A Counterexample-Guided Approach to
Symbolic Simulation of Hybrid Systems

Xian Li, Klaus Schneider
TU Kaiserslautern
xian.li@cs.uni-kl.de
klaus.schneider@cs.uni-kl.de

Abstract

In this paper, we propose a symbolic simulation algorithm for hybrid systems that are specified by parameterized formal models using standard data types (reals, integers and booleans) and non-linear dynamics. To support the proposed algorithm, we develop a prototypical constraint solver for non-linear arithmetic theories over integers and reals with quantifiers. Given a system’s symbolic representation, specifications, and finitely many concrete input values, the algorithm computes ranges of the input parameters by extending each concrete value to a range constraint until some parameter valuation violates the specifications. In this way, no concrete value that belongs to the generated ranges need to be considered for subsequent simulations. The algorithm has been prototypically implemented, and its feasibility is proved by a successful experimental evaluation using parameterized hybrid programs.

1. Introduction

Simulation is a commonly used method for validating behaviors of complex dynamical systems. The integration of simulation and symbolic analysis yields the so-called symbolic simulation that improves the simulation coverage of the system model [Alu11]. Symbolic simulation of hybrid systems evaluates a certain dynamic behavior of the system by using parameterized system models.

There are different kinds of formal models for hybrid systems that could be parameterized: Linear Hybrid Automata (LHAs) and Affine Hybrid Automata (AHAs) are special Hybrid Automata, where LHAs are restricted to linear dynamics for continuous state variables, while affine dynamics are allowed for AHAs. Hybrid programs are based on modeling languages, so that the hybrid systems could be encoded with data types and programming statements. Typical examples are KeYmaera [Pla10], HyDI [CMT11], HybridSAL [Tiw12] and Hybrid Quartz [Bau12]. For the symbolic simulation, only parameterized input variables are considered.

The hybrid systems discussed in this paper are specified by parameterized formal models supporting standard data types (reals, integers and booleans) and affine dynamics, like Hybrid Quartz, HyDI, and HybridSAL. Automated analysis of this kind of models leads to an undecidable satisfiability problem. This is the case since the underlying logic allows boolean combinations of propositional logic atoms as well as atoms from non-linear arithmetic theories over integers and reals with quantifiers. Even though numerous approaches from different backgrounds have been
proposed in literature, each of them could only solve a subclass of the underlying undecidable satisfiability problem. Therefore, we analyze various tools, i.e. constraint solvers, and develop a new prototypical constraint solver by integrating the external tool Bonmin [BBC+08] into the BDD package implemented in our Averest system (www.averest.org). Based on our new solver, we propose a counterexample-guided algorithm for symbolic simulation of hybrid systems. Given a system’s symbolic representation, specifications, and finitely many concrete input values, the algorithm computes ranges of the input parameters by extending each concrete value to a range constraint until some parameter valuation violates the specification. In this way, no concrete value that belongs to the generated ranges need to be considered for subsequent simulations. The algorithm has been prototypically implemented in the Averest system, and its feasibility is proved by a successful experimental evaluation using parameterized Hybrid Quartz programs.

The rest of the paper is organized as follows: Section 2 presents the syntax and semantics of the satisfiability problems that we would like to solve for symbolic simulation of hybrid systems together with a discussion of the available tools. Section 3 gives implementation details of the prototypical constraint solver. The counterexample-guided algorithm for symbolic simulation of hybrid systems will be introduced in Section 4, while Section 5 describes the experiments that we performed for this paper. The paper will be concluded in Section 6.

2. Preliminaries

First, we describe the satisfiability problem that we consider when performing symbolic simulation of hybrid systems that are specified by parameterized formal models using standard data types (reals, integers and booleans) and affine dynamics, so that the decidability of the satisfiability problem and the available tools can be discussed.

2.1. Syntax

We assume a finite set of real, integer and boolean variables \( V = V_{\mathbb{R}} \uplus V_{\mathbb{Z}} \uplus V_{\mathbb{B}} \), and a finite set of real, integer and boolean input parameters \( W = W_{\mathbb{R}} \uplus W_{\mathbb{Z}} \uplus W_{\mathbb{B}} \), where \( W \) is a subset of \( V \), which means \( W_{\mathbb{R}} \), \( W_{\mathbb{Z}} \), and \( W_{\mathbb{B}} \) are subsets of \( V_{\mathbb{R}} \), \( V_{\mathbb{Z}} \), and \( V_{\mathbb{B}} \), respectively. We define boolean expressions and numerical expressions, by the following grammars:

\[
\begin{align*}
e_b & := x \in V_{\mathbb{B}} \mid \neg e_b \mid e_b \land e_b \mid e_b \lor e_b \\
e & := x \in V_{\mathbb{R}} \uplus V_{\mathbb{Z}} \mid e + e \mid e - e \mid e \cdot e \mid e/e
\end{align*}
\]

If \( X_{\mathbb{B}} \subseteq V_{\mathbb{B}} \) is the quantifier set, then boolean expression with \( \exists \)-quantifiers is defined as follows:

\[
(e_b)_Q := (\exists X_{\mathbb{B}}). e_b.
\]

Similarly, given \( X_{\mathbb{R}} \subseteq V_{\mathbb{R}} \) and \( X_{\mathbb{Z}} \subseteq V_{\mathbb{Z}} \) as quantifier sets, numerical expressions \( e \) and \( e' \), and operator \( \odot \in \{ \leq, = \} \), the following formula defines a constraint and a constraint with \( \exists \)-quantifier.

\[
c_Q := (\exists X_{\mathbb{R}}, X_{\mathbb{Z}}). c \quad \text{where} \quad c := e \odot e' \mid c \land c
\]

For convenience of analysis and explanation, the syntax of the satisfiability problem that we consider could be defined as the following disjunctive form:

\[
C_b := \bigvee_{i \in \mathbb{N}} ((e_Q)_i \land ((e_b)_Q)_i)
\]
Formulas without quantifiers are no longer considered. It is reasonable to do that since the constraints and boolean expressions without quantifiers could be represented by those with quantifiers. For the rest of the paper, we assume that the constraint on bound variables that appear in (2-4) are given, denoted as $v_c$. To have some idea on how the formulas look like, we assume that $\mathcal{V}_R = \{r_1, r_2\}$, $\mathcal{V}_Z = \{z_1, z_2\}$, $\mathcal{V}_B = \{b_1, b_2\}$, and list some examples in Table 1.

### Table 1: Formula Examples

<table>
<thead>
<tr>
<th>Category</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e_b$</td>
<td>$E_0 : b_1 \land b_2$</td>
</tr>
<tr>
<td>$c$</td>
<td>$E_1 : r_1 \leq r_2 + z_1 \ast (r_1 - z_2)$</td>
</tr>
<tr>
<td>$(e_b)_Q$</td>
<td>$E_2 : (\exists \mathcal{V}_b). b_1 \land b_2$</td>
</tr>
<tr>
<td>$c_Q$</td>
<td>$E_3 : (\exists \mathcal{V}_b, \mathcal{V}_Z). E_4$</td>
</tr>
<tr>
<td>$C_b$</td>
<td>$E_4 : E_2 \land E_3$</td>
</tr>
<tr>
<td>$e_v$</td>
<td>$E_5 : ((\exists {r_1}). r_1 \geq 1) \lor ((\exists {r_1}). r_1 \leq 0.9)$</td>
</tr>
<tr>
<td></td>
<td>$E_6 : (-2.0 \leq r_1 \ast r_2 \leq 10.0) \land (z_1 \leq z_2 \leq 5)$</td>
</tr>
</tbody>
</table>

#### 2.2. Semantics

The semantics of formulas (1-4) are defined by the interpretation $[\cdot]_v$, which evaluates the inside formula to True or False using the variable valuation function $v$.

**Definition 1 (Variable Valuation)** A variable valuation is a function $v : \mathcal{V} \rightarrow \mathbb{R} \cup \mathbb{Z} \cup \mathbb{B}$ that assigns to each variable a real, integer or boolean value. $v|_d$ is the $x$-variant of the variable valuation $v$. $v$ and $v|_d$ agree on everything except possibly the value of variable $x$, where $v|_d(x) = d$ for some $d \in \mathbb{R} \cup \mathbb{Z} \cup \mathbb{B}$.

**Definition 2 (Semantics)** Given variable $x$, numerical expressions $e_1$ and $e_2$, constraints $c$, boolean expressions $e_b, e_{b_1}$, and $e_{b_2}$, together with operator $\oplus \in \{+,-,\ast,/,\leq,\geq\}$, to which we assign standard numerical relations of reals and integers, the semantics is defined inductively as follows:

- $[x]_v := v(x)$
- $[e_1 \oplus e_2]_v := [e_1]_v \oplus [e_2]_v$
- $[\neg e_b]_v := \begin{cases} True, \text{ iff } [e_b]_v = False \\ False, \text{ Otherwise} \end{cases}$
- $[e_{b_1} \land e_{b_2}]_v := \begin{cases} True, \text{ iff } [e_{b_1}]_v = [e_{b_2}]_v = True \\ False, \text{ Otherwise} \end{cases}$
- $[e_{b_1} \lor e_{b_2}]_v := \begin{cases} True, \text{ iff } [e_{b_1}]_v = [e_{b_2}]_v = False \\ True, \text{ Otherwise} \end{cases}$
- $[(\exists x \in \mathcal{V}_B). e_b]_v := \begin{cases} True, \text{ iff there exists a } d \in \mathbb{B} \text{ such that } [e_b]_v|_d = True \\ False, \text{ Otherwise} \end{cases}$
- $[(\exists x \in \mathcal{V}_R \cup \mathcal{V}_Z). e]_v := \begin{cases} True, \text{ iff there exists a } d \in \mathbb{R} \cup \mathbb{Z} \text{ such that } [e]_v|_d = True \\ False, \text{ Otherwise} \end{cases}$

Taking formulas $E_1$ and $E_0$ in Table 1 as an example, where $E_6$ is the constraint on bound variables, if $v_1$ is a variable valuation that maps $r_1, r_2, z_1$ and $z_2$ to 0.95, 1.0, 1 and $-2$, respectively, then $[E_1]_{v_1}$ evaluates to True.

Based on the base cases of our inductive definition, formula (4) is evaluated by $[\cdot]_v$ as follows:

- $[C_b]_v := \begin{cases} True, \text{ iff exists a } i \in \mathbb{N} \text{ such that } [((c_{Q})_{i})]_v \land [((e_b)_{Q})_{i}]_v = True \\ False, \text{ Otherwise} \end{cases}$
Considering the variable valuation \( v_1 \) in the previous example with \( E_5 \) and \( E_6 \) in Table 1, \( \left[ E_5 \right]_v \) evaluates to \( \text{True} \), since \( v_1 \big|_{E_1} \) is a \( r_1 \)-variant of \( v_1 \), with which \( \left[ r_1 \geq 1 \right]_v \mid_{E_1} = \text{True} \) holds.

2.3. Decidability and Tools

The satisfiability problem defined by formula (4) consists of boolean combinations of propositional logic atoms and atoms of non-linear arithmetic theories over integers and reals with \( \exists \)-quantifiers. However, quantifier alternations are not allowed. Therefore, the satisfiability problem described by formula (4) is undecidable and strictly included in the combination of the first-order logic of the structure \( (\mathbb{R}, +, \cdot, \mathbb{Z}, 0, 1, <) \) and propositional logic.

From the point of view of constraint problems [BHZ06], formula (3) is a standard form formula that describes an instance of a MINLP problem. If we get rid of the boolean variables, then the remaining subset problem defined by formula (4) can be reorganized as follows:

\[
C'_b := \bigvee_{i \in \mathbb{N}} c_{Q_i}
\]  

Solving the satisfiability problem of the above formula amounts to check whether there exists a solution for a set of MINLP problems, where each MINLP problem corresponds to a subformula \( c_{Q_i} \). The solution satisfies at least one MINLP problem in the set. Therefore, we need effective techniques to check the existence of this solution for the set of MINLP problems generated by \( C'_b \).

The MINLP problem is one of the most general modeling paradigms in optimization and includes both Non-Linear Programming (NLP) and Mixed Integer Linear Programming (MILP) as subproblems. Linear constraint problems are a subset of NLP problems, while, MILP problems are a subset of MINLP problems.

When classifying the available tools and techniques for hybrid system analysis, according to the constraint problems they could solve, we have the following results:

- Linear constraint problems: HySAT [FH07] and BACH [BLWL08] are designed for linear arithmetic over reals.
- MILP problems: MathSAT5 [CGJS13], CVC4 [CVC], and Z3 [Z3] contain decision procedures for mixed integer linear arithmetic.
- NLP problems: KeYmaera and MetiTarski [Pau12] accept formulas over reals that are non-linear. Originally based on the inverse method [ACEF09], IMITATOR [And09] is a tool for efficient synthesis for Timed Automata [AD94]. In [FK11], the methodology has been extended for LHAs and AHAs. SpaceEx [FLD+11] facilitates quite a lot of algorithms related to reachability and safety verification in the theory of reals. Z3 also presents partial support of non-linear arithmetic.
- MINLP problems: iSAT [EFH08] could solve large boolean combinations of non-linear arithmetic constraints involving transcendental functions.

Some algebraic computation tools developed in different application areas attracted our attention as well.

- NLP problems: QEPCAD [QEP] can produce quantifier-free equivalent formulas for Tarski formulas. Reduce [Redb] can handle integer and real arithmetic. Its package Redlog [Reda] has some quantifier elimination procedures to solve parameterized non-linear real arithmetic problems. Both of them are widely used as external decision procedures for hybrid system verification tools, like KeYmaera. TReX [ABS01], a tool for automatic analysis of automata-
based models, relies on Reduce to solve the constraint problems in the theory of non-linear arithmetic over reals.

- **MINLP problems**: The survey [BKL+13] introduces a range of approaches to tackle MINLP problems. Bonmin is one of the open source tools mentioned in it. Bonmin contains a combination of techniques that lends itself to be a good candidate for convex MINLP problems. In general, tools for MINLP problems try to find a solution for a single MINLP problem each time. However, they cannot always find a solution. This might be due to the limitations of the tool itself or due to the fact that satisfiability of MINLP problems is undecidable. No solution returned does not mean no solution exists.

Solving the satisfiability problem requires to map the formula to a logical value (True or False). It is beyond the ability of the tools to solve the satisfiability problem for the class of formulas defined by formula (5). This is the case even though assigning a logical value is strongly related to whether a tool can find a solution for each individual MINLP problem or not. Taking $C = c_{Q_1} \lor c_{Q_2}$ as an example, if no solution is found for the MINLP problems that correspond to $c_{Q_1}$ and $c_{Q_2}$ respectively, then it is unclear what logical value $[C]_v$ should be assigned to.

Moreover, iSAT and the algebraic computation tools are developed in different application areas. It is still unclear which tool performs best to solve the satisfiability problem that we consider. Based on the above analysis, we conclude that currently available tools could solve a certain subclass of the undecidable satisfiability problem, but they all have their own limitations. They could still be used as components in the context of other tools, in accordance with the different synthesis requirements to solve different logical formulas or mathematical problems.

3. The Constraint Solver

To bridge the gap between the existing research challenges and available techniques, we develop a prototypical constraint solver by integrating the external tool Bonmin into the BDD package implemented in our Averest system. The new constraint solver has been implemented in our Averest system as F# functions so that engineers can check the formulas in an interactive F# session.

Technically speaking, formula (4) is the standard form input formula. $(c_Q)_i$ can be processed as a MINLP problem. Based on the result returned by Bonmin for the MINLP problem, we use the BDD package to evaluate the conjunctive sub-formula $((c_Q)_i \land ((e_b)_Q)_i)$.

Given $W_R = V_R = \{a\}$, $W_Z = V_Z = \{n\}$, and $n \geq 0 \land 1.26 \leq a \leq 2.24$ as constraint on parameters, let $(\exists n, a). 0.5 * a * (8 * (1 - 0.5^n))^2 \leq 24.5$ be an instance of formula $(c_Q)$. The Bonmin input file that encodes the corresponding MINLP problem is the following:

```
1: var n integer >= 0; var a >= 1.26 <= 2.24;
2: minimize cost: n;
3: subject to
4: 0.5 * a * (8 * (1 - 0.5^n))^2 <= 24.5;
```

Line 1 declares the data types and ranges of variables. The optimal function is given after the statement minimize cost, where engineers decide the minimize cost function according to their preferences. The inequality constraint is displayed in Line 4 following the statement subject to.

Bonmin returns a solution: $n = 0$ and $a = 1.74997$ for this example. Actually, the solution returned by Bonmin is part of a variable valuation for integer and real variables. Thus, a complete variable valuation that satisfies $((c_Q)_i \land ((e_b)_Q)_i)$ can be obtained together with the other part of
the variable valuation decided by \(((c_Q)_i) \vee ((e_b)_Q)_i\). Later in Section 4.2, the variable valuation produced by Bonmin will be used for Algorithm 1. However, as discussed in Section 2.3, Bonmin cannot always find a solution for the MINLP problem. Thus, we extend the interpretation \([\cdot]_v\) by the following interpretation \([\cdot]_3\) for our constraint solver.

**Definition 3 (3-Valued Evaluation)** The semantics of formulas (2-3) is redefined by the interpretation \([\cdot]_3\), which evaluates the inside formula \(\cdot\) to True, False or Unknown.

\[
\begin{align*}
\left[ (\exists X_B). e_b \right]_3 &:= \left[ (\exists X_B). e_b \right]_v \\
\left[ (\exists X_R, X_Z). c \right]_3 &:= \\
&\begin{cases} 
\text{True, if Bonmin could obtain a solution of the MINLP problem for } c \\
\text{Unknown, Otherwise}
\end{cases}
\end{align*}
\]

Truth tables for 3-valued logic of conjunction and disjunction (denoted as \(\land_3\) and \(\lor_3\)) are shown in Table 2, where \(T\), \(F\) and \(U\) represent the truth value True, False and Unknown respectively.

<table>
<thead>
<tr>
<th>(\land_3)</th>
<th>(T)</th>
<th>(F)</th>
<th>(U)</th>
<th>(\lor_3)</th>
<th>(T)</th>
<th>(T)</th>
<th>(T)</th>
<th>(F)</th>
<th>(T)</th>
<th>(F)</th>
<th>(U)</th>
<th>(U)</th>
<th>(U)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T)</td>
<td>(T)</td>
<td>(F)</td>
<td>(U)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(F)</td>
<td>(F)</td>
<td>(F)</td>
<td>(F)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(U)</td>
<td>(U)</td>
<td>(F)</td>
<td>(U)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Therefore, formula (4) is interpreted by \([\cdot]_3\) as below:

\[
[ C_b ]_3 := \\
\begin{cases} 
\text{True, iff there exists a } i \in \mathbb{N} \text{ such that } [ (c_Q)_i ]_3 \land_3 [ ((e_b)_Q)_i ]_3 = \text{True} \\
\text{False, iff for all } i \in \mathbb{N} , [ (c_Q)_i ]_3 \land_3 [ ((e_b)_Q)_i ]_3 = \text{False} \\
\text{Unknown, Otherwise}
\end{cases}
\]

Here is an example to explain the 3-valued valuation. Let \(C = (c_Q)_1 \land (c_Q)_1 \lor (e_b)_Q)_2 \land ((e_b)_Q)_2\), where \((c_Q)_1, (c_Q)_1, (c_Q)_2\), and \((e_b)_Q)_2\) are assigned to True, Unknown, False, and Unknown, respectively by interpretation \([\cdot]_3\). Then \([C]_3\) evaluates to Unknown.

In our method, the dual-rail representation is used, where two BDDs represent the three possible values of a node that corresponds to an individual MINLP problem. Thus, the correctness of our constraint solver is assured by the BDD package and Bonmin. Its capability is restricted by the two tools as well. The BDD package could be replaced by other SAT solvers or SMT solvers as mentioned in Section 2.3. Of course, Bonmin is not the only choice for solving MINLP problems. However, the target of our research is to explore a method for the subclass of formulas defined in formula (4), rather than to develop a new SMT solver to compete with other available tools. Thus, improving the efficiency of the tool integration is not the goal of this paper.

### 4. An SMT-based Algorithm to Determine Ranges for Parameters of Hybrid Systems

Based on our new solver, we propose a counterexample-guided algorithm to compute ranges of input parameters for symbolic simulation of hybrid systems. Section 4.1 gives the general restrictions for parameters that are determined by Algorithm 1. Section 4.2 explains the idea of the algorithm. Specific implementation details are given in Section 4.3.
4.1. Parameter Restriction

Before introducing Algorithm 1, we explain the few parameter restrictions we make. Only $\mathbb{R}$-typed input parameters are considered. We encode parameters in $\mathcal{W}_\mathbb{R}$ as a vector $\vec{p}$. The parameter valuation vector $\vec{v}$ is encoded similarly. An initial state set $I := \{\vec{v}_i \mid i \in \mathbb{N}\}$ contains finitely many parameter valuation vectors. It is required that each parameter $p_i \in \mathcal{W}_\mathbb{R}$ has a range constraint $\Delta_i := [\delta_{i\text{min}}, \delta_{i\text{max}}]$, where $\delta_{i\text{min}}, \delta_{i\text{max}} \in \mathbb{R}$, and $\delta_{i\text{min}} \leq \delta_{i\text{max}}$. $\Delta_i$ could be divided to subranges. Some subranges have parameter valuations that violate the specification, while some other satisfy the specification. Moreover, the parameters are independent so that all the subranges are numerical regions instead of relational regions. For example, given the input parameter set $W$ the specification. Moreover, the parameters are independent so that all the subranges are numerical regions instead of relational regions. For example, given the input parameter set $W = \{p_0, p_1\}$, $\Delta_0 = [0.0, 1.0]$ and $\Delta_1 = [1.0, 2.0]$ are numerical regions for $p_0$ and $p_1$ respectively, while $\Delta_1' = [1.0, p_0]$ is a relational region for $p_1$, since the upper bound of region $\Delta_1'$ depends on parameter $p_0$. For the same example, we have the corresponding parameter vector $\vec{p}_1 = \left(\begin{array}{c} p_0 \\ p_1 \end{array}\right)$ and the range constraint vector $\vec{\Delta} = \left(\begin{array}{c} \Delta_0 \\ \Delta_1 \end{array}\right)$. A parameter valuation vector $\vec{v} = \left(\begin{array}{c} 0.5 \\ 1.5 \end{array}\right)$ assigns 0.5 and 1.5 to $p_0$ and $p_1$ respectively. $\{\vec{v}_0, \vec{v}_1\}$ is an initial state set, where $\vec{v}_0 = \vec{v}$ and $\vec{v}_1 = \left(\begin{array}{c} 0.3 \\ 1.2 \end{array}\right)$.

4.2. Algorithm Explanation

Given a system’s symbolic representation $G$, specification $C_b$, and an initial state set $I$, the algorithm reuses the solution returned by Bonmin that violates the given specification. The recursive function ValueRange computes two subsets of the range constraint vector: $\Delta_f$ has range constraint vectors that lead to an over-approximation of the reachable states violating the given specification, while $\Delta_u$ has not yet found any parameter valuation that will go against the specification.

Function $\text{Extend}$ in Line 5 generates a new range constraint vector $\vec{\Delta}_i$ by extending $\vec{v}_i$. For each parameter, the lower and upper bounds of $\vec{\Delta}_i$ are obtained by subtracting and adding $(j + 1)$ times $\epsilon$ to the value according to $\vec{v}_i$, respectively. Function $\text{Valuation}$ reorganizes the solution produced by Bonmin for this new range constraint vector. Whenever a parameter valuation vector $\vec{v}$ that violates the specification is found, we first exclude $\vec{v}_i$ from the initial state set, and then classify $\vec{\Delta}_i$ to $\Delta_f$ by function $\text{Merge}$ iff $j = 0$ holds, as shown from Line 8 to Line 11. Otherwise, if function $\text{Valuation}$ returns an empty set, then $\vec{\Delta}_i$ is classified to $\Delta_u$ by function $\text{Merge}$ in Line 13. The algorithm executes the above steps for all the elements in the initial state set, before performing a recursive call. The recursive procedure terminates when either the initial state set becomes empty or the maximum recursion step $N$ is reached.

Both $\Delta_f$ and $\Delta_u$ are obtained by extending each parameter valuation in the initial state set to a range constraint vector until some parameter valuation violates the property. The range constraint vector set $\Delta_f$ may include some range vectors that should belong to $\Delta_u$, due to the introduced inaccuracy $\epsilon$, while $\Delta_u$ provides the candidate ranges for parameters that meet the given specifications. Engineers could adapt the value of $\epsilon$ to get more accurate results. However, high accuracy sometimes makes troubles by generating infeasible problems for Bonmin. For the moment, there is no general suggestion to avoid this problem.
Algorithm 1 Computing Ranges for Input Parameters

Input:
- Initial State Set: $I = \{ \vec{v}_i | i \in \mathbb{N} \}$
- Specification: $C_b$
- System’s Symbolic Representation: $G$
- Iteration Length: $N$

Output:
- Range Constraint Vector Set: $\Delta_u$
- Range Constraint Vector Set: $\Delta_f$

Local:
- Step Size: $\epsilon$
- Parameter Vector: $\vec{p}_r$
- Parameter Valuation Vector: $\vec{v}$
- Iteration Counter: $j$

1: $\Delta_u \leftarrow \emptyset$, $\Delta_f \leftarrow \emptyset$, $j \leftarrow 0$
2: procedure VALUE RANGE($I, C_b, G, N, \Delta_u, \Delta_f, j$)
3: if $I \neq \emptyset$ then
4: for all $\vec{v}_i \in I$ do
5: $\vec{\Delta}_i \leftarrow \text{EXTEND}(\vec{v}_i, \epsilon, j)$
6: $\{ \vec{v} \} \leftarrow \text{VALUATION}(\vec{\Delta}_i, C_b, G, \vec{p}_r)$
7: if $\{ \vec{v} \} \neq \emptyset$ then
8: $I \leftarrow I \setminus \vec{v}_i$
9: if $j = 0$ then
10: $\Delta_f \leftarrow \text{MERGE}(\Delta_f, \vec{\Delta}_i)$
11: end if
12: else
13: $\Delta_u \leftarrow \text{MERGE}(\Delta_u, \vec{\Delta}_i)$
14: end if
15: $j \leftarrow j + 1$
16: if $j < N$ then
17: VALUE RANGE($I, C_b, G, N, \Delta_u, \Delta_f, j$)
18: end if
19: else
20: return $(\Delta_u, \Delta_f)$
21: end if
22: end for
23: return $(\Delta_u, \Delta_f)$
24: end if
25: end procedure

4.3. Implementation details

Algorithm 1 has been implemented on top of our Averest system. We benefit from our Averest system, since it offers a constantly evolving infrastructure containing tools for compilation, analysis, synthesis, and different techniques for formal verification. It provides algorithms that translate a Hybrid Quartz program to a set of guarded actions denoted as $G$ [Sch09]. The guarded actions are the basis for the system’s symbolic representation [BS10].

Even though the algorithm has been implemented in Averest for symbolic simulation of Hybrid Quartz programs, it does not mean that the algorithm is applicable only to Hybrid Quartz programs. Performing symbolic simulation on different hybrid models, the algorithm is still usable. However, engineers have to make some adaption for the different symbolic representations obtained by the various hybrid models they choose. Since the modeling language is not a prerequisite to understand the algorithm, we do not introduce the Hybrid Quartz language further. Instead, a brief overview of the language is given in Appendix A to understand the example.

5. Experimental Evaluation

In this section, we use the Ball and Holes scenario to demonstrate the use of our constraint solver and the proposed algorithm. Since both the constraint solver and the proposed algorithm are implemented in the Averest system, we encode this scenario in Hybrid Quartz. Readers could refer to Appendix B for the complete Hybrid Quartz model for this scenario. However, only the scenario description together with the simulation conditions and task will be introduced as prerequisite information. At the end of this section, a comparison between the experimental results and the expected results is given.
5.1. Scenario Description

As shown in Fig. 1a, throwing a ball from the ground with a non-zero speed, the ball will bounce continuously according to the environment conditions, e.g. the wind and the gravity. However, there are some holes on the ground where the ball will not be able to bounce again if it falls into the hole. The safe region starts from the right hand side edge of the furthest hole to the infinite.

For convenience of analysis, the speed is divided into \( V_X \) and \( V_Y \) which stands for the component in the horizontal \( X \) and vertical \( Y \) dimension, respectively, during the whole bouncing procedure. In the horizontal \( X \) dimension, the speed component \( V_X \) is influenced by the wind, so its value changes in accordance with the wind acceleration \( a_X \in [a_{\text{min}}, a_{\text{max}}] \). The speed component in \( Y \) dimension is controlled by the gravity, which means \( a_Y \) is the constant value \( g \). The ball may lose some energy in \( Y \) dimension after hitting the ground, where the energy loss coefficient \( c \) satisfies \( c \in [0.0, 1.0] \). \( B \) and \( H \) encode the bounce trace in two dimensions that are decided by the speed components \( V_X \) and \( V_Y \), respectively. Variable \( n \) increases its value whenever the ball hits the ground. \( B \), \( H \), and \( n \) evolve with physical time \( t \), thus we use \( B(t) \), \( H(t) \), \( n(t) \) to reduce the number of variables appearing in the input formulas for the constraint solver.

5.2. Simulation Conditions and Task

In this scenario, except for \( n \in \mathbb{Z} \), all the other variables are real-valued. We would like to know whether the ball could reach the safety region by restricting the number of bounces.

Assume the initial variable values are: \( V_{X \text{ init}} = 0.0 \) and \( V_{Y \text{ init}} = 19.6 \), \( c \) is constantly equal to 0.5, and \( 0.1 \leq a_{\text{min}} \leq a_{\text{max}} \leq 5.0 \). There are three holes in total, \( S_0 = [0, 8.0] \), \( S_1 = [10, 18] \), and \( S_2 = [22.5, 24.5] \). The wind acceleration \( a_X \) is the only real parameter.

The simulation task is to check whether the ball could meet the specification that requires it to reach the safety region by bouncing at most twice.

5.3. Experimental Result

The following formula \( C_0 \) describes the following two situations that violate the given specification: the ball could not reach the safe region after bouncing twice, described by \( C_1 \); the ball falls into the hole region, described by \( C_2, C_3 \) and \( C_4 \).
\[
C_0 = C_1 \lor C_2 \lor C_3 \lor C_4 \quad \text{where} \\
C_1 = (\exists t) . (2 \leq n(t) \land B(t) \leq 24.5) \\
C_2 = (\exists t) . (n(t) \leq 2 \land H(t) \leq 0.0 \land 22.5 \leq B(t) \leq 24.5) \\
C_3 = (\exists t) . (n(t) \leq 2 \land H(t) \leq 0.0 \land 10 \leq B(t) \leq 18) \\
C_4 = (\exists t) . (n(t) \leq 2 \land H(t) \leq 0.0 \land 0 \leq B(t) \leq 8.0)
\]

Given the parameter vector \( \vec{p} = (a_X) \) and an initial state set \( I = \{ (1.25), (2.5), (3.75) \} \), Fig. 1b plots the symbolic simulation result by applying Algorithm 1 with \( \epsilon = 0.01 \). The three black circles are the three initial parameter valuations. \( \Delta_u \) consists of two elements \( \Delta_{u1} \) and \( \Delta_{u2} \), depicted by the green and blue regions respectively. The red region stands for \( \Delta_f \) that contains some parameter valuations violate the property. Two parameter valuations that violate the property are founded by this symbolic simulation procedure. The first one happened at the first iteration step for concrete value \( \vec{v}_0 = (1.25) \). The other one appeared for \( \vec{v}_1 = (2.5) \) at 24-th iteration step in which the two blue circles locate. Moreover, the iteration length has a big effect on the simulation coverage. The coverage increases generally as the increasing of iteration length. 24 iteration steps lead to a 19.6% coverage for this experiment, while 57.2% after performing 50 iteration steps.

Furthermore, formula (6) shows the returned range constraint vector set \( \Delta_u \) after 50 iteration steps in Fig. 1b. The expected one \( \Delta_i \) is supposed to contain two vectors \( \Delta_1 \) and \( \Delta_2 \) as shown by formula (7). The experimental result is consistent with the expected result.

\[
\begin{align*}
\Delta_u &= \{ \Delta_{u1}, \Delta_{u2} \} \\
\Delta_{u1} &= [3.25, 4.25] \\
\Delta_{u2} &= [2.26, 2.74] \\
\Delta_i &= \{ \Delta_1, \Delta_2 \} \\
\Delta_1 &= (2.25, 2.8125) \\
\Delta_2 &= (3.0625, +\infty)
\end{align*}
\]

6. Conclusion

We developed a prototypical constraint solver for an undecidable logic for non-linear arithmetic theories over integers and reals with quantifiers. Based on which, we proposed a symbolic simulation algorithm for hybrid systems that are specified by parameterized models using standard data types (reals, integers and booleans) and non-linear dynamics. The algorithm has been implemented, and its feasibility is proved through a successful experimental evaluation using parameterized hybrid programs.

References


A reaction step consists of reading new inputs, and executing the code starting from the active
pause
is defined by so-called micro and macro steps
Quartz is a synchronous language that is derived from the Esterel language [Ber00]. The execution of a Quartz program
A. The Hybrid Quartz Language

Quartz is a synchronous language that is derived from the Esterel language [Ber00]. The execution of a Quartz program
is defined by so-called micro and macro steps, where macro steps are defined by pause statements in the program.
A reaction step consists of reading new inputs, and executing the code starting from the active pause statements to
the next reached pause statements as the reaction of the program. Due to parallel statements \( S_1 \parallel S_2 \), more than one
pause statement may be active at a time. Each pause statement introduces a control-flow location that is given a
name by the compiler or the programmer.

The flow statement \( \text{flow } \{ S_1; \ldots; S_n \} \text{ until } (\sigma) \) can replace a pause statement and will then extend the
discrete transition by a continuous transition where the continuous variables behave according to the flow assignments
\( S_1; \ldots; S_n \) that are in the form of either \( x \leftarrow \tau \) or \( \text{drv}(x) \leftarrow \tau \) (that equate variable \( x \) or its derivation on time
\( \text{drv}(x) \) with the expression \( \tau \)). In contrast to the discrete transitions, continuous transitions require physical time
and terminate as soon as the condition \( \sigma \) becomes true.

The continuous transition of the macro step starts with the variable environment determined by the immediate
assignments as initial values. To distinguish between the ‘discrete’ value at the initial time of the continuous transition
and the value during the continuous transitions, a new operator \( \text{cont}(x) \) is introduced: \( x \) always refers to the discrete
value of a variable, whereas \( \text{cont}(x) \) refers to the (changing) value during the continuous evolution. For memorized
and event variables \( x \) and \( \text{cont}(x) \) always coincide as these variables do not change during continuous evolutions.

Due to space limitations, we cannot give an overview of the language here, and refer to [Sch09, Bau12] for full
details. Instead, we just list some of the statements used in the Appendix B and give an idea of their meaning:

- \( x = \tau \) and \( \text{next}(x) = \tau \) (immed./delayed assignments)
- \( \text{assume}(\phi), \text{assert}(\phi) \) (assumptions and assertions)
- \( \ell : \text{pause} \) (start/end of macro step)
- \( S_1; S_2 \) (sequences)
- \( S_1 \parallel S_2 \) (synchronous concurrency)
- \( \text{loop } S \) (iteration)
- \( \text{flow } \{ S_1; \ldots; S_n \} \text{ until } (\sigma) \) (flow statement)
- \( M([\text{params}]) \) (module call)

B. The Scenario Model

Most of the variable notations in Fig. 2 are the same as the ones introduced in Section 5.1. The Ball and Holes Hybrid
Quartz model consists of three parts information.

The macro part gives the static hole region information, including \( N \in \mathbb{Z} \) for the hole number, and two real-valued
arrays \( S_{\text{min}}[N] \) and \( S_{\text{max}}[N] \) for the hole locations.

The main module starts with the input \( \mathbb{R} \)-variables, including \( a_x, V_x_{\text{init}}, V_y_{\text{init}}, T_s \), and \( c \). Among them,
\( a_x \) is the same as \( a_X \) in the previous section for the instant wind acceleration that is sampled by the given period \( T_s \),
and \( a_X \) keeps unchanged until next sample period. \( V_x_{\text{init}} \) and \( V_y_{\text{init}} \) stand for the initial horizontal and vertical
speed components, and \( c \) is the energy loss efficiency.

Some local variables are declared to describe the bounce procedure: \( B \) and \( H \) encode the bounce trace of the two
dimensions that are decided by the speed components \( V_x \) and \( V_y \), respectively. The other time related variable \( T \)
works as a timer to stimulate the sampling procedure, so that both \( a \) and \( T \) should be reset after every \( T_s \) time units.
macro \( N = ?, S_{\text{min}}[N] = ?, S_{\text{max}}[N] = ? \);

module ParametricBall(\( \text{real } a_x, V_x_{\text{init}}, V_y_{\text{init}}, T_s, c \)) {
    hybrid real \( V_x, V_y, T, B, H; \) real \( a, n; \)
    // Initialization configuration
    \( V_x = V_x_{\text{init}}, V_y = V_y_{\text{init}}, a = a_x; \)
    // Ball Bounces
    loop(
        // Continuous dynamics
        flow{
            drv(B) <- cont(V_x); drv(V_x) <- a;
            drv(H) <- cont(V_y); drv(V_y) <- -9.8;
            drv(T) <- 1.0;
        } until((cont(T) >= T_s) or ((cont(H) <= 0.0) & (cont(V_y) <= 0.0)));
        if((T >= T_s) & ((cont(H) <= 0.0) & (cont(V_y) <= 0.0))){
            //Case 1: the ball hits the ground at the sample time
            next(a) = a_x; next(T) = 0.0; next(V_y) = -c*V_y; next(n) = n+1;
        }
        else{
            if((T >= T_s) & !(((cont(H) <= 0.0) & (cont(V_y) <= 0.0))){
                //Case 2: sample time, the ball does not hit the ground
                next(a) = a_x; next(T) = 0.0;
            }
            else{
                //Case 3: the ball hits the ground not at the sample time
                next(V_y) = -c*V_y; next(n) = n+1;
            }
        }
    pause;}
    satisfies // Specification
    assert EF (exists(i = 0..N-1) (S_{\text{min}}[i] <= B) & (B <= S_{\text{max}}[i]) & (H <= 0.0));}

Figure 2: The Ball and Holes Hybrid Quartz Model

Variable \( n \) increases its value whenever the ball hits the ground. Except for \( n \in \mathbb{Z} \), all the other local variables are \( \mathbb{R} \)-variables.

The bouncing procedure is represented by a \texttt{loop} statement, which contains one continuous macro step for the dynamic evolutions, and another discrete macro step for resetting either the wind acceleration at each sample time or the velocity of the ball when it hits the ground.

Checking whether the ball could avoid all holes, is equal to verify whether there exists a path in the future so that the ball may fall into the hole region. Thus, the last \texttt{satisfies} part specifies the specification ignoring the restriction of the bounce number.

For the given scenario, if the wind acceleration is constant, the time duration of each macro step is the time when the ball stays in the air between two sequential bounces. However, the wind acceleration changes its value and is sampled every \( T_s \) time units. It separates the macro step with constant wind acceleration to several sub macro steps. It is safe to eliminate the sample-related variables \( T_s \) and \( T \) to ease the difficulty of synthesis by assuming the wind acceleration is constant. By this, the regions the ball could hit on the ground are over-approximated. If the hole region has no intersection with the approximation regions, then the ball avoids all the holes.