HybridSAL: Modeling and Abstracting Hybrid Systems

Ashish Tiwari
SRI International
Menlo Park, CA

June 24, 2003

Abstract

This document describes the HybridSAL modeling language for hybrid systems and the HybridAbstractor tool for creating sound discrete approximations for HybridSAL models.

The SAL intermediate language is used to describe *discrete* transition systems [1] and forms the core of the verification tool suite being developed at SRI [3]. HybridSAL is an extension of the SAL intermediate language [4] to also allow for specification of *continuous* dynamics. Thus, HybridSAL is a modeling language for hybrid systems. Models in HybridSAL can be abstracted into finite-state discrete transition systems completely automatically using the HybridAbstractor [7]. This document describes the HybridSAL language and the HybridAbstractor tool interface. While describing the HybridSAL language, the focus will be on the differences, or extensions, over the SAL intermediate language. The abstract syntax and the expression language of SAL are described in detail in [4]. However, a familiarity with SAL, though helpful, is not a prerequisite. The algorithm underlying the HybridAbstractor tool is described in [7] and some case studies are presented in [2, 6]. This document will only describe in detail the various parameters of the abstractor that can be set by the user depending on the particular application.

We will use a running example of a simple ThermoStat for illustrating the various language features. The model consists of two modes, defined by heater being "on" or "off", and one continuous variable, the room temperature x. When the heater is "off", the temperature t decreases, say according to the differential equation $\dot{x} = -k*x$, where k is a parameter that could depend on the size of the room, its insulation, etc. When the heater is "on", the temperature x increases, say according to the differential equation $\dot{x} = M - k*x$, where M is another parameter that would depend on the heating capacity of the heater. The discrete behavior of the ThermoStat systems is given by transitions from heater "on" mode to heater "off" mode and vice-versa. Let us assume that heater is switched off whenever the temperature rises above 80 and it is switched "on" whenever the temperature falls below 70. Graphically, this hybrid system is often represented as in Figure 1.

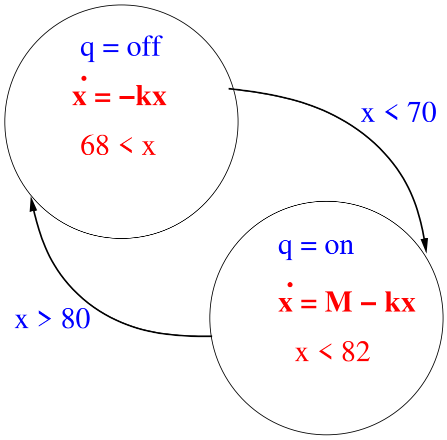

Figure 1: Graphical representation of the ThermoStat hybrid automaton

1 Contexts

At the top level, a HybridSAL specification consists of a CONTEXT declaration. A context consists of models of the system and its subsystems (if any), type and constant declarations that the model uses, and properties of the model. By convention, there is a one-to-one mapping between HybridSAL files and contexts, that is, each HybridSAL file specifies exactly one context, with the same name as the filename.

The abstract syntax of the HybridSAL CONTEXT declaration is identical to that of SAL. The ThermoStat example can be modeled using the following top-level CONTEXT declaration (assumed to be contained in the file "SimpleThermo.sal"):

```
SimpleThermo:  CONTEXT =
BEGIN
   ⟨ module declaration ⟩
   ⟨ property declaration ⟩
END
```

A context can contain multiple module declarations, one for each component of the hybrid system. It can also contain multiple property declarations. In the simple ThermoStat example, we will have one module and one property, see Figure 2 for the complete specification of the ThermoStat in HybridSAL.

Enumeration types are also declared in the context. These types can then be used in the module declarations. An example of enumeration types in a HybridSAL specification can be found in Appendix A.

2 Module

A module represents a self-contained subsystem of the hybrid system. A module is either a *base module* or it is a *composition* of two or more other modules. A *base module* declaration consists of its name, parameters, and body. In the case of the ThermoStat, the base module describing the model can be declared as follows:

```
control[M, k:  REAL] : MODULE =
BEGIN
  ⟨ variable declarations ⟩
  ⟨ invariant declaration ⟩
  ⟨ initialization ⟩
  ⟨ definitions ⟩
  ⟨ transitions ⟩
END
```

The parameters M and k are declared to be real. The dynamics of the temperature of the room depend on these two parameters as discussed above. The string "control" is the name of the module and will be used when talking about properties of this module. The body of a base module consists of five parts: variable declarations, invariant declaration, initialization, definitions, and transition declarations.

Variable declarations.

Variables in HybridSAL can be declared to be of type `REAL`, `BOOLEAN`, or some finite enumeration type. Additionally, each variable is also classified as either `INPUT`, `OUTPUT`, or `LOCAL`. The `INPUT` and `OUTPUT` variables define the interface of the module to other modules (for purposes of composition).

If we model the heater "on" and "off" modes using one boolean variable q so that q is `TRUE` if and only if heater is "on", then the base module representing the ThermoStat would have the following two variable declarations:

```
LOCAL q :  BOOLEAN
OUTPUT x :  REAL
```

The temperature x is declared to be an `OUTPUT` variable because we want it to be visible outside. In particular, the correctness property will be stated in terms of bounds on the possible values of x.

The current version of HybridSAL also requires that a dummy dotted variable be declared for every continuous variable. In particular, we will need to also add the following declaration in the ThermoStat base module:

```
LOCAL xdot :  REAL
```

Invariant.

The invariant declaration contains a boolean formula which is interpreted as the (global) invariant of the system. Typically, constraints on the values of the parameters and variables can be stated here. Semantically, the evolution (traces) of the system are constrained to always lie inside the set specified by the invariant formula.

If M and k are the two parameters of the ThermoStat model, then we have the following constraints:

```
INVARIANT
     k > 0 AND M > 80*k
```

The value of the temperature x is not constrained here, but in principle, values of (state) variables can be constrained via invariants.

Initialization.

There are two possible ways to specify the initial configuration of the base module. Precise valuations for variables can be given in an `INITIALIZATION` section. More generally, however, a set of initial configurations can be specified using a formula to constrain the values of variables. This is specified in the `INITFORMULA` section.

For example, if the initial temperature of the room is 75 and the heater is "on", then this can be specified as follows:

```
INITIALIZATION
    x = 75; q = TRUE
```

However, if initially the temperature can be anywhere between 72 and 78 with the heater being "on", then this set of initial configurations can only be specified using the `INITFORMULA` declaration.

```
INITFORMULA
    72 <= x AND x <= 78 AND q = TRUE
```

Definitions.

The dynamics of state variables is specified in one of two ways: either it can be explicitly specified in the `TRANSITION` section (this is discussed below), or it can be specified implicitly by defining the variable in terms of other variables (whose dynamics are explicitly specified).

In the ThermoStat example, suppose an external observer is interested in knowing whether the temperature of the room is above or below 75. If the variable x is declared as an `OUTPUT` variable, then the external observer can read its value. But we can also do this in a different way. We can declare x to be `LOCAL`, but declare a new boolean output variable:

```
OUTPUT hot?  :  BOOLEAN
```

and define its value using the following:

```
DEFINITION
   hot?  = (x > 75)
```

Such definitions are useful if the predicate $x > 75$ is used often and particularly if it is used outside in other modules. This helps the abstractor tool to compositionally abstract. It explicitly says that the outside world is only interested in whether $x > 75$, and not really on the exact value of x.

Transition.

The transition section of a base module specifies both the discrete and continuous dynamics. Again there are two distinct formats for specification of the dynamics: (i) Regular hybrid automata representation, or (ii) Inside-out hybrid automata representation.

- *Regular hybrid automata representation.* The discrete mode switches are specified using the usual SAL guarded transitions. For example, the two discrete transitions for the ThermoStat are specified as:

  ```
  q = TRUE AND x >= 80 -->
           q' = FALSE
  []
  q = FALSE AND x <= 70 -->
           q' = TRUE
  ```

  The formula before the arrow is the "guard" and the assignment(s) to the right of the arrow are the actions that are taken when the guard is true. Thus, the first transition above states that whenever q is $TRUE$ and x is at least 80, the new value of the variable q (represented by q') is made equal to $FALSE$. Note that the transitions are not "eager", that is, they can be taken whenever the guard is true, but not necessarily at the first instance when the guard becomes true. We use *state invariants* to force the discrete transitions to be taken *before it is too late.*

  State invariants are formulas, one assigned to each discrete mode, which restrict the valuations of variables when inside a mode. Thus, the discrete transitions have to be taken (to exit a mode) before the state invariant of that mode is falsified. We attach the state invariants in the guards of transitions representing continuous dynamics (as discussed below).

  The continuous dynamics are specified using the same syntax as discrete transitions. There is one "continuous transition" for each mode. The guard of the "continuous transition" identifies the mode and the state invariant. The right-hand side consists of assignments to the special "dotted" variables (as oppoed to assignments to regular state variables (q and x) in the discrete transition case).

In the ThermoStat example, we have two modes and the continuous dynamics in these two modes are specified using the following syntax:

```
q = TRUE AND x < 82 -->
         xdot' = (M - k*x)
[]
q = FALSE AND x > 68 -->
         xdot' = -k*x
```

In the first transition, the part $q = TRUE$ of the guard identifies the mode and the rest of the guard, $x < 82$, identifies the state invariant. The right-hand side specifies the continuous dynamics, $\dot{x} = M - k * x$.

Thus, the `TRANSITION` section of the base module representing the ThermoStat contains these four guarded transitions, see the full specification later. Note that this style of representation closely mimics the hybrid automata style specification of hybrid systems, and hence we have called it the regular hybrid automata representation.

- *Inside-out representation.* An alternative to the above specification style is the "inside-out" representation, where the continuous dynamics are specified just once, using parameters. The discrete component is captured via conditional assignments to the parameters. This style of specification is useful for *switched hybrid systems*, and especially for hybrid system models of genetic regulatory pathways. It avoids the enumeration of modes.

  The dynamics of the ThermoStat example can be specified using the following generic formula:

$$\dot{x} = L - k * x$$

  The value of L switches between 0 and M, depending on some boolean conditions. Thus, an inside-out representation of dynamics is achieved by combining a finite number of conditional assignments to certain state variables, with exactly one continuous transition specified using the same syntax as above. The ThermoStat example, in an inside-out representation looks like the following:

```
TRANSITION
    L' = IF q' = TRUE THEN M ELSE 0 ENDIF;
    q' = IF x <= 70 THEN TRUE
         ELSIF x >= 80 THEN FALSE
         ELSE q ENDIF;
    [
    TRUE --> xdot' = L - k*x
    ]
```

  Note, however, that the semantics of the "inside-out" representation above is slightly different from the semantics of the hybrid automata specified before. In particular, the switch from heater "on" to heater 'off" (and vice-versa) is made *as soon as* the temperature hits the 70 and 80 threshold. In the regular hybrid automata representation, these switches could be made non-deterministically in a temperature range.

2.1 Composition

Base modules can be composed together to create larger hybrid system models using either a synchronous `[]` or an asynchronous composition operator. Input variables of the component module with the same name as an output variable of another module match up in the composition. The semantics of the composition is given in terms of the semantics of its components. Assume the semantics of the base modules are given as traces constructed from the union of discrete transitions and continuous evolutions for arbitrary time intervals. In a synchronously composed module, the component modules simultaneously make transitions: each module takes a discrete step, or all the component modules let time elapse (that is, go through continuous evolution). In an asynchronously composed module, only one component module is allowed to make a transition, either discrete or continuous.

3 Properties

Correctness properties about modules can be specified by using LTL operators and state variables of modules. For example, the safety property that the temperature of the room stays between 70 and 80 is stated as:

```
correct:THEOREM
        control[1,100] |- G( 70 <= x AND x <= 80 );
```

The string "correct" is the name of the property, which states that the formula `G( 70 <= x AND x <= 80 )` is true of the module `control[1, 100]`. The module `control[1, 100]` is the module obtained by instantiating the module `module` defined earlier using $k = 1$ and $M = 100$. Currently, the type of symbol G needs to be explicitly specified for the typechecker.

```
G( ss:[ control[1,100].STATE -> BOOLEAN ] ) :
      [ control[1,100].STATE -> BOOLEAN ];
```

The exact instantiation, $k = 1$ and $M = 100$, is *not* used by the Hybrid-Abstractor while constructing the abstraction; and hence the abstractor can be used to verify the above correctness property for the whole family of `control` modules.

Putting all the pieces together, we get the full HybridSal model of the ThermoStat as shown in Figure 2. We illustrate the inside-out representation on an example from the genetic regulatory networks. Specifically, we present the four-cell model of the Delta-Notch lateral inhibition mechanism [2] in Figure 3.

```
%% HybridSAL model of the ThermoStat
SimpleThermo:CONTEXT =
BEGIN
control[k, M: REAL] : MODULE =
   BEGIN
   LOCAL q :  BOOLEAN
   LOCAL x :  REAL
   LOCAL xdot :  REAL
   INVARIANT
         k > 0 AND M > 80*k
   INITFORMULA
         70 <= x AND x <= 80 AND q = TRUE
   TRANSITION
         [
         q = TRUE AND x >= 80 -->
             q' = FALSE
         []
         q = FALSE AND x <= 70 -->
             q' = TRUE
         []
         q = TRUE AND x < 82 -->
             xdot' = (M - k*x)
         []
         q = FALSE AND x > 68 -->
             xdot' = 0 - k*x
         ]
   END;
   G( ss:[ control[1,100].STATE -> BOOLEAN ] ) :
         [ control[1,100].STATE -> BOOLEAN ];
   correct:  THEOREM
         control[1,100] |- G( 70 <= x AND x <= 80 );
END
```

Figure 2: HybridSAL model of the ThermoStat

```
%% Delta-Notch Signaling Hybrid Automaton
%% Number of cells:  2x2
%% Number of state variables:  8
%% Number of discrete states:  256
four_cell_new :  CONTEXT =
BEGIN
system[hD, hN, RD, RN, lD, lN : REAL] : MODULE =
BEGIN
   GLOBAL x1, x2, x3, x4, x5, x6, x7, x8 :  REAL
   LOCAL x1dot, x2dot, x3dot, x4dot, x5dot, x6dot, x7dot, x8dot :  REAL
   LOCAL x1D, x2D, x3D, x4D, x5D, x6D, x7D, x8D: REAL
   INVARIANT
        x1 >= 0 AND lD*x1 <= RD AND x2 >= 0 AND lN*x2 <= RN AND
        x3 >= 0 AND lD*x3 <= RD AND x4 >= 0 AND lN*x4 <= RN AND
        x5 >= 0 AND lD*x5 <= RD AND x6 >= 0 AND lN*x6 <= RN AND
        x7 >= 0 AND lD*x7 <= RD AND x8 >= 0 AND lN*x8 <= RN AND
        lN* hD > -RN AND hD < 0 AND hN > 0 AND lD*hN < RD AND
        RN > 0 AND RD > 0 AND lN > 0 AND lD > 0
   INITFORMULA
        -x2 = hD AND x3+x5 < hN AND -x4 < hD AND x1+x5+x7 = hN AND
        -x6 < hD AND x1+x3+x7 = hN AND -x8 = hD
   TRANSITION
        x1D = IF (-x2 < hD) THEN 0 ELSE -RD ENDIF;
        x2D = IF (x3+x5 < hN) THEN 0 ELSE -RN ENDIF;
        x3D = IF (-x4 < hD) THEN 0 ELSE -RD ENDIF;
        x4D = IF (x1+x5+x7 < hN) THEN 0 ELSE -RN ENDIF;
        x5D = IF (-x6 < hD) THEN 0 ELSE -RD ENDIF;
        x6D = IF (x1+x3+x7 < hN) THEN 0 ELSE -RN ENDIF;
        x7D = IF (-x8 < hD) THEN 0 ELSE -RD ENDIF;
        x8D = IF (x3+x5 < hN) THEN 0 ELSE -RN ENDIF;
        [ TRUE --> %% Parametric Continuous Dynamics
                x1dot' = x1D + lD*x1;
                x2dot' = x2D + lN*x2;
                x3dot' = x3D + lD*x3;
                x4dot' = x4D + lN*x4;
                x5dot' = x5D + lD*x5;
                x6dot' = x6D + lN*x6;
                x7dot' = x7D + lD*x7;
                x8dot' = x8D + lN*x8
        ]
   END;
STATES: TYPE = [ system[0,0,0,0,0,0].STATE -> BOOLEAN ];
G( ss:  STATES) : STATES;
reach:  THEOREM
   system[0,0,0,0,0,0] |- G(x1 >= 0);
END
```

Figure 3: Four cell model of the Delta-Notch lateral inhibition network

4 Hybrid Abstractor

The HybridAbstractor is an automatic tool for creating discrete finite-state sound abstractions for hybrid system models specified in the HybridSAL language described above. The result of the abstraction is a `SAL` file which contains the discrete abstraction. Other `SALENV` tools can be used to analyze this `SAL` file.

The algorithm used by the abstraction tool is described elsewhere [7]. In this document, we will detail the specific parameters that can be set while running the HybridAbstractor and the default values that the abstractor picks for them. The abstractor is run through the HybridSAL Emacs interface, using the following command line:

```
(abstract context-name module-name :option1 val1 :option2 val2 ...)
```

The context *context-name* is assumed to contain the module *module-name* which is to be abstracted. The various keywords for the options are as follows:

- *property.* A property can be specified so that the abstractor constructs an abstraction with the goal of proving that property. The abstractor, in this case, makes sure that the polynomial expressions in the property are used to partition the state space. The name of the property is the value in this case. For example,

  ```
  (abstract 'SimpleThermo 'control :property 'correct)
  ```

  will abstract the ThermoStat model with respect to the property "correct".

- *depth.* The value of depth can be set to any non-negative number. The default value is 2. For most examples, this suffices, but for large examples, smaller values of depth should be used (unless time and space are not serious constraints). This option limits the depth of derivatives the abstractor considers. In the default case, the abstractor uses the first and second derivatives for partitioning the state space.

- *maxdegree.* The value of this can be set to any non-negative number. The default value is 2. It is a limit of the degree of polynomials that are considered by the abstractor for partitioning the state space. If the degree of a derivative of some polynomial of interest is higher than the specified value, then it is ignored.

- *dlevel.* The value of this can be set to any non-negative number. The default value is 4. It sets the debug (verbose) level. Higher number means less verbose.

- *pols.* The value of this option can be set to any list of strings, where each string should parse as a polynomial expression over the state variables of the module that is being abstracted. The abstractor uses these polynomials as seed polynomials to construct the abstraction. The default value is nil.

- *more?*. This value can be set to t (default) or *nil*. It indicates if the abstractor should use polynomials in the guards as seed polynomials.

The hybrid abstractor creates a finite-state discrete transition system and outputs it in the SAL intermediate language [4]. This abstract SAL specification can then be analyzed using tools such as model-checkers from the SALenv tool suite [3].

5 Caveats

We enumerate some of the limitations of the HybridAbstractor tool in this section. First, the tool can only handle specification that are specified using polynomials. Thus, trignometric functions and logarithmic functions are not allowed in the specification. Second, good abstractions result if the disjunction (OR) operator is not used in the guards and conditional assignments. Sometimes, the initialization sections of the generated abstract SAL files might need to be modified by hand for salenv tools to work on them.

The hybrid abstraction tool uses a sound, but incomplete, (decision) procedure for the quantifier-free theory of reals. It uses this decision procedure for computing the feasible states, and abstracting the discrete and continuous dynamics of the hybrid system. The decision procedure produces *proof objects* or *witnesses*. These are used for the feasibility computation. The time taken by the abstractor depends crucially on the size of the abstract discrete state space. The feasibility check phase takes a good proportion of the time. The decision procedure is described in detail in [5].

A Automatic transmission and powertrain

To illustrate some of the features, such as enumeration types and composition, that are not covered in the main text of this document, we give an outline of a nontrivial example of a model of an automatic transmission controller along car engine dynamics. This model was prepared by Aloncrit Chutinan and Ken Butts for a different project.

```
%% ------------------------------------------------------------------------
%% This is the Simplified Closed Loop Model prepared by Aloncrit Chutinan
%% and Ken Butts [Apr 30, 2002 Report].
%% ------------------------------------------------------------------------
SCLM: CONTEXT =
BEGIN
   MODE : TYPE = { first, torque12, torque21, inertia12, inertia21, second };
   CMODE : TYPE = { first, transition12, second, transition21 };
```

```
GEAR : TYPE = { one, two };

%% ---------------------------------------------------------------
%% Plant:  Transmission + Engine + Vehicle.
%% ---------------------------------------------------------------
FullPlantModel :  MODULE =
BEGIN
   LOCAL wt, wcr, Ts, pc2, ...  :  REAL
   INPUT grade, tps :  REAL
   OUTPUT v:  REAL
   INVARIANT ...
   DEFINITION
       wtgewsi?  = (mode = second AND wt >= R2/R1 * wt) OR ...
   TRANSITION
       switch = IF (mode = inertia12 OR mode = inertia21) THEN 1
                ELSE 0 ENDIF;
       ....
       [ TRUE -->
               Tsdot' = Ks * ( Rd * wcr - 1/Hf * v - Ts/1000 );
               ...
       ]
END;

PlantModeSelector:  MODULE =
BEGIN
   OUTPUT mode :  MODE
   INPUT Tc2gt0?, ...  :  BOOLEAN
   INITIALIZATION mode = first
   TRANSITION
       [
       mode = first AND Tc2gt0?  --> mode' = torque12
       []
       ...
       ]
END;

ShiftScheduler:  MODULE =
BEGIN
   INPUT v, tps :  REAL
   OUTPUT to_gear :  GEAR
   INITIALIZATION state = first; to_gear = one
   TRANSITION
       [
       state = first AND ((tps <= 30 AND v > 20) OR ...  -->
               state' = transition12
       []
```

```
          ...
          ]
    END;

    ClosedLoopModel:  MODULE =
       ShiftScheduler [] PlantModeSelector [] FullPlantModel;
END
```

References

[1] S. Bensalem, et.al. An overview of SAL. In B.L. De Vito, editor, *Langley Workshop on Formal Methods, LFMW 2000*, 2000.

[2] R. Ghosh, A. Tiwari, and C. Tomlin. Automated symbolic reachability analysis with application to delta-notch signaling automata. In O. Maler and A. Pnuelu, editors, *Hybrid Systems: Computation and Control HSCC*, volume 2623 of *LNCS*, pages 233–248. Springer, April 2003.

[3] SALenv: A SAL state space exploration toolkit, 2003. Computer Science Laboratory, SRI International, Menlo Park, CA. `http://sal.csl.sri.com/salenv.html`.

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

[5] A. Tiwari. Abstraction based theorem proving: An example from the theory of reals. In *PDPAR: Pragmatics of Decision Procedures in Automated Reasoning 2003*, 2003. To appear.

[6] A. Tiwari. Approximate reachability for linear systems. In O. Maler and A. Pnuelu, editors, *Hybrid Systems: Computation and Control HSCC*, volume 2623 of *LNCS*, pages 514–525. Springer, April 2003.

[7] A. Tiwari and G. Khanna. Series of abstractions for hybrid automata. In C. J. Tomlin and M. R. Greenstreet, editors, *Hybrid Systems: Computation and Control, HSCC 2002*, volume 2289 of *LNCS*, pages 465–478, 2002.