_1), \ldots\rangle \mid i \in \mathbb{N}_0 \wedge s_i \in S \wedge s_0 = 0 \wedge (s_i, s_{i+1}) \in T\}$$

Example II.2. As an example we consider the evolution of the fill level in tank $T1$. Initially the fill level is somewhere in the interval $(0, \textit{Empty})$ and the flow rates *in* and *inout* have a qualitative value $(0..\textit{Max}, 0)$. For presentation purposes we use the QR tool *Garp3* [8] as it generates nice graphical simulation results. Figure 2(a) shows the qualitative behavior M and Figure 2(b) shows according traces projected on the quantity x_1, i.e. $\{tr_1, tr_2, tr_3\} \subset traces(M) \downarrow x_1$, of the simulated scenario. The abstract derivation δ is represented by a circle, by an arrow pointing up and by an arrow pointing down denoting steady, increasing, and decreasing behavior, respectively. Depending on the order relation between the inflow and outflow rates of tank $T1$ the fill level either stays steady (tr_3), increases to the maximum (tr_1), or decreases to zero (tr_2). □

Since a sequence of states in a QR TS abstracts a set of real-valued functions over time each state in the sequence either lasts for a time-point (zero duration) or a time-interval. Thus, time is abstracted to a temporally ordered sequence of states. For example the qualitative value $(0..max, +)$ can abstract an infinite number of monotonic increasing functions like $f(t) = t + \frac{max}{2}, f(t) = 3 \cdot t + 1, f(t) = e^t, \ldots$ which satisfy the following conditions: $0 < f(0) < max$ and $\forall t \in dom(f) \bullet \dot{f}(t) > 0 \wedge f(t) < max$, where dom gives the domain of a function.

Due to overapproximation qualitative behavior can be spurious, i.e. behavior which cannot occur in any real system. A well known example of spurious behavior is a frictionless spring-mass system where in some cycles the amplitude is increasing and decreasing in others. A further problem of Qualitative Simulation is the combinatorial explosion in the size of behavior trees. This is especially the case when processes work in parallel. For instance, consider two water tanks of the same size which are filled with some non-zero inflow rates. Then the behavior tree will explain all possibilities which may occur: One tank gets full before the other one or both tanks reach their maximum fill level at the same time. This is a common property in theories dealing with concurrent processes where the interleaving semantics of parallel behavior may lead to big state spaces. Behavior branching can be reduced by refining the model with additional information about the system, like order relations between quantities. De Jong [11] surveys Qualitative Simulation and discusses issues about and solutions for spurious behavior and combinatorial behavior branching.

The practical effect in our application of QR in model-based testing is that spurious behavior is filtered out automatically during test case execution. Hence, spurious behavior does not influence the correctness of the approach. However, the size of state spaces is a concern.

The next section introduces *Qualitative Action Systems* demonstrated on our running example.

III. Qualitative Action Systems

Action Systems by Back et al. [12], [13] provide a framework for describing discrete and distributed systems. The actions are statements in the form of Dijkstra's guarded commands where the semantics is defined via weakest precondition predicate transformers. An action system, see Equation 2, consists of a block of variable declarations followed by an initialization action S_0 giving to each variable an initial value, and a **do od** block looping over the nondeterministic choice of all actions. Variables declared with a star are exported by an action system and can be imported by others in the import list I at the end of the action system block.

$$AS =_{df} |[\textbf{var } Y : T \bullet S_0; \textbf{do } A_1 \Box \ldots \Box A_n \textbf{od }]| : I \qquad (2)$$

In order to specify distributed concurrent systems several action systems can be composed in parallel. For two action systems A_1 and A_2 the import list of the resulting system gets $(u_1 \cup u_2) \backslash (v_1 \cup v_2)$ where u_1, u_2 are the imported variables and v_1, v_2 are the export variables of A_1 and A_2, respectively.

Parallel composition of several action systems communicating over shared (imported/exported) variables provides a good choice to model distributed and parallel systems.

An action A can be executed if the enabledness guard holds: $g(A) =_{df} \neg wp(A, \mathit{false})$. The execution of an action from states which do not satisfy the enabledness property behaves as *magic*, i.e. any postcondition may be established. When program statements A are totally defined for all possible initial states they satisfy the strictness property: $wp(A, \mathit{false}) \equiv \mathit{false}$. Our actions fulfill this property. Furthermore, the termination guard $t(A) =_{df} wp(A, \mathit{true})$ ensures that when action A is executed it will terminate in some post state. The guard p of an action A must not be weaker than its enabledness guard. This always holds when the actions are strict: $p \rightarrow g(A) = p \rightarrow \neg wp(A, \mathit{false}) = p \rightarrow \mathit{true} = \mathit{true}$.

Our approach of *Qualitative Action Systems (QAS)* is based on the former work of *Hybrid Action Systems (HAS)* by Rönkkö et al. [4]. In HAS the continuous behavior is specified with so called *differential actions* $e :\rightarrow d$ where e is the *evolution guard* and d is a system of ODEs. Differential actions are atomic, i.e. the continuous state of the system evolves as long as the evolution guard is true. The evolution terminates in states where the guard does not hold anymore. As termination condition it must be ensured that the evolution guard becomes false eventually. Differential actions are defined via weakest precondition semantics and are relations between states before and after continuous evolutions. The states in-between are internal and hence hidden from an outside view.

While HAS are a proper framework for specifying hybrid systems and proving properties within the refinement calculus the infinite state space is a problem when dealing with automated techniques like model checking or model-based testing. Hence, the continuous behavior has to be abstracted in some way. A well known technique called predicate abstraction makes infinite state spaces finite. We apply *Qualitative Reasoning* as a means to abstract from continuous behavior to qualitative, discrete behavior.

The work in [9] presents *Qualitative Action Systems* that we use for model-based test case generation. Qualitative actions are defined via weakest precondition semantics, for details see [9]. The evolution of a qualitative action is a transition system which terminates in post states where the evolution guard does not hold. This means that the set of traces is truncated by the evolution guard.

Definition 4 (Termination state): Given the simulation result $TS = (S, s_0, T, v)$, it's trace semantics $traces(TS)$, and an evolution guard e_q, then the termination state for a particular trace $tr \in traces(TS)$ is defined as $\Delta(tr, e_q) =_{df} i$ such that $i \in \mathbb{N}_0 \wedge \neg e_q[Q := tr[i]] \wedge \forall j < i \bullet e_q[Q := tr[j]]$. The expression $e_q[Q := v]$ denotes the substitution of Q in e_q by v and $tr[i]$ denotes the i-th element of the trace tr. If the evolution guard always holds, i.e. there is no termination state, then $\Delta(tr, e_q) =_{df} -1$.

We denote the execution of a qualitative action with the evolution guard e_q and the QR model D_q by $e_q :\rightharpoonup D_q$. This execution can be seen as a nondeterministic assignment $Q := tr[i]$ where $tr \in traces(M)$, M is the simulated behavior of D_q, and $i = \Delta(tr, e_q)$. It states that the qualitative state Q is updated to a new state Q' which is the last state of a terminating trace. If there exists no such state Q', then the action does not terminate.

System $=_{df}$

|[**var** $x_1 : T1, x_2 : T2, \mathit{out}, \mathit{inout} : \mathit{FR}$,
$\mathit{diff}_1, \mathit{diff}_2 : \mathit{NZP}$,
$\mathit{p1\_running}, \mathit{p2\_running}, \mathit{wr} : \mathit{Bool}$

• $x_1 := (0,0); x_2 := (0,0)$;
$\mathit{out} := (0,0); \mathit{inout} := (0,0); \mathit{wr} := \mathit{false}$
$\mathit{p1\_running} := \mathit{false}; \mathit{p2\_running} := \mathit{false}$

alt $\mathit{PUMP1\_ON} : g_1 \rightarrow \mathit{p1\_running} := \mathit{true}$;
$\mathit{inout} := (0..\mathit{Max}, 0)$

□ $\mathit{PUMP1\_OFF} : g_2 \rightarrow \mathit{p1\_running} := \mathit{false}$;
$\mathit{inout} := (0,0)$

□ $\mathit{PUMP2\_ON} : g_3 \rightarrow \mathit{p2\_running} := \mathit{true}$;
$\mathit{out} := (0..\mathit{Max}, 0)$

□ $\mathit{PUMP2\_OFF} : g_4 \rightarrow \mathit{p2\_running} := \mathit{false}$;
$\mathit{out} := (0,0)$

□ $\mathit{WATER\_REQ(X)} : g_5 \rightarrow \mathit{wr} := X$

with $\neg(g_1 \vee g_2 \vee g_3 \vee g_4 \vee g_5) :\rightharpoonup$
$\mathit{add}(\mathit{diff}_2, \mathit{out}, \mathit{inout}) \wedge \mathit{add}(\mathit{diff}_1, \mathit{inout}, \mathit{in}) \wedge$
$\mathit{d/dt}(x_1, \mathit{diff}_1) \wedge \mathit{d/dt}(x_2, \mathit{diff}_2)$

]| : *in*

Fig. 3. Qualitative action system of our running example.

In order to apply model-based testing using labeled transition systems, actions are augmented with names (labels) which may have parameters. Unnamed actions are internal and, when executed, cause τ events. The set of labels is partitioned into inputs and outputs. Then state space exploration of the action system yields an LTS which can be used for test case generation. The tester applies controllable (input) events to the IUT and checks if observable (output) events are allowed by the specification.

Figure 3 shows the qualitative action system of our running example. The **var** section declares the variables with according types T_i. The states S of an action system are a subset of the cross product of its n state variable types, i.e. $S \subset T_1 \times \cdots \times T_n$. Beside the quantities we introduce three Boolean variables: *px_running* models the state of pump1 and pump2, respectively, and *wr* denotes the water request by a user of the system. In the initial phase of the system all Boolean variables are set to *false* and all quantities to steady zero. The imported quantity *in* represents the inflow into tank *T1*. As we do not define an external system which controls the inflow we set it to some constant rate, *in* $=_{df}$*(0..Max, 0)*. Hence, the system is closed and the inflow is actually part of the state variables.

The **alt** and **with** keywords express an alternation between discrete and qualitative actions. In our model we apply *interrupting prioritized alternation* [14]. This means that whenever a discrete action is enabled it has priority over all qualitative actions. For reactive systems this is an important property which allows to interact with the environment at certain points. Hence the environment cannot block the controller from performing its function. In the other direction the controller

eventually has to reach stable states in its computation where it interacts with the environment. Otherwise the environment and hence the progress of time is blocked. Since actions are atomic, interrupting behavior can be realized by interlocking qualitative with discrete actions. In our example the QDEs of the system, see (1), are globally defined, thus the evolution guard is *true*. However, such a qualitative action will never terminate and therefore will prevent the discrete controller from execution. By conjoining all negated guards of discrete actions to the evolution guards of qualitative actions the interrupting alternation behavior can be achieved. Other forms of alternation for hybrid systems are discussed in [14].

The first four actions in the **alt** block are output actions controlling the state of the pumps where the guards g_1 to g_4 are defined according to the requirements as follows:

$$g_1 =_{df} x_2 < Reserve \wedge x_1 = Full \wedge \neg \textit{p1\_running}$$
$$g_2 =_{df} \textit{p1\_running} \wedge (x_1 < Empty \vee x_2 = Full)$$
$$g_3 =_{df} \textit{wr} \wedge \neg \textit{p2\_running} \wedge x_2 > \textit{Empty}$$
$$g_4 =_{df} \textit{p2\_running} \wedge (\neg \textit{wr} \vee x_2 \leq \textit{Empty})$$

The remaining action *WATER_REQ(X)* is an input action to the system. The Boolean parameter X determines if water is required or not, and the guard describes a simple user scenario: A user turns on the water when the tank $T2$ is full and turns it off again when the water drops below the reserve level.

$$g_5 =_{df} (\neg \textit{wr} \wedge x_2 = \textit{Full} \wedge X = \textit{true}) \vee (\textit{wr} \wedge x_2 < \textit{Reserve} \wedge X = \textit{false})$$

The qualitative action in the **with** section updates the environmental state of the system. At the end of an evolution another action may get enabled or the system terminates. Notice, that qualitative actions have no name as they are treated in a different manner than input and output actions. Since the QDEs of qualitative actions are an abstract representation of ODEs, inputs and outputs evolve in parallel over time. In contrast to the sequential input/output behavior of discrete event systems (LTS, ...) the input/output behavior of continuous, linear, and time-invariant systems can be described with transfer functions $H(s) =_{df} \frac{fout(s)}{fin(s)}$. Here, $fout(s)$ and $fin(s)$ are the images of the Laplace transformed continuous, time-dependend functions $fin(t)$ and $fout(t)$. Let us consider a simple integrator $H(s) = \frac{1}{s}$ and an input signal having the function $fin(t) = 3t$. Then

$$\mathscr{L}\{fin(t)\} = fin(s) = \frac{3}{s^2}$$
$$fout(s) = fin(s) \cdot H(s) = \frac{3}{s^3}$$
$$fout(s) \bullet\!\!-\!\!\circ\, fout(t) = 3\frac{t^2}{2}$$

where $F(s) \bullet\!\!-\!\!\circ f(t)$ denotes that $f(t)$ is the inverse Laplace transformed of $F(s)$, i.e. $\mathscr{L}^{-1}\{F(s)\} = f(t)$. The continuous behavior is a vector of two functions $\langle in(t), out(t)\rangle^T$ where in and out evolve simultaneously. Hence, for qualitative actions we introduce a special label *qual* which denotes the evolution of input and output quantities. In our example system the flow rates in, out, and $inout$ are input quantities and x_1, x_2 are output quantities.

In the next section we present *ioco* as conformance relation.

IV. Input-Output Conformance

Conformance relations are used to determine if an IUT behaves correctly regarding a given specification. In order to decide conformance some testing hypotheses have to be stated [15]. One is that the implementation can be represented with the same formalism as the specification. In our application this is always the case since we decide the conformance between two specifications (an original and a mutated version).

The ioco relation expresses the conformance of implementations to their specifications where both are represented as labeled transition systems (LTS). Because ioco distinguishes between inputs and outputs, the alphabet of an LTS is partitioned into inputs and outputs.

Definition 5 (LTS with inputs and outputs): A labeled transition system with inputs and outputs is a tuple $M = (S, L \cup \{\tau\}, \rightarrow, s_0)$, where S is a countable, non-empty set of states, $L = L_I \cup L_U$ a finite alphabet, partitioned into two disjoint sets, where L_I and L_U are input and output alphabets, respectively. $\tau \notin L$ is an unobservable action, $\rightarrow \subseteq S \times (L \cup \{\tau\}) \times S$ is the transition relation, and $s_0 \in S$ is the initial state.

We use the following common notations:

Definition 6: Given a labeled transition system $M = (S, L \cup \{\tau\}, \rightarrow, s_0)$ and let s, s', $s_i \in S, a_{(i)} \in L$ and $\sigma \in L^*$.

$$\begin{aligned}
s \xrightarrow{a} s' &=_{df} (s, a, s') \in \rightarrow \\
s \xrightarrow{a} &=_{df} \exists s' \bullet (s, a, s') \in \rightarrow \\
s \not\xrightarrow{a} &=_{df} \nexists s' \bullet (s, a, s') \in \rightarrow \\
s \overset{\epsilon}{\Rightarrow} s' &=_{df} s = s' \vee \exists s_0 \ldots s_n \bullet \\
&\qquad s = s_0 \xrightarrow{\tau} s_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} s_{n-1} \xrightarrow{\tau} s_n = s' \\
s \overset{a}{\Rightarrow} s' &=_{df} \exists s_1, s_2 \bullet s \overset{\epsilon}{\Rightarrow} s_1 \xrightarrow{a} s_2 \overset{\epsilon}{\Rightarrow} s' \\
s \overset{a_1 \ldots a_n}{\Rightarrow} s' &=_{df} \exists s_0, \ldots, s_n \bullet s = s_0 \overset{a_1}{\Rightarrow} s_1 \overset{a_2}{\Rightarrow} \ldots \overset{a_n}{\Rightarrow} s_n = s' \\
s \overset{\sigma}{\Rightarrow} &=_{df} \exists s' \bullet s \overset{\sigma}{\Rightarrow} s'
\end{aligned}$$

Furthermore, for an LTS M we define:

$$\begin{aligned}
init(s) &=_{df} \{a \in L \cup \{\tau\} \mid s \xrightarrow{a}\} \\
traces(M) &=_{df} \{\sigma \in L^* \mid s_0 \overset{\sigma}{\Rightarrow}\} \\
s \textbf{ after } \sigma &=_{df} \{s' \mid s \overset{\sigma}{\Rightarrow} s'\}
\end{aligned}$$

The first relation *init(s)* defines the set of events enabled in state s. The next definition associates to an LTS the according set of event sequences starting from the initial state. The relation *after* determines the set of states reachable after a trace σ starting from an initial state. Moreover, an LTS M has finite behavior if all traces have finite length and it is deterministic if $\forall \sigma \in L^* \bullet |s_0 \textbf{ after } \sigma| \leq 1$ holds.

For the *ioco* relation IUTs are considered to be weak input enabled, i.e. all inputs (possibly preceded by τ transitions) are enabled in all states: $\forall a \in L_I, \forall s \in S \bullet s \overset{a}{\Rightarrow}$. This class of LTS is referred to as $IOTS(L_I, L_U)$ where $IOTS(L_I, L_U) \subset$

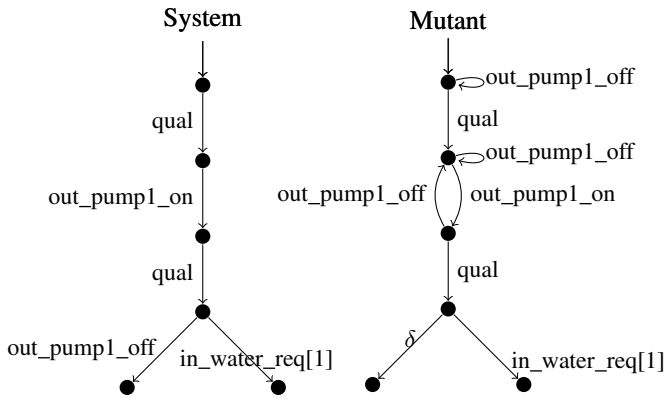


Fig. 4. Two suspension automata showing the behavior of the water tank example and of a mutated version of the water tank specification.

$LTS(L_I \cup L_U)$. A state s from which the system cannot proceed without additional inputs from the environment is called *quiescent*, denoted as $\delta(s)$. In such a state all output and internal events are disabled: $\forall a \in L_U \cup \tau \bullet s \stackrel{a}{\not\rightarrow}$. The special label $\delta \notin L$ denotes the absence of any output event in a state. Hence, the transition relation $\rightarrow$ is extended by adding δ self-loops at quiescent states: $\rightarrow_\delta =_{df} \rightarrow \cup \{(s, \delta, s) \mid s \in S \wedge \delta(s)\}$. Let M_δ be the LTS over the alphabet $L \cup \{\tau, \delta\}$ resulting from adding δ self-loops to an LTS M. Then the deterministic version of M_δ is called *suspension automaton* Γ. The set of suspension traces is

$$Straces(M_\delta) =_{df} \{\sigma \in (L \cup \delta)^* \mid s_0 \stackrel{\sigma}{\Rightarrow}\}.$$

For a state $s \in S$ the set of outputs possible in s is defined as follows:

$$out(s) =_{df} \{a \in L_U \mid s \stackrel{a}{\rightarrow}\} \cup \{\delta \mid \delta(s)\}$$
$$out(S') =_{df} \bigcup_{s \in S'} out(s)$$

Informally, the input-output conformance relation states that for all suspension traces in the specification the outputs of the implementation after such a trace must be included in the set of outputs produced by the specification after the same trace. More formally:

Definition 7: For implementation models $i \in IOTS(L_I, L_U)$ and specifications $s \in LTS(L_I \cup L_U)$ the relation *ioco* is defined as follows:

$$i \text{ ioco } s =_{df} \forall \sigma \in Straces(s) \bullet out(i \textbf{ after } \sigma) \subseteq out(s \textbf{ after } \sigma)$$

Example IV.1. Figure 4 shows two suspension automata with $L_I = \{in\_water\_req[1]\}$ and $L_U = \{qual, out\_pump1\_on, out\_pump1\_off, \delta\}$. These two LTSs do not conform with respect to ioco because out(Mutant **after** $\langle\rangle$) $= \{qual, out\_pump1\_off\} \not\subseteq \{qual\} =$ out(System **after** $\langle\rangle$) □

V. On-the-fly Conformance Checking

A conformance relation is defined between two formal models, in our case $ioco \subset LTS(L_I \cup L_U) \times LTS(L_I \cup L_U)$. Usually one of the models is the implementation model, however we determine the conformance between two specifications. These models represent the behaviors of hybrid systems which are derived on-the-fly during the exploration of our specification models, i.e. QAS.

The labeled transition system of a QAS M is given by considering events as labels in the LTS. Then the LTS semantics of a QAS M is given by the set of event traces M can produce after a finite number of computation steps. More formally: Given a QAS M, and an initial state $s_0 \in S$, the set of event traces is defined as:

$$\begin{aligned} traces(M) =_{df} \{\sigma \downarrow L \mid (\sigma \in (L \cup I)^* \bullet \\ \sigma = \langle a_0, \ldots, a_n \rangle \wedge s_0 \stackrel{a_0}{\rightarrow} s_1 \stackrel{a_1}{\rightarrow} \ldots s_n \stackrel{a_n}{\rightarrow} \wedge \\ \forall i = 0 \ldots n \bullet g(a_i)(s_i))\} \end{aligned} \quad (3)$$

Here, $g(a_i)$ is the enabledness guard of action a_i and I denotes the set of internal actions. If the current state s_i satisfies the enabledness guard the according action is executed and the event name a_i is appended to the trace which led to s_i. Furthermore, the action execution updates state s_i to s_{i+1}. Note, that internal actions, when enabled, cause τ transitions: $\forall i \in I, s \bullet g(i)(s) \implies s \stackrel{\tau}{\rightarrow}$. Since we are only interested in traces over the alphabet L, internal action are removed by projecting σ' onto L, see (3).

Note that we consider *qual* $\in L_U$ as an (observable) output event denoting the end of an environmental evolution, thus the changes of input and output quantities during an evolution are internal.

Example V.1. For example, consider the generation of the LTS "System" in Figure 4 from the system specification in Figure 3. Starting from the initial state of the system specification in Figure 3 only the qualitative action is enabled. This is because all guards $(g_1, \ldots, g_5)$ evaluate to false in the initial state. The execution of the action leads to the *qual* event after the initial state of the LTS "System" in Figure 4. The qualitative action updates x_1 from $(0, 0)$ to $(full, 0)$. In this state the guard g_1 evaluates to true. Hence, the action *PUMP1_ON* can be executed. This results into the *out_pump1_on*-labeled transition in the LTS. The other parts of the system are derived in a similar manner. For the sake of brevity, we do not show the complete LTS of the system. Note, that the Mutant LTS shows parts of an LTS obtained by negating the guard g_2 in the original specification. Also here only parts of the whole LTS are shown. □

The conformance verification between two LTSs is achieved by computing the synchronous product modulo the conformance relation [16], i.e. $LTS_1 \times_{ioco} LTS_2$. In the case of non-conformance the resulting product LTS will contain special *fail* states. The conformance verification is then achieved by a reachability analysis of *fail* states. Any trace leading to a fail state is a counter-example showing non-conformance.

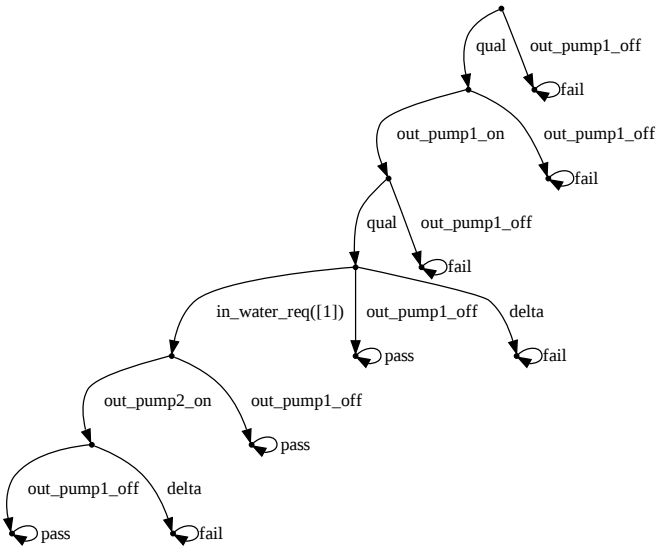


Fig. 5. Conformance verification result with a mutated action 'out_pump1_off'

Informally, the synchronous product $LTS_1 \times_{ioco} LTS_2$ is calculated by applying one of the following rules:

- Transitions that are possible in both LTSs are transitions of the synchronous product.
- Inputs that are allowed in LTS_1 but are not allowed in LTS_2 lead to a sink state labeled with *pass*. This rule reflects that implementations may behave arbitrarily after unspecified inputs.
- For any output that is allowed in LTS_2 but not in LTS_1 a transition to a pass state is added.
- For inputs that are enabled in LTS_2 but not in LTS_1 the labeled transition system LTS_1 is made input-enabled, i.e. such transitions are part of the synchronous product.
- If an output of the left hand LTS LTS_1 is not an output of the right hand LTS LTS_2 a transition leading to a *fail* state is added to the synchronous product.

Example V.2. The calculation of the synchronous product between the two labeled transition systems of Figure 4 leads to the very beginning of the LTS shown in Figure 5. For example, the *qual*-action is enabled in both LTS, thus it is part of the synchronous product. The output action *out_pump1_off* is not allowed by the specification. Consequently, this action leads to a fail state in the synchronous product. After the *qual*-action the *out_pump1_off* action is still not allowed by the specification. Thus, again this action leads to a fail state. Note, that Figure 5 shows the synchronous product for the larger LTSs than those depicted in Figure 4. □

The product calculation is performed on-the-fly. Our Prolog conformance checker explores two given Qualitative Action Systems state by state and computes the synchronous product according to these rules. On-the-fly computation can be quite efficient when traces which reveal the fault are rather short. However, in the worst case the complete specification has to be unfolded during product calculation.

Algorithm 1 *getSuccessors*$(qas, s_1) : L \cup \{\delta\} \mapsto \mathcal{P}(S)$

1: $s_2 := tauClosure(R, s_1, \tau)$;
2: $qstates := s_2 \backslash \{s \in s_2 \mid \exists ev \in L_U \cup \tau, s' \in \mathcal{P}(S) \bullet s' = R(s, ev)\}$;
3: $succ := [ev \mapsto \{s \mid \exists s' \in s_2, s'' \in \mathcal{P}(S) \bullet s'' = R(s', ev) \wedge s \in s''\} \mid ev \neq \tau]$;
4: **if** $qstates \neq \emptyset$ **then**
5: $succ := succ \cup [\delta \mapsto qstates]$;
6: **end if**
7: **return** $succ$;

Algorithm 1 computes the successor events and successor states in the suspension automaton Γ of a given QAS. Because of determinization via subset construction the states in Γ are sets of states. Given a specification qas and a state s_1 *getSuccessors* returns a map from events to successor states. For the successor computation we use a relation $R(s, ev)$ which takes a state s and an event ev and calculates the successor states of s that are reachable by event ev. More formally,

$$R(s, ev) = \{s_1 \mid \exists s_2 \bullet int(qas, s, s_2, ev) \wedge s_1 \in s_2\}$$

Here, the predicate $int : QAS \times S \times \mathcal{P}(S) \times (L \cup \{\tau\}) \mapsto bool$ interprets the given QAS specification by nondeterministically interpreting all actions.

At first all states reachable via internal actions are computed. This process is called τ-closure computation because it generates no observable events. We assume that specifications are strongly converging, i.e. there are no infinite sequences of internal computations. Because of its pattern matching and backtracking capability Prolog is well suited to describe search and exploration problems. Listing 1 shows that *tauClosure* can be written quite compact with only three Prolog clauses.

Listing 1. tauClosure

```
1 tauClosure(QAS, S1, S2)          :-
    tauClosure(QAS, S1, S1, S2).
3 tauClosure(_, [], Cl, Cl)        :- !.
  tauClosure(QAS, S1, A, Cl)       :-
5   findall(S2, (member(S, S1),
                 int(QAS, S, S2, tau)), S3),
7   flatten(S3, States),
    difference(States, A, Frontier),
9   union(Frontier, A, A1),
    tauClosure(QAS, Frontier, A1, Cl).
```

Here, the predicates *difference(A, B, C)* and *union(A, B, C)* are the conventional set operations $C = A \backslash B$ and $C = A \cup B$, respectively. When an action is enabled the interpreter executes it and returns its label plus the set of successor states. In the current version of our tool actions are interpreted with concrete values, thus nondeterministic updates of state variables yield a set of successor states.

For the purpose of τ-closure computation the event variable of the *int* predicate is instantiated with `tau`. This ensures that only internal actions are interpreted. The built-in predicate *findall* collects all successor states $S2$ of executed internal actions, that are enabled in state S, in a list stored in variable

$S3$. The *member* predicate enumerates all states $S \in S1$. Since $S2$ is a list of states, $S3$ is a list of lists. In Line 7 the built-in predicate *flatten* converts S into a single list of states $States$. The set difference in *Frontier* contains new states to be explored and the set union in $A1$ are the states which have already been visited. When the base case in Line 3 is reached the exploration agenda in *Frontier* is empty. Then the variable $S2$ in Line 1 is unified with the list of all states reachable by the execution of internal actions.

Coming back to the description of Algorithm 1, the set of states after τ-closure computation, from which no internal or output action is enabled, are *quiescent* states and stored in $qstates$ (Line 2). Next, the specification is interpreted for all states $s' \in s2$. All successor events $ev \neq \tau$ and successor states s are entered into the successor map $succ : L \cup \{\delta\} \mapsto \mathcal{P}(S)$ via the map comprehension in Line 3. If there are some quiescent states then the successor map is extended with a δ event associated with the set of quiescent states (Line 5). Finally, the successor map is returned by *GetSuccessors*.

During exploration the interpretation of qualitative actions requires to solve QDEs. When the evolution guard of a qualitative action holds in a certain state then the QR tool *ASIM* [10] is called to compute the qualitative transition system. Then, according to Definition 4, a breadth-first search determines the set of states where the evolution terminates. From these states the exploration of further actions proceeds.

Starting from an initial state and by recursively applying the *getSuccessor* algorithm to two given specifications we compute the synchronous product by following the rules described above.

A. Notes on Test Case Generation

An application of the presented conformance checker is mutation-based test case generation. A test case is basically a tree that can be extracted from the synchronous product. However, test cases need to take care of the environmental events.

During conformance verification environmental updates are represented by observable *qual* events. However, the information about qualitative values at the end of an evolution are important in order to decide the enabledness of subsequent events. In addition the valuation is required by a test adapter to determine the end of an evolution for raising *qual* events. The adapter samples the environment while performing qualitative abstraction, i.e. mapping concrete to qualitative values. Then the trigger of *qual* events could be implemented in Prolog via unification. Let us consider an example where an evolution terminates when the environment has the state $x_1 = Empty \vee x_2 = Full$. The corresponding Prolog clause $qual(X1, X2)$:- $X1 = empty; X2 = full$ is true when $X1$ and $X2$ are unified with according values from the test adapter.

Hence, for test cases, *qual* events are extended with parameters and nondeterministic assignments to these parameters. This is an equivalent, but more compact representation than branching over all possible parameter valuations. In the following section we will see an example where this issue is treated.

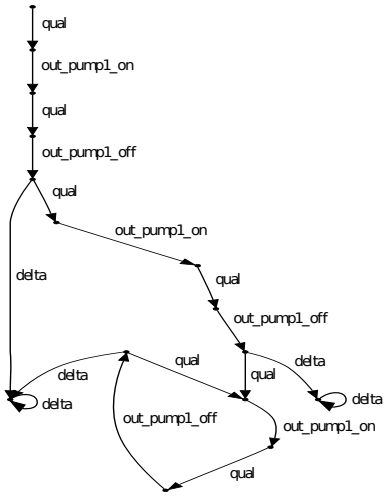


Fig. 6. Scenario where only pump $P1$ is controlled

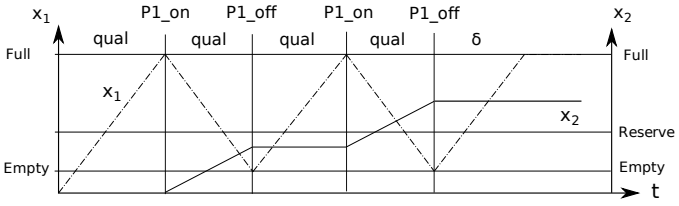


Fig. 7. Example trace of a real system can be replayed on the LTS in Figure 6

VI. Experimental Results

The LTSs in Figure 6 and 5 have been generated by the verification tool. Input and output events are prefixed with 'in_' and 'out_', respectively. When the conformance check of a specification is applied to itself then the resulting LTS is a full exploration of the system behavior. This is useful during system development when one wants to check if the written specification reflects the intended behavior.

Because the visualization of the full system behavior would be too large, Figure 6 shows a smaller scenario. Here, the three actions *PUMP2_ON*, *PUMP2_OFF*, and *WATER_REQ(X)* have been removed from the QAS specification. The controller can only switch pump $P1$ on and off; this happens until the the water level in the upper tank $T2$ has increased above the *Reserve* mark. At this point the system terminates, because pump $P1$ cannot be activated anymore and all guards evaluate to false. Recall that this is our termination condition.

Figure 7 depicts a possible trace of a real system. The system starts in the initial state where both tanks are empty and there is some nonzero inflow into $T1$. When the lower tank gets full the controller starts $P1$. The trace in Figure 7 shows that the flow rate of the pump is higher than the inflow rate, thus the water level in tank $T1$ decreases to the *Empty* mark. However, the inflow rate could also be equal or higher than the pump flow rate. In this case $P1$ would keep pumping until tank $T2$ is full.

TABLE I
RESULTS WHEN APPLYING CONFORMANCE VERIFICATION TO MUTATED SPECIFICATIONS.

Mut. Op.	No.Mut-ations	Avg.Time [ms]	Average No. states	Average No. trans	$=$	$\neq$
ASO	10	2230	33.4	53.2	8	2
ENO	6	777	21.8	35.2	0	6
ERO	20	939	19.6	30.7	0	20
Σ	36	1315	24.9	39.7	8	28

Furthermore, depending on the capacity of the two tanks more or less pump switches are required to fill the upper tank. The model of the continuous environment is abstract enough to explain several numerical variations in real implementations of the system.

The LTS in Figure 5 shows the conformance verification result between the original and a mutated version of the specification. In particular the *Expression Negation Operator* (ENO) was applied to the guard of action *out_pump1_off* in the specification QAS. This operation yields a mutant which is not equivalent to the specification because the resulting LTS contains *fail* states.

Table I depicts first results when applying conformance verification to mutated specifications. We manually applied some well known mutation operators (ASO = association shift operator, ENO = expression negation operator, ERO = event replacement operator) [17] to the specification shown in Figure 3. Table I shows for each mutation operator (1st column) the total number of generated mutants (2nd column) the time needed for the calculation of the complete synchronous product (3rd column), and the size of the synchronous products in terms of number of states (4th column) and number of transitions (5th column). Furthermore, this tables shows the number of equivalent (6th column) and the number of different mutants (7th column).

In the previous section it was mentioned that for test case generation *qual* events are extended with parameters which contain the information of qualitative state variables at the end of an evolution. For instance, after the second *qual* event in Figure 5 it depends on the valuation of the qualitative state variables which successive event is enabled. In this case the input event *in_water_req(X)* requires that the level in tank $T2$ is *Full*.

Hence, in order to obtain meaningful test cases which react and depend on environmental changes the valuation of observed qualitative variables are part of *qual* events.

VII. RELATED RESEARCH

Our work is based on *action systems* [12], [13] and their extension to *hybrid action systems* [4]. Action systems are defined in the theory of refinement calculus where program statements are formalized as predicate transformers in higher order logic. The state of an action system consists of global and local variables. An action system has at least one action. Each action is guarded and in the case of several enabled actions one of them is chosen nondeterministically. The behavior of an action system starts from an initial state where an initialization predicate must hold and is followed by possibly infinite states. Finite behavior can be terminated or aborted. Terminated behaviors end in a state where no further action is enabled and aborted behaviors end in states where termination cannot be guaranteed. Action systems enable compositional modeling via the parallel composition operator. The work in [18] deals with trace refinement of action systems.

The extension to hybrid action systems provides the possibility to model systems of partially defined ordinary differential equations (ODEs). Each differential action represents a partially defined ODE for which the domain is bounded by an invariant. Discrete actions are executed as their guards are enabled and may cause noncontinuous change in the system.

The conformance relation *hioco* [19] is an extension of Tretmans' *ioco* testing theory for labeled transition systems. It states that in every system state the discrete and continuous output behavior of the implementation must conform to the hybrid specification. Transitions are labeled with actions that can be discrete or continuous. Continuous actions are called trajectories $\sigma \in \Sigma$ where $\sigma =_{df} (0, t] \rightarrow val(V)$ valuates a set of variables V. Furthermore, the set of actions is partitioned into input and output actions. In addition to the traditional definition of *ioco* the set of trajectories in the implementation must be included in the set of allowed trajectories in the specification. An abstract test case generation algorithm is presented which is not directly implementable.

The work in [20] deals with randomized test case generation for hybrid systems. Based on the notation of hybrid automata the approach refers to states as (x, q) tuples where $x \in \mathbb{R}^n$ is a valuation of the continuous variables and q is a set of discrete variables which index the system mode. The idea is to explore the state space by building Rapidly Exploring Random Trees (RRTs) [21]. The RRT algorithm has been used in robotics for path planning by computing control signals for trajectories in high dimensional spaces.

The work in [22] deals with the modeling of hybrid systems using interval arithmetic constraints. Interval arithmetic provides a means to deal with rounding errors where the real value of a variable is located somewhere within an interval. Systems are specified in the *CLP(F)* language which can state constraints over real numbers and analytic constraints over differentiable functions. The underlying constraint solver calculates, similar to *QSIM* [7], an over approximation of the solution of a system of ODEs. Due to over approximation the solver returns a set of solution intervals. If there is a correct solution to a query it will be in one of the returned intervals. On the other hand not all solutions in the returned set may contain actual solutions. The CLP(F) system solves analytic constraints by using power series to approximate analytic functions. It is also possible to handle non-linear ODEs. A drawback of the approach is the high resource consumption with increasing modeling time. This is because of an increase of constraints over Taylor coefficients in the according power series.

VIII. Conclusions and Future Work

This paper presented a new approach to verify the input output conformance (ioco) between two hybrid systems. In order to get a discrete event view on a hybrid system, according to the work in [4], only the begin and the end of continuous evolutions in the environment are observed. We described this environmental changes with nondeterministic update statements. Along such a change the observable event *qual* is introduced. For test case generation we propose to extend *qual* events with according parameters of observed qualitative state variables for two reasons. First, the parameter values associate the event with the end of the corresponding evolution. Secondly, the values are required to resolve enabledness decisions for subsequent events.

A discrete event view on a hybrid system allows us to apply well known conformance relations like *ioco* for model-based test case generation. Even though such test cases contain no continuous information they are able to reveal unspecified environmental conditions in hybrid systems whenever they affect the discrete behavior of the system.

After giving an introduction to Qualitative Reasoning, Action Systems, and Labeled Transition Systems with the conformance relation *ioco*, this work presented the exploration of Qualitative Action Systems and the according generation of LTSs. Then it was shown how the input output conformance verification between two given QAS specifications works. After discussing the application of conformance verification for mutation-based test case generation the approach was demonstrated on a two-tank pump control example.

The contribution of this work is threefold: (1) definition of an LTS semantics for hybrid systems by the introduction of *qual* events, (2) implementation of a tool for conformance verification, and (3) first experimental results for mutation testing by means of conformance verification.

For future work the approach can be extended by considering the evolutions inside *qual* events. Therefor, the trajectories of hybrid systems can be verified with the *qrioconf* [23] relation while the discrete events satisfy *ioco*. This combination would result in a discrete version of hioco [19].

References

[1] J. Tretmans, "Test generation with inputs, outputs and repetitive quiescence," *Software - Concepts and Tools*, vol. 17, no. 3, pp. 103–120, 1996.

[2] T. A. Henzinger, "The theory of hybrid automata." IEEE Computer Society Press, 1996, pp. 278–292.

[3] D. A. van Beek, K. L. Man, M. A. Reniers, J. E. Rooda, and R. R. H. Schiffelers, "Syntax and consistent equation semantics of hybrid chi," *J. Log. Algebr. Program.*, vol. 68, no. 1-2, pp. 129–210, 2006.

[4] M. Rönkkö, A. P. Ravn, and K. Sere, "Hybrid action systems," *Theoretical Computer Science*, vol. 290, pp. 937–973, 2003.

[5] R.-J. Back, L. Petre, and I. Porres, "Continuous action systems as a model for hybrid systems," *Nordic Journal of Computing*, vol. 8, no. 1, pp. 2–21, 2001.

[6] T. A. Henzinger, B. Horowitz, R. Majumdar, and H. Wong-toi, "Beyond hytech: Hybrid systems analysis using interval numerical methods," in *in HSCC*. Springer, 2000, pp. 130–144.

[7] B. Kuipers, *Qualitative Reasoning: Modeling and Simulation with Incomplete Knowledge*. MIT Press, 1994.

[8] B. Bredeweg, A. Bouwer, J. Jellema, D. Bertels, F. F. Linnebank, and J. Liem, "Garp3 - a new workbench for qualitative reasoning and modelling," in *Proceedings of 20th International Workshop on Qualitative Reasoning (QR-06)*, Hannover, New Hampshire, USA, 2006, pp. 21–28.

[9] B. K. Aichernig, H. Brandl, and W. Krenn, "Qualitative action systems," in *ICFEM*, ser. LNCS, K. Breitman and A. Cavalcanti, Eds., vol. 5885. Springer, 2009, pp. 206–225.

[10] E. B, I. Bratko, and D. Suc, "Qualitative simulation with clp," in *In Proc. of 16th International Workshop on Qualitative Reasoning (QR02)*, 2002.

[11] H. de Jong, "Qualitative simulation and related approaches for the analysis of dynamical systems," *The Knowledge Engineering Review*, vol. 19, no. 2, pp. 93–132, 2003.

[12] R.-J. Back and R. Kurki-Suonio, "Decentralization of process nets with centralized control," in *Proceedings of the 2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*. Montreal, Quebec, Canada: ACM, 1983, pp. 131–142.

[13] R.-J. Back and K. Sere, "Stepwise refinement of parallel algorithms," *Science of Computer Programming*, vol. 13, no. 2-3, pp. 133–180, Nov 1990.

[14] M. Rönkkö and A. P. Ravn, "Switches and jumps in hybrid action systems," Turku Centre for Computer Science, Tech. Rep. 152, 1997.

[15] G. Bernot, "Testing against formal specifications: A theoretical view," in *TAPSOFT '91: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Volume 2: Advances in Distributed Computing (ADC) and Colloquium on Combining Paradigms for Software Developmemnt (CCPSD)*. London, UK: Springer-Verlag, 1991, pp. 99–119.

[16] M. Weiglhofer and F. Wotawa, ""On the fly" input output conformance verification," in *Proceedings of the IASTED International Conference on Software Engineering*. Calgary, Canada: ACTA Press, February 2008, pp. 286–291.

[17] P. E. Black, V. Okun, and Y. Yesha, "Mutation operators for specifications," in *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*. Washington, DC, USA: IEEE Computer Society, September 2000, pp. 81–88.

[18] R.-J. Back and J. von Wright, "Trace refinement of action systems," in *CONCUR-94:Concurrency Theory*, ser. Lecture Notes in Computer Science, B. Jonsson and J. Parrow, Eds., vol. 836. Uppsala, Sweden: Springer-Verlag, Aug 1994, pp. 367–384.

[19] M. van Osch, "Hybrid input-output conformance and test generation," in *FATES/RV*, ser. LNCS, K. Havelund, M. Núñez, G. Rosu, and B. Wolff, Eds., vol. 4262. Springer, 2006, pp. 70–84.

[20] J. Esposito, "Randomized test case generation for hybrid systems: metric selection," *System Theory, 2004. Proceedings of the Thirty-Sixth Southeastern Symposium on System Theory*, pp. 236–240, 2004.

[21] S. M. LaValle and J. J. K. Jr., "Randomized kinodynamic planning," in *Int'l Conference on Robotics and Automation*, 1999, pp. 473–479.

[22] T. J. Hickey and D. K. Wittenberg, "Rigorous modeling of hybrid systems using interval arithmetic constraints," in *HSCC*, ser. Lecture Notes in Computer Science, R. Alur and G. J. Pappas, Eds., vol. 2993. Springer, 2004, pp. 402–416.

[23] B. K. Aichernig, H. Brandl, and F. Wotawa, "Conformance testing of hybrid systems with qualitative reasoning models," in *Model-Based Testing, MBT 2009*, B. Finkbeiner, Y. Gurevich, and A. K. Petrenko, Eds., 2009, pp. 45–59.