Translating a Modeling and Simulation Language to Hybrid Automata

Kevin Atkinson[1], Adam Duracz[2] and Walid Taha[1]

[1] Rice University and Halmstad University
{kevin.atkinson,walid.taha}@hh.se
[2] Halmstad University
adam.duracz@hh.se

1 Introduction

Tools such as SpaceEx [5] are pushing the limits of formal analysis for hybrid systems and today support models with up to a hundred variables. This still falls orders of magnitude short of what simulation tools are capable of, but it suggests that verification of complex hybrid systems models may be possible.

As the complexity of models increases, the expressiveness of the language in which they are written also becomes a concern. In this sense, verification tools still lag behind simulation tools, which often provide the user with a syntax closer to that of a programming language. Simulation tools are built around languages whose main objective is to allow the user to conveniently and accurately represent the key features of a system; for example, languages like Modelica [6] and Simscape [1] support abstraction through function definitions and object-oriented modeling. This amounts to a large feature set that tends to keep expanding, catering to the needs of domain experts wishing to express themselves efficiently.

Supporting such an expressive syntax poses a challenge to the verification community. Formal analysis requires well-understood and rigorously defined notions on which to operate, and every additional language construct increases the burden of the analysis. We can proceed in two ways to resolve this problem: either we express our analysis directly in terms of the wider syntax, effectively bringing the complexity of the surface language into our analysis; or we can opt for a translation of the wide syntax into a smaller one, in terms of which we can perform our analysis. This paper investigates the latter option by translating a simple modeling language (Acumen [10], shown in in Figure 1a) to hybrid automata [7] that can be formally analyzed [8]. Source code for the implementation of this transformation is available [2].

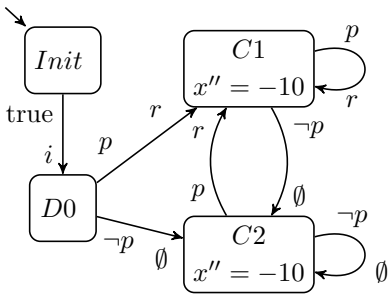
The Translation Challenge Acumen is a lightweight language for modeling hybrid systems. The continuous parts of the system are modeled by ordinary differential equations. The discrete parts are modeled by conditional statements and assignments, executed in a way that makes the order of statements irrelevant.

There are two kinds of interpreters available for the Acumen language: (1) those based on regular floating-point arithmetic that yield simulation results analogous to those obtainable using tools like Simulink [4] or OpenModelica [6]; and, (2) the Acumen enclosure interpreter [8] that yields guaranteed bounds on the simulation trajectory of the system, analogous to those produced by hybrid systems reachability analysis tools [5].

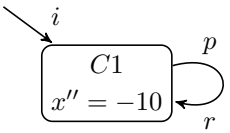
While the first kind of interpreter supports the full Acumen syntax, the enclosure interpreter supports a subset corresponding to a hybrid automaton, such as the one illustrated in Figure 1c. The goal of the algorithm described in Section 2 is to extend the syntax available to users wishing to analyze their model using the enclosure interpreter. For example, the extended syntax should permit writing models whose modes (in the sense of hybrid automata) are not given

```
class Main(simulator)
  private
    x  :=  5;
    x' :=  0;
    x'' := -10
  end
  if x <= 0 && x' <= 0
    x' := -x'/2
  end;
  x'' = -10
end
```

(a) Bouncing ball in Acumen



(b) After initial construction



(c) Final hybrid automaton

Figure 1: Input model, intermediate hybrid automaton and output hybrid automaton. In the above hybrid automata, p is the predicate $x \leq 0 \wedge x' \leq 0$, i is the reset $\{x := 5,\ x' := 0\}$, r is the reset $x' := x'/2$, and $\emptyset$ is the identity reset.

explicitly, and where hierarchies of hybrid behaviours can be expressed. This is accomplished by translating a wider syntax ($\mathcal{A}$), allowing arbitrary nesting of conditional statements intermixed with differential equations and assignments, to the hybrid automaton subset ($\mathcal{B}$).

2 Transformation Algorithm

To explain the basic steps of the algorithm, we use a simple Acumen program, modeling the bouncing ball hybrid system [8], which is shown in Figure 1a. The transformation of this model into a hybrid automaton is simple, but still involves identifying modes and transitions, as this information is not given explicitly in the model. More advanced models require additional passes that due to space are not included in this abstract.

The Basic Algorithm The algorithm starts out by reducing the program into a normal form, consisting of a sequence of `if` statements. Empty `else` branches are first added to any `if` statement lacking an `else` branch and then all `else` branches are converted into `if` statements (with negated predicates). Assignment statements and equations are pushed down into the most nested conditional statements and the conditionals are converted into top-level `if` statements, with predicates corresponding to their path condition.

Each top-level `if` statement C subsequently generates a mode M_C and a transition T_C. The continuous assignment statements of C become the differential equations of the mode M_C. The condition of C becomes the guard of T_C, and the discrete assignments of C, together with the additional discrete assignment $mode := M_C$, become the reset of T_C. The additional discrete assignment encodes the target mode of the transitions in the automaton and is a key step in this algorithm. The source mode of each T_C is derived from its guard.

Two initial modes, *Init* and $D0$, are also added. *Init* serves as the source mode of a transition whose reset corresponds to the initial values of the system. Because the initial mode of the output automaton has not yet been resolved at this stage, a default initial mode $D0$ is added.

The preceeding steps result in the automaton shown in Figure 1b. Additional passes are then used to obtain a more minimal automaton. The first pass merges modes with identical sets of differential equations and out-going transitions. In this pass, $C2$ will merge into $C1$. The second pass attempts to eliminate unnecessary modes containing a transition that must trigger and has a target mode other than its source. In this pass, $D0$ is eliminated, leaving only *Init* and $C1$. The resets in the transition from the *Init* mode become the initial conditions of the automaton leaving only mode $C1$.

Advanced Passes Handling additional language constructs in the source model requires extending the translation with more advanced passes. For example, when the model contains a switch statement, the translation may need to eliminate the variable used to switch between case clauses. To do this, the translation relies on an analysis based on postcondition and precondition predicates that are associated with all transitions and modes, respectively. The source mode of a transition is identified by comparing the precondition of each mode with the guard of the transition. The target mode is defined by matching the postcondition of the transition with the precondition of the mode. Concretely, the target mode of the transition T_C, with reset $mode := M_C$ (inducing a component $mode = M_C$ into the postcondition of T_C), is the mode M_C whose precondition includes $mode = M_C$ as a component.

3 Related Work

Translations from programming language like formalisms to hybrid automata, with the aim of applying formal methods to the source programs, have been studied previously. The approach described by Agrawal, Simon and Karsai [3] is based on a translation of models expressed in Simulink/Stateflow [4] through intermediate languages. Lyde and Might [9] describe the translation of an extended core calculus for the MATLAB programming language, to a subset of Scheme. We are concerned with the translation of a subset of the Acumen language, down to a smaller subset directly corresponding to a hybrid automaton, to enable analysis using the Acumen enclosure interpreter [8]. Crucially, the source programs (in syntax $\mathcal{A}$) that we aim to support do not specify modes and events explicitly.

References

[1] Mathworks. Simscape. `http://www.mathworks.com/products/simscape/`.

[2] Translation algorithm as part of the Acumen source code repository (*extract* branch). `https://bitbucket.org/effective/acumen-dev/branch/extract`.

[3] A. Agrawal, G. Simon, and G. Karsai. Semantic translation of Simulink/Stateflow models to hybrid automata using graph transformations. In *International Workshop on Graph Transformation and Visual Modeling Techniques*, 2004.

[4] L. P. Carloni, R. Passerone, A. Pinto, and A. L. Sangiovanni-Vincentelli. Languages and tools for hybrid systems design. *Foundations and Trends in Design Autom.*, 2006.

[5] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. Spaceex: Scalable verification of hybrid systems. In *International Conference on Computer Aided Verification*, LNCS, 2011.

[6] P. Fritzson, P. Aronsson, A. Pop, H. Lundvall, K. Nystrom, L. Saldamli, D. Broman, and A. Sandholm. OpenModelica - a free open-source environment for system modeling, simulation, and teaching. In *IEEE International Symposium on Intelligent Control*, 2006.

[7] T. A. Henzinger. The theory of hybrid automata. IEEE Computer Society Press, 1996.

[8] M. Konecny, W. Taha, J. Duracz, A. Duracz, and A. Ames. Enclosing the behavior of a hybrid system up to and beyond a zeno point. In *IEEE International Conference on Cyber-Physical Systems, Networks, and Applications*, 2013.

[9] S. Lyde and M. Might. Extracting hybrid automata from control code. In *NASA Formal Methods*, 2013.

[10] W. Taha, P. Brauner, Y. Zeng, R. Cartwright, V. Gaspes, A. Ames, and A. Chapoutot. A core language for executable models of cyber-physical systems (preliminary report). In *International Conference on Distributed Computing Systems Workshops*. IEEE Computer Society, 2012.