Extracting Hybrid Automata from Control Code

Steven Lyde, Matthew Might

University of Utah, Salt Lake City, Utah, USA,
{lyde,might}@cs.utah.edu

Abstract. Formal methods—and abstract interpretation in particular—can assist in the development of correct control code. However, current approaches to deploying formal methods do not always match the way practicing engineers develop real control code. Engineers tend to think in code first—not formal models. Standard practice is for engineers to develop their control code and *then* build a model like a hybrid automaton from which to verify properties. Since the construction of this model is manual, it leaves open the possibility of error. Existing formal approaches, on the other hand, tend to focus on synthesizing control code from a verified formal model. We propose a method for synthesizing a hybrid automaton from the control code directly. Specifically, we use abstract interpretation to create an abstract state transition system, and from this we systematically extract a hybrid automaton. Not only does this eliminate the introduction of error into the model based on the code, it fits with common practice in engineering cyberphysical systems. We test the technique on a couple examples—control code for a thermostat and a nuclear reactor. We then pass the generated automata to the HyTech model-checker to verify safety and liveness properties.

1 Introduction

Avionics, automobiles and medical equipment depend on complex control software. The proscribed approach to developing these systems is "model-first," with analysis, simulation, testing, verification and code generation to follow.

However, in practice, engineers often develop code first, and a model second, if ever. We propose an approach to formal analysis of cyber-physical systems more in line with practice: we demonstrate that a sound model—a hybrid automaton [9] in this case—can be inferred from the control code itself. This gives developers an efficient way to maintain and manipulate the model of a controller more naturally. To achieve our goal, we use abstract interpretation (a higher-order control flow analysis, in particular) to analyze control code, and from the result, we infer hybrid automata. We can then pass these hybrid automata on to model checkers and formally verify properties of program behavior. Source and examples for the tool described in this paper are available [15]. For examples and more details, we refer the reader to the companion technical report [11].

1.1 Contributions

We make the following contributions:

1. We report preliminary work on a method to extract a hybrid automaton from an abstract transition system, which is in turn synthesized from abstract interpretation of control code.
2. We claim a core calculus for a subset of MATLAB as a secondary contribution, developed to facilitate our primary contribution.

2 Language: λ_{M}, a core calculus for MATLAB

We compile control code in MATLAB to an A-Normalized core calculus. A-Normal Form (ANF) [7] forces an order of evaluation and simplifies the transition rules of both the concrete and abstract semantics.

The grammar for the target language is an unsurprising subset of Scheme, except for perhaps the inclusion of `call/ec`, which we use to model exceptions. Another interesting inclusion is that the conditional expression allows a convex predicate *cp* in its test expression. A convex predicate *cp* is a finite conjunction of linear inequalities, e.g., $x_1 \geq 3 \wedge 3x_2 \leq x_3 + 5/2$ [8].

$$
\begin{array}{rcll}
pr \in \mathsf{Prog} & = & \mathsf{Exp} & \text{[programs]} \\
v \in \mathsf{Var} & & \text{is a set of identifiers} & \text{[variables]} \\
c \in \mathsf{Const} & = & \mathit{String} + \mathbb{Z} & \text{[literals]} \\
\mathit{lam} \in \mathsf{Lam} & ::= & \texttt{(}\lambda\ \texttt{(}v_1 \ldots v_n\texttt{)}\ e\texttt{)} & \text{[lambda terms]} \\
f, æ \in \mathsf{AExp} & ::= & \mathit{lam} \mid v \mid c & \text{[atomic expressions]} \\
 & \mid & \texttt{(}op\ æ_1 \ldots æ_n\texttt{)} & \text{[primitive operations]} \\
op \in \mathsf{Op} & \supseteq & \{\texttt{+}, \texttt{-}, \texttt{*}\} & \text{[primitives]} \\
e \in \mathsf{Exp} & ::= & \texttt{(let ((}v\ ce\texttt{))}\ e\texttt{)} & \text{[expressions]} \\
 & \mid & æ & \text{[return]} \\
 & \mid & ce & \text{[tail]} \\
ce \in \mathsf{CExp} & ::= & \texttt{(}f\ æ_1 \ldots æ_n\texttt{)} & \text{[complex expressions]} \\
 & \mid & \texttt{(if}\ cp\ e_1\ e_2\texttt{)} & \text{[physical branching]} \\
 & \mid & \texttt{(if}\ ae\ e_1\ e_2\texttt{)} & \text{[cyber branching]} \\
 & \mid & \texttt{(set!}\ v\ æ\texttt{)} & \text{[variable mutation]} \\
 & \mid & \texttt{(call/ec}\ æ\texttt{)} & \text{[first-class control]}
\end{array}
$$

Fig. 1. Syntax for λ_{M}—a core calculus for MATLAB

To model programs, we inject them into time-stamped CESK-machines [6], whose state-space has five components—the current expression, the current environment, the current store, the current continuation and the current time. The time component does not relate to physical time, but is a way to encode the history of the machine's execution to facilitate abstraction. It is a parameter used when allocating new addresses in the store. In a regular CESK machine there will be an infinite number of addresses that can be allocated in the store, but we can structurally abstract this machine by bounding the set of times available [14]. Bounding time forces the set of states $\hat{\Sigma}$ to be finite.

Figure 2 describes the concrete and abstract state-spaces for the analyzer. To save space, we omit the transition relations $(\Rightarrow) \subseteq \Sigma \times \Sigma$ and $(\leadsto) \subseteq \hat{\Sigma} \times \hat{\Sigma}$

$$\begin{aligned}
\varsigma \in \Sigma &= \mathsf{Exp} \times \mathit{Env} \times \mathit{Store} \times \mathit{Kont} \times \mathit{Time} \\
\rho \in \mathit{Env} &= \mathsf{Var} \rightharpoonup \mathit{Addr} \\
\sigma \in \mathit{Store} &= \mathit{Addr} \rightharpoonup D \\
d \in D &= \mathit{Clo} + \mathit{Kont} + \mathit{String} + \mathbb{Z} \\
\mathit{clo} \in \mathit{Clo} &= \mathsf{Lam} \times \mathit{Env} \\
\kappa \in \mathit{Kont} &::= \mathbf{letk}(v, e, \rho, \kappa) \\
&\;\;\mid\;\; \mathbf{halt} \\
a \in \mathit{Addr} &::= \mathbf{bindaddr}(v, t) \\
t \in \mathit{Time} &\text{ is an infinite, ordered set of times}
\end{aligned}$$

$$\begin{aligned}
\hat{\varsigma} \in \hat{\Sigma} &= \mathsf{Exp} \times \widehat{\mathit{Env}} \times \widehat{\mathit{Store}} \times \widehat{\mathit{Kont}} \times \widehat{\mathit{Time}} \\
\hat{\rho} \in \widehat{\mathit{Env}} &= \mathsf{Var} \rightharpoonup \widehat{\mathit{Addr}} \\
\hat{\sigma} \in \widehat{\mathit{Store}} &= \widehat{\mathit{Addr}} \rightharpoonup \hat{D} \\
\hat{d} \in \hat{D} &= \mathcal{P}\left(\widehat{\mathit{Clo}} + \widehat{\mathit{Kont}} + \widehat{\mathit{String}} + \hat{\mathbb{Z}}\right) \\
\widehat{\mathit{clo}} \in \widehat{\mathit{Clo}} &= \mathsf{Lam} \times \widehat{\mathit{Env}} \\
\hat{\kappa} \in \widehat{\mathit{Kont}} &::= \mathbf{letk}(v, e, \hat{\rho}, \hat{\kappa}) \\
&\;\;\mid\;\; \mathbf{halt} \\
\hat{a} \in \mathit{Addr} &::= \mathbf{bindaddr}(v, \hat{t}) \\
\hat{t} \in \widehat{\mathit{Time}} &\text{ is an finite set of abstract times}
\end{aligned}$$

Fig. 2. The concrete (left) and abstract (right) state-spaces.

and we also omit the abstraction map $\alpha : \Sigma \to \hat{\Sigma}$ that connects them. For examples of such transitions, we refer the reader to [14]. Computing the control-flow analysis consists of constructing the "abstract state transition graph" of reachable states under $(\leadsto)$.

Since our ultimate goal is to extract hybrid automata, we make use of an alternate, labeled formulation of the abstract transition relation:

$$\hat{\varsigma} \overset{cp}{\leadsto} \hat{\varsigma}' \text{ iff } \hat{\varsigma} \leadsto \hat{\varsigma}' \text{ and the physical condition } cp \text{ holds after transition.}$$

Only physical branching expressions—`(if` *cp* `... ...)`—acquire these labels. The label `f` indicates that the transition does not depend on a physical condition.

3 Extracting a hybrid automaton

Control-flow analysis leaves us with an abstract state transition graph. We now briefly describe how to compile this graph into a hybrid automaton.

First, we want to annotate the states with some kind of physical state-change equations, e.g., $\dot{x} = -3$, as well as invariants, e.g. $t < 10$, found in hybrid automata. To do so we require "physical" annotations in the source code. The annotations identify which procedures read physical quantities and which procedure calls initiate physical changes in the system.

3.1 Algorithm

To extract the hybrid automaton from the results of the abstract interpretation, we are going to define hybrid automaton locations as sets (really, partitions) of machine states:

$$p \in \mathit{HyLocation} = \mathcal{P}(\hat{\Sigma})$$

We can lift the transition relation from machine states to hybrid locations:

$$p \leadsto p' \text{ iff there exists } \hat{\varsigma} \in p, \hat{\varsigma}' \in p' \text{ such that } \hat{\varsigma} \leadsto \hat{\varsigma}'.$$

It's convenient to have a special form of the transition relation as well:

$$p \twoheadrightarrow p' \text{ iff } p \leadsto^{\mathtt{f}} p' \text{ and } p'' \leadsto^{\mathtt{f}} p' \text{ implies } p'' = p.$$

Next, we need a function, ∇, to extract governing differential equations:

$$\nabla(\hat{\varsigma}) = \begin{cases} [\dot{x} \mapsto [\![e]\!]] & \text{if } \hat{\varsigma} = (\texttt{(assertCPS } \dot{x} = e\texttt{)}, \ldots) \\ \emptyset & \text{otherwise.} \end{cases}$$

and, for partitions, it combines all equations in a partition:

$$\nabla\{\hat{\varsigma}_0, \ldots, \hat{\varsigma}_n\} = \nabla(\hat{\varsigma}_0) \cup \cdots \cup \nabla(\hat{\varsigma}_n)$$

The procedure Extract accepts an abstract state transition graph generated by the abstract interpretation; it returns a hybrid automaton over the state-space *HyLocation* whose transition relation is $(\leadsto^{cp})$ and whose governing differential equations are determined by the function ∇:

```
procedure EXTRACT($G$):
    $P \leftarrow \{\{\hat{\varsigma}\} \mid \hat{\varsigma} \in G\}$
    do
        $P \leftarrow$MERGE(P)
        $P \leftarrow$SPLIT(P)
    until $P$ does not change
    return $P$
```

The procedure Merge coalesces partitions with compatible equations:

```
procedure MERGE($P$):
    do
        for each partition $p$ in $P$:
            if $p \twoheadrightarrow p'$ and $dom(\nabla(p)) \cap dom(\nabla(p')) = \emptyset$:
                merge partitions $p$ and $p'$ in $P$
            if $p' \leadsto^{\mathtt{f}} p$ and $p'' \leadsto^{\mathtt{f}} p$ and $\nabla(p) = \nabla(p')$:
                merge partitions $p'$ and $p''$ in $P$
    until $P$ does not change
    return $P$
```

The procedure SPLIT duplicates nodes that are blocking a merge due to differential equations:

```
procedure SPLIT($P$):
    for each partition pair $p, p'$ in $P$:
        if $p' \rightsquigarrow p$ and $p'' \rightsquigarrow p$ and $dom(\nabla(p)) \cap dom(\nabla(p')) \neq \emptyset$:
            duplicate $p$ as $p^*$ in $P$
            replace the edge $p \rightsquigarrow p''$ with $p \rightsquigarrow p^*$
    return $P$
```

4 Preliminary evaluation

All code and examples for our implementation are available [15]. We prototyped our technique using Octave (a superset of MATLAB) for the control code. We utilize MATLAB as the input language because in our interactions with practicing engineers, we have found substantial amounts of control code—some production, some prototype—written in MATLAB. We analyzed the control code for a simple thermostat and for a nuclear reactor with our tool. After producing hybrid automata, we verified properties of the control code using HyTech [10].

5 Related Work

To generate the abstract state transition graph, we used a small-step formulation of k-CFA [14], built upon the original formulation of k-CFA by Shivers [12] and the large body of abstract interpretation first started by the Cousots [4, 5].

Our goal was to transform the abstract state-space into a model from which safety properties could be proved. For the model we chose hybrid automaton as formulated by Henzinger [9]. This choice was made not only for their ability to accurately model cyber-physical systems, but also because tools already exist that verify safety and liveness properties [10].

The idea of merging abstract interpretation and physical systems in not unique to our work. Cousot argues that static analysis of control software can be guided by knowledge of the physical system [3]. Our approach claims that given additional information of the physical system, in the form of annotations, abstract interpretation can be used to create a model of the entire system.

Similar work has been done to convert control code to hybrid automata by Bouissou [2]. He provides a semantics preserving transformation from H-SIMPLE to a sampled hybrid automata and then from sampled hybrid automata to a regular hybrid automata. H-SIMPLE is a simple imperative language that has statements to control sensors and actuators, very similar to the annotations provided in our framework. While our language is also simple, it provides first class functions, which allow powerful higher language constructs to be modeled.

Another similar work to ours is in transforming Simulink/Stateflow models into Hybrid Automata [1, 13]. However, our work differs in that we start from the control code of the system, not another model of the system.

This research is partly supported by the National Science Foundation under Grant No. 1035658.

References

1. Agrawal, A., Simon, G., and Karsai, G. Semantic Translation of Simulink/Stateflow Models to Hybrid Automata Using Graph Transformations. *Electron. Notes Theor. Comput. Sci. 109* (Dec. 2004), 43–56.
2. Bouissou, O. From control-command synchronous programs to hybrid automata. In *Analysis and Design of Hybrid Systems* (2012), pp. 291–298.
3. Cousot, P. Integrating physical systems in the static analysis of embedded control software. *Programming Languages and Systems* (2005), 135–138.
4. Cousot, P., and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages* (New York, NY, USA, 1977), ACM Press, pp. 238–252.
5. Cousot, P., and Cousot, R. Systematic design of program analysis frameworks. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (New York, NY, USA, 1979), POPL '79, ACM Press, pp. 269–282.
6. Felleisen, M., and Friedman, D. P. A calculus for assignments in higher-order languages. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (New York, NY, USA, 1987), ACM, pp. 314+.
7. Flanagan, C., Sabry, A., Duba, B. F., and Felleisen, M. The essence of compiling with continuations. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (New York, NY, USA, June 1993), ACM, pp. 237–247.
8. Henzinger, T., Ho, P., and Wong-Toi, H. A user guide to hytech. *Tools and algorithms for the construction and analysis of systems* (1995), 41–71.
9. Henzinger, T. A. The theory of hybrid automata. In *Logic in Computer Science, 1996. LICS &#039;96. Proceedings., Eleventh Annual IEEE Symposium on* (July 1996), IEEE, pp. 278–292.
10. Henzinger, T. A., Ho, P. H., and Toi, H. W. HYTECH: A model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer 1*, 1-2 (1997), 110–122.
11. Lyde, S., and Might, M. Extracting hybrid automata from control code. Tech. rep., University of Utah, 2013. Available from `http://matt.might.net/a/2013/03/03/ha-extract/lyde2013hybrid.pdf`.
12. Shivers, O. G. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1991.
13. Silva, B. I., Richeson, K., Krogh, B., and Chutinan, A. Modeling and verifying hybrid dynamic systems using checkmate. In *Proceedings of 4th International Conference on Automation of Mixed Processes* (2000), pp. 323–328.
14. Van Horn, D., and Might, M. Abstracting abstract machines. In *ICFP '10: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (2010), ICFP '10, ACM Press, pp. 51–62.
15. Hybrid Automata Extraction, `https://github.com/stevenlyde/ha-extraction`