

TOWARDS AUTOMATIC GENERATION OF MODEL CHECKABLE CODE FROM MODELICA

Håkan Lundvall, Peter Bunus and Peter Fritzson
Department of Computer and Information Science,
Linköping University, Sweden

Abstract

Using model components in complex system modeling is sometimes difficult because many semantic properties that should be obeyed during the design are not formalized in the modeling language. There exist rules that users of the components should follow in order to create semantically, mathematically, and physically correct models. Program verification aims at proving that programs meet certain specifications, i.e. that the actual program behavior fulfils certain specified properties. Model checking is a specific approach to verification of temporal properties of reactive and concurrent systems. Verification is usually carried out by using model checking algorithms to demonstrate the satisfiability of certain properties formalized as logical formulae over the model of the system. The model checking approach has proven successful for models based on finite-state automata and is based on state space inspection. To realize the full potential of the simulated and modeled systems with Modelica it is important to verify important properties of models in order to ensure that they meet the required criteria. In this paper we describe an algorithm that translates a non-trivial subset of Modelica to the model checking language of HyTech.

Keywords: Modelica, Model checking

Nomenclature

In the examples the following variables and symbols are used:

alpha Time delay parameter in the railroad example
app,exit Events indicating approach and exit of the train.
clk1, clk2 Discrete variables storing time-stamps.
delay Stopwatch variable with time derivative 0 or 1.
dy Time derivative of y.

lower Boolean value that when becoming true tells the gate to close.
open Boolean value representing the state of the valve.
raise Boolean value that when becoming true tells the gate to open.
t Time variable.
v Velocity of the train.
s1, ..., s4 States of the hybrid automata.
x Distance from the train to the gate.
y Water level.

Introduction

Modelica (Fritzson 2003 [1]) is an object oriented modeling language capable of describing heterogeneous physical system. To take full advantage of the modeling and simulation capabilities of Modelica it is desirable to be able to formally verify important properties of the model. In order to do that the discrete algebraic equations that the Modelica model results in must be translated into a form that a tool capable of performing automated program verification can use. The hybrid automata formalism is such a form where tools like HyTech (Henzinger et al. 1997 [2]) and CheckMate (Kapinski and Krogh 2002 [3]) can perform program verification.

Model Checking

Model checking (Clarke et al. 1999 [4]) is a technique to perform program verification over finite state concurrent systems in a fully automated way. It is done by exploring the state space of the system. Validation is carried out against a formal specification consisting of a set of formulas over the model of the system. These formulas usually express safety properties that should be fulfilled at all times or illegal states that the system should be proven never to enter.

The two main advantages compared to other program verification techniques is that it can be fully automated and that it produces a sequence of steps in the specification leading to the state that does not satisfy the specification. This makes model checking suitable for integration in automated debugging and verification tools.

The specification consists of a temporal logic formula expressing the desired properties of the system. The notion of the Kripke structure is used to model the behavior of the system. A Kripke structure consists of a set of states and transitions between states. Each state also has a valuation for a set of properties that are either true or false in the state. A transition from one state to another represents a step in time that makes a distinguishable difference of the state of the system. A path in a kripke structure represents a computation of the system.

Many systems lead to very large state spaces. In order to be able to handle this, symbolic algorithms can be used. This is especially

important in model checking of hybrid systems since real valued variables are present and the value ranges of these variables need to be represented symbolically.

A temporal logic often used in model checking is called Computational Tree Logic (CTL). CTL formulas describe properties of computation trees. A computation tree is formed by selecting one of the states in the kripke structure to be the initial state. The rest of the infinite tree is then formed by unwinding the kripke structure from the initial state.

CTL formulas are composed of path quantifiers and temporal operators. Path quantifiers express properties of the branching structure of the tree. The two path quantifiers in CTL are **A** and **E**, representing “for all paths” and “for some path”, respectively. These are used at a state to express that some property holds for all or some of the paths starting at that state. The temporal operators express properties of a path through the tree. The five basic operators are presented in Table 1.

Symb.	Phrase	Explanation
X	Next time	Requires a property to hold in the second state of the path.
F	Eventually	Requires a property to hold at some state along the path.
G	Always	Requires a property to hold at every state of the path.
U	Until	This operator combines two properties and requires that there is a state on the path where the second property holds and that at every prior to that state, the first property holds.
R	Release	This operator also combines two properties and requires the second property to hold at every state up to and including the first state at which the first property holds.

Table 1. Basic operators of CTL.

CTL formulas are divided in two categories; state formulas and path formulas. State formulas hold at a specific state in the Kripke structure. Path

formulas hold along a specific path in the kripke structure.

The syntax of a formula in CTL follows the following rules:

All atomic propositions are state formulas. If f and g are state formulas then $\neg f, f \vee g, f \wedge g$ are state formulas. If f is a path formula then $\mathbf{E} f$ and $\mathbf{A} f$ are state formulas. If f is a state formula then f is also a path formula. If f and g are path formulas then $\neg f, f \vee g, f \wedge g, \mathbf{X} f, \mathbf{F} f, \mathbf{G} f, f \mathbf{U} g, f \mathbf{R} g$ are path formulas.

An example of a CTL formula that holds if when a request occurs then it will eventually be acknowledged: $\mathbf{AG}(Req \rightarrow \mathbf{AF} Ack)$.

Hybrid Automata

Existing model checking tools for hybrid systems relies on the formalism of hybrid automata, which is an augmented form of finite automata where a finite set of continuous variables are allowed.

The hybrid automaton is formally defined as follows:

Variables: A finite set $X = \{x_1, x_2, \dots, x_n\}$ of real-valued variables. E.g. y in the Figure 2.

Control modes: A finite set V of control modes. These correspond to states in a finite automaton.

Flow conditions: A labeling function $flow$ that assigns a flow condition to each control mode $v \in V$. The flow condition $flow(v)$ is a predicate over the variables in $X \cup \dot{X}$. While an automaton is in a control mode v the variables in X evolve along a curve such that at all point along the curve the values of the variables and their first derivative satisfy the flow condition. In Figure 2 the equations involving \dot{y} represent the flow conditions.

Invariant condition: A labeling function inv that assigns an invariant condition to each control mode v . The invariant condition $inv(v)$ is a predicate over the variables in X . While a hybrid automaton is in a control mode v , the variables in X must satisfy the invariant condition $inv(v)$. Represented in Figure 2 by the inequalities inside the ellipses.

Initial conditions: A labeling function $init$ that assigns initial conditions to each control mode v . The initial condition $init(v)$ is a predicate over the

variables in X . The control of a hybrid automaton may start in the control mode v when the initial condition $init(v)$ is true.

Control switches: A finite multiset E of control switches. Each control switch is a directed edge between a source mode $v \in V$ and a target mode $v' \in V$. Control switches are denoted by arrows in Figure 2.

Jump conditions: A labeling function $jump$ that assigns a jump condition to each control switch in E . The jump condition $jump(e)$ is a predicate over the variables in $X \cup X'$. The symbols X refer to the values of the variables before the control switch and the symbols in X' refer to the values of the variables after the control switch. The label $\text{delay}'=0$ in Figure 2 denote a jump conditions stating that the value of delay is zero after the control switch.

Events: A finite set Σ of events and labeling function syn that assigns an event in Σ to each control switch in E .

This definition was taken from [2].

In order to be able to perform automatic analysis on hybrid systems certain restrictions have to be put on them. One such restricted class of the hybrid automaton is the linear hybrid automaton. In a linear hybrid automaton the flow conditions are predicates over the derivatives only, so that the derivative of a variable cannot be a function of any variable in X . Further more, the flow conditions, the invariant conditions and the initial conditions are convex linear predicates and for every control switch the jump condition is a convex linear predicate.

HyTech

HyTech is an automatic tool for the analysis of embedded systems. We decided to use HyTech as a target platform for our translator since it allows the symbolic verification and algorithmic analysis of hybrid dynamic systems. However, HyTech can only model linear hybrid automata, which is limiting for most of the simulation models expressed in Modelica. In order to overcome this limitation, we intend to extend our translator to generate code for HyTech+ (Henzinger et al. 2000 [5]) and CheckMate. HyTech+ and CheckMate are both capable of verifying hybrid systems with general continuous expressed with linear and non-

linear differential equations. The input to the HyTech tool consists of a definition of the automaton to check and an analysis section where the specification is represented. It is possible to enter more than one automaton, in which case the HyTech tool transforms them into a single automaton as its initial step. In this step *synclabs*, synchronization labels in HyTech, can be used to synchronize control switches in different automata. Two control switches marked with the same synclab always triggers simultaneously.

Translator algorithm

Modelica is based on the mathematical formalism of DAE (Differential Algebraic Equations) while HyTech uses the linear hybrid automata formalism, therefore we must find a way to isolate the discrete control modes of the model in order to build an automaton that is equivalent to the original model. Since HyTech only handles linear hybrid automata only Modelica models with state variables of constant derivatives can be translated. In models where both constant and variable derivatives exist it is possible to use program slicing (Hatcliff et al. 2000 [6]) to isolate the linear part if there are no dependencies from the linear variables to nonlinear variables.

We start the translation by creating a startup mode and then create transitions to new modes for each Modelica when-equation that can be triggered from that mode. For each new mode the procedure is then repeated according to the stepwise description of the algorithm below.

The steps of the algorithm

Step one: Find all state variables (y in the water tank example below) and add them as continuous variables to the HyTech model. Also add the clock variable τ . In Modelica there is a built in variable `time present`.

Step two: Find all discrete variables that are used together with continuous variables in when-equations (`open` in the water tank example). The discrete variables that do not appear together with continuous variables are not added, but they may appear as synclabs later.

Step three: Find the start value of all state variables and discrete variables and generate a starting control mode in the automaton. In the

water tank example we have `open=true` and `y=1` as the initial condition.

Step four: Find all when-equations that can be triggered from this control mode. When-equations with conditions that are already true or cannot become true in the given mode are discarded; all other when-equations result in corresponding transitions in the hybrid automaton. In the water tank example we have `delay(y,2) >= 10` and `open`. The other clause of the condition can never be triggered in this mode since dy is positive.

Step five: The new transitions result in new values for the variables and the derivatives. If there is an already added control mode that matches all the values, then make the transition point to that control mode; otherwise create a new mode in the automaton. For all added modes start over from step four.

When-equations that only depend on discrete variables trigger only as a result of another mode switch, since discrete variables only change at events. If the when-equation that changes the discrete value resides in the same component then the transitions are merged into a single transition. If the source event resides in a different component a synclab is generated so that the mode switches in the two resulting automata occur simultaneously. If there is no source event that can trigger the transition, then it is omitted.

When a transition that has a guard involving continuous variables is added to the automaton, a corresponding inequality has to be added to the invariant condition of the control mode, otherwise the automaton would have the possibility to stay in the source control mode and not follow the transition.

In Modelica there is a `delay` operator that delays a signal a specified amount of time. In order to translate this into the hybrid automaton an extra control mode is inserted in transitions involving a delayed variable. This extra control mode is a copy of the source control mode of the transition where the delayed transition is replaced by a transition depending on a clock variable. See the WaterTank example below for details.

Example 1: WaterTank

This example is taken from the HyTech system user's guide. Consider a water tank that is leaking water at a constant rate. When the water level falls

below five, a sensor signals a valve to open, which results in the tank being filled at a constant rate. However, the signal is delayed for two seconds before the valve reacts on it.

When the water level reaches ten the valve is signaled to close, again with a two second delay. The Modelica code for such a system is shown in Figure 1.

```

model WaterTank
  Boolean open(start=true);
  Real y(start=1);
  equation
  when delay(y, 2) >= 10 and open
  or delay(y, 2) <= 5 and not open
  then
    open = if pre(open) then
      false else true;
  end when;

  der(y) = if open then 1 else -2;
end WaterTank;

```

Figure 1. WaterTank in Modelica.

The proposed algorithm would result in the automaton shown in Figure 2. In the figure each control mode is denoted by an ellipse and the arrows between them represent control switches. The slopes of the variables and the invariant regions are presented inside the ellipse. Each control switch is labeled by jump conditions, e.g., “ $y \geq 10$ ” and “ $\text{delay}'=0$ ”, which state that in order for that control switch to take place the value of y before the switch must be greater than 10 and the value of delay after the switch is 0.

Running HyTech on the resulting automaton shows us that the water level is kept between 1 and 12 at all times.

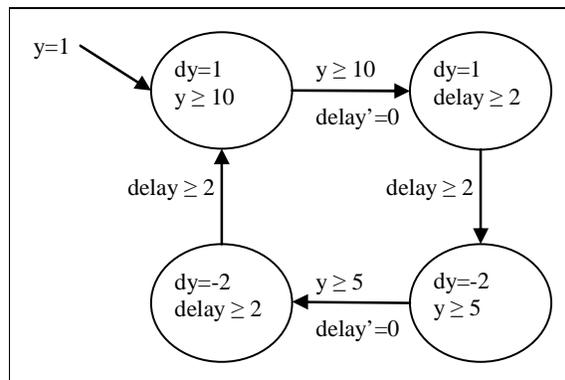


Figure 2. WaterTank automaton.

Example 2: Railroad crossing

This example, also taken from the HyTech user’s guide demonstrates the ability to use model checking to calculate safe parameter values

The example consists of a train that passes a railroad crossing. At a distance of 1000 m a signal is sent to the controller to lower the gate and 100 m past the crossing a signal is sent to raise the gate. The parameter alpha in the controller is the delay from when the signal is sent until it is reacted upon. Here model checking is used to calculate safe values for alpha so that it can be guaranteed that the gate is closed whenever the train is closer than 10 m from the crossing.

The Modelica model is divided into three components that are shown in the Figure 3, the Figure 4 and the Figure 6.

```

import
  Modelica.Blocks.Interfaces.BooleanPort;

package Railroad
  model train
    BooleanPort app;
    BooleanPort exit;
    Real x;
    discrete Real v;
    initial equation
      v = -45;
      x = 2000;
    equation
      der(x) = v;
      when x >= 100 then
        exit = app;
      end when;
      when x <= 1000 then
        v = -40;
      elsewhen x <= 0 then
        v = 35;
      end when;
    end train;

```

Figure 3. Train model in Modelica.

If we first look at the train model, the only state variable is x , so x is added to the variables of the resulting automata. An initial control mode s_1 is also added to the train automata. Since the derivative of x is negative in the initial state only the event occurring when x goes below 1000 is added as a transition from this state. A new control mode, s_2 , is created as a target for the transition since there are no existing modes that match the variable values. In this control mode the only relevant event is when x passes zero. This

results in a new transition and a new control mode, and so on. The resulting automaton is shown in Figure 5.

```

model gate
  Real y;
  discrete Real dy;
  BooleanPort lower;
  BooleanPort raise;
  initial equation
    dy = 0;
    y = 90;
  equation
    der(y) = dy;
    when lower then
      dy = -9;
    elsewhen raise then
      dy = 9;
    elsewhen y <= 0 then
      dy = 0;
    elsewhen y >= 90 then
      dy = 0;
    end when;
  end gate;

```

Figure 4. Gate model in Modelica.

The gate component results in one new continuous variable y . Since y is constant in the initial state of the gate automaton, there is no restriction in the invariant region of this state, but there are two events in the Modelica model that needs to be handled. Since the conditions of these events depend only on discrete variables we must search for the events that make them change. In this case there are events in the controller component that changes the variable. This results in the synclabs `lower` and `raise` that are added to both of the controller automaton and the gate automaton. The code for the gate automaton is shown in Figure 7.

```

var
  x, y : analog;
  t : clock;
  clk1, clk2 : discrete;
  alpha : parameter;
  automaton train
    synclabs: app, exit;
    initially s1 & x = 2000;
    loc s1: while x >= 1000 wait { dx=-45 }
      when x <= 1000 sync app goto s2;
    loc s2: while x >= 0 wait { dx=-40 }
      when x <= 0 goto s3;
    loc s3: while x <= 100 wait { dx=35 }
      when x >= 100 sync exit goto s4;
    loc s4: while True wait { dx=35 }
  end

```

Figure 5. Train automaton in HyTech.

In the controller automaton there exist no state variables but there are two discrete variables that are used in expressions together with continuous variables in when-equations; therefore they are added as discrete variables to the HyTech model. See Figure 8.

```

model controller
  parameter Real alpha=1.0;
  BooleanPort lower;
  BooleanPort raise;
  BooleanPort app;
  BooleanPort exit;
  discrete Real clk1(start=0);
  discrete Real clk2(start=0);
  equation
    when app then
      clk1 = time;
    end when;
    when exit then
      clk2 = time;
    end when;
    when app and time-clk1 > alpha then
      lower = true;
    end when;
    when exit and time-clk2 > alpha then
      raise = true;
    end when;
  end controller;

model test
  controller ctrl;
  train tr;
  gate g;
  equation
    connect(tr.app, ctrl.app);
    connect(tr.exit, ctrl.exit);
    connect(g.lower, ctrl.lower);
    connect(g.raise, ctrl.raise);
  end test;
end Railroad;

```

Figure 6. Controller model in Modelica.

```

automaton gate
  synclabs: lower, raise;

  initially s1 & y = 90;

  loc s1: while True wait {dy=0}
    when True sync lower goto s2;
    when True sync raise goto s3;

  loc s2: while y >= 0 wait {dy=-9}
    when y <= 0 goto s1;

  loc s3: while y <= 90 wait {dy=9}
    when y >= 90 goto s1;

  end

```

Figure 7. Gate automaton in HyTech.

```

automaton controller
synclabs: lower, raise, app, exit;

initially s1;

loc s1: while True wait {}
  when True sync app
    do {clk1'=t} goto s2;
  when True sync exit
    do {clk2'=t} goto s3;

loc s2: while t-clk1 <= alpha wait {}
  when t-clk1 >= alpha sync lower
    goto s1;
  when True sync exit
    do {clk2'=t} goto s3;

loc s3: while t-clk2 <= alpha wait {}
  when t-clk2 >= alpha sync raise
    goto s1;
  when True sync app
    do {clk1'=t} goto s2;

loc s4: while t-clk1 <= alpha &
  t-clk2 < alpha wait {}
  when t-clk1 >= alpha sync lower
    goto s3;
  when t-clk2 >= alpha sync raise
    goto s2;

end

```

Figure 8. Controller automaton in HyTech.

In order to make an analysis of the system we must add a section containing the analysis commands. These are shown in Figure 9.

```

var init_reg, reached, avoid: region;

avoid := y > 0 & x <= 10;

init_reg := loc[train] = s1 & x = 2000 &
  loc[gate] = s1 & y = 90 &
  loc[controller] = s1;

reached :=
  reach forward from init_reg endreach;

print omit all locations
  hide non_parameters in
    reached & avoid
  endhide;

```

Figure 9. Analysis commands for railroad example in HyTech.

The region `avoid` represents the forbidden condition. By printing the resulting region of the intersection between the reachable region and the forbidden region while hiding all non-parameter values and locations we arrive at an expression for the values of `alpha` that leads to unsafe states.

Related Work

Model checking can successfully complement existing software quality assurance techniques such as testing and debugging. Therefore it is important to provide efficient translators from various programming languages to model checkable formal languages. In this way, mature model checking techniques can be reused and applied to software systems that otherwise would not provide support for proving safety and liveness properties. Bridging the gap between high level languages such as C, C++, Java, Ada and Modelica and the input required by model checking tools (finite state automata with properties formulated in temporal logic) require the development of complex tool sets. In this section we present some of the translation frameworks that are most related to ours.

The SLAM project at Microsoft Research (Ball and Rajamani. 2002 [7], Ball and Rajamani 2001 [8]) checks temporal safety properties of sequential C programs. The system requires that the checked properties are encoded in a language called SLIC (Specification Language for Interface Checking).

The Bandera tool set (Corbett et al. 200 [9]) is an integrated collection of program analysis, transformation and visualizations components that enables the extraction of finite state models from Java source code. Bandera is able to generate a description of a finite-state transition system in the Promela and Trans languages that can be interpreted by the SIPN and SMV (Symbolic Model Verifier) models checking systems. Previously to the Bandera project, the same research group at Kansas State University has developed a toolset for translating Ada source code to the input language of the SPIN and SMV model checkers (Dwyer et al. 1998 [10]).

A related project to Bandera is the Java PathFinder (Brat et al. 2000 [11]) that translates Java programs to Promela, the specification language of the Spin model checker. Java PathFinder can detect race conditions, deadlocks, and violations of user specified assertions. The tool has been incorporated as a back-end checker for Bandera.

Conclusion

In this paper we have briefly outlined an algorithm to translate Modelica models to a representation that can be automatically verified against a formal specification using model checking. To be able to perform verifications on more sophisticated models it is possible to continue along this path and generate code for other systems such as CheckMate and HyTech+, both using hybrid automata.

The presented work in this paper should be seen as an important component for a broader attempt to make static analysis (Bunus and Fritzson 2004 [12]), run-time verification through algorithmic debugging (Bunus and Fritzson 2003 [13]) and model checking techniques, more applicable for the development of new automatic debugging tools with enhanced user-interaction for the Modelica language. We intend to implement a prototype translator using the presented algorithm in the numeric and symbolic engine developed for the OpenModelica compiler back-end.

References

- [1] Fritzson Peter. (2003). *Principles of Object-Oriented Modeling and Simulation - with Modelica 2.1*. IEEE Press and John Wiley, 2003.
- [2] Henzinger Thomas, Pei-Hsin Ho, and Howard Wong-Toi. (1997) "HyTech: A Model Checker for Hybrid Systems." *International Journal on Software Tools for Technology Transfer*, vol. 1: 1-2, pp. 110-122, 1997.
- [3] Kapinski James and Bruce H. Krogh. (2002). "A new tool for verifying computer controlled systems." In *Proceedings of the IEEE Conference on Computer-Aided Control System Design*. (Glasgow, Scotland, September, 2002).
- [4] Clark Edmund M., Grumberg Orna, Peled Doron A (1999). *Model Checking*. MIT Press. 1999.
- [5] Henzinger Thomas A., Ben Horowitz, Rupak Majumdar, and Howard Wong-Toi, "Beyond HyTech: Hybrid System Analysis Using Interval Numerical Methods," in *Hybrid Systems: Computation and Control. Lecture Notes in Computer Science 1790*, B. Krogh and N. Lynch, Eds.: Springer Verlag, 2000, pp. 130-144.
- [6] John Hatcliff, Matthew B. Dwyer, Hongjun Zheng "Slicing Software for Model Construction". *Journal of Higher-Order and Symbolic Computation*, Volume 13, Issue 4, December 2000, Pages 315 - 353
- [7] Ball Thomas and Sriram K. Rajamani. (2002). "The SLAM Project: Debugging System Software via Static Analysis." In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. (Portland, Oregon USA, January 16-18, 2002).
- [8] Ball Thomas and Sriram K. Rajamani. (2001). "Automatically Validating Temporal Safety Properties of Interfaces." In *Proceedings of the The 8th International SPIN Workshop on Model Checking of Software*. (Toronto, Canada, May 19-20, 2001). LNCS 2057, Springer Verlag.
- [9] Corbett James, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. (200). "Bandera: Extracting Finite-state Models from Java Source Code." In *Proceedings of the 22nd International Conference on Software Engineering*. (Limerick, Ireland, June 4-11, 200).
- [10] Dwyer Matthew B., Corina S. Pasareanu, and James C. Corbett (1998) "Translating Ada programs for model checking: A tutorial.," Kansas State University, Department of Computing and Information Sciences, Technical Report 98-12, 1998.
- [11] Brat Guillaume, Klaus Havelund, Seung Joon Park, and Willem Visser. (2000). "Java PathFinder Second Generation of a Java Model Checker." In *Proceedings of the In Proceedings of Post-CAV Workshop on Advances in Verification*. (Chicago, July, 2000).
- [12] Bunus Peter and Peter Fritzson. (2004) "Automated Static Analysis of Equation-Based Components." *Simulation: Transactions of the Society for Modeling and Simulation International. Special Issue on Component Based Modeling and Simulation*. To appear, 2004.
- [13] Bunus Peter and Peter Fritzson. (2003). "Semi-automatic Fault Localization and Behaviour Verification for Physical System Simulation Models." In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*. (Montreal, Canada, October 6-10, 2003).