AFRL-RI-RS-TR-2017-147

TEAMBLOCKS: HYBRID ABSTRACTIONS FOR PROVABLE MULTI-AGENT AUTONOMY

BAE SYSTEMS AIT

JULY 2017

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2017-147 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
WILLIAM D. LEWIS
Work Unit Manager

/ S /
JULIE BRICHACEK
Chief, Information Systems Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
JULY 2017	FINAL TECHNICAL REPORT	MAY 2015 – MAY 2017

4. TITLE AND SUBTITLE

TEAMBLOCKS: HYBRID ABSTRACTIONS FOR PROVABLE MULTI-AGENT AUTONOMY

5a. CONTRACT NUMBER
FA8750-15-C-0122

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
62788F

6. AUTHOR(S)

Peter Kingston and Patrick Martin

5d. PROJECT NUMBER
S2MA

5e. TASK NUMBER
VB

5f. WORK UNIT NUMBER
AE

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
BAE Systems AIT
600 District Avenue
Burlington, MA 01803

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RISC
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2017-147

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

In this report, we describe *TeamBlocks*, a software library and underlying theory for the construction, analysis, and execution of provably-correct team autonomy software. *TeamBlocks* includes (a) a framework for working with polynomial hybrid automata, (b) a tool to generate automaton models from C++ code, (c) a runtime validator for Linear Temporal Logic (LTL) specifications, (d) a method for proof search using polynomial Sum of Squares, and (e) a collection of hybrid automaton models.

15. SUBJECT TERMS
Verification and validation (V&V), autonomy, hybrid systems, automata, Linear Temporal Logic (LTL), Sum of Squares (SoS)

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**WILLIAM D. LEWIS**
U	U	U	SAR	39	19b. TELEPHONE NUMBER (*Include area code*) **NA**

Table of Contents

Table of Figures

Chapter 1: Introduction

The reduction of size and cost for computing and electromechanical components is driving the deployment of networked cyber-physical systems (CPS). Defense, healthcare, infrastructure, and transportation industries are building CPS to create new services for their consumers. As these technologies proliferate and are composed to create vast systems of systems, there will be an increased risk of instabilities and cascading failures due to improperly designed control modules.

Some recent examples of control induced failures include power grid blackouts and shutdowns [7,15] due to faulty actuators and human-interaction. Large-scale cloud applications also suffer outages because of improperly designed software control logic [2]. Even simple, stable automatic controllers may cause issues when chained together with other systems, e.g. [4, 18, 21, 23]. These examples indicate that there is a fundamental issue with composition at all levels of system abstraction.

For the past thirty years, there has been considerable effort to model, compose, and verify pure software systems such that they behave as specified by a customer or designer. One way to deal with software system composition is to apply software modeling and analysis techniques that ensure the software operates according to a specification [1]. This model checking process requires a formal model for each software component, typically some finite state machine, and a specification language (e.g. LTL [19] or CTL [5]) that captures system requirements. An analysis tool consumes these two components and applies an algorithm to determine if the desired specification was met by the software component. More sophisticated models of software components were developed to handle concurrent, communicating systems, such as the communicating sequential process language [10] and calculus of communicating systems [14]. These new modeling paradigms introduced the notion of bisimulation to determine the semantic equivalence of one process to another process.

Since CPS have major software components, they inherit the same model checking challenges as pure software, but make the problem more complex through interactions with sensor and actuator hardware as well as heterogeneous communication systems. More recent research has built new CPS software analysis and synthesis tools that target individual components of the system. One important extension from the model checking community is the development of approximate bisimulation, e.g. [8, 9, 24]. This work allows the comparison between dynamical systems to determine if they have similar behavior within some error bound. Approximate bisimulation has been applied to software controller analysis [11] and synthesis [13].

The contribution of this paper is the *TeamBlocks* framework that facilitates the correct construction of cyber-physical systems from source code to collections of systems. We model software and systems using a modified hybrid automata and leverage recent advances in interface controllers, e.g. [6], to link together modules at each step in the abstraction hierarchy. Furthermore, approximate bisimulation is used to generate behavioral certificates for implemented controllers and compositions across abstraction layers such that they approximately match their theoretical design. By integrating these theoretical tools across each abstraction boundary in the CPS hierarchy, *TeamBlocks* guarantees the correct operation of the composed system within user specified bounds, or it detects that the composition fails.

Section 3 introduces *TeamBlocks's* core concepts and models. Section 4 discusses our modeling assumptions and applies the work in Section 3 to model software and systems. The construction of behavioral certificates is presented in Section 5 and the simulation results for a candidate pair of systems are demonstrated in Section 6.3. We conclude with a discussion of our simulation results and potential research extensions in Section 9.

Chapter 2: Use Cases

Before proceeding with formal definitions, it will be helpful to point out a few of the use cases that we have in mind for *TeamBlocks*' software tools, and for the development of the underlying theory, so that we can see where we are headed.

Example: Generate model from C++ source code. Suppose we have developed a controller to be implemented on an embedded system. The implementation has been written in C++. Now we would like to perform further analysis of the system, based on this C++ controller implementation, including whatever quirks it may have. To achieve this, our first step is to generate a model of the C++ implementation that can be used with the rest of the analysis tools. For a supported subset of C++ code, *TeamBlocks* provides exactly this automatic model extraction capability. Give it code; it returns a hybrid automaton model.

Example: Compose and execute models. Suppose next that we are interested in assembling larger systems out of smaller ones, so that we can understand how the larger system (or system of systems) behaves. *TeamBlocks* provides tools to do this as well, and to prove properties of that larger system. That is, *TeamBlocks* provides tools for composition and execution of hybrid automaton models formed from smaller ones.

Example: Check execution against specification. If we have a specification for how a system ought to behave, we would like to test whether a real system actually satisfies that specification. *TeamBlocks* provides a Runtime Validator that does this for a class of specification described using Linear Temporal Logic (LTL).

Example: Use provided models. If we simply wish to model a system-of-systems without starting from scratch, *TeamBlocks* provides a small library of useful models for example, of vehicles that can be used within the framework. These are described using the same, unified representation as software and other systems.

Example: Verify abstraction. As systems are composed, the dimensions of their state spaces are multiplied, and, as a result, one often runs quickly into problems of extremely large state spaces. We use abstraction to combat this complexity, by verifying that a complicated system actually behaves like another, simpler system, to within an approximation bound. Given two systems, we would like an automatic proof that the more complicated system actually does behave approximately like the simpler one from an input-output perspective.

As part of *TeamBlocks,* we have investigated problem formulations and developed prototypes that automatically search for proofs that the one system does in fact approximate the other.

With these motivating examples in mind, we in the next chapter jump into a description of *TeamBlocks*' underlying modeling framework.

Chapter 3: Modeling Framework

3.1 Generalized Hybrid Automata

TeamBlocks requires a modeling paradigm that can describe both control software, which typically has discrete time and state, and physical systems (e.g., the aerodynamics of a UAV), which are typically modeled with continuous time and state. Our approach is to use a single hybrid-automaton representation for both. In particular, we use a kind of controlled hybrid automaton with output map, which we refer to throughout as the generalized hybrid automaton (GHA). Its definition follows:

Definition 1. Let a generalized hybrid automaton (GHA) be defined as the tuple

$$A = (Q,\mathbf{X},U,Y,\Sigma,\mathbf{F},\mathbf{H},E,\Phi,\mathbf{R})$$

consisting of:

- a finite set $Q = \{q^1,...,q_{|Q|}\}$ of modes,
- a set $\mathbf{X} = \{X_q\}_{q\in Q}$, where $X_q \subset \mathrm{R}^{d_q}$, $d_q \in \mathrm{Z}^+$
- a set $U \subset \mathrm{R}^m$ of admissible control inputs,
- a set $Y \subset \mathrm{R}^p$ of possible outputs,
- a finite set Σ of synchronization labels,
- a set $\mathbf{F} = \{f_q\}_{q\in Q}$ of vector fields $f_q : X_q \times U \to X_q$,
- a set $\mathbf{H} = \{h_q\}_{q\in Q}$ of output maps $h_q : X_q \to Y$,
- a set $E \subset Q_1 \times Q \times \Sigma$ of labeled edges where each edge $e = (s,d,\sigma) \in E$ has the following components ,
- a set $\Phi = \{\varphi_e\}_{e\in E}$ of guard functions, $\varphi_e : X_s \times U \to \mathrm{R}^{\dim\Phi_e(X_s,U)}$
- a set $\mathbf{R} = \{R_e\}_{e\in E}$ of reset functions, $R_e : X_s \times U \to X_d$
- an initial condition $W_0 \subset \{(q,x)|q \in Q,x \in X_q\}$.

The GHA state space is the set $W = \{(q,x)|q \in Q,x \in X_q\}$. We say that an edge $e \in E$ is active at time t whenever all elements of $\varphi_e(x(t)) \geq 0$ during the execution; the system may nondeterministically transition, via its reset map, along any edge whenever that edge is active.

[1] We use the subscripts *s* and *d* as shorthand for source and destination modes.

An *execution* of a GHA is a function $w = (q, x): \mathbb{R}_+ \to W$, such that $w(0) \in W_0$, and there exists a sequence of times $T = \{t_1, t_2, \dots\} \subset \mathbb{R}_+$ such that for each interval $[t_k, t_{k+1}) \in \{[0, t_1), [t_1, t_2), \dots\}$ the following equations hold for all $t \in [t_k, t_{k+1})$:

$$\begin{aligned} x(t) &= x(t_k) + \int_{t_k}^{t_{k+1}} f\,(q(\tau))(x(\tau), u(\tau))d\tau \\ q(t) &= q(t_k) \\ \phi_{(s,d)}(x(t), u(t)) &< 0,\ \forall (s,d) \in E \text{ s.t. } s = q(t) \end{aligned}$$

For each $t_i \in T$, there exists $(s, d) \in E$ such that $q(t_k^-) = s$, $q(t_k) = d$, $x(t_k) = R_{(s,d)}(x(t_k^-), u(t_k^-))$, and $\phi_{(s,d)}(x(t_k^-), u(t_k^-)) = 0$.[2]

We define polynomial GHA, denoted A_P, as the subclass of GHAs for which the mappings within **F**,**H**,Φ, and **R** are finite polynomials, and W_0 is a sub-level set of a finite polynomial.

3.2 Composition Operators

Asynchronous Composition of GHA

The *TeamBlocks* analysis operations require the definition of an asynchronous product for two given GHA. We adapt the standard asynchronous product between two discrete automata (e.g. [3, 17]):

Definition 2. Given two GHAs,

$$\begin{aligned} A_1 = &\quad (Q_1, \mathbf{X}_1, U_1, Y_1, \Sigma_1, \mathbf{F}_1, \mathbf{H}_1, E_1, \Phi_1, \mathbf{R}_1, W_{1,0}) \\ A_2 = &\quad (Q_2, \mathbf{X}_2, U_2, Y_2, \Sigma_2, \mathbf{F}_2, \mathbf{H}_2, E_2, \Phi_2, \mathbf{R}_2, W_{2,0}) \end{aligned}$$

their *asynchronous product*, $A_1 || A_2$, is composed of

- mode set $Q = Q_1 \times Q_2$
- continuous state space $\mathbf{X} = \{X_{1,q_1} \oplus X_{2,q_2}\}_{(q_1,q_2)\in Q}$ where is $\oplus$ the direct sum.
- labels $\Sigma = \Sigma_1 \cup \Sigma_2$
- initial condition $W_0 = \{(q, x) | q \in Q_{1,0} \times Q_{2,0}, x \in X_{1,0} \oplus X_{2,0}\}$
- edges from the union of three sets:
  - $E_{1\overline{2}} = \{((s_1, s_2), (d_1, s_2), \sigma) | (s_1, d_1, \sigma) \in E_1 \wedge \sigma \in \Sigma_1 \backslash \Sigma_2 \wedge s_2 \in Q_2\}$ (Advance A_1 but not A_2)

[2] For notational convenience, we let $g(t_k^-)$ denote $\lim_{\tau \to 0} g(t - \tau)$ for any given function g.

- $E_{\overline{1}2} = \{((s_1, s_2), (s_1, d_2), \sigma) | (s_2, d_2, \sigma) \in E_2 \wedge \sigma \in \Sigma_2 \backslash \Sigma_1 \wedge s_1 \in Q_1\}$ (Advance A_2 but not A_1)
- $E_{12} = \{((s_1, s_2), (d_1, d_2), \sigma) | (s_1, d_1, \sigma) \in E_1 \wedge (s_2, d_2, \sigma) \in E_2\}$ (Advance A_1 and A_2 together)

- guard functions for the three cases,
  - A_1 guard condition:

$$\phi_e((x_1, x_2)) = \phi_{1,(s_1,d_1,\sigma)}(x_1) \; \forall e \in E_{1\overline{2}}$$

  - A_2 guard condition:

$$\phi_e((x_1, x_2)) = \phi_{2,(s_2,d_2,\sigma)}(x_2) \; \forall e \in E_{\overline{1}2}$$

  - Conjunction of A_1, A_2 guard conditions:
  - $\phi_e((x_1, x_2)) = (\phi_{1,(s_1,d_1,\sigma)}(x_1), \phi_{2,(s_2,d_2,\sigma)}(x_2)) \forall e \in E_{12}$

- and reset maps for the three cases,
  - $R_e((x_1, x_2)) = (R_{1,q_1}(x_1), x_2) \; \forall e \in E_{1\overline{2}}$
  - $R_e((x_1, x_2)) = (x_1, R_{2,q_2}(x_2)) \; \forall e \in E_{\overline{1}2}$
  - $R_e((x_1, x_2)) = (R_{1,q_1}(x_1), R_{2,q_2}(x_2)) \; \forall e \in E_{12}.$

When a pair of GHA, A_1, A_2, have inputs and outputs, say U_1, U_2 and Y_1, Y_2 respectively, we need to take care that the asynchronous composition properly handles the dimensionality requirements connecting these systems. To handle this operation, we develop a *feedback operator* from one GHA to another.

Feedback Operator

Given a product of automata, we will often want to define the connection of its outputs to its inputs. For this purpose, it will be useful to define a class of simple selection operators $\Pi_{\{i1,\ldots,ik\}} : \mathrm{R}^j \rightarrow \mathrm{R}^k$, $k \leq j$, that return a subset of a vector's elements. With this operator, we define a (partial) feedback map to be a function $K_{I,J} : \Pi_I Y \rightarrow \Pi_J U$, $|I| = |J|$, that maps components (specified by index set I) of the output in Y , to corresponding components (specified by J) of the input in U. Given an automaton A and a feedback map $K_{I,J}$, we get new vector fields at all modes by composing f_q and K as well as new passports and transforms by composing φ_e and R_e with K.

3.3 Timing

The hybrid automaton model defined in the previous section can describe a variety of event-triggered systems. As a special case, this includes discrete-time digital control systems with regular sample intervals. In this section, we describe how this is modeled by means of a clock automaton (Figure 1). Clocks and time will appear in more detail in the next chapter.

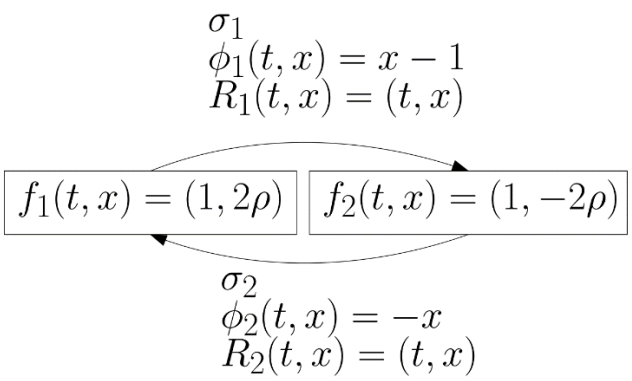


Figure 1 The clock automaton is a polynomial hybrid automaton that, when composed with a controller model via asynchronous product, defines the passage of time and the sample rate ρ. Its modes $\{1,2\} = Q$ have continuous state space $X_1 = X_2 = X = \mathbb{R} \times \mathbb{R} \ni (t, x)$, where t is the physical time, and x is the state of a triangle oscillator.

Chapter 4: Modeling Software Dynamics

TeamBlocks uses the polynomial variant of Definition 1 to model control software. This approach is similar in essence to the work of [22]; however, by using the full generality of hybrid automata, it is possible to leverage approximate bisimulation for validation. This section describes and illustrates how polynomial GHA model software components as well as how the models are automatically generated from C/C++ source code.

4.1 Modeling Assumptions

Figure 2 presents the assumed control architecture that runs software controllers. It is assumed that software controllers are non-blocking functions deployed within a timed process that executes every T seconds. However, operations that access hardware, such as the A/D or D/A, may be a blocking operation through their software APIs. The sample period sets the rate at which the system A/Ds digitize the output signal for control code consumption. Once the newly sampled data is provided to the software controller, it executes as quickly as possible according to the system's processor clock. The control algorithm computes a new value that is written to the D/A for conversion into an actuator signal.

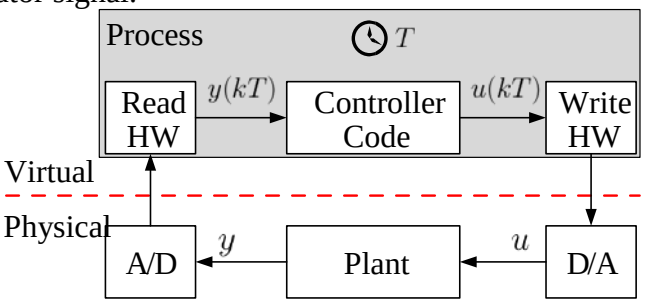


Figure 2 A diagram of the assumed control software execution when deployed on the cyberphysical system.

Modern software controllers are typically implemented on embedded systems with short instruction periods (μs to ns) using a zero-order-hold strategy. For example, an unmanned aerial vehicle autopilot running on a 900 MHz Raspberry Pi 2[3] can easily satisfy control loop periods on the order of 10^{-3}s. Thus, we assume that the time to execute a piece of control code, Δt, is much less that the control loop period, T. Figure 2 visualizes the parallel execution of control software with system dynamics by using the notion of superdense time [12].

[1] https://www.raspberrypi.org/

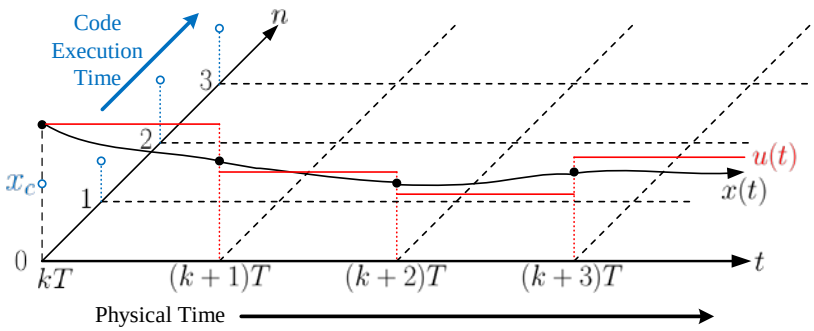


Figure 3: An illustration of how superdense time models the execution of a software controller between autopilot sample times. When a control loop is intiated at its next sample period, the code executes instantaneously along the steps along the index set N. The control value is then written to the actuator and held until the next sample time.

Superdense time is the set $S = \mathrm{R} \times N$ where R is a real valued clock and $N \subset \mathrm{N}_{\geq 0}$ is a finite subset of indices. The continuous portion of system state, $x(t)$, evolves along the physical time axis, t. The software controller state, x_c, evolves along the code execution axis, n, where each instruction is a discrete step. Consider the sample time kT in Figure 4.2, where a sensor provides new data to the control software routine. The computer executes each line of the routine at an index value from the code execution time line. When the control calculation routine completes, it writes the actuator value for that sample time, $u(kT)$, to the memory location used to set actuator speed or position. This actuator value will be held until the next sample time, $(k + 1)T$, and the software controller executes again along its discrete axis.

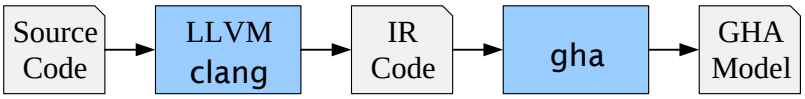
Note that this model of software controller execution is not limited to directly interfacing software and hardware. It can also capture higher level control algorithms that read (write) from (to) data channels. For example, a multi-agent consensus algorithm might block on a queue that provides new neighbor and local state information. The consensus software would compute the resulting control value as soon as the new data arrives in the queue. If this data is provided at a regular interval, the algorithm's execution would operate in a similar way to the hardware example above.

4.2 Generating Dynamical Models of Code

The authors of [22] developed a graphical model of software and created an algorithm to analyze it for runtime behavior. However, there was no process to ingest controller code written in a general purpose language and automatically create a model. Figure 4.3 shows the process that translates source code written in C/C++ into a polynomial hybrid automata. The LLVM compiler, `clang`, compiles the source le into LLVM's intermediate representation (IR) bytecode. The intermediate `gha` application builds the polynomial GHA model by processing a subset of the IR instruction set to build the elements in Definition 1. To facilitate analysis of GHA by future downstream tools, the GHA Model output is generated using a protocol buffer schema[2]

[2]https://github.com/google/protobuf

IR bytecode les produced by `clang` represent a hierarchy of program objects: modules,



functions, basic blocks, and instructions. Modules are the highest level of IR and maintain a list of
Figure 4: The internal process for TB Generator that turns controller source code into a polynomial GHA.

functions; consequently, each function has a list of instructions. Basic blocks are support structures that contain a sequence of instructions that have no branching operations. The `gha` application processes a single module by inspecting each function's IR instructions in order of execution.

We map sequences of instructions that do not branch, or call other functions, into a single mode. Each program variable that is assigned on the stack becomes a state variable for that mode. Using this modeling approach, we assume that there are no dynamics within a mode; instead, state variables evolve through the reset functions along each edge of the produced model. The GHA captures branching behavior by converting comparison instructions into guard functions. The inputs and outputs of a software model are indicated through function calls that read or write hardware registers. For example, a function `sync_in()` might perform a blocking read from the A/D in Figure 4.1. These synchronization functions also generate the events contained in Σ. All other edges in the model receive the empty label, ∈, indicating that no synchronization is required on that edge.

4.3 Code to Model Example

We apply the modeling discussion of the prior section to the example code snippet in Figure 4.4, which increments an integer value, a, from 0 to 9. A diagram of the automatically generated hybrid model is shown in Figure 4.5.

```
1   int main(){
        int N = 10;
        int a = 0;
        for(int i = 0; i < N; i++){
5           a++;
        }
    }
```

Figure 4: A simple for-loop code snippet ingested by the TB Model Generation process.

The formal hybrid automata model, A_{for}, for this source code is generated by applying Definition 1 over the IR code that follows the requirements described in Section 4.2. This code has five modes, $Q = \{q_0, \cdots, q_4\}$, five edges,
$E = \{e_0 = (0,1,\epsilon), e_1 = (1,2,\epsilon), e_2 = (1,3,\epsilon), e_3 = (2,4,\epsilon), e_4 = (4,1,\epsilon)\}$; and a state space set **X** composed of:

$$X_q = \begin{bmatrix} \texttt{N} \\ \texttt{a} \\ \texttt{i} \\ \texttt{retval} \end{bmatrix}, \forall q \in Q$$

Note that the state variable `retval` represents the return status of the `main()` process. This model has no inputs or outputs, so $U = Y = \varnothing$ and $\mathbf{H} = \varnothing$. Since there are no synchronization labels, all

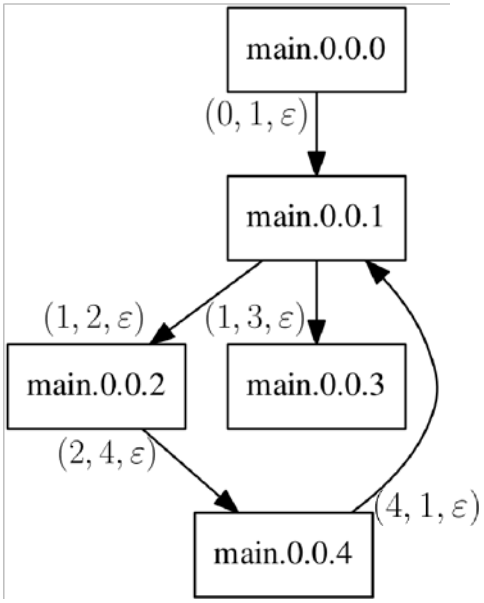


Figure 5: The polynomial hybrid automata that is automatically generated from the source code in Figure 4.

edges have the empty label, . Then the guard function set is given by:

$$\Phi = \{\phi_{e_0} = 1, \phi_{e_1} = \texttt{N} - \texttt{i}, \phi_{e_2} = \texttt{i} - \texttt{N}, \phi_{e_3} = 1, \phi_{e_4} = 1\}$$

Based on the assumptions described in Section 4.2, the physical dynamics of each mode are $f_q = 0$, $\forall q \in Q$. The software state variables are transformed by a set of reset functions, **R**, associated with each edge:

$$\begin{aligned} R_{e_0} &= \begin{bmatrix} 10 & 0 & 0 & 0 \end{bmatrix}^T \\ R_{e_1} &= \begin{bmatrix} \texttt{N} & \texttt{a} & \texttt{i} & \texttt{retval} \end{bmatrix}^T \\ R_{e_2} &= \begin{bmatrix} \texttt{N} & \texttt{a} & \texttt{i} & \texttt{retval} \end{bmatrix}^T \\ R_{e_3} &= \begin{bmatrix} \texttt{N} & \texttt{a}+1 & \texttt{i} & \texttt{retval} \end{bmatrix}^T \\ R_{e_4} &= \begin{bmatrix} \texttt{N} & \texttt{a} & \texttt{i}+1 & \texttt{retval} \end{bmatrix}^T. \end{aligned}$$

Chapter 5: Simulations and Abstraction

5.1 Motivation

The *TeamBlocks* model generation and validation process is illustrated in Figure 6. The TB Model Generator component accepts software controller source code, such as PID or consensus algorithms. Using the LLVM[4] compiler infrastructure, this component translates C or C++ source code into a hybrid dynamical model, which is presented in Section 3.1.

The TB Validator consumes the hybrid automata models for the source code and its ideal description. It uses sum-of-squares (SOS) optimization [16, 20] algorithms to 1) generate an approximate bisimulation function as a behavioral certificate, or 2) report that the code is invalid based on the requirements. Although this paper is focused on the validation of implemented source code against its ideal model, the TB Validator may operate over any pair of models that conform to the definitions presented in Section 3.1.

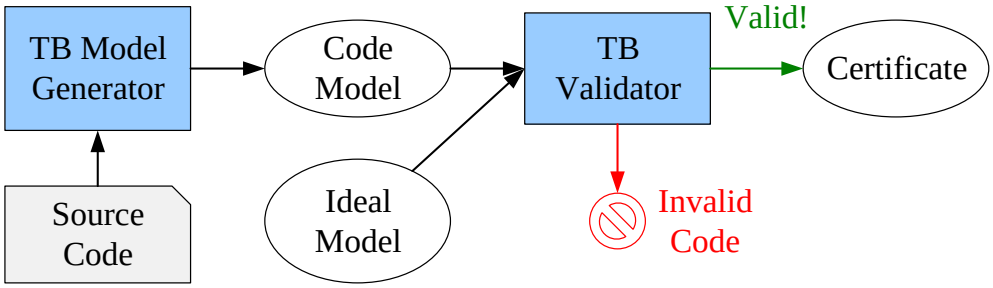


Figure 6: This figure illustrates the *TeamBlocks* model generation and validation process.

The more precise sense in which one system can be made behave to like another one is called approximate (bi)simulation; we describe this next, beginning with the older concepts of exact simulation and bisimulation.

5.2 Definitions

5.2.1 Simulation and Bisimulation

Informally, a transition system T_2 can simulate a system T_1, if there exists a memoryless controller for T_2 that makes it input-output identical to T_1. In symbols, we have:

Definition 3 (Simulation). Let Q_1,Q_2 be the state spaces of T_1,T_2, respectively. A relation $S \subset Q_1 \times Q_2$ is called a simulation relation of T_1 by T_2 is for all $(q_1,q_2) \in S$:

- $h_1(q_1) = h_2(q_2)$
- for all $q_1 \rightarrow_\sigma q_1'$ there exists $q_2 \rightarrow_\sigma q_2'$ such that $(q_1', q_2') \in S$.

[4] http://llvm.org/

If a simulation relation of T_2 by T_1 exists, then T_1 is said to simulate T_2.

Translating back to words, such a relation S has the property that states can be associated between Q_1,Q_2, and inputs to T_2 chosen, such that corresponding outputs of T_2 are identical to outputs of T_1. When we read there exists in the above definition, we can mentally substitute there exists a function to choose, or there exists a controller.

Likewise, a bisimulation relation exists between T_1 and T_2 if T_1 can simulate T_2 and T_2 can simulate T_1, i.e.:

Definition 4 (Bisimulation). Let Q_1,Q_2 be the state spaces of T_1,T_2, respectively. A relation $S \subset Q_1 \times Q_2$ is called a bisimulation relation between T_1 and T_2 if for all $(q_1,q_2) \in S$:

- $h_1(q_1) = h_2(q_2)$
- for all $q_1 \rightarrow_\sigma q_1'$ there exists $q_2 \rightarrow_\sigma q_2'$ such that $(q_1', q_2') \in S$.
- for all $q_2 \rightarrow_\sigma q_2'$ there exists $q_1 \rightarrow_\sigma q_1'$ such that $(q_1', q_2') \in S$.

If a bisimulation relation between T_2 and T_1 exists, then T_1 and T_2 are said to be bisimilar.

In other words, controllers exist to make either system input-output identical to the other.

5.2.2 Approximate Simulation and Bisimulation

In [8], closely-related definitions were introduced that allow for a greater degree of approximation. These modified definitions, appropriately-enough named approximate (bi)simulation, simply introduce a distance metric d and replace the exact equality $h(q_1) = h(q_2)$ by the approximate equality $d(h(q_1), d(q_2)) \leq \epsilon$. The definitions follow:

Definition 5 (Approximate Simulation). Let Q_1,Q_2 be the state spaces of T_1,T_2, respectively. A relation $S \subset Q_1 \times Q_2$ is called an -simulation relation of T_1 by T_2 is for all $(q_1,q_2) \in S$:

- $d(h_1(q_1), h_2(q_2)) \leq \epsilon$
- for all $q_1 \rightarrow_\sigma q_1'$ there exists $q_2 \rightarrow_\sigma q_2'$ such that $(q_1', q_2') \in S$.

If a ϵ-simulation relation of T_2 by T_1 exists, then T_1 is said to *ϵ-simulate* T_2.

Definition 6 (Approximate Bisimulation). Let Q_1,Q_2 be the state spaces of T_1,T_2, respectively. A relation $S \subset Q_1 \times Q_2$ is called a -bisimulation relation between T_1 and T_2 if for all $(q_1,q_2) \in S$:

- $d(h_1(q_1), h_2(q_2)) \leq \epsilon$
- for all $q_1 \rightarrow_\sigma q_1'$ there exists $q_2 \rightarrow_\sigma q_2'$ such that $(q_1', q_2') \in S$.
- for all $q_2 \rightarrow_\sigma q_2'$ there exists $q_1 \rightarrow_\sigma q_1'$ such that $(q_1', q_2') \in S$.

If a ϵ-bisimulation relation between T_2 and T_1 exists, then T_1 and T_2 are said to be *ϵ-bisimilar*.

As [8] argues, the importance of the newer approximate definitions is that they enable additional abstraction. In short, the relaxed definition may allow a system to get away with using fewer states to approximate a more complicated system, than it would have needed to exactly simulate it. For our purposes in *TeamBlocks*, it is primarily simulation that will be of importance. We will look for controllers that cause a complicated system to approximately simulate a simpler one and we will look for certificates that the approximate simulation relation actually exists.

5.2.3 Simulation Functions

Our certificate that one system really does approximately simulate another one will take the form of a simulation function. These are scalar-valued functions of the states of the two systems, which upper-bound the error that may possibly be seen. More precisely:

Definition 7 (Simulation Function). A function $V : Q_1 \times Q_2$ is a simulation function if,

$$V(q_1, q_2) \geq \max \left(d(q_1, q_2), \sup_{q_1 \to_\sigma q_1'} \inf_{q_2 \to_\sigma q_2'} V(q_1', q_2') \right). \tag{5.1}$$

If we have a function $V : Q_1 \times Q_2$, we can easily verify that it is indeed a simulation function, by performing a local test at each joint state (q_1,q_2). In this sense a simulation function is much like a Lyapunov function it's difficult to find but easy to test, and it implicitly carries global information about the dynamical system.

If (5.1) holds with equality, then we arrive at a system of Bellman-like inequalities, and V is known as the directed branching distance. Like the optimal Value Function, it can be computed by Value Iteration, but this can only be done in finite-state systems, and the Curse of Dimensionality makes this impractical in all but the smallest of these. Thus, we will need to find a better way to search for simulation functions. This will be the topic of the next chapter but first, we make one additional simplification.

5.2.4 The Interface Controller and the Error System

If we fix a choice of controller [5] $K : Q_1 \times \Sigma \longrightarrow Q_2$, then (5.1) becomes,

$$V(q_1, q_2) \geq \max \left(d(q_1, q_2), \sup_{q_1 \to_\sigma q_1'} V(q_1', K(q_1, \sigma)) \right). \tag{5.2}$$

Equivalently, we form a new system T by first (a) forming the product system $T_1 \times T_2$, then (b) applying the feedback interconnection operator with feedback map K, and finally (c) defining the output function $h(q) = d(q_1,q_2)$, at which point, letting $Q = Q_1 \times Q_2$ be the state space of T, (5.2) can be expressed,

[5] K is called the interface controller because it causes T_2 to satisfy the interface defined by T_1.

$$V(q) \geq \max\left(h(q),\ \sup_{q \to_\sigma q'} V(q')\right). \tag{5.3}$$

The transition system T is illustrated by Figure 7. In short, for any choice of interface controller K we will have an upper bound on the directed branching distance (and thus a simulation function), and, fixing that choice of controller, the problem reduces to bounding the output of the error system T. We will tackle this problem with Sum of Squares optimization in the next chapter.

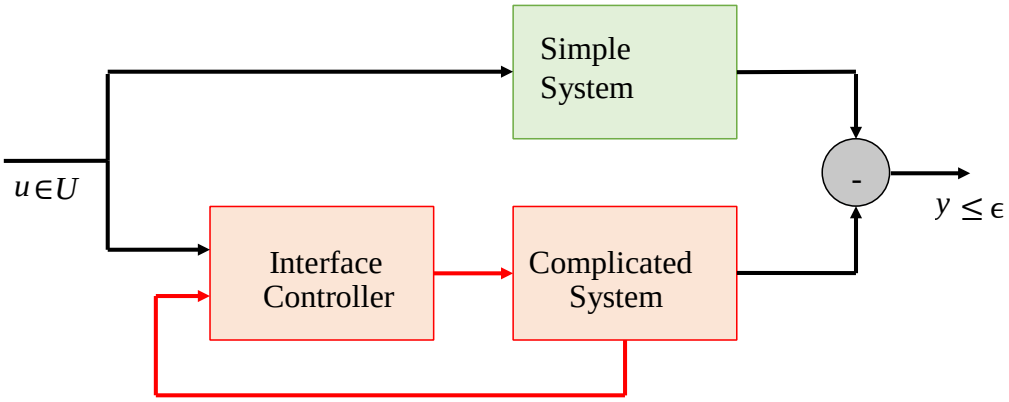


Figure 7: The error system T is formed through a product of the complicated system T_2 with the simple (or abstract) system T_1, and its feedback interconnection with an interface controller K.

Chapter 6: Sum of Squares

In an effort to tame the Curse of Dimensionality, in this chapter we will attempt to search for polynomial simulation functions, given a fixed choice of interface controller. To do this, we will form a convex relaxation (really a stricter, sufficient problem), that in principle will allow us to efficiently search for these polynomials by solving a convex optimization problem. If we succeed in finding such a polynomial, then we will have an easily-checked certificate that one system -simulates another. (If on the other hand the optimization is infeasible, then we will have proven nothing.) The method by which we search for these polynomials is known as Sum of Squares programming an efficient method for finding positive polynomials.

6.1 Introduction

A polynomial p in one or more variables is a sum of squares (SoS) if it is of the form

$$p = \sum_{i=1}^{N} p_i^2 \tag{6.1}$$

where $N \geq 1$ and $p_1,...,p_N$ are arbitrary polynomials. Clearly, such a polynomial is everywhere positive. What is more interesting, however, is that whereas it is NP-hard to test whether an arbitrary multivariate polynomial is pointwise positive, one can test whether it is a sum-of-squares in polynomial time by solving a semidefinite program (SDP). Thus SoS-ness is, in the general (i.e. multivariate) case, a sufficient but not necessary condition for positivity. Our approach, then, will be to form systems of polynomial inequalities, to turn these into positivity constraints on polynomials, and to then to replace those positivity constraints by sufficient (but not necessary) SoS constraints. In fact, the procedure for converting SoS constraints to SDP constraints is relatively simple:

Given an appropriately-large [6] vector $b = (b_1,...,b_n)$ of basis polynomials (e.g., monomials to sufficient degree), p is a sum-of-squares if and only if

$$p = b^T S b \tag{6.2}$$

for positive-semidefinite (PSD) matrix $S = S^T \succ 0$. Thus, constraints that polynomials be SoS, can quickly be reduced to constraints that real matrices be PSD.

6.2 Simulation Function Search

Our algorithm will take the following form:

1. Form product automaton of model (in feedback with interface controller) with abstraction.

[6] There are a number of techniques for reducing the number of basic elements needed, e.g. by exploiting properties of the Newton Polytope; for more information, see [20].

2. Define error output polynomial for the product automaton e.g. $h_2(x_2) - h_1(x_1)$.
3. Choose a polynomial basis (e.g., monomials) in which to search for a candidate simulation function.
4. Generate constraints to link (the coefficients of) various polynomial-valued decision variables.
5. Solve a convex program for the polynomial coefficients.

The main question arises at step 4: What constraints should be formed to define a simulation function for a polynomial hybrid automaton? And what decision variables will we need to define? We answer these questions next.

6.2.1 The Optimization Problem

Decision variables: For each mode $q \in Q$, we define a polynomial decision variable V_q; together these describe the hybrid simulation function $(x, q) \mapsto V_q(x)$. (The degree of the V_q is an algorithm parameter.) Additionally, following the *s-procedure*, we define for each edge $e \in E$ a "Lagrange multiplier" polynomial S_e (again with some chosen degree). Finally, we also introduce a scalar decision variable $\gamma > 0$ to upper-bound the simulation function.

Mode constraints: For each mode $q \in Q$, we define constraints that the following polynomials be SoS:

$V_q(x) - h_q(x)^T h_q(x)$ The simulation function upper − bounds the instantaneous error

$\frac{dV_q}{dx}(x) f_q(x)$ The simulation function is non − decreasing within a mode

$\gamma - V_q(x)$ γ upper − bounds the simulation function

Edge constraints: For each edge $e = (s, d) \in E$, we define constraints that the following polynomials be SoS:

$V_s(x) - V_d(R_e(x)) + S_e(x)\Phi_e(x)$ Meaning: $\Phi(x) \geq 0 \Rightarrow V_s(x) \geq V_d(R_e(x))$

$S_e(x)$

Optimization Problem: Solve,

$$\min_{\{V_q\}_{q\in Q}, \{S_e\}_{e\in E}, \gamma} \gamma \tag{6.8}$$

subject to the preceding constraints.

6.3 Example

Simulation-function certificate generation was demonstrated by comparing two relatively simple two-well polynomial systems of the form,

$$\dot{x} = v$$
$$\dot{v} = -\frac{d}{dx}U_i(x) - \zeta v + u$$

for polynomials U_1,U_2 (potential functions), scalar $\zeta > 0$ (damping factor), and control input u (force). The system is illustrated in Figure 6.1; in this case, the polynomials were given by,

$$U_1(x) = \frac{x^2(x^2-2)+1}{8}$$
$$U_2(x) = \tfrac{1}{2}(x+1)^2 - \tfrac{3}{6}(x+1)^3 + \tfrac{61}{480}(x+1)^4.$$

After the parallel composition of Systems 1 and 2, the certificate polynomial V was computed by sum-of-squares optimization. For a polynomial basis, all monomials of order less than or equal to 8 were used. The YALMIP frontend was used to generate SoS constraints, and the SDPT3 was used to solve the resulting convex optimization problem.

The resulting certificate polynomial V of the joint state (x_1,v_1,x_2,v_2) has 495 terms, a few of which are shown below, together with global upper bound γ on the error, obtained via SoS constraints:

$V(x) = 20.00081495 - 0.0027x_1 + 0.0033v_1 + \cdots - 1.7699e-05x_2v_2^7 - 4.5964e-06v_2^8$

$\gamma = 20.2882$

The polynomial V is plotted in Figure 9.

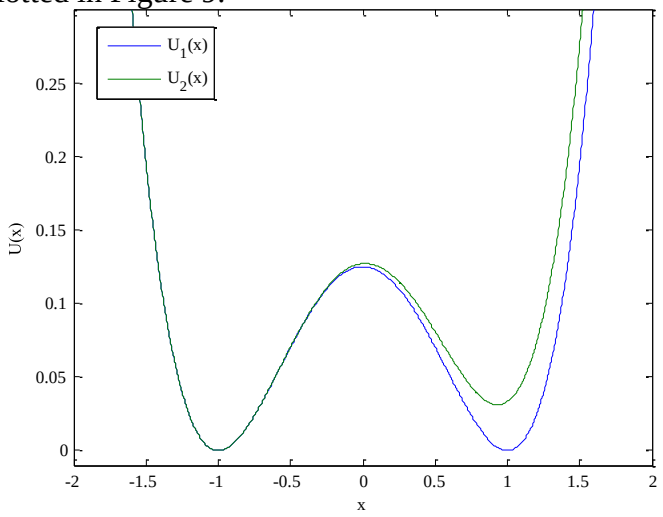


Figure 8: Two similar polynomial potential wells

A few things are worth noting. First, we are able to compute a certificate polynomial; and, moreover, we obtain a global upper bound for the error between the two systems (20.2882). However, note also how ill-conditioned the problem is when using this monomial representation: Products of

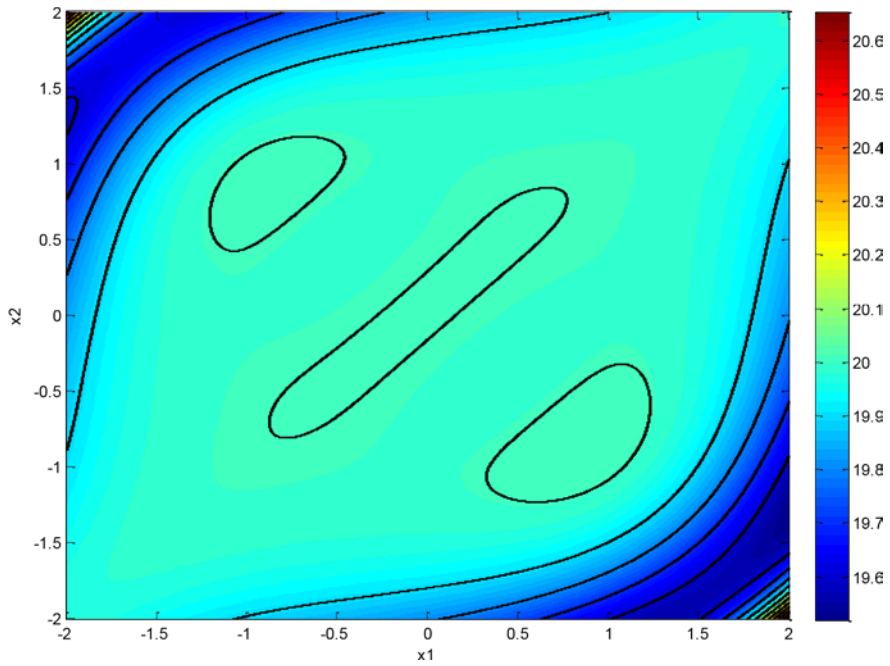


Figure 9: The simulation function polynomial V for the two-well system

very small numbers (e.g. 4.5964e-06) with large powers (in this case v_2^8) appear. This quickly starts to present difficulties for the interior point solvers (e.g. SDPT3) that we use, and as a result the technique will require modifications for use on larger problems. In particular, future research may investigate the use of other polynomial and trigonometric bases.

Chapter 7: Modeling Vehicles and Teams

7.1 Models Overview

In this section, we describe a collection of continuous-time state-space models that are provided as reference models for use with the *TeamBlocks* Framework. The models provided in the current version are tailored for *TeamBlocks* Demonstration 1 (scheduled for December 2015) and are intended to form the basis of a hierarchy of abstraction in multiagent systems that spans low-level vehicle dynamics to higher-level team behaviors.

Each of the examples that follows has a single mode of operation, which maps to a single `Mode` in the *TeamBlocks* schema, without any edges. The dynamics are represented by the `Mode`'s `vector_field` field, which describes a polynomial that is the right hand side of an ordinary differential equation (ODE) in the state and input variables.

7.2 Vehicle Models

Here, we describe vehicle models that appear in TeamBlocks Demonstration 1 abstraction hierarchy.

We begin with the simplest, highest-level kinematic descriptions of vehicles as integrators, and finish with more complicated models of ight dynamics, before proceeding to team abstractions that aggregate these models into larger formations.

7.2.1 Cartesian Single-Integrator

Summary The single-integrator is the simplest kinematic vehicle abstraction. It is useful in situations when dynamics can, relative to the required tolerances, be entirely abstracted by lowerlevel controllers. This model allows the application of the extremely well-developed tools of linear control theory.

Parameters Models in this class are parameterized by the spatial dimension $n \in \{2,3\}$.

Modes The system consists of a single mode.

Inputs The input to the system is an instantaneous velocity command $u \in \mathrm{R}^n = U$.

State Space The state $x \in X = \mathrm{R}^n$ of the vehicle is its n-dimensional position in Euclidean space.

Dynamics The position $x \in X$ evolves according to the rst-order linear ODE $\dot{x} = u$.

7.2.2 Cartesian Double-Integrator

Summary The double-integrator is used to abstract vehicles in cases when their acceleration is bounded but their orientation is unimportant. As in the single-integrator case, systems that can be abstracted to this model can be treated with the many tools of linear control theory.

Parameters Models in this class are parameterized by the spatial dimension $n \in \{2,3\}$.

Modes The system consists of a single mode.

Inputs The input to the system is an instantaneous acceleration command $u \in \mathrm{R}^n = U$.

State Space The state space $X = \mathrm{R}^n \times \mathrm{R}^n$ 3 (p,v) consists of n-dimensional positions and velocities.

Dynamics The state evolves simply according to the linear ODE,

$$\dot{p} = v$$
$$\dot{v} = u\ .$$

7.2.3 SE(2) Integrator

Summary The $SE(2)$ integrator is a kinematic model of a nonholonomic vehicle in two-dimensional space. It is used to abstract ground vehicles, surface vessels, and fixed-wing aircraft constrained to constant-altitude flight.

Modes The system consists of a single mode, containing the ODEs described in the next paragraphs.

Inputs An input $u = (v,\omega) \in \mathrm{R}^2 = U$ to the system consists of a (signed) speed command v, together with a angular velocity command ω.

State Space The state space $X = \mathrm{R}^2 \times SO(2)$ 3 (p,R) of the system consists of all 2d rigid-body positions and orientations.

Dynamics The state evolves according to the ODEs,

$$\dot{p} = vRe_1$$
$$\dot{R} = \omega RJ$$

where

$$J = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

is the 90-degree rotation matrix (named J for its analogy to the imaginary number), and $e_1 = (1,0)$ is the rst element of the natural basis for R^2.

7.2.4 SE(3) Integrator

Summary The $SE(3)$ integrator is a simple kinematic model of a nonholonomic vehicle in three-dimensional space. It is used to abstract fixed-wing aircraft and other vehicles (e.g., unmanned underwater vehicles (UUVs)) that perform three-dimensional maneuvers by steering.

Modes The system consists of a single mode, containing the ODEs described in the next paragraphs.

Inputs An input $u = (v,\omega) \in \mathrm{R}\times\mathrm{R}^3 = U$ to the system consists of a (signed) speed command v, together with a angular velocity command ω.

State Space The state space $X = \mathrm{R}^3 \times SO(3) \sim SE(3)$ 3 (p,R) of the system consists of all 3d rigid-body positions and orientations.

Dynamics The state evolves according to the ODEs,

$$\dot{p} = vRe_1$$

$$\dot{R} = R\Omega$$

where

$$\Omega = \begin{bmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{bmatrix}$$

and $e_1 = (1,0,0)$ is the fi rst element of the natural basis for R^3.

7.2.5 Flight Model

Summary This is a quasi-static model of 6-degree-of-freedom fixed-wing flight with 12 state dimensions. Quasi-static in this case means that the aerodynamic forces are entirely a function of the aircraft's orientation and (angular and translational) velocity, as opposed to also depending on additional state possessed by the surrounding fluid.

Modes The flight model described here consists of a single mode, [7] containing the equations of motion that are described next.

Inputs The input space for the aircraft is $U = \mathrm{R}_{\geq 0} \times \mathrm{R}^3$; an input $u = (u_v,u_\omega)$ consists of,

- $u_v \in \mathrm{R}_{\geq 0}$, the engine thrust, in units of force, and
- $u_\omega = (u_r,u_e,u_a) \in \mathrm{R}^3$, the rudder, elevator, and aileron deflections, in radians.

[7] We anticipate that subsequent models will capture piecewise features of lift and drag curves through separate glide and stall modes, each subject to different aerodynamic forces, and with a passport (or guard) dependent on the angle of attack.

State Space The state space of the aircraft is $X = \mathrm{R}^3 \times SO(3) \times \mathrm{R}^3 \times \mathrm{R}^3$; states consist of

- $p_I \in \mathrm{R}^3$, the inertial position of the aircraft,
- $R_{IB} \in SO(3)$, the rotation from the aircraft's body frame to the inertial frame,
- $v_B \in \mathrm{R}^3$, the body velocity of the aircraft, and
- $\omega_B \in \mathrm{R}^3$, the angular velocity of the aircraft in the body frame.

The subscripts *I* and *B* are used above to denote inertial and body frames, respectively. Additionally, we denote by

$$\Omega_B = \begin{bmatrix} 0 & -\omega_{B,3} & \omega_{B,2} \\ \omega_{B,3} & 0 & -\omega_{B,1} \\ -\omega_{B,2} & \omega_{B,1} & 0 \end{bmatrix}$$

the cross-product matrix corresponding to ω_B.

Dynamics The state variables evolve according to the ODEs,

$$\begin{aligned} \dot{p}_I &= R_{IB} v_B \\ \dot{R}_{IB} &= R_{IB} \Omega_B \\ \dot{v}_B &= M^{-1}(-\Omega_B M v_B + f_v(x) + G_v(x) u_v) \\ \dot{\omega}_B &= J^{-1}(-\Omega_B J \omega_B + f_w(x) + G_\omega(x) u_\omega) \end{aligned}$$

where,

- $f_v(x) = f_v(R_{IB}, v_B, \omega_B)$ and $f_\omega(x) = f_\omega(R_{IB}, v_B, \omega_B)$ are aerodynamic forces and moments, to be specified,
- $M = mI \in \mathbb{R}^{3\times3}$ for $m > 0$ is the aircraft mass matrix,
- $J = J^T \succ 0 \in \mathbb{R}^{3\times3}$ is the body-frame inertia tensor about the aircraft center of mass
- $G_v(x) = G_v(R_{IB}, v_B, \omega_B) \in \mathbb{R}^{3\times1}$ and $G_\omega = G_\omega(R_{IB}, v_B, \omega_B) \in \mathbb{R}^{3\times3}$ are the *decoupling matrices* that relate control surface deflections to moments.

The functions f_v, f_ω, G_v, and G_ω are required to be polynomial for use in the larger framework, and in particular in the Sum of Squares optimization.

7.3 Team Abstractions

When multiple vehicles are combined into a team or platoon, they are then controlled to behave as a single composite system, called the Team Abstraction. The Team Abstraction enables coordinated behavior among the agents, and, because it generally has lower state dimension than

the product of the vehicles' states, it additionally helps to abstract or simplify the state space for planning and verification purposes.

7.3.1 Formation Abstractions

Summary The formation abstraction treats the formation as a single rigid body, together with a number of shape parameters that describe degrees of freedom e.g. the spacing of agents in a line, or the apex angle of a V formation. It is formed as a Cartesian product of an integrator in the shape parameters, with an $SE(3)$ integrator, as described next. [8]

Parameters A formation abstraction is parameterized on,

- a number $m \in \mathrm{N}$ of degrees of freedom for the formation,
- a set $Q \subset \mathrm{R}^m$ of possible shape parameters, and
- a shape function $g = (g_1,...,g_N) : Q \Rightarrow (\mathrm{R}^3)^N$, where N is the number of platforms.

The example of a "V" formation is given later in this section.

Modes The system consists of a single mode.

Inputs An input $u = (v,\omega,\xi) \in \mathrm{R} \times \mathrm{R}^3 \times Q = U$ to the system consists of a (signed) speed command v, an angular velocity command ω, and a shape velocity command ξ.

State space The state of the formation abstraction is a tuple $x = (p,R,q) \in \mathrm{R}^3 \times SO(3) \times Q = X$.

Dynamics The formation abstraction's state evolves according to,

$$\dot{p} = vRe_1$$
$$\dot{R} = R\Omega$$
$$\dot{q} = \xi$$

where again

[8] More generally, a formation abstraction can be formed as a Cartesian product of (a) a model with any dynamics on the shape parameters (e.g., double or single integrator), and (b) a vehicle model whose state space contains an $SE(3)$ subspace (e.g., the aircraft model).

$$\Omega = \begin{bmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{bmatrix}$$

Outputs The formation abstraction generates an output $y = [Rg_i(q) + p]_{i=1}^{N} \in Y = (\mathbb{R}^3)^N$.

Example The "V" formation on *N* platforms, as e.g. in Figure 7.1, is defined by,

- $m = 2$ degrees of freedom,
- shape parameters $(\theta, L) \in (0, \pi] \times \mathbb{R}_{\geq 0} = Q$, that define the V angle and integrant spacing, respectively, and

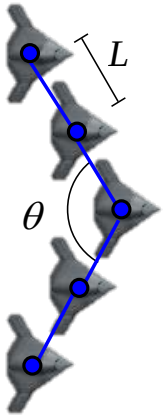


Figure 10: A formation abstraction for "V" -type formations.

- the shape function $g = (g_1,...,g_N) : Q \Rightarrow (\mathbb{R}^3)^N$, defined by

$$\begin{aligned} g_1(\theta, L) &= (0, 0, 0) \\ g_k(\theta, L) &= \tfrac{k}{2}(-\cos(\theta/2), \sin(\theta/2), 0) && k > 1 \text{ is even} \\ g_k(\theta, L) &= \tfrac{k+1}{2}(-\cos(\theta/2), -\sin(\theta/2), 0) && k > 1 \text{ is odd .} \end{aligned}$$

Thus the "V" formation summarizes the state of *N* agents by a total of 8 state dimensions.

Chapter 8: *TeamBlocks* Tools

8.1 GHA Schema

The `tbautomaton` command line tool interacts with the *TeamBlocks* Framework through a serialization schema, which provides the concrete realization of the abstract *TB* automaton. This section describes *TeamBlocks'* first such schema.

One motivation for providing a concrete serialization is that it allows the *TeamBlocks* Framework to support analysis tools implemented in multiple languages. In particular, while *TeamBlocks* now performs LLVM-based code analysis in C++, its current version performs bisimulation function computations separately in MATLAB using the SOSTOOLS toolbox for Sum of Squares programming.

Typical choices for a serialization format include s-expressions, JSON, and Google Protocol Buffers (protobuf). Matlab and C++ both have industry and community support for Google Protocol Buffers and JSON; by using one of these formats, *TB* automaton models can be imported/exported using mature deserialization/serialization libraries. The current version of *TeamBlocks*, including the the `tbautomaton` program, therefore now supports a concrete protobuf serialization of the abstract *TB* automaton. We expect that other tools may interact with `tbautomaton` by reading *TB* automaton objects following the schema defined in this section.

Polynomials Bisimulation function computation, using the Sum of Squares technique, assumes that systems and their controllers are formulated in terms of polynomial expressions. Thus, we define the Expression Schema, shown in Figure 12, together with its required components, to represent these polynomials. Polynomial expressions are naturally expressed in a tree structure where each node in the tree may be a binary operation, such as addition (+), a variable (x), or a numeric literal (1.0). For example, consider the simple two-variable polynomial $p(x,y) := 2x + y$. The tree structure for this polynomial is shown in Figure 11. The leaf nodes of the tree are always a variable or literal, i.e. 2, x, and y. The intermediate nodes represent sub-expressions of the total polynomial.

Using these facts, we define the Expression Schema that is composed of one of the following: 1) a BinaryOperator, 2) a double-valued literal, or 3) a string-valued variable. BinaryOperators are themselves composed of left and right operand Expressions as well as a mathematical Operator: +,−, or ×. These objects are all required to make a valid BinaryOperator object.

Modes and Edges The *TB* automaton definition in Section 3 requires these polynomial Expression objects, as well as several other components. Figure 11 shows the schema for Modes and Edges that represent a *TB* automaton object. Each Mode object has an integer identifier, id, as well as

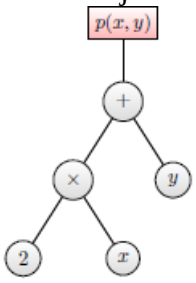


Figure 11: A tree structure representing the expression 2x + y.

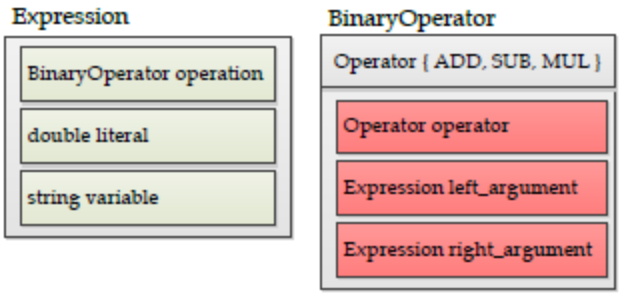


Figure 12: This diagram shows the structure of the Polynomial Schema used in the *TeamBlocks* Framework. Green colored blocks are optional fields. one or more Expression objects that represent the dynamics of the Mode. Edges maintain their source and destination Modes using the integers src_id and dest_id as well as a label string that may be used in synchronization operations. They must also contain a passport expression and one or more transformation expressions.

Schema Realization in protobuf format The current *TeamBlocks* Framework realizes the schema concretely using the Google Protocol Buffers serialization library; the descriptor (.proto) is shown in Figure 12. The process that generates a *TeamBlocks* Automaton in this format is described in the next section.

Schema Realization in MATLAB Corresponding to their protobuf representation, GHAs also have a native MATLAB struct representation as shown in Figure 8.5.

8.2 Functions

8.2.1 Core Functions

tbproduct(tb1, tb2)

- Inputs: two GHA Models.

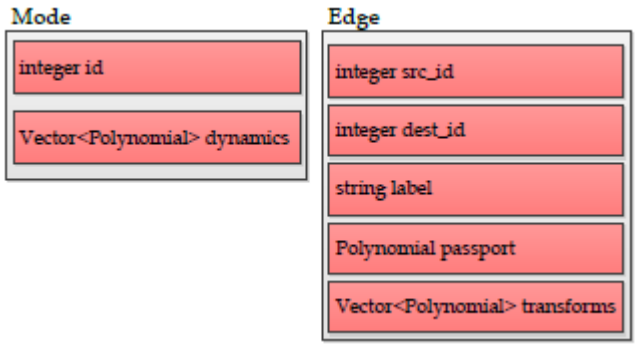


Figure 13: This diagram illustrates the high level structure of TB automaton models. This schema is ingested by optimization tools to perform the bisimulation function search for code certification. Red colored blocks indicate required fields.

- Returns: the composed GHA model.

- This function performs a composition that synchronizes two GHA on edge labels.

tbsim(tb, startModeId, x0, time_variable, input, maxTime, maxIters, display)

- Inputs: A GHA model, the start mode identifier, state variable initial condition(s), the time variable. The input signal, maximum simulation time, maximum iterations, and display are all optional.

- Returns: A struct of the simulation history.

- This function simulates a GHA model from initial conditions until a maximum time or number of iterations.

tbIoConnect(tb, output_names, input_names)

- Inputs: A GHA model, output variable names, and input variable names.

- Returns: A new GHA with the IO variables substituted.

- This function connects the outputs of an GHA model to its inputs based on the given name vectors. For example, the tbdemo_ghaexecution script uses this operation to make feedback system out of the product of the linear system and PI controller.

tbread(lename)

- Inputs: String of a filename.

- Returns: A GHA struct built from the protocol buffer binary file.

- This function loads a protobin schema instance and creates a Matlab representation using the structs described above.

```
import "google/protobuf/descriptor.proto";

package TbMsg;

message TbMsg {
    optional google.protobuf.FileDescriptorSet file_descriptor_set = 1;
    required Mode mode = 2;
    required Edge edge = 3;
}

message Mode {
    required uint64     id           = 1;
    repeated Expression vector_field = 2;
}

message Edge {
    required uint64     src_id    = 1;
    required uint64     dst_id    = 2;
    required string     label     = 3;
    required Expression passport  = 4;
    repeated Expression transform = 5;
}

message Expression {
    optional BinaryOperator operation = 1;
    optional double         literal   = 2;
    optional string         variable  = 3;
}

message BinaryOperator {
    enum Operator {
        ADD = 1;
        MUL = 2;
        SUB = 3;
    };
    required Operator   operator       = 1;
    required Expression left_argument  = 2;
    required Expression right_argument = 3;
}
```

Figure 14: The Protocol Buffer implementation of the schema in Figures 12 & 13.

```
%% Mode Struct
mode = struct();

mode.id                 % String identifier
mode.vector_field       % Vector of polynomials
mode.output_function    % Expression/function
mode.state_variable     % Vector of variables
mode.is_accept          % Boolean
mode.outgoing_edge_idx  % Integer

%% Edge Struct
edge = struct();

edge.src_id             % String
edge.dst_id             % String
edge.label              % String
edge.passport           % Vector of polynomials
edge.transform          % Vector of polynomials

%% The GHA Struct
gha = struct();

gha.mode                % Vector of modes
gha.edge                % Vector of edges
gha.io_variables        % Polynomials
```

Figure 15: The MATLAB struct representation of GHAs. tbCompileToMat(tb, name)

tbCompileToMat(tb, name)

- Inputs: A GHA struct, and name for the output function.
- This function takes a GHA model and compiles it into a set of Matlab functions. One would use this function to make a GHA model execution more performant than running tbsim on the same GHA model.

tbCertificate(tb, degree, tolerance)

- Inputs: A GHA struct, the maximum degree of polynomial used in certification, the constraint checking tolerance.
- Returns: A struct with polynomial objects that represents the squared norm bound on theoutput of the system.
- This function computes, if possible, the polynomial error bound certificate of the GHA.

-

8.2.2 Model Templates

tbclock(rate)

- Inputs: The rate in Hz for the clock.
- Returns: A GHA-based clock model.

tbLinearSystem(A,B,C)

- Inputs: The canonical A, B, and C linear system matrices.
- Returns: A GHA-based linear model.

tbIdealPi(nMeasurement, nControl, kp, ki)

- Inputs: The number of measurement and control signals, proportional gain, and integral gain.
- Returns: A GHA-based model of the theoretical PI controller.

tbSE2Integrator(speed)

- Inputs: Speed of the vehicle.
- Returns: A GHA-based model of an SE(2) vehicle.

tbVee(N,speed)

- Inputs: The number of agents and speed of the formation.
- Returns: A GHA-based model of a collection of SE(2) vehicles in a “vee” formation

8.2.3 LTL Translation and Integration

ltlToTb(ltlCharacterString)

- Inputs: A string of characters that represent an LTL expression.
- Returns: An executable LTL specification automaton using the GHA data structure. Thetbdemo_ghaexecution script illustrates how this function is invoked.

Chapter 9: Conclusion

TeamBlocks introduces theory and software tools that address the modeling, composition, and abstraction of hybrid systems. It does this using a single, unified representation the generalized (polynomial) hybrid automaton (GHA) that represents both continuous systems (i.e., systems described by differential equations) and discrete ones, like control software. Within this framework, one can compose, interconnect, and execute a variety of models, whether generated automatically from C++ code by our tool, or compiled from Linear Temporal Logic (LTL) specifications, or constructed by hand in MATLAB.
The LLVM-based model generator, in particular, is a key product of the *TeamBlocks* effort.

We have also shown that, despite being restricted to polynomials, GHAs are sufficiently expressive to describe a variety of systems of interest, from basic models like linear systems, to standard models of nonholonomic vehicles, to actual C++ implementations of controllers and that these models behave as expected. Indeed, the *TeamBlocks* library currently ships with a number of useful model templates provided. In future, relatively straightforward extensions to include trigonometric factors can only increase the expressiveness of GHA-type models (and may improve numerical stability).

By compiling Linear Temporal Logic (LTL) specifications to monitor automata, we are also able to perform runtime verification efficiently and within the same framework of automaton execution.

The most challenging part of this work has been the development of automatic proof techniques, based on Sum of Squares (SoS) optimization, to find polynomial certificates that verify that one system abstracts another. Here we have developed a formulation of an optimization problem over polynomials as a sum-of-squares optimization problem, which in principle has the advantage of convexity and therefore tractability; moreover, we developed prototype software that is able to search for these polynomials using standard, state-of-the-art solvers. However, the resulting optimization problems appear to be numerically ill-conditioned in practice, which may be a symptom of the use of monomial bases; as a result, the generation of certificates for more complicated systems will likely require the use of other polynomial or trigonometric basis functions, and must be left for future work.

Bibliography

[1] C. Baier and J-P. Katoen. Principles of Model Checking. MIT Press, 2007.

[2] Jon Brodkin. Why gmail went down: Google misconfigured load balancing servers. http://arstechnica.com/information-technology/2012/12/why-gmail-went-down-googlemisconfigured-chromes-sync-server/, December 2012.

[3] C. Cassandras and S. Lafortune. Introduction to Discrete Event Systems. Springer, 2008.

[4] R. Caudil and L. Garrard. Vehicle followed longitudinal control for automated transit vehicles. Journal of Dynamical Systems, Measurements and Control, 18(3), 1969.

[5] E.M. Clarke and E.A. Emerson. Logic of Programs, volume 131 of Lecture Notes in Computer Science, chapter Design and synthesis of synchronization skeletons using branching time temporal logic, pages 52 71. Springer-Verlag, 1981.

[6] G. Fainekos, A. Girard, H. Kress-Gazit, and G. Pappas. Temporal logic motion planning for dynamic robots. Automatica, 45(2):343 352, 2009.

[7] US-Canada Power System Outage Task Force. Final report on the august 14, 2003 blackout in the united states and canada. Technical report, US Department of Energy, 2004.

[8] A. Girard and G. Pappas. Approximation metrics for discrete and continuous systems. IEEE Transactions on Automatic Control, 52(5):782 798, 2007.

[9] A. Girard and G. Pappas. Hierarchical control system design using approximate simulation. Automatica, 45(2):566 571, 2009.

[10] C.A.R. Hoare. Communicating sequential processes. Communications of the ACM, 21(8):666 677, 1978.

[11] J. Kapinski, A. Donzˆ', F. Lerda, H. Maka, S. Wagner, and B.H. Krogh. Control software model checking using bisimulation functions for nonlinear systems. In Proceedings of the 47th IEEE Conference on Decision and Control, pages 4024 4029, December 2008.

[12] Edward A. Lee and Sanjit A. Seshia. Introduction to Embedded Systems, A Cyber-Physical Systems Approach. http://LeeSeshia.org, 2nd edition, 2015.

[13] M. Mazo, A. Davitian, and Paulo Tabuada. Pessoa: towards the automatic synthesis of correctby-design control software. In Hybrid Systems: Computation and Control, 2010.

[14] R. Milner. A Calculus of Communicating Systems. Springer-Verlag, 1980.

[15] US NRC. Backgrounder on the three-mile island accident. http://www.nrc.gov/readingrm/doc-collections/fact-sheets/3mile-isle.html, February 2013.

[16] P.A. Parrilo. Structured semide finite programs and semialgebraic geometry methods in robustness and optimization. PhD thesis, California Institute of Technology, 2000.

[17] D. Peled. Software Reliability Methods. Springer, 2001.

[18] L. Peppard. String stability of relative motion pid vehicle control systems. IEEE Transactions on Automatic Control, 19(5):579 581, 1979.

[19] A. Pnueli. The temporal logic of programs. In 18th IEEE Symposium on Foundations of Computer Science (FOCS), pages 46 67. IEEE Computer Society Press, 1977.

[20] S. Prajna, A. Papachristodoulou, and P.A. Parrilo. Introducing sostools: A general purpose sum of squares programming solver. In Proceedings IEEE Conference on Decision and Control, 2002.

[21] W. Ren and E. Atkins. Second-order consensus protocols in multiple vehicle systems with local interactions. In AIAA Guidance, Navigation, and Control Conference and Exhibit, 2005.

[22] M. Roozbehani, A. Megretski, and E. Feron. Optimization of lyapunov invariants in verification of software systems. IEEE Transactions on Automatic Control, 58(3):696 711, 2013.

[23] Yuki Sugiyama, Minoru Fukui, Macoto Kikuchi, Katsuya Hasebe, Akihiro Nakayama, Katsuhiro Nishinari, Shin ichi Tadaki, and Satoshi Yukawa. Tra c jams without bottlenecks: experimental evidence for the physical mechanism of the formation of a jam. New Journal of Physics, 10:1 7, 2008.

[24] Paulo Tabuada. Verification and Control of Hybrid Systems. Springer, 2009.