HybridSAL Relational Abstractor: User's Guide

Ashish Tiwari

December 9, 2013

1 Introduction

HybridSal Relational Abstractor is a formal verification tool for verifying safety properties of cyber-physical (hybrid dynamical) systems. Here, we briefly describe one standard way of using the tool for performing verification. This report does *not* describe the technical approach, which can be found in other publications [2, 4, 5]. We focus on the inputs to the tool, how to run the tool, and the expected outputs of the tool here.

2 Components of the Tool

The HybridSal Relational Abstractor has three main components:

1. A `modelica2hsal` translater (optional): This component converts an XML dump generated by OpenModelica into a HybridSal model.
2. A `hsal2hasal` relational abstracter: This component takes as input a model (in HybridSal, such as the one generated by `modelica2hsal`) and creates an abstraction that is output in SAL syntax.
3. A `SAL model checker` that performs verification on the abstract SAL model.

These three components are run sequentially in that order. There are two possible outcomes of the verification effort: either the property is proved, or a counter-example is found and reported (by the last component, the SAL model checker, in the chain).

While it is not necessary to know about the existence of these three components, it is useful to know about them for making best use of the HybridSal verification technology. In particular, for maximum benefit, the human user may need to "close-the-loop" – when verification fails and the user believes the property should have been valid, the user needs to modify the model or the property by analyzing the counter-example.

3 Installing the Tool

The `modelica2hsal` translater and the `hsal2hasal` relational abstracter are available open-source and are essentially Python scripts that require some standard Python packages. For Windows platform, there is a pre-built executable available [4].

The third component, namely the `SAL model checker`, needs to be installed separately [1]. For Linux, pre-compiled executables are available and for Windows, Cygwin is required.

4 Running the Tool

An XML dump of a Modelica model can be converted into SAL using the command:

```
modelica2sal <filename.xml>
```

The executable `modelica2sal` is in the directory `HSalROOT/bin`, where `HSalROOT` is the root directory where the HybridSal tool is installed. This commands performs both the first and second steps – it first creates a HybridSal model and then creates a SAL abstraction for it. It generates a SAL model. The SAL model can be model checked as follows:

```
sal-inf-bmc -d 4 <filenameModel.sal> <propertyName>
```

Here `<filenameModel.sal>` is the file generated by the previous command above. Inside the file `<filenameModel.sal>`, there should be a property named `<propertyName>`.

There are two options for generating the property:

- The user can add the property by hand in the file `<filenameModel.sal>` that is created by `modelica2sal`.
- The property can be specified in JSON format and passed on as an argument to the `modelica2sal` tool:

  ```
  modelica2sal <filename.xml> <propertyFile.json>
  ```

Rather than using Modelica XML dump as the starting point, the user can built a HybridSal model by hand. In that case, verification is performed using the commands:

```
hsal2hasal <filenameModel.hsal>
sal-inf-bmc -d 4 <filenameModel.sal> <propertyName>
```

Properties are of the form `G(`ϕ`)`, which is read as "in every reachable state of the system, the formula ϕ is true". For example, ϕ can be $x < 5$, where x is a state variable in the model.

5 Interpreting the Results

The final output of the verification process, which is generated by the tool `sal-inf-bmc`, is either an affirmative answer (saying that the property is valid in the model), or a negative answer (saying that the property is false in the model). A negative answer is accompanied with a trace that shows the violation of the property.

If the property is not true, the user can modify the model (initial states, values of certain parameters, constraints on inputs) or the property and re-verify. The modification is done by directly manipulating the HybridSal model (in the file `<filenameModel.hsal>`).

6 Caveats

Verification of complex cyber-physical systems is a hard, and in general an undecidable, problem. Hence verification tools are limited in various ways. Verification using the HybridSal relational abstraction tool can fail due to many reasons:

Nonlinearity The tool works only on linear hybrid systems. Such systems can be nonlinear – but all nonlinearity should be in the form of "mode changes". Within each mode, the dynamics have to be linear.

State space representation The HybridSal modeling language represents the system in its state-space form. Modelica models contain plenty of non-state variables. The modelica2hsal translater tries to eliminate the non-state variables and create a HybridSal model. But this process can fail due to many reasons, which causes failure of the verification process itself.

References

[1] The SAL intermediate language, 2003. Computer Science Laboratory, SRI International, Menlo Park, CA. `http://sal.csl.sri.com/`.

[2] S. Sankaranarayanan and A. Tiwari. Relational abstractions for continuous and hybrid systems. In *Proc. 23rd Intl. Conf. on Computer Aided Verification, CAV*, volume 6806 of *LNCS*, pages 686–702. Springer, 2011.

[3] A. Tiwari. Approximate reachability for linear systems. In *Proc. 6th Intl. Workshop on Hybrid Systems: Computation and Control, HSCC 2003*, volume 2623 of *Lecture Notes in Computer Science*, pages 514–525. Springer, 2003.

[4] A. Tiwari. Hybridsal relational abstracter. In *Proc. CAV*, volume 7358 of *LNCS*, 2012. `http://www.csl.sri.com/~tiwari/relational-abstraction/`.

[5] A. Zutshi, S. Sankaranarayanan, and A. Tiwari. Timed relational abstractions for sampled data control systems. In *Proc. CAV*, volume 7358 of *LNCS*, pages 343–361, 2012.

A Detailed Example and the HybridSal Relational Abstractor Tool

Consider a simple 2-dimensional continuous system defined by

$$\begin{aligned} \frac{dx}{dt} &= -y + x \\ \frac{dy}{dt} &= -y - x \end{aligned}$$

Assume we are given the initial condition $x = 1, y = 2$ and the invariant that y is always non-negative. We wish to prove that x always remains non-negative.

This example can be encoded in HybridSAL as shown in Figure 1. The HybridSAL syntax is almost identical to the syntax of SAL [1], but for a few modifications that enable encoding of continuous dynamical systems. The key changes in HybridSAL are:

- Variables whose name ends in *dot* denote the derivative. In Figure 1, there are two state variables x, y, and their derivatives are denoted by variables named *xdot*, *ydot* respectively. These special dot variables can only be used as left-hand sides of simple definitions (equations) that appear in guarded commands. Thus, the equation $xdot' = -y + x$ denotes the differential equation $dx/dt = -y + x$.

- The guard of the guarded command that encodes the system of differential equation denotes the state invariant; that is, the system is forced to remain inside the invariant set while evolving as per the differential equations. This meaning is consistent with the usual semantics of guards in SAL. In Figure 1, the guard $y \geq 0 \wedge y' \geq 0$ says that $y \geq 0$ is the mode invariant for the only mode in the system.

The definition of contexts, modules, and properties (theorems) are exactly as in the SAL language [1].

The user creates a HybridSAL model, similar to the one shown in Figure 1, using a text editor. Following the SAL convention for naming files, the HybridSAL file containing the model in Figure 1 is stored as file `Linear1.hsal`. Thereafter, to prove the two properties contained in `Linear1.hsal`, the user executes the following commands.

`bin/hsal2hasal examples/Linear1.hsal` . This command is run from the root directory of the HybridSAL relational abstraction tool. It will create a set of files in the subdirectory `examples/`, including the file `Linear1.sal` that contains the relational abstraction of the original model. The process

```
Linear1: CONTEXT =
BEGIN

control: MODULE =
BEGIN
 LOCAL x,y:REAL
 LOCAL xdot,ydot:REAL
 INITIALIZATION
  x = 1; y = 2
 TRANSITION
 [
 y >= 0 AND y' >= 0 -->
  xdot' = -y + x ;
  ydot' = -y - x
 ]
END;

% proved using sal-inf-bmc -i -d 2 Linear1 helper
helper: LEMMA
 control |- G(0.9239 * x >= 0.3827 * y);

% proved using sal-inf-bmc -i -d 2 -l helper Linear1 correct
correct : THEOREM
 control |- G(x >= 0);
END
```

Figure 1: HybridSAL file describing a simple two-dimensional continuous dynamical system, along with two safety properties of that system.

of going from `Linear1.hsal` to `Linear1.sal` involves the following stages (that are all performed in a single run of the above command):

$$\texttt{Linear1.hsal} \longrightarrow \texttt{Linear1.hxml} \longrightarrow \texttt{Linear1.haxml} \longrightarrow \texttt{Linear1.xml} \longrightarrow \texttt{Linear1.sal}$$

The hsal2hxml converter is in the subdirectory `hybridsal2xml`. It is simply the HybridSAL parser that parses a files and outputs it in `.hxml` format. The program `bin/hsal2hasal` can take as input either a `.hsal` file or a `.hxml` file. It creates `.hasal` and `.haxml` – which is a file in *extended HybridSAL* syntax – that contains the original model as well as its relational abstraction. It is an intermediate file that is useful only for debugging purposes at the moment. In the last stage, the final `.xml` and `.sal` files are easily extracted from the `.haxml` files.

`sal-inf-bmc -i -d 2 Linear1 correct` . Once the relational abstraction has been created, it can be model checked. Note that the relational abstraction (in file `Linear1.sal`) is an *infinite* state system. Hence, we can not use finite state model checkers. We can, however, use the SAL infinite bounded model checker (`sal-inf-bmc`) and the k-induction prover (`sal-inf-bmc -i`). The k-induction prover can sometimes fail to prove a correct assertion because the assertion is not inductive. In such a case, auxiliary lemmas may be needed to complete a proof. In the running example, we need a helper lemma. Using the lemma `helper`, the property `correct` can be proved using the command:

```
sal-inf-bmc -i -d 2 -l helper Linear1 correct
```

The lemma `helper` can itself be proved using k-induction as:

```
sal-inf-bmc -i -d 2 Linear1 helper
```

This completes the discussion of the application of the HybridSAL relational abstraction tool on the running example. For more complex examples, including examples of hybrid systems, the reader is referred to the `examples/` subdirectory in the tool. We note a few points here.

Initialization The initial state of the system need not be a single point. It can be a region of the state space. For example, in Figure 1, the initialization section can be replaced by the initialization

```
INITIALIZATION
 x IN {z:REAL|0 <= z AND z <= 1};
 y IN {z:REAL|2 <= z AND z <= 3};
```

The new model can again be verified using the same set of commands given above.

Composition The model need not be a single module, and it can be a composition of modules. Modules can be purely discrete – they need not all have dynamics given by differential equations. For example, the example in the File `TGC.hsal` describes a model of the train-gate-controller in HybridSAL that is a composition of five modules. Only one of the five has continuous differential equations in the dynamics.

Apart from the syntax for writing differential equations, the HybridSAL input language supports two additional features that are not part of the SAL language [1]. These features are:

Invariant Apart from `INITIALIZATION` and `TRANSITION` blocks, each base-module can also have an `INVARIANT` block. The invariant block contains a formula that is an (assumed) invariant of the system.

In our running example, we can add the Invariant block:

```
INVARIANT y >= 0
```

in the basemodule, for example, just before/after the `INITIALIZATION` block. Then, we could replace the guard $y \geq 0 \wedge y' \geq 0$ by the new guard *True* in the (only) transition. If ϕ is declared as the invariant, then it has the effect of adding the formula $\phi \wedge \phi'$ in the guards of *all* transitions. This is performed as a preprocessing step by the HSal Relational Abstractor.

INITFORMULA Instead of `INITIALIZATION` block, a HybridSAL input file can contain a `INITFORMULA` block that has a formula as the initialization predicate.

In our running example, the initialization block shown above can be replaced by

```
INITFORMULA 0 <= x AND x <= 1 AND 2 <= y AND y <= 3
```

without changing the meaning of the HybridSAL model.

The `INITFORMULA` block is also handled during the preprocessing phase. The preprocessing phase also handles *defined constants*.

B HSal Relational Abstractor: Flags

The HybridSAL Relational Abstractor tool accepts the following options/flags.

-n, –nonlinear With this option, the tool creates a more precise relational abstraction, but it may be nonlinear. Even when the input model is a linear continuous dynamical system, the output could be a nonlinear discrete time system.

Currently, the SAL model checker can not analyze nonlinear discrete time systems. Hence, this flag is useful only if using other backend tools that can handle discrete time nonlinear systems.

By default, this option is *not* turned on, and the tool creates linear relational abstractions that can be analyzed by SAL infinite bounded model checker.

-c, –copyguard With this option, the tool explicitly handles the guard in the continuous dynamics as state invariants. Recall the HybridSAL code for the differential equations $dx/dt = -y + x, dy/dt = -y - x$.

```
 [
y >= 0 AND y' >= 0 -->
 xdot' = -y + x ;
 ydot' = -y - x
 ]
```

Here the guard $y \geq 0 \wedge y' \geq 0$ says that y should be nonnegative both *before* and *after* the transition. In other words, $y \geq 0$ is the mode invariant. We could have written the same dynamics as follows:

```
[
y >= 0 -->
 xdot' = -y + x ;
 ydot' = -y - x
]
```

The new HybridSAL file, when processed with the flag `-c`, produces the same output as the original file would produce without the `-c` flag. Thus, the `-c` flag causes copying of the guard of the continuous transitions, but with variables replaced by their prime forms.

By default, this option is *not* turned on, and the tool assumes that the input HybridSAL file already contains guards on prime variables.

-o, –opt This flag turns on some optimizations in the relational abstractor. Currently, this flag causes the tool to construct a relational abstraction that assumes that a certain amount of time is always spent in each mode. This is an unsound assumption. Hence, the output of the relational abstractor can be unsound when the `-o` flag is used.

The `-o` flag is useful if the (sound) relational abstraction is too large to be analyzable (in reasonable time) by the model checker. In that case, the user could try to create a simpler, but unsound, abstraction using the `-o` flag and model checking it.

By default, this option is *not* turned on.

-t ⟨timestep⟩, –time ⟨timestep⟩ This flag is still under development. It constructs a timed relational abstraction of the given system. This is useful in cases where all discrete mode switches are *time triggered.* The value of `timestep` should be a real number, which is interpreted as the inverse of the sampling frequency. It is important to choose the timestep carefully.

The tool is expected to include more options in the future as new extensions and features are implemented.

C Background: HSal Relational Abstractor

Hybrid dynamical systems are formal models of complex systems that have both discrete and continuous behavior. It is well-known that the problem of verifying hybrid systems for properties such as safety and stability is quite hard, both in theory and in practice. There are no automated, scalable and compositional tools and techniques for formal verification of hybrid systems.

HybridSAL is a framework for modeling and analyzing hybrid systems. HybridSAL is built as an extension of SAL (Symbolic Analysis Laboratory). SAL consists of a language and a suite of tools for modeling and analyzing discrete state transition systems. HybridSAL extends SAL by allowing specification of continuous dynamics in the form of differential equations. Thus, HybridSAL can be used to model hybrid systems. These models can be abstracted into discrete finite state transition systems using the HybridSAL abstractor. The abstracted system is output in the SAL language, and hence SAL (symbolic) model checkers can be used to model check the abstraction. While HybridSAL can be used to verify intricate hybrid system models, it is based on constructing qualitative abstractions – which can be very coarse at times. Furthermore, the HybridSAL abstractor is not compositional, and can not abstract modules independently separately without being very coarse.

To alleviate the shortcomings of the HybridSAL abstractor, we developed the concept of relational abstractions of hybrid systems. A relational abstraction transforms a given hybrid system into a purely discrete transition system by summarizing the effect of the continuous evolution using relations. The state space of system and its discrete transitions are left unchanged. However, the differential equations describing the continuous dynamics (in each mode) are replaced by a relation between the initial values of the variables and final values of the variables. The abstract discrete system is an infinite-state system that can be analyzed using standard techniques for verifying systems such as k-induction and bounded model checking.

Relational abstractions can be constructed compositionally by abstracting each mode separately. Abstraction and compositionality are crucial for achieving scalability of verification. We have also developed techniques for constructing good quality relational abstractions. The details are technical and can be found in papers [2, 3]. The HybridSAL verification framework has been extended by an implementation of relational abstraction.