SYNT 2015

Workshop on Synthesis

July 18, 2015 (collocated with CAV 2015)

PRE-Proceedings

Preface

This volume contains the papers presented at SYNT 2015: 4th Workshop on Synthesis held on July 17, 2015 in San Francisco.

Each submission was reviewed by at least 3, and on the average 3.8, program committee members. The committee decided to accept 5 papers. The program also includes 2 invited talks, by Sumit Gulwani and Aditya Nori and 2 invited presentations of synthesis competitions.

We are grateful to our sponsor, the project ExCAPE: Expeditions in Computer Augmented Program Engineering.

July 12, 2015
Lausanne

Pavol Cerny
Viktor Kuncak
Madhusudan P.

Table of Contents

Program Committee

Pavol Cerny	University of Colorado Boulder
Colin De-La-Higuera	
Rüdiger Ehlers	University of Bremen
Bernd Finkbeiner	Saarland University
Dana Fisman	University of Pennsylvania
Carlo A. Furia	ETH Zurich
Barbara Jobstmann	EPFL, Jasper DA, and CNRS-Verimag
Viktor Kuncak	EPFL
P. Madhusudan	University of Illinois at Urbana-Champaign
Daniel Neider	RWTH Aachen
Doron Peled	Bar Ilan University
Ingo Pill	Institute for Software Technology, TU Graz
Ruzica Piskac	Yale University
Arjun Radhakrishna	University of Pennsylvania
Leonid Ryzhyk	Carnegie Mellon University
Sven Schewe	University of Liverpool
Ute Schmid	Faculty Information Systems and Applied Computer Science, University of Bamberg
Johann Schumann	SGT, Inc/NASA Ames
Rishabh Singh	MIT
Douglas Smith	Kestrel Institute
Armando Solar-Lezama	MIT
Eran Yahav	Technion
Steve Zdancewic	University of Pennsylvania

Programming by Examples applied to Data Manipulation (Invited Talk)

Sumit Gulwani
Principal Researcher @ Microsoft Research
Adjunct Faculty @ IIT Kanpur
Affiliate Faculty @ Univ. of Washington
sumitg@microsoft.com

Programming by Examples (PBE) involves synthesizing intended programs in an underlying domain-specific language from example based specifications (Espec). In this talk, I will formalize our notion of Espec and describe some principles behind designing appropriate domain-specific languages. A key technical challenge in PBE is to search for programs that are consistent with the Espec provided by the user. We have developed a divide-and-conquer based search paradigm that leverages deductive rules and version space algebras to achieve real time efficiency. Another technical challenge in PBE is to resolve the ambiguity that is inherent in the Espec. We use machine learning based ranking techniques to predict an intended program within a set of programs that are consistent with the Espec. We also leverage active-learning based user interaction models to help resolve ambiguity in the Espec. In this talk, I will demo few PBE technologies (FlashFill, FlashExtract, and FlashRelate) that have been developed using these principles for the domain of data manipulation. These technologies are useful for data scientists who typically spend 80% of their time cleaning data, and for 99% of those end users who do not know programming.

Submitted to:
SYNT 2015, Workshop on Synthesis

Probabilistic Programming: Algorithms, Implementation and Synthesis

Aditya V. Nori
Microsoft Research
adityan@microsoft.com

June 15, 2015

Recent years have seen a huge shift in the kind of programs that most programmers write. Programs are increasingly *data driven* instead of being *algorithm driven*. They use various forms of machine learning techniques to build models from data, for the purpose of decision making. Indeed, search engines, social networks, speech recognition, computer vision, and applications that use data from clinical trials, biological experiments, and sensors, are all examples of data driven programs.

We use the term "probabilistic programs" to refer to data driven programs that are written using higher-level abstractions. Though they span various application domains, all data driven programs have to deal with uncertainty in the data, and face similar issues in design, debugging, optimization and deployment.

In this talk, we describe connections this research area called "Probabilistic Programming" has with programming languages and software engineering—this includes language design, static and dynamic analysius of programs, and program synthesis. We survey current state of the art and speculate on promising directions for future research.

The Second Syntax-Guided Synthesis Competition (SyGuS-COMP 2015)

Rajeev Alur[1], Dana Fisman[1], Rishabh Singh[2], and Armando Solar-Lezama[3] *

[1] University of Pennsylvania
[2] Microsoft Research
[3] Massachusetts Institute of Technology

Abstract. *Syntax-Guided Synthesis (SyGuS)* is the computational problem of finding an implementation f that meets both a semantic constraint given by a logical formula φ in a background theory T, and a syntactic constraint given by a grammar G, which specifies the allowed set of candidate implementations. Such a synthesis problem can be formally defined in SyGuS-IF, a language that is built on top of SMT-LIB.
The *Syntax-Guided Synthesis Competition (SyGuS-COMP)* is an effort to facilitate, bring together and accelerate research and development of efficient solvers for SyGuS by providing a platform for evaluating different synthesis techniques on a comprehensive set of benchmarks. In this year's competition we added two specialized tracks: a track for conditional linear arithmetic, where the grammar need not be specified and is implicitly assumed to be that of the LIA logic of SMT-LIB, and a track for invariant synthesis problems, with special constructs conforming to the structure of an invariant synthesis problem.

* This research was supported by NSF Expeditions in Computing award CCF-1138996

The Second Reactive Synthesis Competition (SYNTCOMP 2015)

Roderick Bloem
Graz University of Technology
Graz, Austria

Swen Jacobs
Saarland University
Saarbrücken, Germany

The reactive synthesis competition (SYNTCOMP) is intended to stimulate and guide advances in the design and application of synthesis procedures for reactive systems in hardware and software. The foundation of stimulating such advancement is a common benchmark format for synthesis problems, and an extensive benchmark library in this format. The first SYNTCOMP was held in 2014.

We report on the design and results of the second SYNTCOMP, held in 2015. We have extended our benchmark library with 6 completely new sets of benchmarks, and additional challenging instances for 4 of the benchmark sets that were already used last year. Overall, we have added more than 2000 problem instances to the existing 569 benchmarks, including many difficult instances. To enhance the analysis of experimental results, we introduce an extension of our benchmark format with meta-information tags that can contain for example a difficulty rating or a reference size for solutions of the benchmark.

Tools are evaluated on a set of 250 benchmarks, selected to provide a good coverage of benchmarks from all classes and difficulties. Like in the previous year, the competition is divided into four different tracks: two tracks for solving the realizability problem and two tracks for synthesis, with one of each in parallel and one in sequential execution mode. In contrast to the previous year, ranking of tools is mainly based on the number of problems that can be solved within the timeout of one hour. Solving time is only used as a tie-breaker, and in the synthesis tracks the size of synthesized artifacts gives raise to an additional quality ranking. Finally, we describe the entrants into SYNTCOMP 2015, as well as the results of our experimental evaluation. In our analysis, we emphasize progress over the tools that participated last year.

Submitted to:
SYNT 2015

Synthesizing a Lego Forklift Controller in GR(1): A Case Study

Shahar Maoz Jan Oliver Ringert
School of Computer Science
Tel Aviv University

Reactive synthesis is an automated procedure to obtain a correct-by-construction reactive system from a given specification. GR(1) is a well-known fragment of linear temporal logic (LTL) where synthesis is possible using a polynomial symbolic algorithm. We conducted a case study to learn about the challenges that software engineers may face when using GR(1) synthesis for the development of a reactive robotic system. In the case study we developed two variants of a forklift controller, deployed on a Lego robot. The case study employs LTL specification patterns as an extension of the GR(1) specification language, an examination of two specification variants for execution scheduling, traceability from the synthesized controller to constraints in the specification, and generated counter strategies to support understanding reasons for unrealizability. We present the specifications we developed, our observations, and challenges faced during the case study.

1 Motivation and Context

Reactive synthesis is an automated procedure to obtain a correct-by-construction reactive system from a given specification, if one exists. The time complexity for synthesis of a reactive system from a linear temporal logic (LTL) formula is double exponential in the length of the formula [18]. However, limited fragments of LTL together with symbolic implementations exhibit more practical time complexities. One such fragment is General Reactivity of rank 1 (GR(1)), where synthesis is possible using a polynomial symbolic algorithm [3, 17].

The availability of efficient synthesis algorithms, as in the case of GR(1), and the guarantee of implementations being correct by construction, motivate applications in software engineering. GR(1) synthesis has been recently used in various application domains, including robotics [11], scenario-based specifications [14], aspect languages [13], and event-based behavior models [4], to name a few.

We conducted a case study to explore the benefits and current challenges of using existing tools and implementations of GR(1) synthesis to develop a reactive system. The research objectives of our case study were to learn about the following questions:

- What are challenges faced when using a GR(1) synthesis tool to synthesize a software controller for a robotic system?
- Is the use of LTL specification patterns helpful to formulate assumptions and guarantees?
- Is it easy to understand reasons for unrealizability?
- Do successfully synthesized controllers work as expected? If not, how can one understand why?

Our goal is to learn what reactive GR(1) synthesis offers and what it lacks to be successfully applied by software engineers in a model-based development process. On a wider scale, we want to understand what is required to make a synthesis specification language and synthesis tools available and accepted by reactive systems software engineers.

Submitted to:
SYNT 2015

It is important to note that our case study concerns the initial development of a specification and synthesized controller. This case study was not about applying new methods and tricks for specification rewriting to make synthesis faster or to optimize the synthesized controller, e.g., as in case studies of the AMBA AHB protocol [2, 7]. When including excerpts of the developed specifications in this paper we decided to not rewrite LTL formulas in a more *clever* way but to present them as written during specification development.

The context of our case study is the synthesis of software for a robotic system. Synthesis is limited to a single controller that interacts with its environment. The specification is written and analyzed by a software engineer and the synthesized controller is used without modification for automatic, direct code generation and deployment to a robotic system. The tools used in our case study are:

- a symbolic GR(1) synthesis algorithm implementation from [3] using the JTLV framework [19] including the synthesis of counter strategies for unrealizable specifications;
- the AspectLTL [13] input language, with syntax similar to SMV, for specifying environment assumptions and system guarantees, with syntax highlighting and code completion;
- an implementation for traceability between a synthesized controller and its temporal specification based on [16]; and
- a catalog of LTL specification patterns [5] and their GR(1) templates [12].

The GR(1) synthesis problem is to find a controller that realizes a given specification over a set of environment variables and a set of system variables. A GR(1) specification consists of:

- constraints on initial assignments (constraints without temporal operators),
- safety constraints over current and next assignments (constraints of the form `G (exp)` where the expression `exp` is limited to past time temporal operators and the **`next`** operator), and
- liveness constraints, more specifically justice constraints, that should hold infinitely often (constraints of the form `G F (exp)` where the expression `exp` has no temporal operators).

All constraints are either assumptions, i.e., obligations of the environment, or guarantees, i.e., obligations of the system. Intuitively, if all assumptions are satisfied by the environment a synthesized controller satisfies all guarantees. If no such controller exists, the specification is called unrealizable. A detailed introduction to LTL, past time LTL, and GR(1) synthesis is available from [3].

In this case study we use an extension of the GR(1) synthesis problem where assumptions and guarantees may also be specified using LTL specification patterns [5]. The extension is described in [12].

Apart from the above GR(1) synthesis implementations we have used the MontiArcAutomaton language [21] for modeling the components of the robotic system and as a concrete syntax for the synthesized controllers. We have used code generators from this intermediate representation to Java for deployment to the Lego Mindstorms NXT platform[1].

2 Case Design and Variant

The task of the case study was to develop a specification for a Lego forklift robot shown on the left side of Figure 1. The forklift is an actual Lego robot[2] we have constructed and experimented with in our lab.

[1] LEGO Mindstorms website: `http://www.lego.com/en-us/mindstorms`

[2] Robot based on these building instructions: `http://www.nxtprograms.com/NXT2/forklift/steps.html`

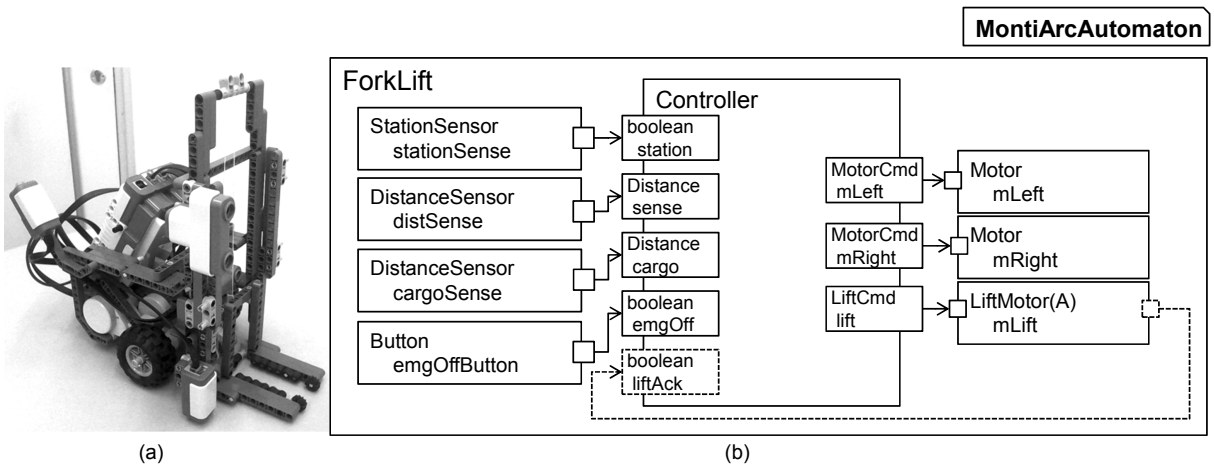


Figure 1: (a) The Lego forklift robot with four sensors and three actuators. (b) The logical software architecture of the robot with wrappers for sensors and actuators and the main component `Controller` to be synthesized (data types of input and output ports defined in Figure 2). The dashed elements describe a second variant with feedback of the lift motor to acknowledge completion of lifting or dropping the fork.

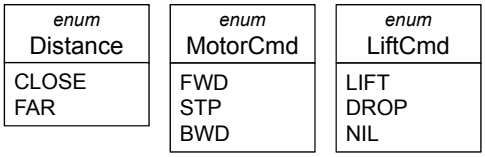


Figure 2: Data type definitions for ports of component `Controller` shown in Figure 1

A criterion for success of synthesis was not only to obtain and inspect a synthesized controller but also to deploy it to the real robot and see that its behavior makes sense.

The forklift shown in Figure 1 has a sensor to determine whether it is at a station, two distance sensors to detect obstacles and cargo, and an emergency button to stop it. It has two motors to turn the left and right wheels and one motor to lift the fork. The case definition consists of an initial set of informal requirements for the behavior of the forklift:

1. Do not run into obstacles.
2. Only pick up or drop cargo at stations.
3. Do not attempt to lift cargo if cargo is lifted.
4. Always keep on delivering cargo.
5. Never drop cargo at the station where it was picked up.
6. Stop moving if emergency off switch is pressed.

Formalization and refinement of these requirements into guarantees and the elicitation of suitable assumptions was part of the case study execution.

Together with the above list of informal requirements the logical software architecture of the forklift was defined before case study execution. It is depicted as a component and connector model in Figure 1 (b). The components on the left side are hardware wrappers that read sensor values and publish

them as messages on their output ports. The output ports of the sensor components are connected to input ports of component `Controller`. The output ports of component `Controller` are connected to three components on the right that receive commands and encapsulate access to the motors of the forklift. The datatypes of input and output ports as well as their names are written on the ports of component `Controller`. Datatypes other than `boolean` are defined as enumerations in Figure 2.

The execution of the robot is performed in a cycle: read sensor data, execute controller, perform actions. When deploying the synthesized controller on the forklift robot the output produced by transitions between states of the controller manipulates the robots environment through its actuators. We decided on two scheduling variants to integrate the synthesized controller and the real world.

V1.Delayed In the first variant the execution of transitions of the synthesized controller is delayed to give the physical robot enough time to execute tasks of the actuators. The delay has to be large enough to, e.g., completely lift or drop the fork, complete a 90 degrees turn, or back up from an obstacle. We set the delay to 2000ms.

V2.Continuous In the second variant the controller is executed continuously without any delay. This variant uses a technique inspired by Raman et al. [20] to synthesize a reactive controller for continuous control. The setting requires the controller to be aware whether actions have completed or not. In our case driving actions get feedback from distance sensors but an additional feedback signal had to be added from the lift motor to acknowledge completion of its actions (shown as a dashed line in Figure 1).

Depending on the scheduler and actuator implementations more variants are possible, e.g., a variant where the scheduler pauses execution of the synthesized controller while an actuator performs a task (e.g., similarly by Kress-Gazit et al. [10]).

3 Resulting Specifications

Development of the specifications started with the first variant **V1.Delay** in 7 versions with incremental addition of features. When the first variant was almost complete the development of variant **V2.Continuous** started based on the existing specification. The final specifications for both variants are available from [22].

The first and common part to both specifications is a schematic translation of the input and output ports of component `Controller` shown in Figure 1 to environment variables (`VARENV`) and system variables (`VAR`) declared in Listing 1. The following excerpts of the two specifications refer to variable names and short names defined in the `DEFINE` block of Listing 1. The only difference between the variable declarations for **V1.Delay** and **V2.Continuous** is the environment variable `liftAck` in line 6 added only in variant **V2.Continuous**.

The two system variants are however very different. The main difference between both systems is that the first variant of the forklift is scheduled in steps of 2000ms (it waits for 2 seconds after each read input, compute, write output iteration), while the second variant does not use any delay. To illustrate this difference: when executing the controller synthesized from **V1.Delay** without the delay the forklift runs over cargo it was expected to have lifted up; it may initiate dropping cargo at a station but only completes dropping it after leaving the station; during this process it detects the dropping cargo as an obstacle and tries to drive clear from it.

```
1  VARENV
2    cargo : { CLEAR, BLOCKED };
3    sense : { CLEAR, BLOCKED };
4    station : boolean;
5    emgOff : boolean;
6    liftAck : boolean; -- only in V2.Continuous
7
8  VAR
9    mLeft : { FWD, STOP, BWD };
10   mRight : { FWD, STOP, BWD };
11   lift : { LIFT, DROP, NIL };
12
13 DEFINE -- defines to ease reading specs
14   backing := mLeft = BWD & mRight = BWD;
15   stopping := mLeft = STOP & mRight = STOP;
16   turning := mLeft = BWD & mRight = FWD | mLeft = FWD & mRight = BWD;
17   forwarding := mLeft = FWD & mRight = FWD;
18   dropping := lift = DROP;
19   lifting := lift = LIFT;
```

Listing 1: Environment variables (`VARENV`), system variables (`VAR`), and definitions of short names (`DEFINE`) of the controller to synthesize as an implementation of component `Controller` from Figure 1 (b)

3.1 Specifications

We give an overview of both specifications developed as part of the case study. We start with **V1.Delay** and later discuss the changes in **V2.Continuous** compared to **V1.Delay**.

V1.Delay: Assumptions and Guarantees

The specification document for **V1.Delay** contains 7 assumptions, 12 guarantees, and 1 auxiliary variable.

Of the 7 assumptions, 1 is a safety constraint, and 6 are LTL specification patterns: 5 instances of the response pattern P26 and 1 instance of a bounded existence pattern P15 (patterns numbered as appearing on the website of [5] and in our catalog of GR(1) templates [12]). An example for a typical response pattern is shown in Listing 2, ll. 1-3. The pattern expresses the assumption that if the robot is backing or turning it will reach a state where both distance sensors are clear unless it decides to go forward or stop, i.e., the robot may *escape* obstacles. A safety assumption about the environment is that the reading of the sensor `station` will not change when the forklift is stopping (see Listing 2, ll. 5-6). Another assumption is formulated using pattern P15 is shown in Listing 2, l. 8-9. It expresses that the robot will encounter at most two low obstacles between stations.

Of the 12 guarantees of **V1.Delay**, 1 constrains the initial state, 8 are a safety constraints, 1 is a justice constraint, and 2 are LTL specification patterns (P09 and P20). One auxiliary variable is defined (we treat the variable definition and assignments as a special case although the assignments are currently implemented as four safety guarantees). Our current implementation distinguishes manually added auxiliary variables from other variables by the prefix `spec_` preceding the name of the variable.

The specification uses pattern P09 as a guarantee that the forklift has to leave the station between lifting cargo and delivering it (Listing 3, ll. 1-2). Another guarantee uses pattern P20 to express that after leaving a station cargo cannot be dropped until the forklift reaches a station (Listing 3, ll. 4-5).

```
ASSUMPTION -- backing or turning clears sensors
  Globally (backing | turning) leads to
               ((sense=CLEAR & cargo=CLEAR) | forwarding | stopping)); --P26

ASSUMPTION -- station does not change when stopping
  G (stopping -> station = next(station));

ASSUMPTION -- at most blocked twice between stations
  After (!atStation) have at most two (lowObstacle) until (atStation); --P15
```

Listing 2: Three assumptions of variant **V1.Delay** using LTL specification patterns [5] with equivalent representations in GR(1) [12]

```
GUARANTEE -- have to leave station to deliver
  (!atStation) occurs between (lift=LIFT) and (lift=DROP);  --P09

GUARANTEE -- don't drop cargo after leaving station until arriving
  Globally (lift!=DROP) after (!atStation) until (atStation) --P20
```

Listing 3: Two guarantees of of variant **V1.Delay** using LTL specification patterns [5] with equivalent representations in GR(1) [12]

```
VAR -- new auxiliary variable to "remember" when cargo is loaded
  spec_loaded : boolean;
GUARANTEE
  !spec_loaded; -- initial value false
  G (lifting -> next (spec_loaded)); -- set loaded when lifting
  G (dropping -> ! next (spec_loaded)); -- unset loaded when dropping
  G (lift = NIL -> next (spec_loaded) = spec_loaded); -- preserve value
```

Listing 4: Definition of the auxiliary variable `spec_loaded`

```
1  ASSUMPTION -- find station to drop when forwarding (unless backing or stopping)
2    Globally (forwarding & spec_loaded) leads to
3                                    ((station & cargo=CLEAR) | backing | stopping));
4
5  ASSUMPTION -- find station and cargo when forwarding (unless backing or stopping)
6    Globally (forwarding & !spec_loaded) leads to
7                                  ((station & cargo=BLOCKED) | backing | stopping));
8
9  GUARANTEE -- do not lift again when cargo is loaded
10   G (spec_loaded -> !lifting);
```

Listing 5: An assumption and a guarantee of variant **V1.Delay** using the auxiliary variable `spec_loaded` introduced in Listing 3

```
1  GUARANTEE -- emergency stop
2    G (emgOff -> (stopping & lift=NIL));
3
4  GUARANTEE -- main goal to always eventually deliver cargo
5    G F ((lift = DROP) | emgOff | lowObstacle);
```

Listing 6: More guarantees of variant **V1.Delay**: the emergency stop feature and the main justice constraint of the forklift

In Listing 4 a new auxiliary variable `spec_loaded` is introduced for the purpose of simplifying the specification. This variable is used by the specifier to keep track on whether the forklift has loaded cargo or not. The value of the variable is determined by safety constraints. The variable is used in the specification in both assumptions and guarantees as shown in Listing 5. The assumption in lines 1-3 expresses that if the forklift goes forward and has cargo loaded it will eventually find a station where it can deliver cargo unless it stops or goes backward.

Two additional guarantees in Listing 6 describe the emergency stop feature of the forklift and the main justice constraint to always eventually deliver cargo. The main justice constraint of the forklift is shown in Listing 6, ll. 4-5. It does not only contain the expected `lift = DROP`, i.e., delivering cargo, but also the alternatives `emgOff` and `lowObstacle`. These alternatives represent environment actions that if occurring infinitely often liberate the forklift from its obligation.

V2.Continuous: Assumptions and Guarantees

We now describe the differences between the specification of the first variant **V1.Delay** and the second variant **V2.Continuous** of the forklift specifications. The second variant applies a method inspired by Raman et al. [20] for continuous control. The main idea is to add a new variable for every action which signals completion of the action. In our case study this method is only necessary for the completion of lifting and dropping cargo. The environment variable `liftAck` is added (see Listing 1) to signal completion of the actions of component `LiftMotor` (see Figure 1).

The specification document for variant **V2.Continuous** contains 9 assumptions, 14 guarantees, and 2 auxiliary variable definitions. None of the assumptions or guarantees of **V1.Delay** were removed, three assumptions were added, and two guarantees were added.

Of the 2 added assumptions, 1 is a safety constraint, and 1 is an instance of pattern P26. The auxiliary variable `spec_waitingForLifting` is declared and completely defined in Listing 7, ll. 9-15 in the same way as `spec_loaded`. The variable is **`true`** iff a lifting command was issued and completion has not been acknowledged yet. The assumption in Listing 8, ll. 1-2 expresses that an acknowledgment

```
VAR
  spec_loaded : boolean;
GUARANTEE -- updated assignments of var spec_loaded
  !spec_loaded;
  G (liftAck & !spec_loaded -> next(spec_loaded));
  G (liftAck & spec_loaded -> next(!spec_loaded));
  G (!liftAck ->  spec_loaded = next(spec_loaded));

VAR -- variable to keep track of waiting for lifting
  spec_waitingForLifting : boolean;
GUARANTEE
  !spec_waitingForLifting;
  G ((lift=LIFT | lift=DROP) -> next(spec_waitingForLifting));
  G (liftAck -> ((lift=LIFT | lift=DROP) | next(!spec_waitingForLifting)));
  G (!((lift=LIFT | lift=DROP) | liftAck) -> spec_waitingForLifting = next(
      spec_waitingForLifting));
```

Listing 7: Auxiliary variable definitions in **V2.Continuous**: the assignments of `spec_loaded` were modified and variable `spec_waitingForLifting` was added

```
ASSUMPTION -- always eventually complete lifting
  Globally (spec_waitingForLifting) leads to (liftAck);

ASSUMPTION -- only acknowledge if waiting for it
  G (next(liftAck) -> (spec_waitingForLifting & !liftAck));
```

Listing 8: New assumptions added in variant **V2.Continuous** about acknowledgments of executing the `LIFT` and `DROP` commands

`liftAck` will eventually follow every waiting. A second assumption ensures that acknowledgments are only sent if `spec_waitingForLifting` is **`true`**. The assignment of variable `spec_loaded` has been modified. The assignment now depends on the previous value of variable and the new input `liftAck` (see ll. 1-6).

Both added guarantees are safety constraints, as shown in Listing 9. The first expresses that the forklift stops when it waits for the completion of a lifting action (see Listing 9, ll. 1-2), and the second expresses that it does not issue new lifting commands when waiting for completion (see Listing 9, ll. 4-5).

3.2 Synthesis Times and Controller Sizes

For both variants a controller that implements the specification can be synthesized in a few seconds using the Java-based BDD engine that comes with JTLV [19]. The sizes of both specification variants are summarized in the upper half of Table 1. We report the assumption and guarantee constraints, the total number of variables including auxiliary variables (the state space referred to as N in the time complexity $O(nmN^2)$ of GR(1) synthesis is 2 to the power of the number of Boolean variables required to repre-

```
GUARANTEE -- while lift performs action stop
  G (spec_waitingForLifting -> stopping);

GUARANTEE -- do not send another request when waiting for completion
  G (spec_waitingForLifting -> lift=NIL);
```

Listing 9: Added guarantees of variant **V2.Continuous**

	V1.Delay	V2.Continuous
Assumptions (patterns)	1 safety, 5 times P26, P15	2 safety, 6 times P26, P15
Guarantees (patterns)	1 initial, 8 safety, 1 justice, P09, P20	1 initial, 8 safety, 1 justice, P09, P20
Boolean Variables (auxiliary)	4 environment, 6 system, 1 manual, 12 pattern	5 environment, 6 system, 2 manual, 13 pattern
Checking Realizability	0.2 sec	0.7 sec
Controller Construction	1.8 sec	1.3 sec
States of Controller	3412	2888

Table 1: For both variants we report the size of the specification, times for checking realizability and controller construction in seconds, and the size of a synthesized controller.

sent environment, system, and auxiliary variables). Environment and system variables amount to 2^{10} in **V1.Delay** and to 2^{11} in **V2.Continuous**. For the specifications of variant **V1.Delay** and **V2.Continuous** developed in this case study 13 and 15 auxiliary variables were automatically (and implicitly) added to support specification patterns or explicitly added as auxiliary variables in the specification. The GR(1) synthesis problem after the translation of patterns to GR(1), via the templates described in [12], had 6 environment and 3 system justice goals in variant **V1.Delay** and 7 environment and 3 system justice goals in variant **V2.Continuous**.

In the lower half of Table 1 we report the time it took the synthesis algorithm to decide realizability and the additional time consumed by the controller construction phase, in seconds. Times are shown as reported by JTLV running on an ordinary laptop with Java 7, Windows 7 64bit, 8GB RAM, and an Intel i7 CPU with 3.0 GHz. Synthesis times for all intermediate versions during specification development confirmed that synthesis is conveniently fast.

3.3 Running Synthesized Controllers on the Forklift

We used code generators implemented in our group for the NXJ LeJOS platform[3] to generate code and directly deploy the software components shown in Figure 1 (b) to the LEGO NXT forklift shown in Figure 1 (a). The components `StationSensor` (light value measured on the ground), `Distance-Sensor` (ultrasonic distance sensor), and `Button` (touch sensor) have generic implementations in Java that wrap the LeJOS sensor API. The components `Motor` and `LiftMotor` have generic implementations that wrap the LeJOS motor API.

To execute the synthesized controller of variant **V1.Delay** on the forklift we had to adapt a set of platform specific parameters: number of degrees motors `mLeft` and `mRight` rotate backward and forward, number of degrees the lifting motor rotates, distance to obstacles detected by `distSense`, and distance to cargo detected by `cargoSense`. In the second variant **V2.Continuous** the motors `mLeft` and `mRight` do not rotate a fixed amount of degrees but move continuously.

The components on the robot are executed in execution cycles. Every execution cycle starts with reading all sensor values, executes the controller, and ends with executing all actuators. In variant **V1.Delay** we added a fixed delay of 2000ms after the execution of the actuators to allow the forklift finishing all actions. In variant **V2.Continuous** a delay was not added and the execution time of one

[3] Website of LeJOS NXJ: `http://www.lejos.org/nxj.php`

cycle on the robot was around 50ms.

We observed that variant **V1.Delay** provided more reliable sensor readings than **V2.Continuous** because it measured values after completing its actions in a resting position. For variant **V2.Continuous** erroneous sensor readings had serious effects. As one example, when the station sensor falsely detected a station the forklift stopped to drop cargo. A second reading when stopped did not report the station and thus caused a violation of an environment assumption (Listing 2, ll. 5-6), and so the forklift went into an infinite loop of stopping. As another example, unstable readings of the ultrasonic sensor led to detecting and not detecting obstacles at the sensor's threshold level. This happened when the forklift was not at a station and led to a violation of the assumption that at most two obstacles occur (Listing 2, ll. 8-9). In this case the forklift kept going forward in an infinite loop not stopping at obstacles anymore.

Some works have addressed the challenge of synthesizing controllers which are more robust to assumption violations, e.g., Ehlers and Topcu [6] suggested an approach where the synthesized controller allows violations of safety assumptions up to some constant number of times.

4 Observations and Challenges

We now report some of our observations and challenges faced during specification development.

O1: Differences between V1.Delay and V2.Continuous

The way that the synthesized controllers of variants **V1.Delay** and **V2.Continuous** are executed and interact with their environment is fundamentally different. Thus, we found it surprising that their GR(1) specifications are still very similar. Only three assumptions and two guarantees were added. One reason for the similarity is that **V2.Continuous** is based on **V1.Delay**. It is still interesting that most existing assumptions and guarantees also remain valid for the second variant, although it is based on a continuous execution scheduling, without delays. The main reason we did not have to adapt many assumptions seems to be the use of the response patterns already in **V1.Delay** (five instances) instead of explicitly referring to immediate successor states.

O2: Manually adding auxiliary variables helpful

During the development of the two specifications we found it very helpful to add auxiliary variables, as they assisted us in making relevant states explicit. As an example, the auxiliary variable `spec_loaded` (Listing 4) is defined to be **`true`** iff the forklift has loaded cargo. This information is derived and not provided as a sensor input. The variable `spec_loaded` appears in two assumptions and two guarantees. The past LTL formula

```
PREV (lift!=DROP SINCE lift=LIFT)
```

could replace the auxiliary variable but we believe that it helps readability of the formulas to make the property explicit with a new name.

Technically, adding an auxiliary variable to the specification requires that its value is determined by complete and deterministic safety constraints. This mechanism is discussed by Bloem et al. [3] for supporting past LTL in GR(1) synthesis.

O3: Support for patterns helpful

LTL specification patterns [5] allow one to express high-level temporal patterns in a convenient way that are otherwise complicated and error-prone to express in LTL. In the limited fragment of GR(1) correctly expressing these behaviors becomes even more complicated due to the limitation of available operators and no nesting. We provide GR(1) templates for 52 of the 55 LTL specification patterns [12].

During the case study we found that using patterns is very helpful for expressing more complicated temporal properties. Moreover, using patterns gave us better confidence that the specification matches our intention.

A specifically useful pattern in assumptions was the response pattern P26. Instances of this pattern appeared as 5 out of 7 assumptions in **V1.Delay** (6 out of 9 in **V2.Continuous**). This pattern seems to be well suited for describing robotic systems where one should assume that some actions of actuators eventually have an impact on sensor values.

It is important to note that the addition of patterns comes at a price. Most patterns cannot be expressed without the addition of auxiliary variables. In both variants of our case study more than half of the Boolean variables encoding the statespace were auxiliary variables, implicitly added in the translation of the patterns to the GR(1) form based on our templates. The resulting synthesis problems of the case study are however still solved in a few seconds.

C1: Environment vs. real environment

During the case study it turned out that it is difficult to use assumptions to describe realistic environments. A very simple example that appeared early during development of **V1.Delay** is the assumption that turning of the robot will make the distance sensor signal clear:

```
G (turning -> next(sense=CLEAR));
```

Our inspection of the synthesized controller concluded positively. When deploying the controller to the forklift it turned and stopped in a corner of the room. The assumption was too strong for the forklift's real environment. In a corner, turning 90 degrees leaves the robot facing another wall. The response pattern `Globally (turning) leads to (sense=CLEAR)` would solve this particular problem but the expressed assumption is still too strong and we observed another undesired behavior: after being blocked the forklift turned once but then stopped and waited for the obstacle to disappear. A corrected version (also handling driving backwards) is shown in Listing 2, ll. 1-3.

A different challenge appeared when adding a guarantee that the robot should pick up cargo and not drop it between stations:

```
Globally (lift!=DROP SINCE lift=LIFT)
                              after (!atStation) until (atStation);
```

The addition of this guarantee made the specification unrealizable. The past time formula `lift!=DROP SINCE lift=LIFT` embedded in the specification pattern requires that the forklift leaves a station only when cargo has been picked up. We decided that it might be reasonable to assume that at every station the forklift may pick up cargo. It turned out too complicated for us to formulate this assumption in GR(1). A possible assumption might require encoding the area of each station and valid navigation of the forklift to allow it to explore the station for cargo. Finally, we weakened the above guarantee to the one in Listing 3, ll. 4-5. The modified version allows the forklift to leave stations without cargo.

To summarize, in the case study we faced both the challenge of formulating assumptions that are too strong for the real environment of the forklift and the challenge of not being able to formulate reasonable assumptions due to difficulties in expressing them. Both cases are not easy to address. Specifically the first can lead to successful synthesis of a controller that fails in a real environment.

C2: Undesired realizable case: system forces environment to violate assumptions

When the system can force the environment to violate its assumptions the controller often does not act as the engineer would have expected it to act. Consider the following two assumptions from an early version of **V1.Delay**. The first assumption is that if the forklift does not move the station sensor value will not change. The second assumption is that the forklift will eventually leave a station if it moves forward.

```
G (stopping -> station = next(station));
Globally (forwarding) leads to (!station);
```

We synthesized a controller and in some cases the forklift running the controller stopped and did not continue to move. To understand this behavior in the example above we enabled our synthesis tool to annotate every transition with one of three reasons for it to be included in the controller. The possible reasons are [3]: satisfying a justice constraint, working towards satisfying a justice constraint, or preventing a justice constraint of the environment. The annotation of each transition of the controller includes the justice constraint from the specification (see [16] for more details about traceability). Using this traceability information, which links transitions in the controller to elements of the specification, we learned the reasons for the synthesized strategy of the controller. We found out that in the example above, the forklift goes forward on a station. If it is still on the station in the next step it stops forever and thus prevents the environment from satisfying the justice constraint of the response pattern.

In this simple example with a controller of 80 states we were able to find an explanation we could trace to the specification and modify the response pattern as shown in Listing 5, ll. 5-7. It would have been helpful to automatically find cases where the system forces the environment to violate assumptions for an engineer to decide whether this behavior is desired or not. Towards the end of our specification development process synthesized controllers simply became too large for manual inspection.

Klein and Pnueli [8] defined environments where the system cannot force a violation as *well-separated*. They suggest a modified GR(1) game to check whether an environment is well-separated. This may be a direction towards addressing this challenge.

C3: Unrealizable case

In many cases adding a new feature expressed as a set of assumptions and guarantees led to an unrealizable specification. In some cases we were able to find mistakes quickly in the added assumptions and guarantees. Many times we initially forgot to add alternatives in safety constraints leading to their unsatisfiability by the system. In more complicated cases we asked our tool to synthesize a counter strategy that represents an environment forcing all system strategies to lose. We (interactively, as described in [16, 15]) executed moves of our intended system and learned where our strategy loses against the environment. This often led to better understanding of the reasons for unrealizability.

In some cases it was however very difficult to understand the reason for unrealizability, trace it back to the specification, and fix it. As an example, consider the introduction of the new environment variable

`liftAck` and the auxiliary variable `spec_waitingForLifting` (see Listing 7). We added the assumption that lifting can only be acknowledged if the controller is expecting it:

```
G (next(liftAck) -> spec_waitingForLifting);
```

The specification of variant **V2.Continuous** with the above assumption is unrealizable. A synthesized counter strategy has 3735 states. The Java code generated by our synthesis tool for interactive exploration of the counter strategy failed to compile due to its size. Printing the counter strategy including information on successors disabled by safety properties ran out of memory. It was not easily possible to reduce the synthesis problem by removing irrelevant parts from the specification. Lifting of cargo is related to movement of the motors and the cargo sensor. The movement of motors is related to the distance sensor.

We manually executed the counter strategy by inspecting the generated text output. Many steps were repetitive (long chains of apparently similar states) as we were working towards forcing the environment to present a station with cargo, and after lifting forcing it to acknowledge lifting. Right after acknowledging lifting the environment acknowledged lifting again. The double acknowledgment set and immediately afterwards disabled the variable `spec_loaded` (see Listing 7, ll. 5-6). The double acknowledgment was possible because `liftAck` disables `spec_waitingForLifting` only in the next step (see Listing 7, ll. 14). We adapted the above assumption as shown in Listing 8, l. 5.

While counter strategies help understanding reasons of unrealizability their handling by our tools turned out to be insufficient for larger specifications. In the future we plan to examine how recent work by others, e.g., [1, 9], may help in addressing the unrealizability challenge.

5 Conclusion

We have presented a case study of the development of a software controller for a forklift robot using GR(1) synthesis tools. Rather than examining how to write most elegant and efficient specifications we focused on challenges for software engineers in the process of specification development. We showed the specifications of two variants of the controller. On the one hand, our observations are that extensions of the specification language with auxiliary variables and higher-level specification patterns support writing specifications with better confidence. On the other hand, with growing specification size, understanding reasons for synthesized behavior and for unrealizability turned out to be major challenges.

This case study is part of our larger project on bridging the gap between the theory and algorithms of reactive synthesis on the one hand and software engineering practice on the other. In many aspects it demonstrates the different challenges awaiting us.

Acknowledgments The authors thank the anonymous reviewers of the SYNT 2015 workshop for their helpful comments. Jan O. Ringert acknowledges support from a postdoctoral Minerva Fellowship, funded by the German Federal Ministry for Education and Research. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 638049, SYNTECH).

References

[1] Rajeev Alur, Salar Moarref & Ufuk Topcu (2013): *Counter-strategy guided refinement of GR(1) temporal logic specifications*. In: *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR,*

USA, October 20-23, 2013, IEEE, pp. 26–33. Available at http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6679387.

[2] Roderick Bloem, Swen Jacobs & Ayrat Khalimov (2014): *Parameterized Synthesis Case Study: AMBA AHB*. In Krishnendu Chatterjee, Rüdiger Ehlers & Susmit Jha, editors: *Proceedings 3rd Workshop on Synthesis, SYNT 2014, Vienna, Austria, July 23-24, 2014.*, *EPTCS* 157, pp. 68–83. Available at http://dx.doi.org/10.4204/EPTCS.157.9.

[3] Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli & Yaniv Sa'ar (2012): *Synthesis of Reactive(1) Designs*. *J. Comput. Syst. Sci.* 78(3), pp. 911–938. Available at http://dx.doi.org/10.1016/j.jcss.2011.08.007.

[4] Nicolás D'Ippolito, Víctor A. Braberman, Nir Piterman & Sebastián Uchitel (2013): *Synthesizing nonanomalous event-based controllers for liveness goals*. *ACM Trans. Softw. Eng. Methodol.* 22(1), p. 9. Available at http://doi.acm.org/10.1145/2430536.2430543.

[5] Matthew B. Dwyer, George S. Avrunin & James C. Corbett (1999): *Patterns in Property Specifications for Finite-State Verification*. In: *ICSE*, ACM, pp. 411–420.

[6] Rüdiger Ehlers & Ufuk Topcu (2014): *Resilience to intermittent assumption violations in reactive synthesis*. In Martin Fränzle & John Lygeros, editors: *17th International Conference on Hybrid Systems: Computation and Control (part of CPS Week), HSCC'14, Berlin, Germany, April 15-17, 2014*, ACM, pp. 203–212, doi:10.1145/2562059.2562128. Available at http://doi.acm.org/10.1145/2562059.2562128.

[7] Yashdeep Godhal, Krishnendu Chatterjee & Thomas A. Henzinger (2013): *Synthesis of AMBA AHB from formal specification: a case study*. *STTT* 15(5-6), pp. 585–601, doi:10.1007/s10009-011-0207-9. Available at http://dx.doi.org/10.1007/s10009-011-0207-9.

[8] Uri Klein & Amir Pnueli (2010): *Revisiting Synthesis of GR(1) Specifications*. In Sharon Barner, Ian G. Harris, Daniel Kroening & Orna Raz, editors: *Hardware and Software: Verification and Testing - 6th International Haifa Verification Conference, HVC 2010, Haifa, Israel, October 4-7, 2010. Revised Selected Papers*, *Lecture Notes in Computer Science* 6504, Springer, pp. 161–181. Available at http://dx.doi.org/10.1007/978-3-642-19583-9_16.

[9] Robert Könighofer, Georg Hofferek & Roderick Bloem (2013): *Debugging formal specifications: a practical approach using model-based diagnosis and counterstrategies*. *STTT* 15(5-6), pp. 563–583, doi:10.1007/s10009-011-0221-y. Available at http://dx.doi.org/10.1007/s10009-011-0221-y.

[10] Hadas Kress-Gazit, Georgios E. Fainekos & George J. Pappas (2007): *Where's Waldo? Sensor-Based Temporal Logic Motion Planning*. In: *2007 IEEE International Conference on Robotics and Automation, ICRA 2007, 10-14 April 2007, Roma, Italy*, IEEE, pp. 3116–3121, doi:10.1109/ROBOT.2007.363946. Available at http://dx.doi.org/10.1109/ROBOT.2007.363946.

[11] Hadas Kress-Gazit, Georgios E. Fainekos & George J. Pappas (2009): *Temporal-Logic-Based Reactive Mission and Motion Planning*. *IEEE Trans. Robotics* 25(6), pp. 1370–1381. Available at http://dx.doi.org/10.1109/TRO.2009.2030225.

[12] Shahar Maoz & Jan Oliver Ringert (2015): *GR(1) Synthesis for LTL Specification Patterns*. In: *ESEC/FSE*, ACM. http://smlab.cs.tau.ac.il/syntech/patterns/.

[13] Shahar Maoz & Yaniv Sa'ar (2011): *AspectLTL: an aspect language for LTL specifications*. In Paulo Borba & Shigeru Chiba, editors: *AOSD*, ACM, pp. 19–30. Available at http://doi.acm.org/10.1145/1960275.1960280.

[14] Shahar Maoz & Yaniv Sa'ar (2012): *Assume-Guarantee Scenarios: Semantics and Synthesis*. In: *MODELS*, *LNCS* 7590, Springer, pp. 335–351. Available at http://dx.doi.org/10.1007/978-3-642-33666-9_22.

[15] Shahar Maoz & Yaniv Sa'ar (2013): *Counter play-out: executing unrealizable scenario-based specifications*. In: *ICSE*, IEEE / ACM, pp. 242–251. Available at http://dl.acm.org/citation.cfm?id=2486821.

[16] Shahar Maoz & Yaniv Sa'ar (2013): *Two-Way Traceability and Conflict Debugging for AspectLTL Programs*. *T. Aspect-Oriented Software Development* 10, pp. 39–72. Available at http://dx.doi.org/10.1007/978-3-642-36964-3_2.

[17] Nir Piterman, Amir Pnueli & Yaniv Sa'ar (2006): *Synthesis of Reactive(1) Designs*. In: *VMCAI*, *LNCS* 3855, Springer, pp. 364–380. Available at http://dx.doi.org/10.1007/11609773_24.

[18] Amir Pnueli & Roni Rosner (1989): *On the Synthesis of a Reactive Module*. In: *POPL*, ACM Press, pp. 179–190.

[19] Amir Pnueli, Yaniv Sa'ar & Lenore D. Zuck (2010): *JTLV: A Framework for Developing Verification Algorithms*. In: *CAV*, *LNCS* 6174, Springer, pp. 171–174. Available at http://dx.doi.org/10.1007/978-3-642-14295-6_18.

[20] Vasumathi Raman, Nir Piterman & Hadas Kress-Gazit (2013): *Provably correct continuous control for high-level robot behaviors with actions of arbitrary execution durations*. In: *2013 IEEE International Conference on Robotics and Automation, Karlsruhe, Germany, May 6-10, 2013*, IEEE, pp. 4075–4081, doi:10.1109/ICRA.2013.6631152. Available at http://dx.doi.org/10.1109/ICRA.2013.6631152.

[21] Jan Oliver Ringert, Bernhard Rumpe & Andreas Wortmann (2014): *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. *Aachener Informatik-Berichte, Software Engineering* 20, Shaker Verlag.

[22] *SYNTECH forklift website*. http://smlab.cs.tau.ac.il/syntech/forklift/.

A multi-paradigm language for reactive synthesis

Ioannis Filippidis Richard M. Murray
{ifilippi@, murray@cds.}caltech.edu
Control and Dynamical Systems,
California Institute of Technology,
Pasadena CA 91106, USA

Gerard J. Holzmann
gerard.j.holzmann@jpl.nasa.gov
Laboratory for Reliable Software,
Jet Propulsion Lab, Caltech,
Pasadena CA 91109, USA

This paper proposes a language for describing reactive synthesis problems that integrates imperative and declarative elements. The semantics is defined in terms of two-player turn-based infinite games with full information. Currently, synthesis tools accept linear temporal logic (LTL) as input, but this description is less structured and does not facilitate the expression of sequential constraints. This motivates the use of a structured programming language to specify synthesis problems. Transition systems and guarded commands serve as imperative constructs, expressed in a syntax based on that of the modeling language Promela. The syntax allows defining which player controls data and control flow, and separating a program into assumptions and guarantees. These notions are necessary for input to game solvers. The integration of imperative and declarative paradigms allows using the paradigm that is most appropriate for expressing each requirement. The declarative part is expressed in the LTL fragment of generalized reactivity(1), which admits efficient synthesis algorithms, extended with past LTL. The implementation translates Promela to input for the Slugs synthesizer and is written in Python. The AMBA AHB bus case study is revisited and synthesized efficiently, identifying the need to reorder binary decision diagrams during strategy construction, in order to prevent the exponential blowup observed in previous work.

1 Introduction

Over the past three decades, system formal verification has aided design and become practical for industrial application. In the past decade, synthesis of systems from specifications has seen significant development [60, 100], partially owing to the discovery of temporal logic fragments that admit efficient synthesis algorithms [83, 20, 30, 8]. Applications range from protocol synthesis for hardware circuits [20], to correct-by-construction controllers for hybrid systems [57, 56, 101].

Many languages and tools have been developed for modeling and model checking systems. Unlike verification using model checking, the tools for synthesis have been developed much more recently. One reason is that centralized synthesis from linear temporal logic (LTL) [85] has doubly exponential complexity in the length of the specification formula [89], a result that did not encourage further development initially.

Currently, LTL is the language used for describing specifications as input to synthesis tools. There are many benefits in using a logic for synthesis tasks, its declarative nature being a major one, because it allows expressing individual requirements separately, and in a precise way. It also makes explicit the implicit conventions present in programming languages [61]. Another aspect of synthesis problems that makes declarative descriptions appropriate is that we want to describe as large a set of possible designs as possible, in order to avoid overconstraining the search space.

However, not all specifications are best described declaratively. There exist synthesis problems whose description involves graph-like structures that are cumbersome for humans to write

Preliminary Report. Final version to appear in:
SYNT 2015

in logic. Robotics problems typically involve graph constraints that originate from possible physical configurations. For example, considering a wheeled robot, its physical motion is modeled by possible transitions that avoid collisions with other objects, whereas an objective to patrol between two locations can more appropriately be described with a temporal logic formula. Properties that specify sequential behavior also lead to graph-like structures, and require use of auxiliary variables that serve as memory. Expressing sequential composition in logic leads to long, unstructured formulas that deemphasize the specifier's intent. The resulting specifications are difficult to maintain, and writing them is error-prone. In addition, the specifier may need to explicitly write clauses that constrain variables to remain unchanged, in order to maintain imperative state. This leads to longer formulas in which the intent behind individual clauses is less readable.

Another motivation relates to the temporal logic hierarchy [71, 92]. Synthesis from LTL has time complexity polynomial in the state space size, and doubly exponential in the size of the formula. In contrast, algorithms with linear time complexity in the size of the formula are known for the fragment of generalized reactivity of rank one, known as GR(1).

In the automata hierarchy, the GR(1) fragment corresponds to an implication of deterministic Büchi automata (BAs) [83, 92, 78, 93]. The consequent requires some system behavior, provided that the environment satisfies the antecedent of the implication, as described in Section 2.3. Deterministic automata can describe recurrence properties ($\square\diamondsuit$), but not persistence ($\diamondsuit\square$). Intuitively, the behavior of variables uniquely determines the associated behavior of a deterministic BA. This drops the complexity of synthesis, because the algorithm does not have to keep track of branching in the automata that is not recorded in the problem's variable.

A large subset of properties that are of practical interest in industrial applications [28, 69, 20] can be expressed in GR(1). There do exist properties that cannot be represented by deterministic Büchi automata, e.g., persistence $\diamondsuit\square p$. Of these properties, those with Rabin rank equal to one are still amenable to polynomial time algorithms (by solving parity games) [30]. Higher Rabin ranks are not expected to admit polynomial time solution, unless $\mathrm{P} = \mathrm{NP}$ [30]. This motivates formulating the required properties in GR(1), which corresponds to Streett properties with rank one. A game with Streett objective of rank one can be solved with the same time complexity as a Rabin objective of rank one. Therefore, properties in the lower Rabin ranks are known to be at least as hard to synthesize, as GR(1). This motivates formulating the required properties in GR(1), which trades off expressive power for computational efficiency.

Translating properties to deterministic automata can be done automatically, but may lead to more expensive synthesis problems than manually written properties, as reported in [78]. So the ability to write deterministic automata directly in a structured and readable language avoids the need for automated translation, and allows fine tuning them, based on the specifier's understanding of the problem. The trade-off is that the translation has to be performed by a human.

Another reason why specifications are not always purely declarative is that in many cases we want to synthesize a system using *existing* components. In other words, we already have a *partial* model, which describes the possible behavior of components that already exist, e.g., because we purchased them off the self, to interface them with the part of the system that we are synthesizing. We *declare* to our synthesis tool what properties the controller under design should satisfy with respect to this model. This restricts what the system should achieve using these components, but not how exactly that will be achieved. So the partial model is best described imperatively, whereas the goal declaratively, using temporal logic.

Educationally, the transition for students from a general purpose programming language like PYTHON or C, directly to temporal logic constitutes a significant leap. Using a multiparadigm language can make this transition smoother.

This work proposes a language that can describe synthesis problems for open systems that react to an adversarial environment. The syntax is derived from that of PROMELA, whereas the semantics interprets it as a two-player turn-based game of infinite duration. Both synchronous and asynchronously scheduled centralized systems with full information can be synthesized. In Section 2, we review temporal logic and relevant notions about two-player games. The presence of two players requires declaring who controls each variable (Section 3.1), as well as the data flow, and control flow in transition systems (Section 3.2). In addition, the specification needs to be partitioned into assumptions about the environment, and guarantees that the system must satisfy (Section 2, Section 3.2). The integration of declarative and imperative semantics is obtained by defining imperative variables (Section 3.1), deconstraining, and executability of actions (Section 3.3). In order to be synthesized, the program is translated to temporal logic, as described in Section 4. In Section 6, we discuss the implementation and the significant improvements in the AMBA case study [20] that were possible by merging fairness requirements into a single Büchi automaton. Relevant work is collected in Section 7, and conclusions in Section 8.

2 Preliminaries

2.1 Linear Temporal Logic

Linear temporal logic with past is an extension of Boolean logic used to reason about temporal modalities over sequences. The temporal operators "next" $\bigcirc$, "previous" $\ominus$, "until" $\mathscr{U}$, and "since" $\mathscr{S}$ suffice to define the other operators [85, 10]. Let AP be a set of variable symbols p that can take values over $\mathbb{B} \triangleq \{\bot, \top\}$. A model of an LTL formula is a sequence of variable valuations called a *word* $w : \mathbb{N} \to \mathbb{B}^{AP}$. A well-formed formula is inductively defined by $\varphi ::= p|\neg\varphi|p \wedge p|\bigcirc\varphi|\varphi \mathscr{U} \varphi|\ominus\varphi|\varphi\mathscr{S}\varphi$. A formula φ is evaluated over a word w at a time $i \geq 0$, and $w,i \models \varphi$ denotes that φ holds at position i of word w. Formula $\bigcirc\varphi$ holds at position i if φ holds at position $i+1$, $\varphi \mathscr{U} \psi$ holds at i if there exists a time $j \geq i$ such that $w,j \models \psi$ and for all $i \leq k < j$, it is $w,k \models \varphi$. The operator $\diamondsuit p \triangleq \mathsf{true}\, \mathscr{U} p$ requires that p be "eventually" true, and the operator $\square p \triangleq \neg\diamondsuit\neg p$ requires that p be true over the whole word. The past fragment of LTL extends it with the "previous" and "since" operators, $\ominus, \mathscr{S}$ respectively [66, 70, 54]. Formula $\ominus\varphi$ holds at i iff $i > 0$ and $w,i-1 \models \varphi$, and formula $\varphi\mathscr{S}\psi$ holds at i iff there exists a time j with $0 \leq j \leq i$ such that $w,j \models \psi$, and for all k such that $j < k \leq i$ it is $w,k \models \varphi$. The *weak previous* operator $\circleddash$ is defined as $\circleddash\varphi \triangleq \neg\ominus\neg\varphi$, "once" $\diamondminus$ as $\diamondminus\varphi \triangleq \top\mathscr{S}\varphi$, and "historically" $\boxminus$ as $\boxminus\varphi \triangleq \neg\diamondminus\neg\varphi$.

2.2 Turn-based games

In many applications, we are interested in designing a system that does not have full control over the behavior of all variables that are used to model the situation. Some problem variables represent the behavior of other entities, usually collectively referred to as the "environment". The system reads these *input* variables and reacts by writing to *output* variables that it controls,

continuing indefinitely. Such a system is called *open* [6, 84], to distinguish it from closed systems that have no inputs, and so full control.

The synthesis of an open system can be formulated as a two-player adversarial game of infinite duration. The two players in the game are usually called the protagonist (system) and antagonist (environment). We control the protagonist, but not the antagonist. If the players move in turns, then the game is called *alternating*. Each pair of consecutive moves by the two players is called a *turn* of the game. In each turn, player 0 moves first, without knowing how player 1 will choose to move in that turn of the game. Then player 1 moves, knowing how player 0 moved in that turn. Depending on which player we control, there are two types of game. If the protagonist is player 1, then the game is called *Mealy*, otherwise *Moore* [75, 76]. Due to the difference in knowledge about the opponent's next move between the two flavors of game, more specifications are realizable in a Mealy game, than in a Moore game. There exist solvers for both Moore and Mealy games. Here we will consider Mealy games only.

2.3 Games in logic

Temporal logic can be used to describe both the possible moves in a game (the *arena* or *game graph*), as well as the winning condition. Let $\mathscr{X}$ and $\mathscr{Y}$ be two sets of propositional variables, controlled by the environment and system, respectively. Let $\mathscr{X}'$ and $\mathscr{Y}'$ denote primed variables, where x' represents the next value $\bigcirc x$ of variable x. We abuse notation by using primed variables inside temporal formulae.

Synthesis from LTL specifications is in 2ExpTime [87, 89], motivating the search for fragments that admit more efficient synthesis algorithms. Generalized reactivity of index one, abbreviated as GR(1), is a fragment of LTL that admits synthesis algorithms of time complexity polynomial in the size of the state space [20]. GR(1) [50, 88, 16, 67, 32] is used in the following, but the results can be adapted to larger fragments of LTL, provided that another synthesizer be used [49, 31, 29, 21, 33].

The possible moves in a Mealy game can be specified by initial and transition conditions that constrain the environment and system. Initial conditions are described by propositional formulae over $\mathscr{X} \cup \mathscr{Y}$. Transition conditions are described by safety formulae of the form $\square \varphi_i$ where, for the environment $i = e$ and φ is a formula over $\mathscr{X} \cup \mathscr{X}' \cup \mathscr{Y}$, and for the system $i = s$ and φ is a formula over $\mathscr{X} \cup \mathscr{X}' \cup \mathscr{Y} \cup \mathscr{Y}'$. Note that the system plays second in each turn, so it can see $\mathscr{X}'$, whereas the environment cannot see $\mathscr{Y}'$, because it represents future values. The winning condition in a GR(1) game is described using progress formulae of the form $\square\diamondsuit \psi_i, i \in \{e, s\}$, where φ_i is a propositional formulae over $\mathscr{X} \cup \mathscr{Y}$.

The overall specification of a GR(1) game is of the form

$$\underbrace{\Big(\bigwedge_{i=0}^{n_0} \theta_{e,i} \wedge \bigwedge_{i=0}^{n_1} \square\varphi_{e,i} \wedge \bigwedge_{i=0}^{n_2} \square\diamondsuit \psi_{e,i}\Big)}_{assumption} \rightarrow \underbrace{\Big(\bigwedge_{i=0}^{m_0} \theta_{s,i} \wedge \bigwedge_{i=0}^{m_1} \square\varphi_{s,i} \wedge \bigwedge_{i=0}^{m_2} \square\diamondsuit \psi_{s,i}\Big)}_{assertion} \tag{1}$$

Note that requirements that constraint the environment are called *assumptions* and guarantees that the system must satisfy are called *assertions*. Assumptions limit the set of admissible environments, because, in practice, it is impossible to satisfy the design requirements in arbitrarily adversarial environments [6]. The implication above is interpreted by prioritizing between safety and liveness, to prevent the system from violating its safety assertion in case this would allow it

to prevent the environment from satisfying its liveness assumption. The synthesis algorithm for GR(1) has time complexity $O\left(nm|\Sigma|^2\right)$ [20], where n is the length of the assumption formula, m the length of the assertion formula, and Σ is the set of all possible variable valuations.

3 Language definition

The language we are about to define is syntactically an extension of Promela [47], but its semantics is defined by a translation to turn-based infinite games with full information. Promela is a guarded command language that can represent transition systems, non-deterministic execution, and guard conditions for determining whether statements are executable [47, 27]. Its syntax can be found in the language reference manual [47, 46]. Here we will introduce syntactic elements only as needed for the presentation. Briefly, we mention that a program comprises of transition systems and automata, whose control flow can be described with sequential composition, selection and iteration statements, `goto`, as well as blocks that group statements for atomic execution.

3.1 Variables

Ownership In a game, variables from $\mathscr{X}$ are controlled by the environment and variables from $\mathscr{Y}$ by the system. We call *owner* of a variable the player that controls it. We use the keywords `env` and `sys` to signify the owner of a variable. Variables can be of Boolean, bit, byte, (bounded) integer, or bitfield type.

Declarative and imperative semantics In imperative languages, variables remain unchanged, unless explicitly assigned new values. In declarative languages, variables are free to change, unless explicitly constrained [97]. In verification, both declarative languages like TLA [62] and SMV [24] have been used, as well as imperative languages like Promela and Dve [12].

In a synthesis problem, there are variables that are more succinct to describe declaratively, whereas others imperatively. For this reason, we combine the two paradigms, by introducing a new keyword `free` to distinguish between imperative and declarative variables. Variables whose declaration includes the keyword `free` are by default allowed to be assigned any value in their domain, unless explicitly constrained otherwise. Variables without the keyword `free` have imperative semantics, so their value remains unchanged, unless otherwise explicitly stated. Let V^{free} denote free variables, and V^{imp} imperative variables, and V_p the variables of player $p \in \{e, s\}$.

Ranged integer data type Symbolic methods for synthesis use reduced ordered binary decision diagrams (BDDs) [23, 10], which represent sets of states, and relations over states. As operations are performed between BDDs, these can grow quickly, consuming more memory. The growth can be ameliorated by reordering the variables over which a BDD is defined. Reordering variables can be prohibitively expensive, as discussed in Section 6, so reducing the number of bits is a primary objective. In addition, the complexity of GR(1) synthesis is polynomial in the number $|\Sigma|$ of variable valuations, which grows exponentially with each additional bit. We can reduce the number of bits by using bitfields whose width is tailored to the problem at hand. For convenience, the *ranged* integer type `int(MIN, MAX)` is introduced to define a

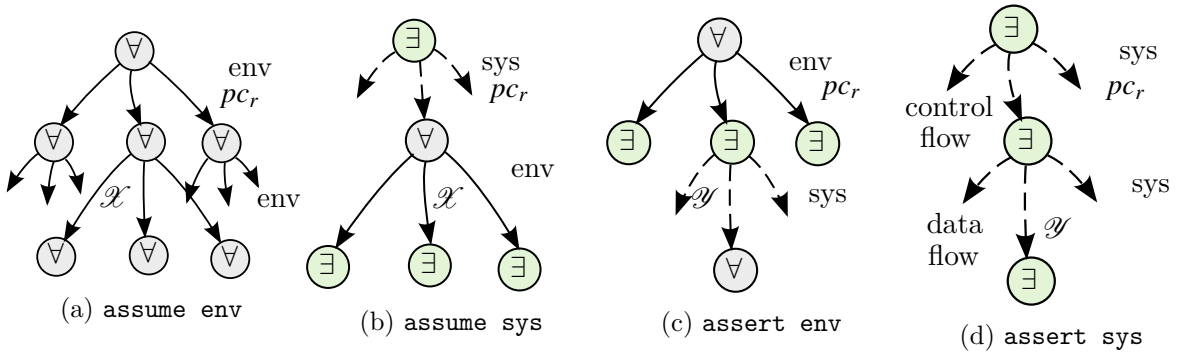


Figure 1: An assumption (assertion) process constrains the environment (system) variables, and `env` (`sys`) declares who chooses the next statement to be executed (when there are multiple).

variable $x \in \{\mathsf{MIN}, \mathsf{MIN}+1, \ldots, \mathsf{MAX}\}$, with saturating semantics [42]. An integer with saturating semantics cannot be incremented when its value reached the maximal in its range, i.e., MAX.

A ranged integer is represented by a bitfield. The bitfield comprises of bits, so it can only range between powers of two. The ranged integer though may have an arbitrary range. For this reason, safety constraints are automatically imposed on the bitfield representing the ranged integer. In other words, if x is the integer value of the bitfield, and it represents an integer that can take values from MIN to MAX, then the constraint $\square(\mathsf{MIN} \leq x \leq \mathsf{MAX})$ is added to the safety formula of the player that owns the ranged integer.

Other numerical data types have mod wrap semantics. The value of an integer with mod wrap semantics overflows to MIN (underflows to MAX) if incremented when equal to the maximal value MAX (minimal value MIN). Mod wrap semantics are available only for integers that range over all values of a (signed) bitfield, because the modulo operation would otherwise be needed. Any BDD describing a modulo operation is at best of exponential size [23].

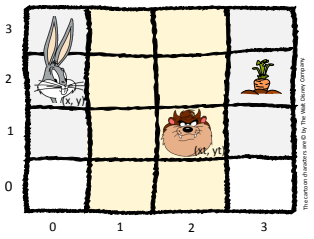
3.2 Programs representing games

In many synthesis problems, the specification includes graph-like constraints. These may originate from physical configurations in robotics problems, deterministic automata to express a formula in GR(1), or describe abstractions of existing components that are to be controlled. These constraints can be described by processes. A process describes both control and data flow. In order to discuss control and data flow, we will refer to *program graphs*. A program graph is an intermediate representation of a process after parsing and control flow analysis. For our purposes, a *program graph* is a rooted directed multi-graph $P_r \triangleq (V_r, E_r)$ whose edges E_r are labeled with program statements, and nodes V_r abstract states of the system [53, 10]. Execution starts from the graph's root. A multi-digraph is needed, because, between two given nodes, there may exist edges labeled with different program statements.

Control flow is the traversal of edges in a program graph (i.e., execution of statements), whereas data flow is the behavior of program variables along this traversal. A *program counter* pc_r is a variable used to store the current node in V_r. A natural question to ask is who controls the program counter. Another question is whose data flow is constrained by the program graph P_r. In the next section, we define syntax that allows declaring the player that controls the program counter, and the player that is constrained to manipulate the variables it owns, according to

the statements selected by the program counter. This allows defining both processes where control and data flow are controlled and constrain the same player, but also processes with mixed control. If one player controls the program counter, and the opponent reacts by choosing a compliant data flow, then the process itself describes a game.

As an example, suppose that the environment controls the program counter, and the system the data flow. At each node of the program graph, the environment can pick any successor node, and the system must react by selecting a data flow compatible with the program statement that labels the edge that the environment picked. So *paths* in this program graph are *universally* quantified, whereas data flow is *existentially* quantified. The notion of path quantification corresponds to universal and existential nodes in alternating tree automata [25, 79, 98, 99, 59], although the program graphs presented here differ in how edges are labeled. If paths are universally quantified, then control flow nondeterminism is known as *demonic*, p.85 [45], [95], otherwise as *angelic* [73, 37, 22] If we unrolled the game described by a process, then we would get a game graph, with nodes that correspond to valuations of the variables and the program counter. When the control flow player takes a turn, it picks the next statement to be executed, i.e., an edge in the program graph of a process. This edges corresponds to an edge in the game graph. If the control flow player is the system, then the choice of edge in the game graph is existentially quantified, otherwise it is universally quantified. In this way, the path that is constructed through the game graph has alternating quantification. The dataflow player must respond, by choosing the values of variables accordingly. This choice corresponds to selecting an edge from the next node in the game graph, as shown in Fig. 1.



(a) Adversarial game.

```
1  #define H 3
2
3  free env int(1, 2) xt;
4  env int(0, H) yt;
5
6  assume env proctype taz(){
7      do
8      :: yt = yt - 1
9      :: yt = yt + 1
10     :: skip
11     od
12 }
13
14 assume ltl { []<>(yt == 0) }
15
16 sys int(0, 3) x;
17 sys int(0, H) y;
18
19 assert sys proctype bunny(){
20     do
21     :: x = x - 1
22     :: x = x + 1
23     :: y = y - 1
24     :: y = y + 1
25     :: skip
26     od
27 }
28
29 assert ltl {
30   [] ! ((x == xt) && (y == yt)
         ) &&
31   /* [] ! -X ((xt' == x ) &&
         (yt' == y )) && */
32   [] -X ! ((xt' == x) && (yt'
         == y)) &&
33   []<>((x == 3) && (y == 2)) }
```

(b) Specification for the game.

Figure 2: Simple example.

3.2.1 Syntax

Program graphs are declared with the `proctype` keyword of `Promela` followed by statements enclosed in braces. The keyword `assume` (`assert`) declares a process that constrains the environment's (system's) data flow. These keywords are common in theorem proving and program verification languages [64]. The keyword `env` (`sys`) declares that the environment (system) controls the program counter pc_r of a process, Fig. 1. The implementation of `assume sys` is the most interesting, and is described in Section 4. We will call program graphs *processes*, noting that these processes have full information about each other, so they correspond to centralized synthesis, not distributed. The program counter *owner* is the player that controls variable pc_r. The *process player* is the player constrained by the program graph.

Example For example, the specification in Fig. 2b defines a game between two players: the Bunny, and Taz, that move in turns, as depicted in Fig. 2a. Each logic time step includes a move by Taz from (x_t, y_t) to (x'_t, y'_t), followed by a move by the Bunny from (x, y) to (x', y'). The Bunny must reach the carrot, without moving through a cell that Taz is in (`assert ltl`). Taz can only move between $x_t \in \{1, 2\}$, and has to keep visiting the lower row (`assume ltl`). Taz can move diagonally, but the Bunny only vertically or horizontally. Both players have an option to stay still (`skip`). Note that x_t is a declarative variable, so it can change unless constrained.

The process `taz` constrains the environment variables x_t, y_t (`assume`) and the environment controls its program counter (`env`). The `do` loops define alternatives that each player must choose from to continue playing the game. Note that nondeterminism in process `taz` is demonic (universally quantified), whereas in `bunny` angelic (existentially quantified), i.e., the design freedom given to the synthesis tool. Each player has full information about all variables in the game, both local, as well as global, and auxiliary. The solution is a strategy represented as a Mealy transducer [75] that the Bunny can use to win the game.

It is interesting to consider the conjuct $\varphi \triangleq \Box\ominus\neg((x = x'_t) \wedge (y = y'_t))$, which prevents the Bunny from moving next to Taz, from where Taz can catch it in the next turn. Note that `-X` and `--X` are the weak and strong previous operators, and $\Box$ requires that, at each time step, the formula with $\ominus$ as main operator be true. Using the strong previous, this is equivalent to $\Box\neg\ominus((x = x'_t) \wedge (y = y'_t))$ (commented in code).

A naive first attempt could be $\Box\neg((x = x'_t) \wedge (y = y'_t))$. However, during solution of the Mealy game, this leads to a controllable predecessor operation [20, 96] of the form $\forall x'_t, y'_t, \ldots \exists x', y', \ldots . (x = x'_t) \wedge (y = y'_t)$. This is false, because variables x, y are not quantified (fixed already in the previous time step). Instead, we apply the axiom $\Box p = \Box\ominus p$, P4, p.58 [70]. This shifts the expression $\neg((x = x'_t) \wedge (y = y'_t))$ to the past, and yields the equivalent formula φ, which is suitable for controllable predecessor computations. Past LTL is implemented by a translation using temporal testers [54].

3.3 Statements

Control flow can be defined using selection (`if`) and repetition (`do`) statements, `else`, `break`, `goto`, and labeled statements. The statements `run`, `call`, `return` are not supported, because dynamic process creation would dynamically add BDD variables. In this section, we define expressions, assignments, and their executability.

Expressions Primed variables (that correspond to using the "next" operator $\bigcirc$) can appear in expressions to refer to the "next" values of those variables, as in the syntax of synthesis tools and TLA [61]. Following TLA, we will call *(state) predicate* an expression that contains no primed variables and *action* an expression that contains primed variables [61]. Actions can be regarded as generalized assignments, in a sense that will be made precise later. Primed system variables cannot appear in assumption processes, because they refer to values not yet known to the environment. Using a GR(1) synthesizer as back-end, multiple priming within a single statement is not allowed, but can be allowed if a full LTL synthesizer is used as back-end [49, 36].

Deconstraining By default, imperative variables are constrained to remain invariant. If any assumption (assertion) process executes a statement that contains a primed environment (system) variable, then that variable is not constrained to remain unchanged in that time step. For

example, in the assertion `sys bit x = 0; (x == 0); (x' == 1 - y)` the variable `x` is constrained by $x' = x$ when `x == 0` is executed, but the synthesizer is allowed to pick its next value as needed, in order to satisfy `x' == 1 - y`. Note that statements in assumption (assertion) processes that contain primed imperative system (environment) variables do *not* deconstrain those variables, because assumptions (assertions) are relevant only to the environment's (system's) data flow.

Assignments In PROMELA, expressions are evaluated by first converting all values to integers, then evaluating the expression with precision that depends on the operating system and processor, and updating the assigned variable's value, truncating if needed. Let $\mathsf{trunc}(y, w)$ denote a function that truncates the value of expression y to bitwidth w. An assignment `x = expr` is translated to the logic formula $x' = \mathsf{trunc}(expr, \mathsf{width}(x))$, if variable x has mod wrap semantics, and to $x' = expr$ otherwise. If variable x is imperative, then it is deconstrained.

Statement executability A condition called *guard* is associated to each statement [27]. The process can execute a statement only if the guard evaluates to true. If a process currently has no executable statement, then it *blocks*. For each statement, its guard is defined by existential quantification of the primed variables of the data flow player. The quantification is applied after the statement is translated to a logic formula. So the guard of a statement is the realizability condition for that statement. It means that, from the local viewpoint of that statement only, given the current values of variables in the game, the constrained player can choose a next move. So the scheduler cannot pick as next process to execute a process that has blocked. Clearly, if all processes block, then that player deadlocks.

Using this definition, the guard of a state predicate is itself, as in PROMELA. The implementation quantifies variables using the PYTHON binary decision diagram `dd` [4]. If an unsatisfiable guard is found, then the implementation raises a warning. For example, if we inserted the statement `xt && xt' && y'` in the process `bunny` (see example), then its guard would be $\exists y'.x_t \wedge x'_t \wedge y' = x_t \wedge x'_t$. Similarly, the guard of an expression `xt && xt'` in the process `taz` is x_t.

4 Translation to logic

In this section, we describe how a program is translated to temporal logic, in particular GR(1). For each process, the starting point is its program graph, which has edges labeled by program statements, and describes the control flow of a process in the source code. The construction of program graphs from source code is the same as for PROMELA [47], and described in detail in [35].

Here we give a brief example. Consider the process `maintain_lock` in Listing 1. It has two `do` loops, with two outgoing edges each. The corresponding program graph is shown in Fig. 3b. Each statement labels one edge, and that edge can be traversed if the guard associated to the statement evaluates to true. The guard can contain primed variables, requiring that the dataflow player manipulates them so as to make the edge's guard true. Otherwise, the player cannot traverse an edge with false guard. This program graph is translated further to logic, as described next. The semantics of the language are defined by this translation to logic.

There are three groups of elements in a program: processes, `ltl` blocks, and the scheduler that picks processes for execution. The scheduler is not present in the source code, but is

added during translation, to represent the products between processes. The translation can be organized into a few thematically related sets of formulae. Due to lack of space, we are going to discuss the most interesting and representative of these at a high level. The full translation can be found in [35], and in the implementation. There are four groups of formulae: (i) control and data flow, (ii) invariance of variables, (iii) process scheduler, (iv) exclusive execution (atomic).

Control and data flow The translation of processes is reminiscent of symbolic model checking [74], but differs in that there are two players, and both play in each logic time step. This requires carefully separating the formulae into assumptions and guarantees (assertions).

Suppose that the scheduler selects process r to execute (how is explained later). At a given time step, a process is at some node i in its program graph, and will transition to a next node j, by traversing an edge labeled by a program statement. The player that controls the program counter pc_r selects the next statement, so the edge in the program graph. The player that is constrained by that process has to make sure that it *complies*, by picking values for variables that it controls such that the statement is satisfied. Recall that the scheduler can only pick from processes that have a satisfiable statement, so whenever a process executes, there will exist a satisfiable next statement. Of course, conflicts can arise between different synchronous processes that can lead to deadlock, and it is the synthesizer's task to avoid such situations, to avoid losing the game. The transition constraints are encoded by the formula

$$\mathrm{trans}(p,r) \triangleq \bigwedge_{i\in N_r} \big((pc_r = i) \rightarrow \bigvee_{(i,j,k)\in E_r} \varphi_{r,i,j,k} \wedge (\tilde{pc}_r = j) \wedge (\tilde{\mathrm{key}}_r = k) \wedge \mathrm{exclusive}(p,r,i,j,k)\ \big) \tag{2}$$

where N_r denotes the set of nodes, and E_r the multi-edges of the program graph of a process, p denotes the player (e,s). The logic formula equivalent to bitblasting the statement labeling edge (r,i,j,k) is $\varphi_{r,i,j,k}$. For `assume sys` processes, the system selects the next edge one time step before the scheduler decides whether that environment process will execute, so two copies are needed, pc_r,$\hat{pc}_r$ (system variables). So in an `assume sys` process, $\tilde{pc}_r \triangleq \hat{pc}_r, \tilde{\mathrm{key}}_r \triangleq \mathrm{key}(r)$, and in other processes $\tilde{pc}_r \triangleq pc'_r, \tilde{\mathrm{key}}_r \triangleq \mathrm{key}(r)'$. The variable $\mathrm{key}(r)$ selects among multi-edges, and is controlled by the same player as the program counter pc_r of the process with pid r. In a system process, if node j is in an atomic block, then $\mathrm{exclusive}(p,r,i,j,k)$ sets the auxiliary variables ex'_s and pm'_s to request atomic execution from the scheduler. The integer variable ex_s stores the identity of the process that requests atomic execution, and the bit pm_s requests that the environment be preempted, if the scheduler grants the request for atomic execution.

$$\begin{aligned}
\mathrm{dataflow}(r) &\triangleq (ps(r)' = m(r)) \rightarrow \mathrm{trans}(p,r)\\
\mathrm{selectable}(r) &\triangleq \mathrm{blocked}(r) \rightarrow (ps(r)' \neq m(r))\\
\mathrm{control_flow}(r) &\triangleq \mathrm{ite}\big((ps(r)' = m(r)),\ \mathrm{pc_trans}(p,r),\ \mathrm{inv}(pc_r)\big)\\
\mathrm{blocked}(r) &\triangleq \bigvee_{i\in N_r} \big((pc_r = i) \wedge \bigwedge_{(i,j,k)\in E_r} \neg \mathrm{guard}_{r,i,j,k}\big)
\end{aligned} \tag{3}$$

The environment variables $ps(r)$ select the process or synchronous product that will execute next inside an asynchronous product (top context is an asynchronous product). For this purpose, each process and product have a local integer id $m(\cdot)$ among the elements inside the product that contains them. The transition relation for the program counter depends on the type of process. For `assume sys` processes, $\mathrm{pc\_trans}(p,r) \triangleq (pc'_r = \hat{pc}_r)$, and for other processes it equals

$\mathsf{guards}(r)$. The condition $\mathsf{guards}(r)$ constrains the program counter to follow unblocked edges in a process. It is necessary when the control and data flow are controlled by different players, because whoever moves the program counter, can otherwise pick an edge with a statement that blocks the other player. In addition, for `assume sys` processes, a separate constraint with same form as $\mathsf{guards}(r)$, but different priming of sub-expressions is imposed on the program counter copy $\widehat{pc}_r$. The ternary conditional is denoted by $\mathsf{ite}(a,b,c)$.

Invariance of variables When a process is not executing, its declarative local variables must be constrained to remain invariant ($x' = x$). Also, imperative variables must remain invariant whenever no process executes a statement (edge) that either is an expression and contains a primed copy of that variable, or is an assignment. These are ensured by the following equations

$$\begin{aligned}
&\mathsf{local_free}(p,r) \triangleq (ps(r)' \neq m(r)) \to \bigwedge_{x \in V_{p,r}^{\mathrm{free}}} \mathsf{inv}(x) \\
&\mathsf{imperative_inv}(p,r) \triangleq \mathsf{array_inv}(p,r) \wedge \bigwedge_{x \in V_{p,r}^{\mathrm{imp}}} (\mathsf{inv}(x) \vee \bigvee_{(i,j,k) \in E_r . x \in \mathsf{deconstrained}(r,i,j,k)} \mathsf{edge}(r,i,j,k))
\end{aligned} \tag{4}$$

where $V_{p,r}^{\mathrm{free}}$ are free local variables of player p in process r. For `assume sys` processes, it is $\mathsf{edge}(r,i,j,k) \triangleq (ps(r)' = m(r)) \wedge (pc_r = i) \wedge (\hat{pc}_r = j) \wedge (\mathsf{key}(r) = k)$, and for other types of processes $\mathsf{edge}(r,i,j,k) \triangleq (ps(r)' = m(r)) \wedge (pc_r = i) \wedge (pc_r' = j) \wedge (\mathsf{key}(r)' = k)$. Primed references to elements in imperative arrays deconstrain only the referenced array element, ensured by $\mathsf{array\_inv}$.

Scheduler The scheduler (environment) has to select the processes that will execute. Products of processes can be defined in the source code by enclosing processes, or other products, in braces preceded by the keywords `async` and `sync`. They can be nested. `async` defines an asynchronous product, and the scheduler picks some unblocked process or product inside it to execute next. If all processes/products in an asynchronous product k have blocked, then the scheduler sets the corresponding variable ps_k to a reserved value (n_k). The reserved value is also used if the asynchronous product is nested in a synchronous product that currently is not selected to execute.

If the top product blocks, then the player has deadlocked, losing the game. The only exception is when the environment is preempted by a request from a system process for atomic execution. At a high level, this behavior is expressed as follows for scheduling the environment processes.

$$\begin{aligned}
\mathsf{product_selected}(k) &\triangleq (ps(k)' \neq m(k)) \leftrightarrow (ps_k' = n_k) \\
\mathsf{selectable_element}(r) &\triangleq \mathsf{element_blocked(r)} \to (ps(r)' \neq m(r)) \\
\mathsf{element_blocked}(r) &\triangleq \begin{cases} \mathsf{blocked}(r), \text{ if } r \text{ is a process} \\ \mathsf{sync_blocked}(r), \text{ if } r \text{ is a synchronous product} \\ \mathsf{async_blocked}(r), \text{ if } r \text{ is an asynchronous product.} \end{cases}
\end{aligned} \tag{5}$$

$$\begin{aligned}
\mathsf{sync_blocked}(k) &\triangleq \bigvee_{r \in R_k} \mathsf{element_blocked}(r) \\
\mathsf{async_blocked}(k) &\triangleq \bigwedge_{r \in R_k} \mathsf{element_blocked}(r)
\end{aligned} \tag{6}$$

$$\text{pause\_env\_if\_req} \triangleq (ps'_{\text{env\_top}} = n_e) \leftrightarrow (pm_s \wedge (ps'_{\text{sys\_top}} = ex_s < n_s)). \tag{7}$$

The expression $\text{element\_blocked}(r)$ depends on the $\text{blocked}(z)$ expressions, and ensures that the scheduler doesn't select a synchronous product containing some blocked process, neither an asynchronous product where all processes are blocked. Recall also selectable from earlier, which applies to individual processes. Analogous formulae apply to system processes. For system processes, the top-level asynchronous product implication in $\text{async\_blocked}(r)$ must be replaced with equivalence, to force the environment to choose some system process (or product) to execute, when there exist unblocked ones. Note that the asynchronous products here are in the context of full information, so the system is *not* asynchronous in the sense of [55, 86].

Exclusive execution A system process of the top asynchronous product can request to execute atomically by setting the variables pm_s, ex_s, Eq. (2). If that process remains unblocked in the next time step, then the scheduler will grant it uninterrupted execution, until it exits atomic context (either blocked, or reached statements outside the `atomic{...}` block).

$$\text{grant}_s \triangleq \bigwedge_{r \in \text{pids}(s)} \big(((ex_s = m(r)) \wedge \text{frozen\_unblocked}(r)) \rightarrow (ps(r)' = m(r))\big) \tag{8}$$

Recall also that the environment is allowed to pause only if preempted by the system, otherwise it loses the game (pause_env_if_req). The formula $\text{frozen\_unblocked}(r)$ checks whether the system would block, in case the environment froze, granting it exclusive execution. In case the system will block, then the request is not granted, and atomicity lost. This requires substituting primed environment variables with unprimed ones, as follows

$$\begin{aligned}
\text{frozen_unblocked}(r) &\triangleq \begin{cases} \bigvee_{i \in N_r} \big((pc_r = i) \wedge \bigvee_{(i,j,k) \in E_r} \text{guard_test}(r,i,j,k)\big), \text{ if } \text{player}(r) = s \\ \neg \text{blocked}(r), \text{ otherwise} \end{cases} \\
\text{guard_test}(r,i,j,k) &\triangleq \begin{cases} \text{guard}(r,i,j,k)|_{\,x/x' \text{ for } x \in \mathscr{X}}\,, \text{ if } i \text{ in atomic context} \\ \text{guard}(r,i,j,k), \text{ otherwise.} \end{cases}
\end{aligned} \tag{9}$$

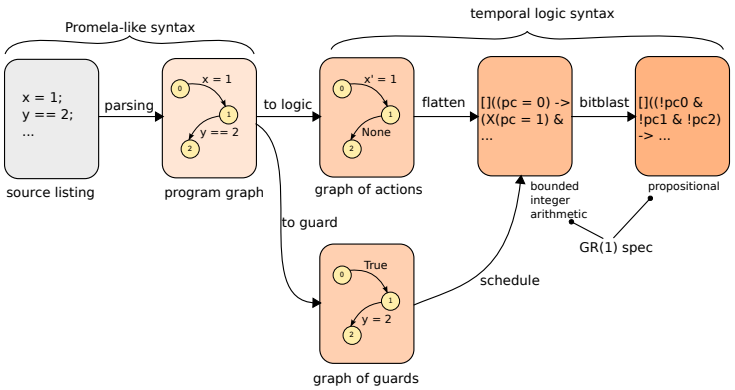
The reason is that this formula corresponds to the case that the environment sets $x' = x$ for the program variables it owns. If atomicity is lost in this turn, then the environment does not need to set $x' = x$, and this is ensured by the definition of $\text{guard\_test}(r,i,j,k)$.

As in PROMELA, LTL formulae that express safety are deactivated during atomic execution (in implementation, an option allows making atomic execution visible to LTL properties). They are re-activated as soon as atomicity is lost.

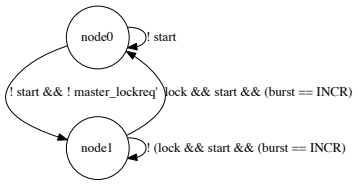
$$\text{mask\_env\_ltl} \triangleq \text{ite}(pm_s \wedge (ps'_{\text{sys\_top}} = ex_s < n_s),\ \text{freeze\_env\_free},\ \psi_{\text{env safety } \texttt{ltl}}) \tag{10}$$

For the system, mask_sys_ltl is defined similarly. The formula freeze_env_free constrains declarative environment variables in global context and inside system processes to remain unchanged while the system is granted exclusive execution.

If an `atomic` block appears in a process, then the `ltl` properties in the program must not contain primed variables, to ensure that the above translation yields the intended interpretation (stutter invariance). If unbounded loops appear inside an atomic context, then there can be



(a) Compiler architecture.



(b) Program graph that corresponds to process `maintain_lock` in Listing 1.

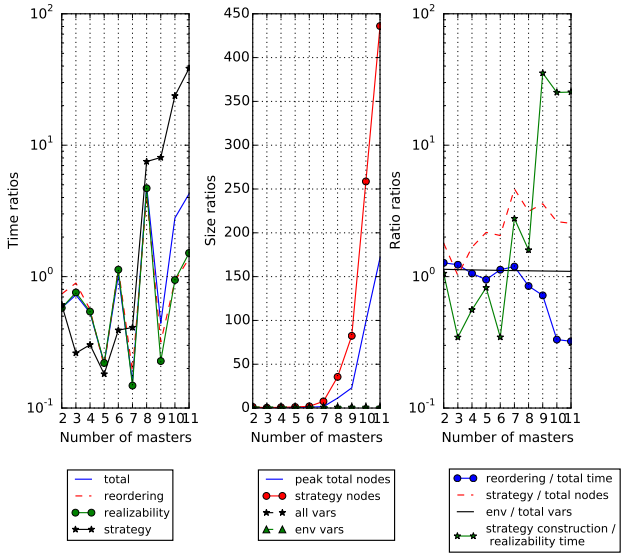
Figure 3: Compiling programs to temporal logic.

no liveness assumptions. The reason is that the system can "hide" in atomic execution forever, preventing the environment from satisfying its liveness assumptions, thus winning trivially. In order to avoid this, the environment liveness goals must be disjoined with strong fairness, a persistence property ($\Diamond\Box$), which is outside of the GR(1) fragment. An extension to use a full LTL synthesizer is possible, though not expected to scale as well. Labels in the code that contain "progress" result in accepting states (liveness conditions). Those expressions described but not defined above, the initial conditions, and a listing into assumptions and assertions can be found in the technical report [35].

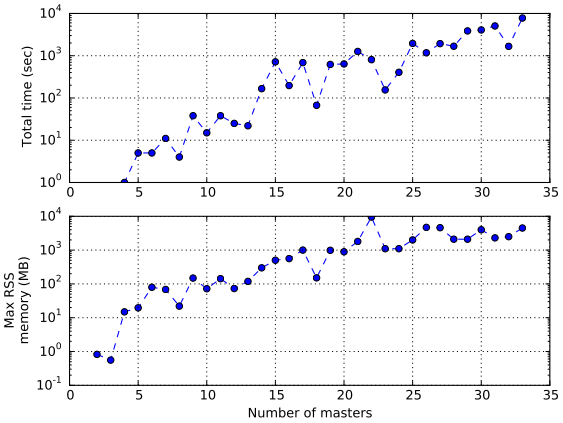
5 Implementation

The implementation is written in Python and available [1, 2, 5] under a BSD license. The frontend comprises of a parser generator that uses Ply (Python `lex-yacc`) [14]. The parser for the proposed language subclasses and extends a separate parser for Promela [2], to enable use of the latter also by those interested in verification activities. After parsing and program graph construction, the translation described in Section 4 is applied [1]. This results in linear temporal logic formulae that contain modular integer arithmetic. At this point, each `ltl` block is syntactically checked to be in the GR(1) fragment, and split into initial condition, action, and recurrence conjuncts [5]. The past fragment is then translated using temporal testers [54]. In the future, the syntactic check can be removed, and a full LTL synthesis algorithm used.

The next step encodes signed arithmetic in bitvector logic using two's complement representation [58]. The resulting formulae are in the input syntax recognized by the Slugs synthesizer [32]. This prefix syntax includes *memory buffers*, which enable avoiding repetition of formulae. For example, `$ 3 x a b  & ?1 ?0  | ?2 ! ?0` describes the ternary conditional $\mathsf{ite}(x,a,b)$. Memory buffers prevent the bitblasted formulae from blowing up. The Slugs distribution includes an encoder of unsigned addition and comparison into bitvector logic using memory buffers. Here, signed arithmetic and arrays are supported. The bitblaster code is a separate module, which can be reused as a backend to other frontends. The resulting formula is passed to the Slugs synthesis tool to check for realizability and construct a winning strategy as a Mealy transducer.



(a) Conjunction divided by BA, no reordering.



(b) BA, no reordering.

Figure 4: Selection of experimental measurements for the revised AMBA specification.

6 AMBA AHB Case study

Revised specification The ARM processor Advanced Microcontroller Bus Architecture (AMBA) [9] specifies a number of different bus protocols. Among them, the Advanced High-performance (AHB) architecture has been studied extensively in the reactive synthesis literature [17, 18, 77, 20, 91, 43, 19].

The AHB bus comprises of masters that need to communicate with slaves, and an arbiter that controls the bus and decides which master is given access to the bus. The arbiter receives requests from the masters that desire to access the bus, and must respond in a weakly fair way. In other words, every master that keeps uninterruptedly requesting the bus must eventually be granted access to it. Note that the AMBA technical manual [9] does not specify any fairness requirement, but instead leaves that decision to the designer. For automated synthesis, weak fairness is one possible formalization that ensures servicing of all the masters.

In addition, a master can request that the access be locked. In the ARM manual, the arbiter makes no promises as to whether a request for the lock will be granted. If the arbiter does lock the access, then it guarantees to maintain the lock, until the request for locking is withdrawn by the master that currently owns the bus. Note that the specification used here requires the arbiter to lock the bus, whenever requested by the master to be granted next.

A specification for the arbiter appeared in [17], and is presented in detail in [20]. Here, we expressed the specification of [20] in the proposed language, Listing 1 on 19. In doing so, some assumptions were weakened and assumption A1 modified, to improve the correspondence with the ARM technical manual, and reduce the number of environment variables (thus universal branching). First, we describe the AHB specification, referring to Listing 1. After that, we summarize the changes, and discuss the experiments.

The specification in Listing 1 has both environment and system variables, as well as assumptions and guarantees. The arbiter is the system, and the environment comprises of slaves and $N+1$ masters. An array `request` of bits represents the request of each individual master

to be given bus ownership, for sending and receiving from a slave of interest. Communication proceeds in bursts. The bus owner selects which type of burst it desires, by setting the integer `burst`. Three lengths of bursts are modeled: single time step (`SINGLE`), four consecutive time steps (`BURST4`), and undefined duration (`INCR`). The currently addressed slave sets the bit `ready` (to true) to acknowledge that it has successfully received data for a burst. While `ready` is false, the bus owner cannot change (G1,6), and a `BURST4` time step is not counted towards completion (G3). For this reason, the slaves (environment) are required to recur setting `ready` to true (A2). The master can also request that, when it is granted the bus, it should be locked. Each master can do so by setting a signal. Only two of these signals are modeled here, using the bit variables `grantee_lockreq` and `master_lockreq` (described below).

The arbiter works in primarily two phases, as introduced in [20]. These phases are extraneous to the standard, and used only to aid in describing the specification. Firstly, the arbiter decides to which master it will next grant the bus to. The arbiter sets the bit `decide` to true during that period. The decision is stored in the form of two variables, `grant` and `lockmemo`, which don't change while `decide` is false (G8). The integer `grant` indicates the master that has been decided to receive bus ownership after the current owner. For performance reasons, the arbiter can only grant the bus to a master that requested it (G10), with the exception of a default master (with index 0).

The bit `lockmemo` is set to the value of the environment bit `grantee_lockreq` (G7). The value `grantee_lockreq'` represents whether the master `grant` had requested locked ownership. In the original specification, an array *lockreq* of *N* environment bits is used (denoted by *HLOCK* in [20]). This increases significantly the variables with universal quantification. Here, this array is abstracted by the bit `grantee_lockreq`. In implementation, the transducer input `grantee_lockreq'` should be set equal to the lock request of master `grant` in the previous time step, i.e., $grantee\_lockreq' \triangleq (\ominus lockreq)[grant]$. In [20], some assumptions are expressed to constrain the array *lockreq*, i.e., when masters request locked ownership. The assumptions can be weakened [34], and by modifying assumption A1 (described below), the array *lockreq* can be abstracted by the two bits `grantee_lockreq` and `master_lockeq`.

The arbiter promises to lock the bus, until the bus owner `master` interrupts requesting it. The owner indicates its lock request by the value `master_lockreq'`. In implementation, the input value `master_lockreq'` should be set equal to the lock request of `master`, i.e., $master\_lockreq' \triangleq lockreq[master]$.

In the second phase, the master changes the bus owner, by updating the integer `master` to `grant` (G4,5). If the grantee had requested a lock, via `grantee_lockreq`, then that request is propagated to the bit `lock` (G4,5). With the bit `lock`, the arbiter indicates that `master` has been given locked access to the bus.

To be weakly fair, each master that keeps uninterruptedly requesting the bus should be granted ownership. This requirement is described as a Büchi automaton (G9). This

The assumption A1 of [20] requires that for locked undefined-length bursts, the masters eventually withdraw their request to access the bus. This assumption is not explicit in the ARM standard, so we modify it, by requiring that masters withdraw only their request for the lock, *not* for bus access. This is described as the Büchi automaton `withdraw_lock` that constrains the environment. The arbiter grants `master` locked access by setting the bit `lock` to true. If `lock` is false, then the master (environment) remains in the outer loop, at the `else`. If `lock` becomes true, then the automaton enters the inner loop. In order for the automaton `withdraw_lock` to exit the inner loop, the environment must set `master_lockreq'` to false. This obliges the owner

`master` to eventually stop requesting locked ownership.

For a `SINGLE` burst, the burst is completed at the next time step that `ready` is true, so the arbiter does not need to lock the bus (since the owner remains unchanged while `ready` is false). For a `BURST4` burst, the arbiter locks the bus for a predefined length of four successful beats (G3). This requirement is described by the safety automaton `count_bursts`. Note that assumption A1 is not needed for this case. For a `INCR` burst, the duration is unspecified a priori. While the owner `master` continuously requests locked access (with `master_lockreq`), the arbiter cannot change the bus owner (G2). This is described by the safety automaton `maintain_lock`. When the arbiter grants locked access to the bus for a burst of undefined duration, then the guard `lock && start && (burst == INCR)` is true. The process `maintain_process` enters the inner loop, and remains there until `master_lockreq` becomes true. This is where assumption A1 is required, to ensure that the owner will eventually stop requesting the lock. The arbiter can then exit the inner loop of `maintain_lock`. Then, the arbiter can wait outside (`start` is false throughout the burst), until the addressed slave sets `ready` to true, signifying the successful completion of that burst, and allowing the arbiter to set `start` and change the bus owner, if needed.

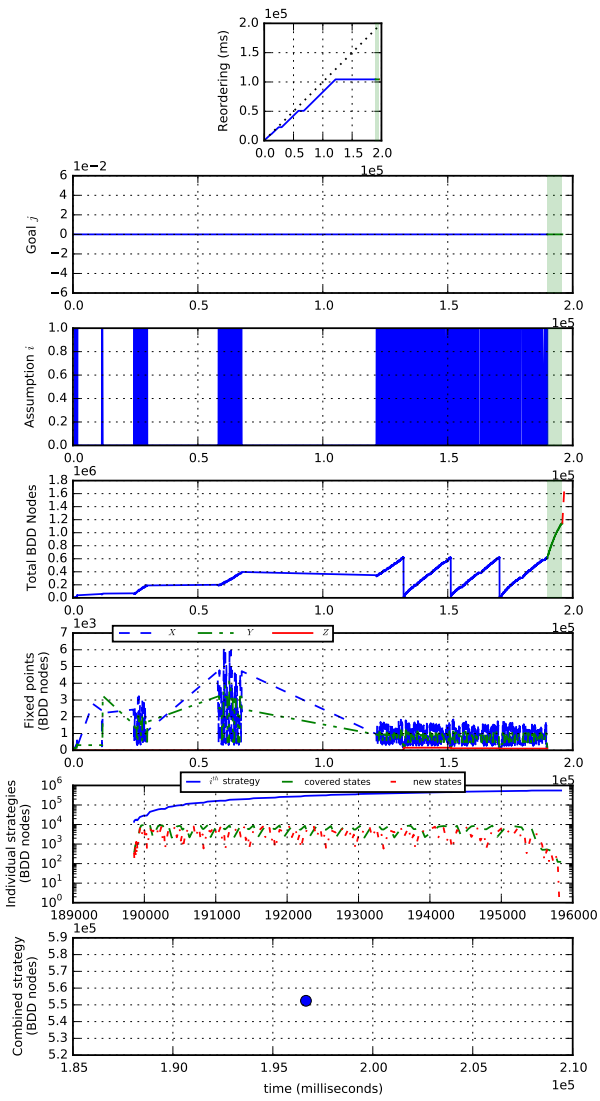
Some properties not in GR(1) are translated to deterministic Büchi automata in [20]. The resulting formulae are much less readable, and not easy to modify and experiment with. Above, we specified these properties directly as processes, with progress states where needed.

Observations By encoding `master` and `grant` as integers, and abstracting the array *lockreq* by the two variables `master_lockreq` and `grantee_lockreq`, the synthesis time was reduced significantly (by a factor of 100 [34]), but are not sufficient to prevent the synthesized strategies from blowing up. By also merging the N weak fairness guarantees $\bigwedge_{i=0}^{N-1} \square\diamondsuit(request[i] \rightarrow master = i)$ into the Büchi automaton (BA) `weak_fairness` with one accepting state, we were able to prevent the strategies from blowing up, and synthesize up to 33 masters, Fig. 4b. The synthesis time for 16 masters is in the order of 5 minutes, and peak memory consumption less than 1GB. To our knowledge, in previous works, the maximal number of masters has been 16, the strategies were blowing up, and the runtimes were significantly longer (21 hours for 12 masters in [20], and more than an hour in [43] for 16 masters).

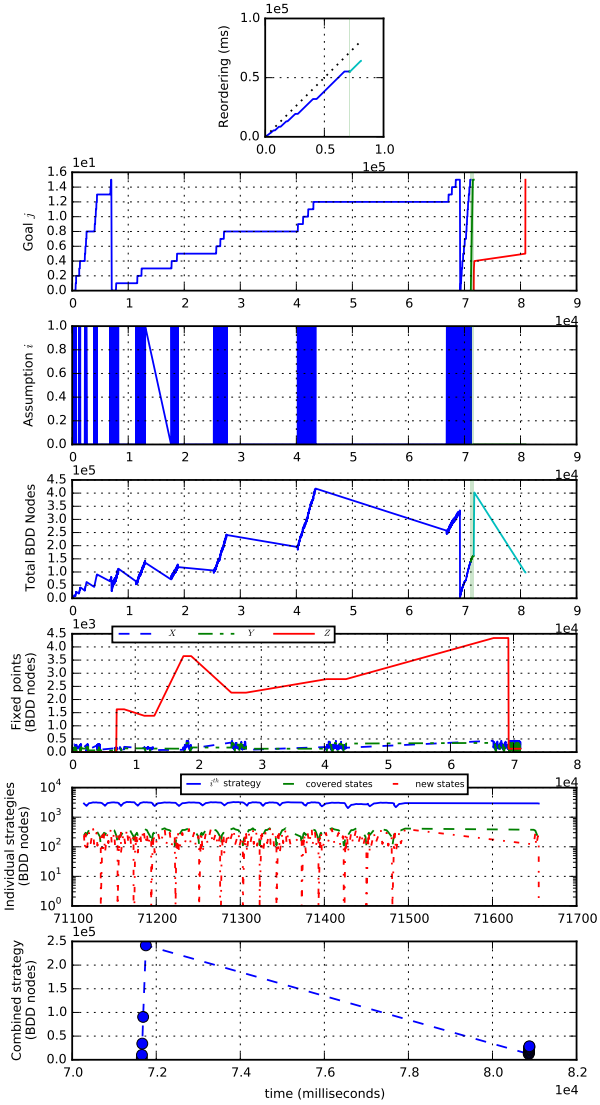
Measurements To identify what caused this difference, we conducted experiments for 8 different combinations: original vs revised spec, conjunction vs BA, reordering during strategy construction enabled/disabled, Table 1. We collected detailed measurements with instrumentation that we inserted into Slugs, available at [3]. Some of these are shown in Fig. 5, and the complete set can be found in the technical report [34] (the language is described in [35]). The experiments were run on an Intel(R) Xeon® X5550 core, with 27 GB RAM, running Ubuntu 14.04.1. The maximal memory limit of Cudd [94] was set to 16 GB.

We found that dynamic BDD reordering during construction of a strategy was the reason for poor performance of conjoined liveness goals, as opposed to a single BA. The implementation of the GR(1) synthesis algorithm in Slugs has three phases:

1. Computing the winning region, while memoizing the iterates of the fixpoint iteration, as BDDs.
2. Construction of individual strategies, one for each recurrence goal.



(a) BA, no reordering, $N = 16$.



(b) Conjunction, with reordering, $N = 16$.

Figure 5: Measurements during phases of: (1) realizability, (2) sub-strategy, and (3) combined strategy construction. The top 4 plots are over all phases, the fixpoints over (1), the individual strategies over (2), and the bottom plot over (3). The revised specification is used.

3. Combination of the individual strategies into a single transducer, which iterates through them.

In Slugs, variable reordering [90] is enabled during the first two phases, but disabled in the last one. If the recurrence goals are conjoined into a formula of the form $\bigwedge \Box\Diamond$, then the memory needed for synthesis blows up Fig. 4a, for both the original and revised specifications. Using a BA, the revised specification scales without blowup.

If reordering is enabled during the last phase (combined transducer construction), then the specification with a conjunction can be synthesized without blowup. With a BA, turning on reordering in the last phase has mildly negative effect, because it can trigger unnecessary reordering. We used the group sifting algorithm [82, 90] for reordering.

Table 1: Overview of results.

	Strategy reordering	Specification original	Specification revised
Conjunction of fairness	with	slow	fast
	w/o	memory blowup	memory blowup
Büchi automaton	with	very slow	ok (slower)
	w/o	slow	ok

Enabling dynamic BDD variable reordering is necessary to prevent the blowup. The conjunction with reordering enabled in phase 3 outperforms the BA with reordering turned off in phase 3. This is a consequence mainly of the fact that the BA chains the goals inside the state space, leading to deeper fixpoint iterations, and has slightly larger state space, due to the nodes of the automaton `maintain_lock`.

Reordering typically accounts for most of the runtime (top plot in Fig. 5a). The second plot shows the currently pursued goal during realizability, and later the sub-strategy under construction, and the sub-strategy being combined in the final strategy. Each drop in total BDD nodes ("teeth" in 4th plot) corresponds to an outer fixpoint iteration. The first outer iteration takes the most time, due to reordering. Later iterations construct subsets, for which the obtained order remains suitable. The highlighted period corresponds to the construction of individual strategies. Plots for the other experiments can be found in [34].

Conclusions The major effect of reordering in the final phase of strategy construction can be understood as follows. Using a BA reduces the goals to only one, so no disjunction of individual sub-strategies is needed [83]. Also, this encoding shifts the transducer memory (a counter of liveness goals), from the strategy construction, to the realizability phase (attractor computations). This slightly increases the state space. Nonetheless, this symbolic encoding allows the variable ordering more time to gradually adjust to the represented sets.

In contrast, by conjoining liveness goals, the variable order is oblivious during realizability checking that the sub-strategies will be disjoined at the end. The disjunction of strategies acts as a shock wave, disruptive to how far from optimal the obtained variable order is. If, by that phase, reordering has been disabled, then this effect causes exponential blowup.

Overall, the proposed language made experimentation easier and revisions faster, helping to study variants of the specification. It can be used to explore the sensitivity of a specification, in the following way. A formula, e.g., requiring weak fairness, can be temporarily replaced with a process that is one possible refinement of that formula, potentially simplified. In the AMBA example, one can fix a round robin schedule for selecting the next grantee (temporarily dropping G10). This is reminiscent of the manual implementation [20]. By doing so, it can be evaluated whether the synthesizer finds it difficult to pick requestors only, or whether some other factor is more important, either another part of the specification, or some external factor. For the AMBA problem, such a simplification showed that some other factor controls the runtime, in particular reordering, and the increased number of environment variables. Therefore, we believe that it can prove useful in exploring the sensitivity of specifications, to help the specifier direct their attention to improve those parts of the specification that impact the most synthesis performance.

Listing 1: AMBA AHB specification in the proposed language.

```
#define N 2  /* N + 1 masters */
#define SINGLE 0
#define BURST4 1
#define INCR 2
/* variables of masters and slaves
A4: initial condition */
free env bool ready = false;
free env int(0, 2) burst;
free env bool request[N + 1] =false;
free env bool grantee_lockreq=false;
free env bool master_lockreq =false;
/* arbiter variables */
/* G11: sys initial condition */
free bool start = true;
free bool decide = true;
free bool lock = false;
free bool lockmemo;
free int(0, N) master = 0;
free int(0, N) grant;
/* A2: slaves must progress with
    receiving data */
assume ltl { []<> ready }
/* A3: dropped, weakening the
    assumptions */
/* A1: */
assume env proctype withdraw_lock(){
    progress:
    do
    :: lock;
        do
        :: ! master_lockreq'; break
        :: true /* wait */
        od
    :: else
    od
}
assert ltl {
[](
/* G1: new access starts only when
    slave is ready */
(start' -> ready)
/* G4,5 */
&& (ready -> ((master' == grant) &&
    (lock' <-> lockmemo')))
/* G6 */
&& (! start' -> (
    (master' == master) &&
    (lock' <-> lock)))
/* G7: remember if lock requested */
&& ((--X decide) -> (lockmemo' <->
    grantee_lockreq'))
/* G8 */
&& (! decide -> (grant' == grant))
&& ((! --X decide) -> (lockmemo' <->
    lockmemo))
/* G10: grant only to requestors */
&& ((grant' == grant) || (grant' ==
    0) || request[grant'])
)
}
sync{ /* synchronous product */
/* G9: weak fairness */
assert proctype weak_fairness(){
    int(0, N) count;
    do
    :: (! request[count] || (master
        == count));
        if
        :: (count < N) && (count' ==
            count + 1)
        :: (count == N) && (count'
            == 0);
            progress: skip
        fi
    :: else
    od
}
/* G2: lock until no lock req */
assert sys proctype maintain_lock(){
    do
    :: (lock && start && (burst ==
        INCR));
        do
        :: (! start && !
            master_lockreq'); break
        :: ! start
        od
    :: else
    od
}
/* G3: for a BURST4 access, count
    the "ready" time steps. */
assert sys proctype count_burst(){
    int(0, 3) count;
    do
    :: (start && lock &&
        (burst == BURST4) &&
        (!ready || (count' == 1)) &&
        (ready || (count' == 0)) );
        do
        :: (! start && ! ready)
        :: (! start && ready && (
            count < 3) &&
            (count' == count + 1))
        :: (! start && ready && (
            count >= 3)); break
        od
    :: else
    od
}
}
```

7 Relevant work

Our approach has common elements with program repair [51], program sketching [65], and syntax-guided synthesis [7]. Program repair aims at modifying an existing program in a conventional programming language. Syntax-guided synthesis uses a grammar to "slice" the admissible search space of terminating programs. Here, we are interested in reactive programs. Similarly, program sketching uses templates to restrict the search space and give hints to the synthesizer for obtaining a complete program. In [15], the authors propose another constraint-based approach to games, but start directly from logic formulae.

TLA [61, 62] subsumes our proposed language, since it includes quantification, but is intended as a theorem proving activity, is declarative, and is aimed at verification. Nonetheless, one can view the proposed translation as from open-Promela to TLA. SMV is a declarative language [24], and Jtlv [88] an SMV-like language for synthesis specifications, but with no imperative constructs. AspectLTL is a further declarative extension for aspect-oriented programming [72].

RPromela is an extension of Promela that adds synchronous-reactive constructs (not in the sense of reactive synthesis) that include synchronous products and channels called ports [80, 81]. Its semantics are defined in terms of *stable states*, where the synchronous product blocks, waiting for message reception from its global ports. RPromela does not address modeling of the environment, nor declarative elements. Besides, synchronous-reactive languages like Esterel, Quartz (imperative textual), Statecharts, Argos, SyncCharts (imperative graphical), Lustre, and Lucid Synchrone (declarative textual) and Signal (declarative graphical) are by definition *deterministic* languages intended for direct design of transducers [41, 44, 52]. In synthesis, non-determinism is an essential feature of the specification.

Our approach has common elements with *constraint imperative programming* (CIP), introduced with the experimental language Kaleidoscope [38, 39, 40, 68], one of the first attempts to integrate the imperative and declarative constraint programming paradigms. An observation from [38], which applies also here, is that specifiers need to express two types of relations: long-lived (best described declaratively), and sequencing relations (more naturally expressed in an imperative style). However, CIP does *not* ensure correct reactivity, because the constraints are solved online. Constraints are a related approach that uses constraints for indirect assignment to imperative variables is [63].

The translation from Promela to declarative formalisms has been considered in [11, 48, 26] and decision diagrams in [13]. These translations aim at verification, do not have LTL as target language, and either have limited support for atomicity [26], no details [11], or programs graphs semantics that do not match Promela [48].

8 Conclusions

We have presented a language for reactive synthesis that combines declarative and imperative elements to allow using the most suitable paradigm for each requirement, to write readable specifications. By expressing the AMBA specification in a multi-paradigm language, it became easier to experiment and transform it into one that led to efficient synthesis that improved previous results by two orders of magnitude. Besides the AMBA specification, other examples can be found in the code repository [1].

Acknowledgments This work was supported by STARnet, a Semiconductor Research Corporation program, sponsored by MARCO and DARPA. The first author was partially supported by a graduate research fellowship from the Jet Propulsion Laboratory, over the summer of 2014.

References

[1] *open-*PROMELA *compiler (*PYTHON *package).* Available at `https://github.com/johnyf/openpromela`.

[2] PROMELA *parser (*PYTHON *package).* Available at `https://github.com/johnyf/promela`.

[3] SLUGS *instrumentation on branch* `printstats`. Available at `https://github.com/johnyf/slugs`.

[4] `dd`*: Decision diagrams (*PYTHON *package).* Available at `https://github.com/johnyf/dd`.

[5] `omega`*: Symbolic and enumerated data structures and algorithms for manipulating* ω*-regular sets (*PYTHON *package).* Available at `https://github.com/johnyf/omega`.

[6] Martín Abadi & Leslie Lamport (1994): *Open Systems in TLA.* In: *PODC*, pp. 81–90, doi:10.1145/197917.197960.

[7] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak & Abhishek Udupa (2013): *Syntax-guided synthesis.* In: *FMCAD*, pp. 1–17, doi:10.1109/FMCAD.2013.6679385.

[8] Rajeev Alur & Salvatore La Torre (2004): *Deterministic Generators and Games for LTL Fragments. ACM Trans. Comput. Logic* 5(1), pp. 1–25, doi:10.1145/963927.963928.

[9] ARM Ltd. (1999): *AMBA™ Specification*, Rev 2.0 edition. Available at `http://www-micro.deis.unibo.it/~magagni/amba99.pdf`.

[10] Christel Baier & Joost-Pieter Katoen (2008): *Principles of Model Checking.* The MIT Press.

[11] Michael Baldamus & Jochen Schröder-Babo (2001): *P2B: A Translation Utility for Linking* PROMELA *and Symbolic Model Checking.* In: *SPIN*, pp. 183–191, doi:10.1007/3-540-45139-0_11.

[12] Jiří Barnat, Luboš Brim, Vojtěch Havel, Jan Havlíček, Jan Kriho, Milan Lenčo, Petr Ročkai, Vladimír Štill & Jiří Weiser (2013): DIVINE *3.0 – An Explicit-State Model Checker for Multi-threaded C & C++ Programs.* In: *CAV*, 8044, pp. 863–868, doi:10.1007/978-3-642-39799-8_60.

[13] Vincent Beaudenon, Emmanuelle Encrenaz & Sami Taktak (2010): *Data decision diagrams for promela systems analysis. STTT* 12(5), pp. 337–352, doi:10.1007/s10009-010-0135-0.

[14] David M. Beazley: *PLY (Python Lex-Yacc) v3.4.* Available at `http://www.dabeaz.com/ply/ply.html`.

[15] Tewodros Beyene, Swarat Chaudhuri, Corneliu Popeea & Andrey Rybalchenko (2014): *A constraint-based approach to solving games on infinite graphs.* In: *POPL*, pp. 221–233, doi:10.1145/2535838.2535860.

[16] Roderick Bloem, Alessandro Cimatti, Karin Greimel, Georg Hofferek, Robert Könighofer, Marco Roveri, Viktor Schuppan & Richard Seeber (2010): RATSY*–A new requirements analysis tool with synthesis.* In: *CAV*, pp. 425–429, doi:10.1007/978-3-642-14295-6_37.

[17] Roderick Bloem, Stefan Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli & Martin Weiglhofer (2007): *Interactive Presentation: Automatic Hardware Synthesis from Specifications: A Case Study.* In: *Design, Automation and Test in Europe (DATE)*, pp. 1188–1193. Available at `http://dl.acm.org/citation.cfm?id=1266366.1266622`.

[18] Roderick Bloem, Stefan Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli & Martin Weiglhofer (2007): *Specify, Compile, Run: Hardware from* PSL. *ENTCS* 190(4), pp. 3–16, doi:10.1016/j.entcs.2007.09.004.

[19] Roderick Bloem, Swen Jacobs & Ayrat Khalimov (2014): *Parameterized Synthesis Case Study: AMBA AHB*. In Krishnendu Chatterjee, Rüdiger Ehlers & Susmit Jha, editors: *SYNT*, *EPTCS* 157, pp. 68–83, doi:10.4204/EPTCS.157.9. Available at `http://arxiv.org/abs/1407.6580v1`.

[20] Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli & Yaniv Sa'ar (2012): *Synthesis of Reactive(1) designs*. *Journal of Computer and System Sciences (JCSS)* 78(3), pp. 911–938, doi:10.1016/j.jcss.2011.08.007.

[21] Aaron Bohy, Véronique Bruyère, Emmanuel Filiot, Naiyong Jin & Jean-François Raskin (2012): Acacia+*, a tool for LTL synthesis*. In: *CAV*, pp. 652–657, doi:10.1007/978-3-642-31424-7_45.

[22] Manfred Broy (1986): *A theory for nondeterminism, parallelism, communication, and concurrency*. *TCS* 45(0), pp. 1–61, doi:10.1016/0304-3975(86)90040-X.

[23] Randal E. Bryant (1986): *Graph-Based Algorithms for Boolean Function Manipulation*. *IEEE Trans. Comput.* 35(8), pp. 677–691, doi:10.1109/TC.1986.1676819.

[24] Roberto Cavada, Alessandro Cimatti, Charles Arthur Jochim, Gavin Keighren, Emanuele Olivetti, Marco Pistore, Marco Roveri & Andrei Tchaltsev (2010): NuSMV *2.5 User Manual*. Technical Report, Fondazione Bruno Kessler, 18 Via Sommarive, 38055 Povo (Trento), Italy.

[25] Ashok K. Chandra, Dexter C. Kozen & Larry J. Stockmeyer (1981): *Alternation*. *JACM* 28(1), pp. 114–133, doi:10.1145/322234.322243.

[26] Frank Ciesinski, Christel Baier, Marcus Größer & David Parker (2008): *Generating Compact MTBDD-Representations from Probmela Specifications*. In: *SPIN*, pp. 60–76, doi:10.1007/978-3-540-85114-1_7.

[27] Edsger W. Dijkstra (1975): *Guarded Commands, Nondeterminacy and Formal Derivation of Programs*. *CACM* 18(8), pp. 453–457, doi:10.1145/360933.360975.

[28] M.B. Dwyer, G.S. Avrunin & J.C. Corbett (1999): *Patterns in property specifications for finite-state verification*. In: *ICSE*, pp. 411–420, doi:10.1145/302405.302672.

[29] Rüdiger Ehlers (2011): *Experimental aspects of synthesis*. *EPTCS* 50, doi:10.4204/EPTCS.50.

[30] Rüdiger Ehlers (2011): *Generalized Rabin(1) synthesis with applications to robust system synthesis*. In: *NFM*, pp. 101–115, doi:10.1007/978-3-642-20398-5_9.

[31] Rüdiger Ehlers (2011): *Unbeast: Symbolic Bounded Synthesis*. In: *TACAS*, pp. 272–275, doi:10.1007/978-3-642-19835-9_25.

[32] Rüdiger Ehlers & Vasumathi Raman (2014): *Low-Effort Specification Debugging and Analysis*. *EPTCS* 157, pp. 117–133, doi:10.4204/EPTCS.157.12.

[33] Ioannis Filippidis & contributors (2013): *List of verification and synthesis tools*. Available at `https://github.com/johnyf/tool_lists/blob/master/verification_synthesis.md`.

[34] Ioannis Filippidis & Richard M. Murray (2015): *Revisiting the AMBA AHB bus case study*. Technical Report CaltechCDSTR:2015.004, California Institute of Technology, Pasadena, CA. Available at `http://resolver.caltech.edu/CaltechCDSTR:2015.004`.

[35] Ioannis Filippidis, Richard M. Murray & Gerard J. Holzmann (2015): *Synthesis from multi-paradigm specifications*. Technical Report CaltechCDSTR:2015.003, California Institute of Technology, Pasadena, CA. Available at `http://resolver.caltech.edu/CaltechCDSTR:2015.003`.

[36] Bernd Finkbeiner & Sven Schewe (2013): *Bounded synthesis*. *International Journal on Software Tools for Technology Transfer (STTT)* 15(5-6), pp. 519–539, doi:10.1007/s10009-012-0228-z.

[37] Robert W. Floyd (1967): *Nondeterministic Algorithms*. *JACM* 14(4), pp. 636–644, doi:10.1145/321420.321422.

[38] Bjorn N. Freeman-Benson (1990): Kaleidoscope*: Mixing Objects, Constraints, and Imperative Programming*. In: *OOPSLA/ECOOP*, pp. 77–88, doi:10.1145/97946.97957.

[39] Bjørn N. Freeman-Benson & Alan Borning (1992): *Integrating Constraints with an Object-Oriented Language*. In: *ECOOP*, pp. 268–286, doi:10.1007/BFb0053042.

[40] B.N. Freeman-Benson & A Borning (1992): *The design and implementation of* KALEIDOSCOPE*'90-A constraint imperative programming language.* In: *ICCL*, pp. 174–180, doi:10.1109/ICCL.1992.185480.

[41] Abdoulaye Gamatié (2010): *Designing embedded systems with the Signal programming language: synchronous, reactive specification.* Springer, doi:10.1007/978-1-4419-0941-1.

[42] Jeff Gennari, Shaun Hedrick, Fred Long, Justin Pincar & Robert Seacord (2007): *Ranged Integers for the C Programming Language.* Technical Note CMU/SEI-2007-TN-027, Software Engineering Institute, Carnegie Mellon University. Available at `http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=8265`.

[43] Yashdeep Godhal, Krishnendu Chatterjee & Thomas A Henzinger (2013): *Synthesis of AMBA AHB from formal specification: a case study. International Journal on Software Tools for Technology Transfer (STTT)* 15(5-6), pp. 585–601, doi:10.1007/s10009-011-0207-9.

[44] Nicolas Halbwachs (1993): *Synchronous Programming of Reactive Systems. Engineering and Computer Science* 215, Springer, doi:10.1007/978-1-4757-2231-4. Available at `http://www-verimag.imag.fr/~halbwach/newbook.pdf`.

[45] Charles Antony Richard Hoare (1985–2004): *Communicating sequential processes.* 178, Prentice-Hall. Available at `http://www.usingcsp.com/cspbook.pdf`.

[46] Gerard J. Holzmann: PROMELA *Language Reference (`http://spinroot.com/spin/Man/promela.html`).* Available at `http://spinroot.com/spin/Man/promela.html`.

[47] Gerard J. Holzmann (2003): *The SPIN Model Checker, the: Primer and Reference Manual.* Addison-Wesley.

[48] Yong Jiang & Zongyan Qiu (2012): *S2N: model transformation from* SPIN *to* NUSMV. In: *SPIN*, pp. 255–260, doi:10.1007/978-3-642-31759-0_20.

[49] Barbara Jobstmann & Roderick Bloem (2006): *Optimizations for LTL synthesis.* In: *FMCAD*, pp. 117–124, doi:10.1109/FMCAD.2006.22.

[50] Barbara Jobstmann, Stefan Galler, Martin Weiglhofer & Roderick Bloem (2007): ANZU*: A tool for property synthesis.* In: *CAV*, pp. 258–262, doi:10.1007/978-3-540-73368-3_29.

[51] Barbara Jobstmann, Andreas Griesmayer & Roderick Bloem (2005): *Program repair as a game.* In: *CAV*, pp. 226–238, doi:10.1007/11513988_23.

[52] Muriel Jourdan, Fabienne Lagnier, R Maraninchi & Pascal Raymond (1994): *A multiparadigm language for reactive systems.* In: *ICCL*, pp. 211–218, doi:10.1109/ICCL.1994.288379.

[53] Robert M. Keller (1976): *Formal Verification of Parallel Programs. CACM* 19(7), pp. 371–384, doi:10.1145/360248.360251.

[54] Yonit Kesten, Amir Pnueli & Li-on Raviv (1998): *Algorithmic verification of linear temporal logic specifications.* In: *ICALP*, 1443, pp. 1–16, doi:10.1007/BFb0055036.

[55] Uri Klein, Nir Piterman & Amir Pnueli (2012): *Effective synthesis of asynchronous systems from GR(1) specifications.* In: *VMCAI*, pp. 283–298, doi:10.1007/978-3-642-27940-9_19.

[56] M. Kloetzer & C. Belta (2008): *A Fully Automated Framework for Control of Linear Systems from Temporal Logic Specifications. TAC* 53(1), pp. 287–297, doi:10.1109/TAC.2007.914952.

[57] H. Kress-Gazit, G.E. Fainekos & G.J. Pappas (2009): *Temporal-Logic-Based Reactive Mission and Motion Planning. IEEE Transactions on Robotics (TRO)* 25(6), pp. 1370–1381, doi:10.1109/TRO.2009.2030225.

[58] Daniel Kroening & Ofer Strichman (2008): *Decision procedures: An algorithmic point of view.* Springer.

[59] O. Kupferman & M.Y. Vardi (2005): *Safraless decision procedures.* In: *FOCS*, pp. 531–540, doi:10.1109/SFCS.2005.66.

[60] Orna Kupferman (2012): *Recent Challenges and Ideas in Temporal Synthesis*. In: *SOFSEM*, pp. 88–98, doi:10.1007/978-3-642-27660-6_8.

[61] Leslie Lamport (1994): *The Temporal Logic of Actions*. *ACM Trans. Program. Lang. Syst.* 16(3), pp. 872–923, doi:10.1145/177492.177726.

[62] Leslie Lamport (2002): *Specifying Systems: The* TLA+ *Language and Tools for Hardware and Software Engineers*. Addison-Wesley. Available at `http://research.microsoft.com/en-us/um/people/lamport/tla/book.html`.

[63] Leslie Lamport & Fred B. Schneider (1985): *Constraints: A uniform approach to aliasing and typing*. In: *POPL*, pp. 205–216, doi:10.1145/318593.318640.

[64] K Rustan M Leino (2010): *Dafny: An automatic program verifier for functional correctness*. In: *LPAR*, 6355, pp. 348–370, doi:10.1007/978-3-642-17511-4_20.

[65] A Solar Lezama (2008): *Program synthesis by sketching*. Ph.D. thesis, Citeseer. Available at `http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-177.html`.

[66] Orna Lichtenstein, Amir Pnueli & Lenore Zuck (1985): *The glory of the past*. In: *Logics of Programs*, 193, pp. 196–218, doi:10.1007/3-540-15648-8_16.

[67] S.C. Livingston, R.M. Murray & J.W. Burdick (2012): *Backtracking temporal logic synthesis for uncertain environments*. In: *ICRA*, pp. 5163–5170, doi:10.1109/ICRA.2012.6225208.

[68] Gus Lopez, Bjorn Freeman-Benson & Alan Borning (1994): *Implementing Constraint Imperative Programming Languages: The* Kaleidoscope*'93 Virtual Machine*. In: *OOPSLA*, pp. 259–271, doi:10.1145/191080.191118.

[69] Z Manna & Amir Pnueli (1990): *Tools and rules for the practicing verifier*. Technical Report CS-TR-90-1321, Stanford University, CA, USA. Available at `http://i.stanford.edu/pub/cstr/reports/cs/tr/90/1321/CS-TR-90-1321.pdf`.

[70] Zohar Manna & Amir Pnueli (1989): *The anchored version of the temporal framework*. In: *Linear time, branching time and partial order in Logics and models for concurrency*, *LNCS* 354, Springer, pp. 201–284, doi:10.1007/BFb0013024.

[71] Zohar Manna & Amir Pnueli (1990): *A Hierarchy of Temporal Properties*. In: *PODC*, pp. 377–410, doi:10.1145/93385.93442.

[72] Shahar Maoz & Yaniv Sa'ar (2011): AspectLTL*: An Aspect Language for LTL Specifications*. In: *Aspect-oriented Software Development (AOSD)*, ACM, pp. 19–30, doi:10.1145/1960275.1960280.

[73] John McCarthy (1959): *A basis for a mathematical theory of computation*. In P. Braffort & D. Hirschberg, editors: *Computer Programming and Formal Systems*, *Studies in Logic and the Foundations of Mathematics* 26, North-Holland, pp. 33–70, doi:10.1016/S0049-237X(09)70099-0.

[74] Kenneth Lauchlin McMillan (1992): *Symbolic Model Checking: An Approach to the State Explosion Problem*. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA, doi:10.1007/978-1-4615-3190-6. Available at `http://www.kenmcmil.com/pubs/thesis.pdf`. UMI Order No. GAX92-24209.

[75] George H Mealy (1955): *A method for synthesizing sequential circuits*. *Bell System Technical Journal* 34(5), pp. 1045–1079, doi:10.1002/j.1538-7305.1955.tb03788.x.

[76] Edward F Moore (1956): *Gedanken-experiments on sequential machines*. *Automata studies* 34, pp. 129–153.

[77] A. Morgenstern (2010): *Symbolic controller synthesis for LTL specifications*. Ph.D. thesis, Computer Science. Available at `http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:hbz:386-kluedo-25721`.

[78] A. Morgenstern & K. Schneider (2011): *A LTL Fragment for GR(1)-Synthesis*. *EPTCS* 50, pp. 33–45, doi:10.4204/EPTCS.50.3.

[79] David E Muller, Ahmed Saoudi & Paul E Schupp (1986): *Alternating automata, the weak monadic theory of the tree, and its complexity*. In: *ICALP*, pp. 275–283, doi:10.1007/3-540-16761-7_77.

[80] Elie Najm & Frank Olsen (1996): *Protocol Verification with Reactive* PROMELA*/RSPIN*. In: *SPIN*, pp. 109–128. Available at `http://spinroot.com/spin/Workshops/ws96/Ol.pdf`.

[81] Elie Najm & Frank Olsen (1996): *Reactive EFSMs — Reactive Promela/RSPIN*. In: *TACAS*, pp. 349–368, doi:10.1007/3-540-61042-1_54.

[82] Shipra Panda & Fabio Somenzi (1995): *Who are the variables in your neighbourhood*. In: *ICCAD*, pp. 74–77, doi:10.1109/ICCAD.1995.479994.

[83] Nir Piterman, Amir Pnueli & Yaniv Sa'ar (2006): *Synthesis of Reactive(1) designs*. In: *VMCAI*, pp. 364–380, doi:10.1007/11609773_24.

[84] A. Pnueli & R. Rosner (1989): *On the Synthesis of a Reactive Module*. In: *POPL*, pp. 179–190, doi:10.1145/75277.75293.

[85] Amir Pnueli (1977): *The temporal logic of programs*. In: *FOCS*, pp. 46 –57, doi:10.1109/SFCS.1977.32.

[86] Amir Pnueli & Uri Klein (2009): *Synthesis of programs from temporal property specifications*. In: *MEMOCODE*, pp. 1–7, doi:10.1109/MEMCOD.2009.5185372.

[87] Amir Pnueli & Roni Rosner (1989): *On the Synthesis of an Asynchronous Reactive Module*. In: *ICALP*, pp. 652–671, doi:10.1007/BFb0035790.

[88] Amir Pnueli, Yaniv Sa'ar & Lenore D Zuck (2010): JTLV*: A framework for developing verification algorithms*. In: *CAV*, pp. 171–174, doi:10.1007/978-3-642-14295-6_18.

[89] Roni Rosner (1992): *Modular synthesis of reactive systems*. Ph.D. thesis, Weizmann Institute of Science, Rehovot, Israel. Available at `http://www.researchgate.net/publication/238759536_Modular_synthesis_of_reactive_systems/file/50463527f8b648c3ba.pdf`.

[90] Richard Rudell (1993): *Dynamic variable ordering for ordered binary decision diagrams*. In: *ICCAD*, pp. 42–47, doi:10.1109/ICCAD.1993.580029.

[91] Matthias Schlaipfer, Georg Hofferek & Roderick Bloem (2012): *Generalized reactivity(1) synthesis without a monolithic strategy*. In: *HSVT*, pp. 20–34, doi:10.1007/978-3-642-34188-5_6.

[92] Klaus Schneider (2004): *Verification of reactive systems: formal methods and algorithms*. Springer, doi:10.1007/978-3-662-10778-2.

[93] Saqib Sohail, Fabio Somenzi & Kavita Ravi (2008): *A hybrid algorithm for LTL games*. In: *VMCAI*, pp. 309–323, doi:10.1007/978-3-540-78163-9_26.

[94] Fabio Somenzi (2012): CUDD*: CU Decision Diagram package - release 2.5.0*. University of Colorado at Boulder. Available at `http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html`.

[95] Harald Søndergaard & Peter Sestoft (1992): *Non-determinism in functional languages*. *The Computer Journal* 35(5), pp. 514–523, doi:10.1093/comjnl/35.5.514.

[96] Wolfgang Thomas (2008): *Solution of Church's Problem: A tutorial*. *New Perspectives on Games and interaction* 5.

[97] Peter Van-Roy & Seif Haridi (2004): *Concepts, techniques, and models of computer programming*. MIT press.

[98] Moshe Y Vardi (1995): *Alternating automata and program verification*. In: *Computer Science Today*, Springer, pp. 471–485, doi:10.1007/BFb0015261.

[99] Moshe Y Vardi (1996): *An automata-theoretic approach to linear temporal logic*. In: *Logics for concurrency*, *LNCS* 1043, Springer, pp. 238–266, doi:10.1007/3-540-60915-6_6.

[100] Igor Walukiewicz (2004): *A Landscape with Games in the Background*. *LICS* 0, pp. 356–366, doi:10.1109/LICS.2004.1319630.

[101] Tichakorn Wongpiromsarn, Ufuk Topcu & Richard M Murray (2013): *Synthesis of control protocols for autonomous systems*. *Unmanned Systems* 1(01), pp. 21–39, doi:10.1142/S2301385013500027.

Compositional Algorithms for Succinct Safety Games

Romain Brenguier[*], Guillermo A. Pérez[†],
Jean-François Raskin[*], Ocan Sankur[*]

{rbrengui,gperezme,jraskin,osankur}@ulb.ac.be
Université Libre de Bruxelles – Brussels, Belgium

We study the synthesis of circuits for succinct safety specifications given in the AIG format. We show how AIG safety specifications can be decomposed automatically into sub-specifications. Then we propose symbolic compositional algorithms to solve the synthesis problem compositionally starting for the sub-specifications. We have evaluated the compositional algorithms on a set of benchmarks including those proposed for the first synthesis competition organised in 2014 by the Synthesis Workshop affiliated to the CAV conference. We show that a large number of benchmarks can be decomposed automatically and solved more efficiently with the compositional algorithms that we propose in this paper.

1 Introduction

We study the synthesis of circuits for succinct safety specifications given in the AIG format. An AIG file for synthesis describes a circuit that compactly defines a transition relation between valuations for latches, *uncontrollable* and *controllable* input signals. The circuit contains a special latch called the *error latch*. Initially, all latches are false, and the controller chooses values for the controllable input signals so as to always keep the error latch *low* (safety objective), no matter how the environment chooses values for the uncontrollable input signals. The AIG format is *monolithic* in the sense that it is not explicitly structured into subsystems. This is unfortunate as in general, complex systems or specifications are built of smaller sub-parts and taking into account this structure may be a definite advantage.

And-Inverter Graphs (AIG) have been proposed as a way to provide a simple and compact file format for a model checking competition affiliated to CAV 2007 (see `http://fmv.jku.at/aiger/FORMAT`). This format has been extended to be the input format for the 2014 *reactive synthesis competition*. Because the synthesis competition uses the AIG format, and this format is monolithic, all the tools that took part in the 2014 reactive synthesis competition solved the synthesis problems *monolithically*. Nevertheless, the specifications that were proposed during the 2014 synthesis competition are, for a large part of them, generated from higher level descriptions of systems that bear structure. For example, two of the most interesting sets of benchmarks, GenBuf and AMBA, are generated from Reactive(1) specifications (a tractable subset of LTL specifications) [12], or directly from LTL specifications that are conjunctions of smaller LTL sub-formulas.

In this paper, we show that part of the structure lost during the AIG format translation can be recovered and used to solve the synthesis problem *compositionally*. First, we propose a static analysis of the AIG file that returns, when possible, a decomposition of the circuit into smaller sub-circuits with their own safety specifications. Then we provide three different algorithms that first solve the sub-games

[*]Authors supported by the ERC inVEST (279499) project.
[†]Author supported by F.R.S.-FNRS fellowship.

Submitted to:
SYNT 2015

corresponding to the sub-circuits and then aggregate, following three different heuristics, the results obtained on the sub-games. Namely, once we have the solution of all the sub-games we aggregate them by (i) taking their intersection – which, we show, over-approximates the actual solution of the general game – and applying the usual fixpoint algorithm to it; (ii) assigning a score to each pair of solutions based on the number of variables shared and the size of the BDDs obtained after their intersection and using said score to aggregate (pair by pair) all the solutions; (iii) trying to refine them using information from a single step of the fixpoint computation on the general game (*i.e.*projecting the resulting "bad" states onto each sub-game). We have implemented the decomposition, the compositional synthesis algorithms, and evaluated the approach on the 2014 reactive synthesis competition benchmarks as well as on new benchmarks produced from large LTL specifications.

Related Work. In [8, 10], compositional algorithms are proposed for the LTL realizability problem. The LTL formulas considered there are assumed to be conjunctions of smaller LTL formulas, and so the structure of the specification is directly available to them, while in our case it has to be recovered. Also, the main data-structures used there are based on antichains while we use BDDs. In symbolic model checking algorithms, partitioned transition relations [5] are widely used whenever the system is made of several components. Here, the goal is to compute the one-step successor states without explicitly computing the conjunction of the transition relations for each component. The image computation is rather done using *quantification scheduling* heuristics which tries to apply variable quantification as early as possible inside the conjunction; see *e.g.* [17]. We also use partitioned transition relations in our algorithms: the next-state function for each latch is stored separately. Unlike forward model checking algorithms, synthesis algorithms proceed backwards, so we can use the *composition* operation provided by BDD libraries to compute predecessors, and we do not need any early quantification heuristics.

Structure of the paper. In Section 2, we fix notation and recall the definitions needed to present our results. Then, in Section 3, we describe the class of decompositions our algorithms accept as input, we give some examples of how to decompose a succinct safety specification given by an extended AIGER file and outline the algorithm we implemented to get such a decomposition. Our algorithms are described in detail in Section 4 and the results of our tests are presented in Section 5.

2 Preliminaries

Let $\mathbb{B} = \{0,1\}$. Given a set of variables A, a *valuation over* A is an element of $\mathbb{B}^A$, and a set of valuations over A is represented by its characteristic function $f : \mathbb{B}^A \to \mathbb{B}$. We will write $f(A)$ to make the dependency on the variables A explicit. Given two disjoint sets of variables A, B, let us write $\mathbb{B}^{A,B}$ for $\mathbb{B}^A \times \mathbb{B}^B$. Consider variable sets $A \subseteq B$. We define the *projection* of a valuation $v : \mathbb{B}^B$ to A as $v\downarrow_A : \mathbb{B}^A$, with $v\downarrow_A(a) = 1$ if, and only if $v(a) = 1$. We extend this notation to functions $f : \mathbb{B}^B \to \mathbb{B}$ by $f\downarrow_A : \mathbb{B}^A \to \mathbb{B}$, defined as $f\downarrow_A(v)$ if, and only if $\exists v' \in \mathbb{B}^B, f(v')$, and $v = v'\downarrow_A$. We define the *lifting* of a set $f : \mathbb{B}^A \to \mathbb{B}$ in $\mathbb{B}^B$ by $f\uparrow_B(v) = 1$ if, and only if $f(v\downarrow_A) = 1$. For a set of variables $A = \{a_1, a_2, \ldots\}$, let us write $A' = \{a'_1, a'_2, \ldots\}$ the set of *primed variables*. For $f(A)$, let $f(A')$ denote the characteristic function $f(A)$ where each variable $a \in A$ has been renamed as its primed copy $a' \in A'$.

Symbolic Games. We formalize the reactive synthesis problem as a two-player turn-based game with safety objective described symbolically. We consider games defined by sequential synchronous circuits, encoded in the AIGER format. More precisely, a *game* is a tuple $G = \langle L, X_u, X_c, (f_l)_{l\in L}, \mathsf{err}\rangle$, where:

1. X_u, X_c, L are finite disjoint sets of Boolean variables representing *uncontrollable inputs*, *controllable inputs*, and *latches* respectively;

2. for each latch $l \in L$, $f_l \colon \mathbb{B}^L \times \mathbb{B}^{X_u} \times \mathbb{B}^{X_c} \to \mathbb{B}$ is the *transition function* that gives the valuation of l in the next step. In practice these functions will be given by And-Inverter Graphs (see below for a definition).

3. $\mathsf{err} \in L$ is a distinguished latch which indicates whether an error has occurred. We will often modify the circuit by replacing f_{err} by some other Boolean function e, which we denote by $G[f_{\mathsf{err}} \leftarrow e]$.

A *state* q of game G is a valuation of latches, that is an element of $\mathbb{B}^L$. A *valuation* v in game G is a valuation of latches and inputs, that is an element of $\mathbb{B}^{L,X_u,X_c}$. We denote the *global transition function* $\delta \colon \mathbb{B}^L \times \mathbb{B}^{X_u} \times \mathbb{B}^{X_c} \to \mathbb{B}^L$ such that $\delta(v)(l) = f_l(v)$ for each latch l. An *execution* from valuation v of the game G is a sequence of valuations $(v_i)_{i\in\mathbb{N}} \in \left(\mathbb{B}^{L,X_u,X_c}\right)^\omega$ such that $v_0 = v$ and for all i,

$$v_{i+1}\downarrow_L = \delta(v_i\downarrow_L, v_i\downarrow_{X_u}, v_i\downarrow_{X_c}).$$

The execution is *safe* if, for all $i \geq 0$, we have that $v_i(\mathsf{err}) = 0$.

Note that symbolic games define game arenas of exponential size but we will only work on their symbolic representations.

Controller synthesis. The goal of *controller synthesis* is to find a strategy to determine the controllable inputs given uncontrollable inputs and the current state (*i.e.*, valuation of the latches) to ensure that the error state is not reachable. A *strategy* is a function $\lambda : \mathbb{B}^{L,X_u} \to \mathbb{B}^{X_c}$. An execution $(v_i)_{i\in\mathbb{N}}$ is *compatible* with λ if for all $i \in \mathbb{N}$,

$$v_i\downarrow_{X_c} = \lambda(v_i\downarrow_L, v_i\downarrow_{X_u}).$$

A strategy λ is *winning* if all executions that are compatible with λ are safe. A valuation v is *winning* if there exists a strategy λ that is winning from v. We denote $W(L,X_u,X_c)$ the *winning valuations* of G, that is the set of valuations that are winning.

And-Inverter Graphs. An *And-Inverter Graph* (AIG) is a directed acyclic graph with two-input nodes representing logical conjunction (AND gates), terminal nodes representing inputs, and edges that are possibly *inverted* to denote logical negation (NOT gate). Formally, an AIG is a tuple $G = \langle V, E, \iota\rangle$ such that (V,E) is a directed graph with every vertex having 0 or 2 outgoing edges, and $\iota : E \to \mathbb{B}$ labels inverted edges with 1. We depict edges (not) labelled by ι as arrows (not) marked with a dark dot. Figure 1 shows a simple AIG with Boolean variables x_1, x_2, x_3, x_4. Each node in the AIG defines a Boolean function. For example, v_1 defines the Boolean function $\varphi_{v_1} \equiv x_1 \wedge \neg\varphi_{v_2}$, where φ_{v_2} is the corresponding formula defined by v_2, since the edge from v_1 to v_2 is marked as inverted.

The AIGER format (`http://fmv.jku.at/aiger/FORMAT`) was defined as a standard file format to describe sequential synchronous circuits (the logic defined as an AIG), and has been used in model checking and synthesis competitions. In the latter case, the inputs are partitioned into *controllable* and *uncontrollable* (`http://www.syntcomp.org/wp-content/uploads/2014/02/Format.pdf`). This is the format that we will assume as representation of the input game for our algorithms. We call an *AIG game*, a symbolic game described in the AIGER format.

Binary Decision Diagrams. Internally, our tool uses *binary decision diagrams* (BDD) [4] to represent Boolean functions used to represent sets of states or (parts of) transition relations. We use classical operations and notation on BDDs and refer the interested reader to [1] for a gentle introduction to BDDs. Projection and lifting of functions are easily implemented with BDDs: projecting is done by an existential quantification and lifting is a trivial operation because it only extends the domain of the function but its logical representation, *i.e.* its Boolean formula, stays the same.

In our algorithms, we often use BDD operations which implement heuristics to reduce the size of the given BDD, namely, *generalized cofactors* [13,16]. A generalized cofactor $\hat{f}(X)$ of $f(X)$ with respect to $g(X)$ yields a BDD that matches $f(X)$ inside $g(X)$, and is defined arbitrarily outside $g(X)$. This degree of freedom outside $g(X)$ allows heuristics to reduce the BDD size. We write $\hat{f}(X) = f(X) \Downarrow g(X)$. Formally, we have that $\hat{f}(X) \wedge g(X) = f(X) \wedge g(X)$ and $\hat{f}$ has at most the size of f. BDD libraries implement the operations *restrict* or *constrain* (see, *e.g.* [14]), which are specific generalized cofactors.

Classical Algorithms to Solve Safety Games. We recall the basic fixpoint computation for solving safety games, applied here on symbolic safety games. Let $G = \langle L, X_u, X_c, (f_l)_{l \in L}, \mathsf{err} \rangle$ be a symbolic game. The complement of the set $W(L, X_u, X_c) \downarrow_L$ can be computed by iterating an *uncontrollable predecessors* operator. For any set of states $S(L)$, the *uncontrollable predecessors* of S is defined as

$$\mathtt{upre}_G(S) = \{q \in \mathbb{B}^L \mid \exists x_u \in \mathbb{B}^{X_u}.\ \forall x_c \in \mathbb{B}^{X_c} : \delta(q, x_u, x_c) \in S\};$$

the dual *controllable predecessors* operator is defined as

$$\mathtt{cpre}_G(S) = \{q \in \mathbb{B}^L \mid \forall x_u \in \mathbb{B}^{X_u}.\ \exists x_c \in \mathbb{B}^{X_c} : \delta(q, x_u, x_c) \in S\};$$

We denote by $\mathtt{upre}^*_G(S) = \mu X.(S \cup \mathtt{upre}_G(X))$, the *least fixpoint* of the function $F : X \to S \cup \mathtt{upre}_G(X)$ in the μ-calculus notation (see [7]). Note that F is defined on the powerset lattice, which is finite. It follows from Tarski-Knaster theorem [15] that, because F is monotonic, the fixpoint exists and can be computed by iterating the application of F starting from any value below it, *e.g.*the least value of the lattice. Similarly, we denote by $\mathtt{cpre}^*_G(S) = \nu X.(S \cap \mathtt{cpre}_G(X))$, the *greatest fixpoint* of the function $F : X \to S \cap \mathtt{cpre}_G(X))$. Dually, we have that, because F is monotonic, the fixpoint exists and can be computed by iterating the application of F starting from any value above it, *e.g.*the greatest value of the lattice. When G is clear from the context, we simply write `upre` (`cpre`) instead of $\mathtt{upre}_G$ ($\mathtt{cpre}_G$). The Proposition follows from well-known results about the relationship between safety games and these operators (see, *e.g.*, [2]).

Proposition 1. *For any symbolic game $G = \langle L, X_u, X_c, (f_l)_{l \in L}, \mathsf{err} \rangle$, we have $\mathtt{cpre}^*((\mathsf{err} \mapsto 0) \uparrow_L) = \mathtt{cpre}(W(L, X_u, X_c) \downarrow_L)$; dually, $\mathtt{upre}^*((\mathsf{err} \mapsto 1) \uparrow_L) = \neg\,\mathtt{cpre}(W(L, X_u, X_c) \downarrow_L) = \mathtt{upre}(\neg W(L, X_u, X_c) \downarrow_L)$.*

In the rest of the paper, we assume a black-box procedure `solve_vals` which, for a given symbolic game, computes the corresponding winning valuations. In practice, `solve_vals` can be implemented using `upre` or `cpre`. Formally,

$$\mathtt{solve\_vals}(G) = \{(q, x_u, x_c) \in \mathbb{B}^{L, X_u, X_c} \mid q(\mathsf{err}) = 0 \wedge \delta(q, x_u, x_c) \notin \mathtt{cpre}^*((\mathsf{err} \mapsto 0) \uparrow_L)\}.$$

Note that `solve_vals` gives the set of winning valuations, and not the set of winning states. The interpretation of $\mathtt{solve\_vals}(G)$ is that it is the maximal permissive strategy: any strategy for the controller that ensures to stay within this set is a winning strategy. We also consider procedure $\mathtt{solve\_states}(G) = \{q \in \mathbb{B}^L \mid q \in \mathtt{cpre}^*((\mathsf{err} \mapsto 0) \uparrow_L)\}$ which returns the set of winning states.

Optimizations Using Generalized Cofactors. Let us now establish the correctness of two optimizations we use in the sequel.

We first formalize the dependence on latches as follows. The *cone of influence* (see, *e.g.*, [6]) of e_i, written $\texttt{cone}(e_i)$, is the set of variables on which e_i depends, that is, $\texttt{cone}(\Phi) \subseteq L \cup X_u \cup X_c$ is the minimal set of variables such that if $x \in \texttt{cone}(\Phi)$ then either $(\exists x : \Phi) \not\Leftrightarrow \Phi$ or $x \in \texttt{cone}(f_y)$ for some $y \in \texttt{cone}(\Phi) \cap L$. For convenience, we denote by $\texttt{cone}_L(\Phi)$ the set $\texttt{cone}(\Phi) \cap L$.

Observe that we have defined the cone of influence of a Boolean function semantically. That is to say, a variable x is in the cone of influence of a function Φ if and only if the set of valuations satisfying Φ changes for some fixed valuation of x. Since we consider functions given by AIGs, the cone of influence can be over-approximated by exploring the AIG starting from the vertex corresponding to function Φ, adding all latches and inputs visited and the cones of influence of the latches – computed recursively. In our implementation we use this over-approximation when working on the AIG only and we use the definition on the semantics to obtain an algorithm on BDDs – which we use when working with BDDs.

Given an over-approximation Λ of the winning valuations (i) we first simplify the transition relation and keep it precise only in Λ, (ii) we further modify the transition relation by making every transition not allowed by Λ go to an error state, *i.e.*change f_{err}. In fact, correctness of the first optimization requires that the second one be used as well. The following result summarizes the properties of these optimizations.

Lemma 1. *For any symbolic game $G = \langle L, X_u, X_c, (f_l)_{l\in L}, \mathsf{err}\rangle$, and any $\Lambda(L, X_u, X_c) \supseteq W(L, X_u, X_c)$, if we write $f'_l = f_l \Downarrow \Lambda$ for all $l \in L$, we have*

$$\mathit{solve\_vals}(G) = \mathit{solve\_vals}(\langle \mathit{cone}_L(\Lambda), X_u, X_c, (f'_l)_{l\in \mathit{cone}_L(\Lambda)}\rangle[f'_{\mathsf{err}} \leftarrow \neg\Lambda]) \uparrow_L .$$

Proof. We first show that solving the game with error function $\neg\Lambda$ yields the same winning valuations as for f_{err}. For that we will use two basic properties of the winning valuations: first if $f \subseteq f'$ then

$$\texttt{solve\_vals}(G[f_{\mathsf{err}} \leftarrow f']) \subseteq \texttt{solve\_vals}(G[f_{\mathsf{err}} \leftarrow f]);$$

secondly

$$\texttt{solve\_vals}(G[f_{\mathsf{err}} \leftarrow \neg\texttt{solve\_vals}(G)]) = \texttt{solve\_vals}(G),$$

this is because if an execution compatible with strategy λ reaches $\neg\texttt{solve\_vals}(G)$, then by definition of winning valuations it can be extended from there to an execution compatible with λ that is unsafe. Together with the fact that $f_{\mathsf{err}} \subseteq \neg\Lambda \subseteq \neg W$, these properties imply that $\texttt{solve\_vals}(G) = \texttt{solve\_vals}(G[f_{\mathsf{err}} \leftarrow \neg\Lambda])$. It is clear that one can consider only the variables in $\texttt{cone}_L(\Lambda)$ for this computation, and thus considering $H = (\langle \texttt{cone}_L(\Lambda), X_u, X_c, (f_l)_{l\in\texttt{cone}_L(\Lambda)}\rangle[f_{\mathsf{err}} \leftarrow \neg\Lambda])$, we have

$$\texttt{solve\_vals}(G) = \texttt{solve\_vals}(H) \uparrow_L .$$

It remains to show that the same set $\texttt{solve\_vals}(H)$ is obtained when the functions f'_l are used transition functions f_l. Let us denote $G' = (\langle \texttt{cone}_L(\Lambda), X_u, X_c, (f'_l)_{l\in\texttt{cone}_L(\Lambda)}\rangle[f'_{\mathsf{err}} \leftarrow \neg\Lambda])$. We note that, for any $u \supseteq \neg\Lambda$, the following holds:

$$\texttt{upre}'_G(u) \cup u = \texttt{upre}_H(u) \cup u.$$

Hence, it is straightforward to show by induction that $\texttt{solve\_vals}(H) = \texttt{solve\_vals}(G')$. □

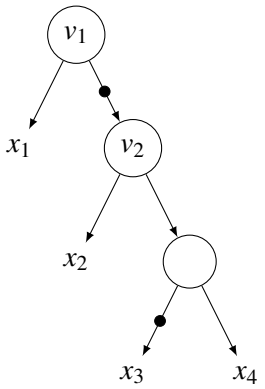


Figure 1: Example AIG

3 Decomposing the Specification

In this section, we describe how we decompose the error function f_{err} of a given symbolic game into a disjunction *i.e.* $f_{\mathsf{err}} \equiv \left(\bigvee_{1\leq i\leq n} e_i\right)$. Notice that if a strategy $\lambda(L,X_u,X_c)$ ensures that f_{err} is never true then it also ensures that e_i is never true. We will then give algorithms that solve the game where each e_i is seen as the error function, and combine the obtained solutions into a global solution.

The rationale behind this approach is that the functions e_i do not depend on all latches in general, so solving the game for e_i is often efficient.

Sub-game. Given a decomposition of f_{err}, we define a *sub-game* G_i by replacing the error function by e_i and considering only variables in its cone of influence. Formally, we write

$$G_i = \langle \mathtt{cone}_L(e_i), X_u, X_c, (f_l)_{l\in\mathtt{cone}_L(e_i)}\rangle[f_{\mathsf{err}} \leftarrow e_i].$$

We will often use the notation $G[f_{\mathsf{err}} \leftarrow e_i]$, which consists in replacing the function f_{err} by e_i. In practice, the size of the symbolic representation of the sub-games are often significantly smaller than that of the original game. Recall also that winning all the sub-games is necessary to win the global game. We write $W_i(\mathtt{cone}_L(e_i), X_u, X_c)$ for the winning valuations of G_i. In the implementation, S_i and $S_i \uparrow_L$ are represented by the same BDD.

Example 1. Consider the AIG shown in Figure 1 where x_1, x_2, x_3, x_4 are all input variables. We would like to decompose the function defined by the sub-tree rooted at v_1 (*i.e.* the whole tree) which we will denote by φ_{v_1}. It should be clear that $\varphi_{v_1} \equiv x_1 \wedge \neg\varphi_{v_2}$ where φ_{v_2} is the function defined by the sub-tree rooted at v_2. In turn, we also have that $\varphi_{v_2} \equiv x_2 \wedge \neg x_3 \wedge x_4$. If we distribute the disjunction from $\neg f_{v_2}$ we get that $\varphi_{v_1} \equiv (x_1 \wedge \neg x_2) \vee (x_1 \wedge x_3) \vee (x_1 \wedge \neg x_4)$. Thus, one possible decomposition of φ_{v_1} would be to take $e_1 = x \wedge \neg x_2$, $e_2 = x_1 \wedge x_3$, and $e_3 = x_1 \wedge \neg x_4$.

The general steps followed in Example 1 above can be generalized into an algorithm which outputs a decomposition of the error function whenever one exists. Intuitively, the algorithm consists in exploring all non-inverted edges of the AIG graph from the vertex which defines the error function. If there are no inverted edges which stopped the exploration, or if all of them lead to leaves, the error function is in fact a conjunction of Boolean variables and can clearly not be decomposed. Otherwise, there is at least one inverted edge leading to a node representing an AND gate. In this case, we can push the

negation one level down and obtain a disjunction which can be distributed to obtain our decomposition. Algorithm 2 details the procedure we have implemented. It takes as input an AIG, whether the error function is inverted, and the vertex v_{err} which defines the error function. It outputs a set of functions whose conjunction is logically equivalent to the error function.

We have kept our description of Algorithm 2 and Algorithm 1 (called by the former) informal. A more formal discussion on their correctness is given in Appendix B.

Algorithm 1: `get_minput_and`(V,E,ι,v_0)

```
1  to_visit := {v_0};
2  pos := {};
3  neg := {};
4  while |to_visit| > 0 do
5      Pop u ∈ to_visit;
6      if u is not a leaf then
7          Let e = (u,v) and e' = (u,v') be s.t. e,e' ∈ E;
8          if ι(e) = 1 then
9              neg := neg ∪ {v};
10         else
11             to_visit := to_visit ∪ {v};
12         if ι(e') = 1 then
13             neg := neg ∪ {v};
14         else
15             to_visit := to_visit ∪ {v};
16     else
17         pos := pos ∪ {u};
18 return (pos, neg)
```

Algorithm 2: `decompose`$(V,E,\iota,inv,v_{\mathsf{err}})$

```
1  (pos,neg) := get_minput_and(V,E,ι,v_err);
2  if inv = 1 then
3      return {¬φ_v | v ∈ pos} ∪ {φ_v | v ∈ neg}
4  if inv = 0 and all v ∈ neg are leaves then
5      return {φ_v_err};                          /* No decomposition possible */
6  Take v_0 ∈ argmax{||get_minput_and(V,E,ι,v)|| | v ∈ neg};   /* where ||(S_1,S_2)|| := |S_1|+|S_2| */
7  res := ⋀_{u∈pos} φ_u ∧ ⋀_{v∈neg\{v_0}} ¬φ_v;
8  (pos,neg) := get_minput_and(V,E,ι,v_0);
9  return {res ∧ (¬φ_v) | v ∈ pos} ∪ {res ∧ φ_v | v ∈ neg}
```

Example 2. Consider a formula given by a set of assumption formulas $\{A_i(L,X_u) \mid 1 \leq i \leq n\}$ and a set of guarantees $\{G_j(L,X_u,X_c) \mid 1 \leq j \leq m\}$.[1] The system we want to synthesize is expected to determine the controllable inputs in way such that if the assumptions are true, then the guarantees are met. This is formally stated as Equation 1.

$$\Phi = \Big(\bigwedge_{1\leq i\leq n} A_i \Big) \implies \Big(\bigwedge_{1\leq j\leq m} G_j \Big) \tag{1}$$

[1] This is actually the way in which the error formula is stated for, *e.g.*, the AMBA benchmarks.

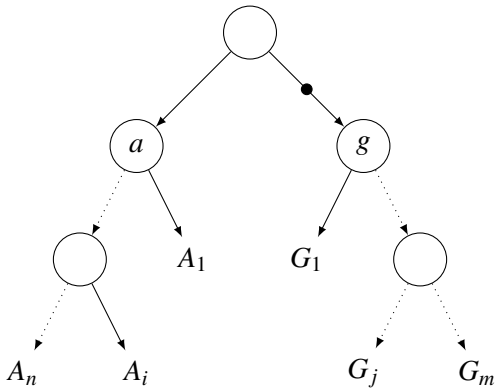


Figure 2: One possible AIG for Equation 1

A natural decomposition for the error function $\neg\Phi$ would be the following: $\bigvee_{1\leq j\leq m}\left(\neg G_j \wedge \bigwedge_{1\leq i\leq n} A_i\right)$. If $\neg\Phi$ were given as the AIG depicted in Figure 2, then it is not hard too see that Algorithm 2 would yield a very similar decomposition. Indeed, as we have not assumed anything in particular about the formulas A_i and G_j we cannot tell whether Algorithm 1 will explore beyond each G_j, thus giving us more sub-games than the proposed decomposition. However, in practice, this is even better as smaller sub-games usually depend on less variables. This, in turn, could lead to them being easier to solve.

Lemma 2. *For each sub-game G_i with new error function e_i, we have that*

$$W(L,X_u,X_c) \subseteq (W_i \uparrow_L)(L,X_u,X_c).$$

Proof. For each valuation $v' \in W(L,X_u,X_c)\downarrow_{\mathtt{cone}_L(e_i)\cup X_u\cup X_c}$, we select a valuation $v \in W(L,X_u,X_c)$. Let λ_v be a winning strategy in G from v. Since there is no losing outcome for λ_v, for all $x_u \in \mathbb{B}^{X_u}$, $\lambda_v(\delta(v),x_u)$ is such that $(\delta(v),x_u,\lambda_v(\delta(v),x_u)) \in W(L,X_u,X_c)$. For all $x_u \in \mathbb{B}^{X_u}$, we fix $\lambda'(\delta(v'),x_u)$ to be $\lambda_v(\delta(v),x_u)$. We have that $(\delta(v'),x_u,\lambda'(\delta(v'),x_u)) \in W(L,X_u,X_c)\downarrow_{\mathtt{cone}_L(e_i)\cup X_u\cup X_c}$ because the transition relations of G and G_i coincide on $\mathtt{cone}_L(e_i)\cup X_u\cup X_c$. The strategy λ' ensures that any execution which starts in $W(L,X_u,X_c)\downarrow_{\mathtt{cone}_L(e_i)\cup X_u\cup X_c}$ stays inside $W(L,X_u,X_c)\downarrow_{\mathtt{cone}_L(e_i)\cup X_u\cup X_c}$. Since e_i evaluates to false on $W(L,X_u,X_c)$, these states are not error states in G_i. Therefore λ' is winning for all states in $W(L,X_u,X_c)\downarrow_{\mathtt{cone}_L(e_i)\cup X_u\cup X_c}$. This implies that W_i contains the projection of all winning states of G and therefore $W \subseteq W_i \uparrow_L$. □

4 Compositional Algorithms

In this section, we give three algorithms to solve AIG games compositionally. Each algorithm first solves the sub-games, and then combines the solutions using different heuristics. We denote by `decompose` the procedure that implements the decomposition of f_{err} described in Section 3, and returns the set of error functions e_i. In all three algorithms, we start by solving each sub-game and obtaining the winning valuations $W_i(L,X_u,X_c)$, for $1 \leq i \leq n$. These steps are given in lines 1–3, and are common to ll our algorithms; we assume that `solve_vals` raises an exception and terminates the program if the sub-game cannot be won. Otherwise, we aggregate the results and solve the global game; for the latter, we adopt a different approach in each of the three algorithms.

Algorithm 3: $\texttt{comp\_1}(\langle L, X_u, X_c, (f_l)_{l\in L}\rangle)$

$\{e_1,\dots,e_n\} := \texttt{decompose}(f_{\mathsf{err}})$; /* Formulas $e_i(L,X_u,X_c)$ s.t. $f_{\mathsf{err}} \equiv \bigvee_{1\le i\le n} e_i$ */
for $1 \le i \le n$ **do**
  $w_i(L,X_u,X_c) := \texttt{solve\_vals}(\langle \texttt{cone}_L(e_i), X_u, X_c, (f_l)_{l\in\texttt{cone}_L(e_i)}[f_{\mathsf{err}} \leftarrow e_i]\rangle)\uparrow_{L,X_u,X_c}$;
$\Lambda(L,X_u,X_c) := \bigwedge_{1\le i\le n} w_i(L,X_u,X_c)$;
for $l \in \mathit{cone}_L(\Lambda)$ **do** $f'_l(L,X_u,X_c) := f_l(L,X_u,X_c) \Downarrow \Lambda(L,X_u,X_c)$;
return $\texttt{solve\_vals}(\langle \texttt{cone}_L(\Lambda), X_u, X_c, (f'_l)_{l\in\texttt{cone}_L(\Lambda)}\rangle[f'_{\mathsf{err}} \leftarrow \neg\Lambda])\uparrow_{L,X_u,X_c}$;

4.1 Global aggregation

In Algorithm 3, we start by computing the intersection of the winning valuations: $\Lambda = \bigwedge_{1\le i\le n} W_i$. In fact, any valuation that is not in Λ is losing in one of the sub-games; thus in the global game. Conversely, a strategy that stays in Λ is winning for each sub-game. Therefore, we solve the global game with the new safety objective of avoiding $\neg\Lambda$. Before solving the global game, the algorithm attempts to reduce the size of the transition relations by virtue of Lemma 1.

Theorem 1. *Algorithm 3 computes the winning valuations of the given AIG game.*

Proof. We prove first that $W \subseteq \Lambda$ (that is for all valuation v, $W(v) \Rightarrow \Lambda(v)$). Since $\neg e_i \supseteq W(L,X_u,X_c)$, we get – by Lem. 1 – that each $w_i(L,X_u,X_c)$ is $W_i\uparrow_L$ where W_i is the winning valuations of the sub-game G_i. If $q \notin \Lambda(L,X_u,X_c)$, there is a sub-game G_i such that $\pi_i(q)$ is not winning. By Lem. 2, this implies that q is not winning in G, hence $q \notin W(L,X_u,X_c)$.

From Lem. 1 it then follows that $\texttt{solve\_vals}(G) = \texttt{solve\_vals}(G')\uparrow_L$ and therefore the algorithm computes the correct result. □

4.2 Incremental aggregation

Algorithm 4: $\texttt{comp\_2}(\langle L, X_u, X_c, (f_l)_{l\in L}\rangle, \alpha, \beta, \gamma)$

$\{e_1,\dots,e_n\} := \texttt{decompose}(f_{\mathsf{err}})$; /* Formulas $e_i(L,X_u,X_c)$ s.t. $f_{\mathsf{err}} \equiv \bigvee_{1\le i\le n} e_i$ */
for $1 \le i \le n$ **do**
  $w_i(L,X_u,X_c) := \texttt{solve\_vals}(\langle \texttt{cone}_L(e_i), X_u, X_c, (f_l)_{l\in\texttt{cone}_L(e_i)}[f_{\mathsf{err}} \leftarrow e_i]\rangle)\uparrow_{L,X_u,X_c}$;
$E := \{w_i \mid 1 \le i \le n\}$;
while $|E| > 1$ **do**
  $(r,s) := \operatorname{argmax}_{(i,j)\in|E|^2: i\neq j}$ $\{\alpha \cdot \texttt{bddsize}(\neg(w_i \wedge w_j))$
    $+\beta|\texttt{cone}_L(w_i) \cap \texttt{cone}_L(w_j)|$
    $+\gamma|\texttt{cone}_L(w_i) \cup \texttt{cone}_L(w_j)|\}$;
  for $l \in \mathit{cone}_L(w_r \wedge w_s)$ **do** $f'_l(L,X_u,X_c) := f_l(L,X_u,X_c) \Downarrow (w_r \wedge w_s)$;
  $w(L,X_u,X_c) := \texttt{solve\_vals}(\langle \texttt{cone}_L(w_r \wedge w_s), X_u, X_c, (f'_l)_{l\in\texttt{cone}_L(w_r\wedge w_s)}\rangle[f_{\mathsf{err}} \leftarrow \neg w_r \vee \neg w_s])\uparrow_{L,X_u,X_c}$;
  Remove w_r, w_s and add w to E;
return last $w(L,X_u,X_c) \in E$;

In Algorithm 4, we aggregate the results of the sub-games *incrementally*: given the list of winning valuations w_i for the sub-games, at each iteration, we choose and remove two sub-games i and j, solve their conjunction (as in Algorithm 3, with error function $\neg(w_i \wedge w_j)$), and add the newly obtained winning valuations back in the list. To choose the sub-games, we use the following heuristics; we assign a score to each pair of sub-games based on the size of the BDD of the error function $\neg(w_i \wedge w_j)$, and on the number

of shared latches, and the number of the latches that appear in either of the sub-games. Intuitively, we prefer to work with small BDDs, and to merge sub-games that share a lot of latches, while yielding a small number of total latches. We thus use a linear combination at line 6 to choose the best scoring pair. In our experiments, we used $\alpha = -2, \beta = 1, \gamma = -1$.

Theorem 2. *Algorithm 4 computes the winning valuations of the given AIG game.*

Proof. Let us denote by $w^i_1, \ldots, w^i_{n_i}$ the content of E at the beginning of iteration i. We define a function F from winning valuations w^i_j to subsets of $\{1, \ldots, n\}$. Intuitively, $F(w^i_j)$ is the set of sub-games that were solved to obtain w^i_j. For instance, at the first iteration, if sub-games r, s are combined – and the result, w, is added to E – then we get $F(w) = \{r, s\}$. For convenience, we assume that w is appended at the end of the sequence $w^i_1, \ldots, w^i_{n_i}$ at line 9.

We proceed by induction on i to define F. Initially $F(w^1_i) = \{i\}$ for all $1 \leq i \leq n$. For $i > 1$, for all $j \neq r, s$, the element w^i_j remains in the list so F is already defined on w^i_j. For the newly element $w^i_{n_i}$ we let $F(w^i_{n_i}) = F(w^{i-1}_r) \cup F(w^{i-1}_s)$.

We claim that at any iteration i, w^i_j is the winning valuations of the game whose error function is the disjunction of the negation of the winning valuations of the sub-games in $F(w^i_j)$. More precisely,

$$w^i_j = \texttt{solve\_vals}(\langle L, X_u, X_c, (f_l)_{l \in L} \rangle [f_{\mathsf{err}} \leftarrow \bigvee_{k \in F(w^i_j)} e_k]).$$

The correctness of the algorithm will follow since the sets $F(\cdot)$ are merged at each iteration, and the algorithm always stops with $|E| = 1$ and $F(w) = \{1, \ldots, n\}$.

The condition holds initially as shown in Theorem 1. Let $i > 1$. As shown in Lem. 1, the generalized cofactor operation applied before the call to `solve` does not affect the returned set. Let us denote $E_r = \bigvee_{k \in F(w^{i-1}_r)} e_k$ and $E_s = \bigvee_{k \in F(w^{i-1}_s)} e_s$. Let us write $\mathscr{E} = E_r \vee E_s$. We have $E_r \Rightarrow \neg w_r$ by induction, and similarly $E_s \Rightarrow \neg w_s$; thus $\mathscr{E} \Rightarrow \neg w_r \vee \neg w_s$. Moreover, for any $q(L, X_u)$ if the controller plays strategy $x_u \in \mathbb{B}^{X_c}$ with $\neg w_r(q, x_u)$, or $\neg w_s(q, x_u)$, then he loses for the error function defined by $\mathscr{E}$. In other terms, $\neg w_r \vee \neg w_s$ is a subset of losing valuations for error function $\mathscr{E}$, and contains $\mathscr{E}$, the set of states losing in one step. It follows that $w(L, X_u, X_c)$ computed at step 8 is the winning valuations for the error function $\mathscr{E}$. □

4.3 Back-and-forth

In Algorithm 5, we interleave the analysis of the global game (with objective Λ) and the analysis of the sub-games. At each iteration, we extend the losing states $u(L)$ by one step, by applying once the `upre` operator. We then consider each sub-game, and check whether the new set $u'(L)$ of losing states (projected on the sub-game), changes the local winning states. Here, $p_i(L)$ is this projection on the local state-space of sub-game i. We update the strategies λ_i of the sub-games when necessary, and restart until stabilization. Because analyzing the sub-games is often more efficient than analyzing the global game, this algorithm improves over Algorithm 3 in some cases (see the experiments' section). A similar idea was used in [9] for the problem of synthesis from LTL specifications.

Theorem 3. *Algorithm 5 computes the winning valuations of the given AIG game.*

Proof. Let $W(L)$ denote the set of winning states of the game G. We consider the following invariant.

$$\begin{array}{l} \forall i \in \{1, \ldots, n\}, W(L) \subseteq s_i(L), \\ \mathsf{err} \subseteq u'(L) \subseteq \neg W(L). \end{array} \tag{2}$$

Algorithm 5: $\texttt{comp\_3}(\langle L, X_u, X_c, (f_l)_{l\in L}\rangle)$

1 $\{e_1,\dots,e_n\} := \texttt{decompose}(f_{\mathsf{err}})$; /* Formulas $e_i(L,X_u,X_c)$ s.t. $f_{\mathsf{err}} \equiv \bigvee_{1\le i\le n} e_i$ */
2 **for** $1 \le i \le n$ **do**
3 $w_i(L,X_u,X_c) := \texttt{solve\_vals}(\langle \texttt{cone}_L(e_i), X_u, X_c, (f_l)_{l\in\texttt{cone}_L(e_i)}[f_{\mathsf{err}} \leftarrow e_i]\rangle)\uparrow_{L,X_u,X_c}$;
4 $s_i(L) := \texttt{solve\_states}(\langle \texttt{cone}_L(e_i), X_u, X_c, (f_l)_{l\in\texttt{cone}_L(e_i)}[f_{\mathsf{err}} \leftarrow e_i]\rangle)\uparrow_L$;
5 $\Lambda(L,X_u,X_c) := \bigwedge_{1\le i\le n} w_i(L,X_u,X_c)$;
6 $G' := \langle \texttt{cone}_L(\Lambda), X_u, X_c, (f_l)_{l\in\texttt{cone}_L(\Lambda)}\rangle[f_{\mathsf{err}} \leftarrow \neg\Lambda]$;
7 $u(L) := 0$;
8 $u'(L) := \bigvee_{1\le i\le n} \neg s_i(L)$; /* The union of all the losing states */
9 **while** $u \neq u'$ **do**
10 $u(L) := u'(L)$;
11 $u'(L) := u(L) \vee \texttt{upre}_{G'}(u)$;
12 **for** $1 \le i \le n$ **do**
13 $p_i(L) := \forall L \setminus \texttt{cone}_L(e_i) : u'(L)$; /* Universal projection of latches not present in local sub-game */
14 **if** $p_i \wedge s_i \neq 0$ **then**
15 **for** $l \in \texttt{cone}_L(p_i)$ **do** $f'_l(L,X_u,X_c) := f_l(L,X_u,X_c) \Downarrow \neg p_i(L)$;
16 $s_i(L) := \texttt{solve\_states}(\langle X_u, X_c, \texttt{cone}_L(p_i), (f'_l)_{l\in\texttt{cone}_L(p_i)}\rangle[f_{\mathsf{err}} \leftarrow p_i \uparrow_{L,X_u,X_c}])\uparrow_L$;
17 $u'(L) := u'(L) \vee \neg \bigwedge_{1\le i\le n}(s_i(L)\downarrow_L)$;
18 **return** $\neg u(L)$;

In words, in every iteration, $u'(L)$ is contained in the losing valuations of the global game, and each $s_i(L)$ contains the winning valuations of G_i.

Initially, by Algorithm 3, $W \subseteq s_i(L)$ for all i, and we have $\mathsf{err} \subseteq \neg s_i(L)$. So $\mathsf{err} \subseteq \neg \wedge_i s_i(L)$. Thus, $\mathsf{err} \subseteq u'(L)$. Moreover, since $\vee_i \neg s_i(L) \subseteq \neg W$, we have that $u'(L) \subseteq \neg W$.

Consider now iteration $i > 1$, and assume the invariant holds at the beginning. $u'(L)$ is updated at line 11. The property $\mathsf{err} \subseteq u'(L) \subseteq \neg W$ still holds by the definition of the $\texttt{upre}$ operation, and by the fact that the set u' can only grow at this step (because of the union).

We consider now the for loop, and show that $W \subseteq s_i$ after each iteration. Assume $p_i \cap s_i \neq \emptyset$ since otherwise s_i is not modified. By definition $p_i \subseteq u'$ thus $p_i \subseteq \neg W$. Then the $\texttt{solve}$ function computes the set of states from which the controller can avoid $\mathsf{err} \vee p_i$. Since $\mathsf{err} \vee p_i \subseteq \neg W$, we get that $s_i \subseteq W$. It follows that $\neg \bigwedge_{i=1}^n s_i \subseteq \neg W$. Thus, at the last line of the while loop, we have $\mathsf{err} \subseteq u'(L) \subseteq \neg W$.

Now, line 11 ensures that after iteration i, $u'(L)$ contains the i-th iteration of the $\texttt{upre}$ fixpoint computation. Hence, the test $u \neq u'$ of the while loop ensures that the while loop terminates with $u(L)$ being equal to $\texttt{upre}^*(G)$. □

5 Experiments

We implemented our algorithms in the synthesis tool AbsSynthe [3]. We compare their running times against the most efficient algorithm of AbsSynthe that implements a backward fixpoint algorithm.[2] This algorithm was the winner of the 2014 **Synthesis Competition** synthesis track, and the winner of the realizability track at the same competition implemented a similar backward algorithm.

Let us first illustrate the advantage of the compositional approach with two examples. In the first

[2]The new version of AbsSynthe with the implementation of the compositional algorithms can be fetched from `https://github.com/gaperez64/abssynthe`.

set of benchmarks we consider, the controller is to compute the multiplication of two Boolean matrices given as (uncontrollable) input. Since each cell of the resulting matrix depends only on a subset of inputs, namely, on one row and one column, these benchmarks are well adapted for compositional algorithms. Figure 3 compares the performances of the classical algorithm with Algorithm 3. The classical algorithm was able to solve 36 instances, while the compositional algorithm solved all 75 instances and was significantly faster. The x-axis shows the number of solved benchmarks within the running time given by the y-axis. The second set of benchmarks we consider consist in a washing system made of n tanks. An uncontrollable input can request at any time the tank to be activated, at which point the controller should fill the tank with water, and empty it after at least k steps. Moreover, some subsets of tanks cannot be filled at the same time, and a light is to be on if at least one tank is filled with water. Note that the control strategy for each tank is not independent due to mutual exclusion constraints, and to the light indicator. Algorithm 3 was also more efficient on these benchmarks, as shown on Fig. 4. The classical algorithm solved 132 benchmarks out of 256, while Alg. 3 solved 152.

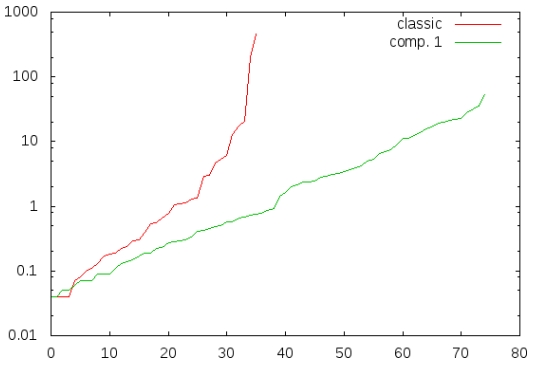


Figure 3: Performances for 75 Boolean matrix multiplication benchmarks for Algorithm 3 and the classical algorithm.

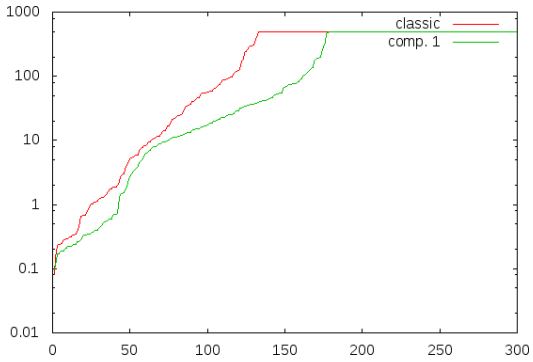


Figure 4: Peformances for the 256 washing system benchmarks for Algorithm 3 and the classical algorithm.

We now evaluate all three compositional algorithms and compare them with the classical algorithm on a large benchmark set of 674 benchmarks. 562 of these benchmarks were provided for the 2014 **Synthesis Competition** and 105 have been generated by the new version of LTL2AIG [11] which translates conjunctions of LTL specifications into AIG.[3] Among those benchmarks, 351 are decomposable by our static analysis into at least 2 smaller sub-games. More specifically, the average number of sub-games our decomposition algorithm outputs is 29; the median is 21.

In general, the performances of the three compositional algorithms can differ, but they are complementary. Figures 5 to 8 show the performances of the algorithms on several sets of benchmarks. All benchmarks in Figures 5 and 6 are decomposable. Figure 7 shows all the benchmarks we used and Figure 8 shows only those benchmarks from last year's synthesis competition which were based on specifications of the AMBA arbiter.

Conclusion. Even if AIG synthesis problems are monolithic, the experiments show that our compositional approach was able to solve problems that can not be handled by the monolithic backward algorithm; our compositional algorithms are sometimes much more efficient. There are also examples that

[3] A collection of benchmarks, including the ones mentioned here, can be fetched from `https://github.com/gaperez64/bench-syntcomp14` and `https://github.com/gaperez64/bench-ulb-syntcomp15`.

can be decomposed but which are not solved more efficiently by the compositional algorithms. So, it is often a good idea to apply all the algorithms in parallel. This portfolio approach improved the performance and was able to solve 20 benchmarks that could not be solved by the fastest algorithms of last year's reactive synthesis competition.

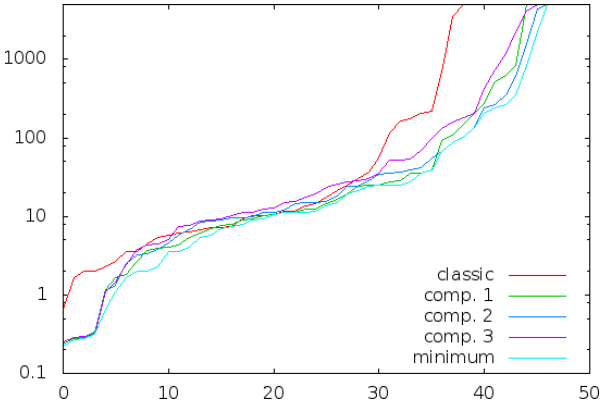


Figure 5: Performances for 68 load-balancing benchmarks translated from LTL. The classical algorithm solves 38 benchmarks, comp.1 44, comp.2 45, comp.3 45. In total there are 46 benchmarks that can be solved. The largest example that can be solved has 4005 latches and the smallest example that cannot be solved has 670 latches.

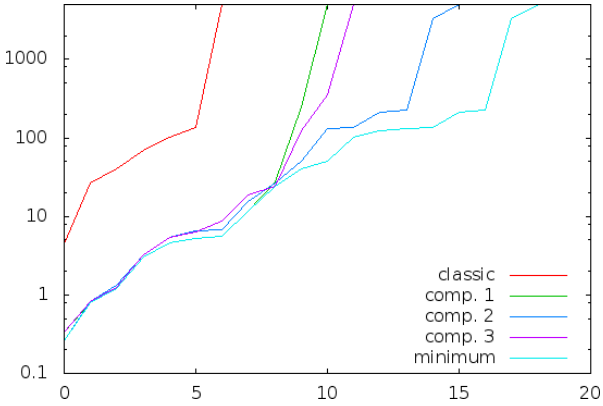


Figure 6: Performances for 46 generalized buffer benchmarks translated from LTL. The classical algorithm solves 6 benchmarks, comp.1 10, comp.2 15, comp.3 11. In total there are 18 benchmarks that can be solved. The largest example that can be solved has 22662 latches and the smallest example that cannot be solved has 590 latches.

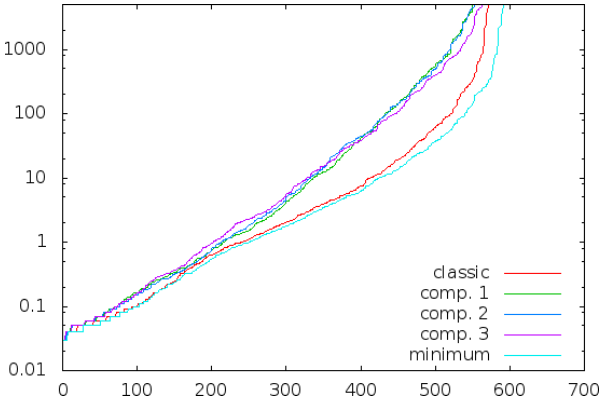


Figure 7: Performances for the 674 benchmarks. The classical algorithm was able to solve 572 benchmarks. 20 more benchmarks were solved by one of the three compositional algorithms.

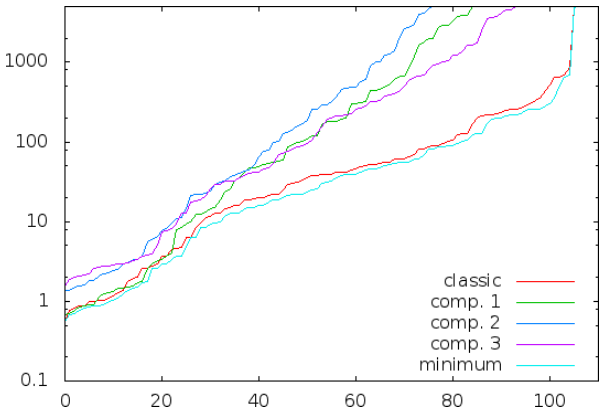


Figure 8: Performances for 108 AMBA benchmarks. The classical algorithm was able to solve 106 benchmarks, comp.1 84, comp.2 76, comp.3 93.

References

[1] Henrik Reif Andersen (1997): *An introduction to binary decision diagrams*. Technical Report, Course Notes on the WWW.

[2] Krzysztof R. Apt & Erich Grädel (2011): *Lectures in game theory for computer scientists*. Cambridge University Press.

[3] Romain Brenguier, Guillermo A. Pérez, Jean-François Raskin & Ocan Sankur (2014): *AbsSynthe: abstract synthesis from succinct safety specifications*. In: *Proceedings 3rd Workshop on Synthesis, SYNT 2014, Vienna, Austria, July 23-24, 2014.*, pp. 100–116, doi:10.4204/EPTCS.157.11. Available at `http://dx.doi.org/10.4204/EPTCS.157.11`.

[4] Randal E. Bryant (1986): *Graph-based algorithms for boolean function manipulation*. *Computers, IEEE Transactions on* 100(8), pp. 677–691, doi:10.1109/TC.1986.1676819.

[5] Jerry Burch, Edmund M Clarke & David Long (1991): *Symbolic model checking with partitioned transition relations*. *Computer Science Department*, p. 435.

[6] Edmund M. Clarke, Orna Grumberg & Doron Peled (2001): *Model checking*. MIT Press. Available at `http://books.google.de/books?id=Nmc4wEaLXFEC`.

[7] E. Allen Emerson & Charanjit S. Jutla (1991): *Tree automata, mu-calculus and determinacy*. In: *FOCS*, IEEE, pp. 368–377, doi:10.1109/SFCS.1991.185392.

[8] Emmanuel Filiot, Naiyong Jin & Jean-François Raskin (2010): *Compositional Algorithms for LTL Synthesis*. In: *Automated Technology for Verification and Analysis - 8th International Symposium, ATVA 2010, Singapore, September 21-24, 2010. Proceedings*, *Lecture Notes in Computer Science* 6252, Springer, pp. 112–127.

[9] Emmanuel Filiot, Naiyong Jin & Jean-François Raskin (2010): *Compositional Algorithms for LTL Synthesis*. In: *Automated Technology for Verification and Analysis - 8th International Symposium, ATVA 2010, Singapore, September 21-24, 2010. Proceedings*, pp. 112–127, doi:10.1007/978-3-642-15643-4_10.

[10] Emmanuel Filiot, Naiyong Jin & Jean-François Raskin (2011): *Antichains and compositional algorithms for LTL synthesis*. *Formal Methods in System Design* 39(3), pp. 261–296, doi:10.1007/s10703-011-0115-3. Available at `http://dx.doi.org/10.1007/s10703-011-0115-3`.

[11] Guillermo A. Pérez: *LTL2AIG*. `https://github.com/gaperez64/acacia_ltl2aig`.

[12] Nir Piterman, Amir Pnueli & Yaniv Sa'ar (2006): *Synthesis of reactive (1) designs*. In: *Verification, Model Checking, and Abstract Interpretation*, Springer, pp. 364–380.

[13] Thomas R. Shiple, Ramin Hojati, Alberto L. Sangiovanni-Vincentelli & Robert K. Brayton (1994): *Heuristic minimization of BDDs using don't cares*. In: *Proceedings of the 31st annual Design Automation Conference*, ACM, pp. 225–231.

[14] Fabio Somenzi (1999): *Binary Decision Diagrams*. In: *Calculational system design*, 173, IOS Press, p. 303.

[15] Alfred Tarski et al. (1955): *A lattice-theoretical fixpoint theorem and its applications*. *Pacific journal of Mathematics* 5(2), pp. 285–309, doi:10.2140/pjm.1955.5.285.

[16] H.J. Touti, H. Savoj, B. Lin, R.K. Brayton & A. Sangiovanni-Vincentelli (1990): *Implicit enumeration of finite state machines using bdd's*. In: *IEEE Int. Conference on CAD*.

[17] Chao Wang, Gary D Hachtel & Fabio Somenzi (2003): *The compositional far side of image computation*. In: *Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*, IEEE Computer Society, p. 334.

A Repeating the experiments

The new version of AbsSynthe with the implementation of the compositional algorithms can be fetched from `https://github.com/gaperez64/abssynthe`.

B Correctness of the decomposition algorithm

We now give a more formal explanation on how our decomposition procedure works. For two nodes n, n' of an AIG, we will write $n \equiv n'$ if they define the same Boolean function. We will also apply

Boolean operators on nodes (*e.g.* $n \vee n'$) which means that they are applied on the corresponding Boolean functions.

In an AIG each f_l is given by a (possibly inverted) edge from a node n_l. We first explore the graph until finding a negation. This is exactly what Algorithm 1 achieves. Formally, given a node n or an edge e, we define set of nodes $N(n)$, $N(e)$, $M(n)$ and $M(e)$ recursively as follows:

- if n is a terminal node then $N(n) = \{n\}$ and $M(n) = \varnothing$;
- if e is a regular edge from a node n then $N(e) = N(n)$ and $M(e) = M(n)$;
- if e is an inverted edge (a NOT gate) from a node n then $N(e) = \varnothing$ and $M(e) = \{n\}$;
- otherwise n is a two-input node (an AND gate) with two incident edges e_1 and e_2, then $N(n) = N(e_1) \cup N(e_2)$ and $M(n) = M(e_1) \cup M(e_2)$.

Lemma 3. *For every node l, $f_l \equiv \bigwedge_{n \in N(l)} n \wedge \bigwedge_{m \in M(l)} \neg m$.*

Proof. This proof is by induction. If n is a terminal node then the property is obvious. If e is a regular edge, the property follows by induction. If e is a negated edge to n, then its semantic is the negation of n: $f_e \equiv \neg f_n$ and the property follows. Now if n is an AND gate of two edges e_1 and e_2, its semantic is the conjunction: $f_n \equiv f_{e_1} \wedge f_{e_2}$. By induction, we have that $f_{e_i} \equiv \bigwedge_{n \in N(e_i)} n \wedge \bigwedge_{m \in M(e_i)} \neg m$ and therefore $f_n \equiv \bigwedge_{n \in N(e_1) \cup N(e_2)} n \wedge \bigwedge_{m \in M(e_1) \cup M(e_2)} \neg m$ and the property follows. □

If $M(\mathsf{err})$ is empty, or all its nodes are terminal (*i.e.* inputs or latches), there is no decomposition. Otherwise, we can choose an AND gate $m_0 \in M(\mathsf{err})$. We then define $\texttt{decompose}(f_l)$ as $\{e_p \mid p \in N(m_0)\} \cup \{e_{\neg p} \mid p \in M(m_0)\}$ where $e_p = \neg p \wedge \bigwedge_{n \in N(\mathsf{err})} n \wedge \bigwedge_{m \in M(\mathsf{err}) \setminus \{m_0\}} \neg m$. This is the decomposition given by Algorithm 2.

Lemma 4. *For every node l, $f_l \equiv \bigvee_{e_i \in \texttt{decompose}(f_l)} e_i$.*

Proof. Thanks to Lem. 3, $f_l \equiv \bigwedge_{n \in N(l)} n \wedge \bigwedge_{m \in M(l) \setminus \{m_0\}} \neg m \wedge \neg m_0$ and $\neg f_{m_0} \equiv \bigvee_{p \in N(m_0)} \neg p \vee \bigvee_{p' \in M(m_0)} p'$. We then rewrite f_l by distributing this disjunction and obtain $f_l \equiv \bigvee_{p \in N(m_0)} \left(\neg p \wedge \bigwedge_{n \in N} n \wedge \bigwedge_{m \in M \setminus \{m_0\}} \neg m\right) \vee \bigvee_{p' \in M(m_0)} \left(p' \wedge \bigwedge_{n \in N} n \wedge \bigwedge_{m \in M \setminus \{m_0\}} \neg m\right)$. This yields the required decomposition. □

Specification Format for Reactive Synthesis Problems*

Ayrat Khalimov
Graz University of Technology, Austria

Automatic synthesis from a given specification automatically constructs correct implementation. This frees the user from the mundane implementation work, but still requires the specification. But is specifying easier than implementing? In this paper, we propose a user-friendly format to ease the specification work, in particularly, that of specifying partial implementations. Also, we provide scripts to convert specifications in the new format into the SYNTCOMP format, thus benefiting from state of the art synthesizers.

1 Introduction

Specifying reactive synthesis tasks is not easy. First, writing non-trivial specifications in e.g. linear temporal logic (LTL) requires experience, and even an experienced user of LTL may notice that some properties are easier to implement oneself than to specify. Thus, it is desirable to be able to mix imperative and declarative paradigms when specifying reactive synthesis tasks, which makes a call for a new convenient specification format.

The full set of features of the new specification format might include:

1. *Modularity.* A synthesis task may require to synthesize several communicating modules where each module has its own properties. Thus, the new format should allow for specifying module interfaces and connections between them. These interfaces specify the amount of information each module knows about others.
2. *Imperative and declarative.* Some modules may already be given to the user, and some modules or parts of it may be easier to implement than to specify. Thus, the new format should allow for specifying module implementations.
3. *Conversion to the SYNTCOMP format.* The SYNTCOMP format [9] was recently proposed as the common ground format for reactive synthesis competitions, and at least four synthesizers were competing in 2014. Thus, to let the user to benefit from state of the art synthesizers, the new format should be convertible into the SYNTCOMP format.
4. *Property language agnostic.* The new format should allow the user to choose the best suited language for writing properties: linear temporal logic, linear dynamic logic [14], regular expressions, automata, etc.

These features requirements are our subjective suggestions and arise from the domain of synthesis of reactive systems that usually represent some hardware. The features certainly depend on the synthesis domain: for example, in the case of fault-tolerant algorithms the user also needs to specify the ratio of faulty to normal processes, the type of faults, etc.

In this the paper we:

- propose a specification format for reactive synthesis tasks, and

*This work was supported by the Austrian Science Fund via project RiSE (S11406).

To appear in EPTCS.

- provide scripts to convert from the new format into the SYNTCOMP format.

The new format can be extended to support features (1), (2), (3), and (4), but the current version has limitations. Some of the limitations are: (i) the user can separate the system into modules, but each module has the full information about others, (ii) only deterministic Büchi automata are allowed for specifying properties, and (iii) assumptions must be safety properties.

The new format is based on the SMV format [7] – it is convenient for describing hardware systems: it allows the user to define finite state machines that operate on variables of enumeration and range types, and to separate the system into modules, etc. Another advantage of using the SMV format as the starting point is that there is a solid support of the SMV format in the AIGER distribution [1], which greatly simplifies the task of the development of the conversion scripts.

Outline. We describe the new format and its restrictions in Section 2. Section 3 describes the conversion scripts and also introduces the SYNTCOMP format extended with liveness which is one of the supported target formats (alongside the standard SYNTCOMP format). Section 4 illustrates the use of the format and of the scripts – we write the specification that describes the task: when given an implementation of a Huffman decoder for the English alphabet, synthesize an encoder for it. Section 5 points to other possible ways of writing specifications and converting them into the SYNTCOMP format. And we conclude in Section 6.

2 Specification Format

We assume that the reader is familiar with the SMV (cf. [7]) and the SYNTCOMP [9] formats. We introduce a new section into the SMV format, and the comments of special form that allow for specifying synthesis problems. The specification in the extended SMV format is then translated into the SYNTCOMP format.

An example of the extended SMV format is shown in Listing 1. [1]

As in the usual SMV format, it consists of modules and the main module. In the main module, variables to be controlled by the system are marked with the comment '`--controllable`' (Mealy-type). The new sections ENV_AUTOMATON_SPEC and SYS_AUTOMATON_SPEC contain definitions of the assumptions and guarantees respectively. Every assumption and guarantee in the corresponding sections is expressed by a file path to a Büchi automaton in the GOAL format [13]. A file path can be preceded by '!' to indicate that the property is the negation of the automaton. These property automata will be converted into SMV modules.

Restrictions

The framework we describe in Section 3 converts a given specification in the extended SMV format into a deterministic game in the AIGER circuit format. AIGER circuits are inherently deterministic and so should be automata used in sections SYS_AUTOMATON_SPEC and ENV_AUTOMATON_SPEC. We require that:

- guarantees automata are deterministic (or determinizable),
- assumptions automata represent safety properties.

These conditions are sufficient (but not necessary) for the game to be deterministic, and are required by the conversion script spec_2_aag.py described in Section 3.

[1] The format is under active development and may slightly differ from the one described here.

Listing 1: Format structure (special elements are in blue color).

```
MODULE helper1(input1,input2)  //we can define and use SMV modules as usually
VAR
  state: 0..100;
DEFINE
  reached42 := state=42;
  ...

MODULE main  // module 'main' contains a specification
VAR
  CPUread: boolean;   // only boolean is allowed

VAR --controllable
  valueOut: boolean;  // only boolean is allowed

VAR
  h: helper1(readA, valueOut);  // we can instantiate modules as usually

DEFINE
  //signals defined in the module can be referred to in the property automata
  a := TRUE;
  b := FALSE;

  writtenA := CPUwrite & valueIn=a & done;
  readA := CPUread & valueOut=a & done;
  is42 := h.reached42;
  ...
  // thus we can use variables 'is42', 'readA', 'writtenA' in property automata below

SYS_AUTOMATON_SPEC // guarantees in the GOAL automata format
  guarantee1.gff;
  !guarantee2.gff; // '!' signals to negate the automaton

ENV_AUTOMATON_SPEC // assumptions in the GOAL automata format
  assumption1.gff;
  !assumption2.gff;
  ...
```

3 Conversion into the SYNTCOMP Format

We will convert specifications in the extended SMV format into standard and extended SYNTCOMP formats. Specifications in the standard SYNTCOMP can be given to any synthesis tool from the SYNTCOMP competition. Specification in the extended format can either be converted into the standard SYNTCOMP format using justice_2_safety.py, or can be given to our synthesizer aisy.py that supports it.

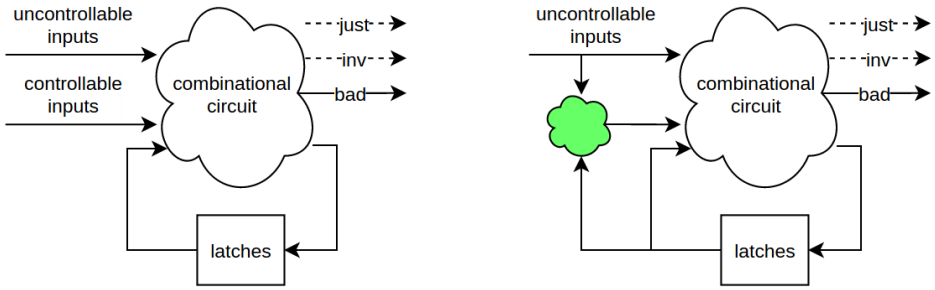
The scripts are available at `https://bitbucket.org/art_haali/spec-framework`.

Standard and extended SYNTCOMP

In this section we remind what the standard SYNTCOMP format is and then introduce the extension.

The standard SYNTCOMP is a circuit in the old AIGER format [2] with special comments that allow for specifying controllable (by the system) and uncontrollable (thus controllable by the environment) signals. The following figure shows the standard SYNTCOMP format [9] (ignore the dotted arrows – they are part of the extended format):

The goal is to synthesize the controllable signals (i.e., replace them with combinational circuits that as inputs use the memory and uncontrollable signals) such that the output *bad* never raises. Thus, the



(a) SYNTCOMP specification (b) SYNTCOMP model

semantics of the standard SYNTCOMP is $\mathsf{G}\neg bad$, which allows for specifying safety properties.

The natural extension is to allow liveness properties. This is what the extended SYNTCOMP format proposes. It also uses signal *inv* though it does not add the expressiveness. These signals are 'introduced' using the standard capabilities of the new AIGER format [4] (which allows for specifying 'bad' signals, 'invariant' signals, and 'justice' signals). The extended SYNTCOMP is shown on the same figure as the standard one if you take into account the dotted signals.

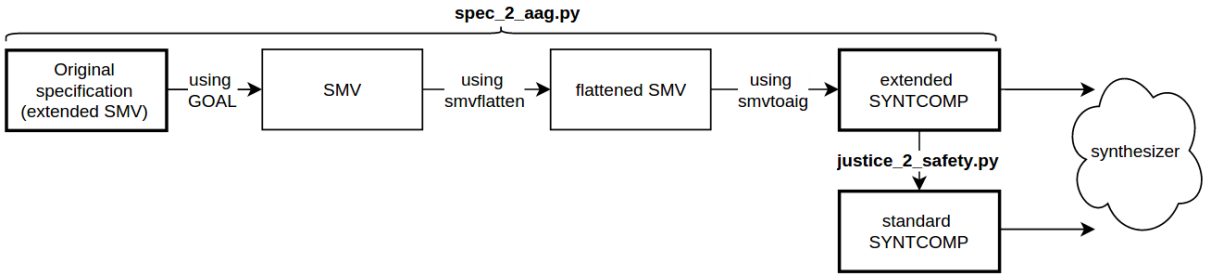
The semantics of the extended SYNTCOMP format is

$$(\neg bad\,\mathsf{W}\,\neg inv) \wedge (\mathsf{G}\,inv \rightarrow \mathsf{GF}\,just) \tag{1}$$

Note: the meaning of the signal *just* is reversed compared to the new AIGER format [4]: in that case a witness liveness trace satisfies $\mathsf{G}\,inv \wedge \mathsf{GF}\,just$, while in our case it satisfies $\mathsf{G}\,inv \wedge \neg\,\mathsf{GF}\,just$. We reversed the meaning of the signal *just* to be able to specify properties like $\mathsf{G}(r \rightarrow \mathsf{F}\,g)$ ("every request is granted") or $\mathsf{GF}\neg r$ ("request is lowered infinitely often"). Such properties can be represented by deterministic Büchi automata but not by deterministic co-Büchi automata. And we need specification automata to be deterministic to be able to convert them into inherently deterministic AIGER circuits.

Converting specifications into SYNTCOMP

The figure shows how we convert a given specification into SYNTCOMP format:



The main script is spec_2_aag.py:

1. Given a specification in extended SMV format (Section 2), we first convert all the automata in the GOAL format into SMV modules. At this step we might need to complement or determinize a given automaton – this is done using GOAL. Then we parse the result and convert it into an SMV

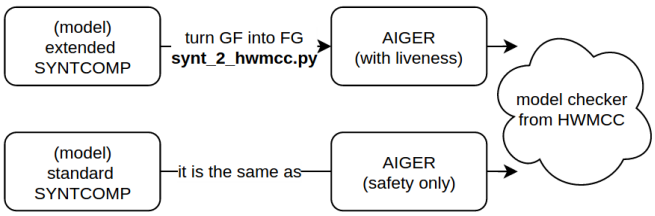
module. Such SMV module contains two special signals: *bad* and $fair$. In such SMV module, signal $fair$ is risen when we visit an accepting state of the automaton, and *bad* is risen when we visit a non-accepting state with a self-loop labelled $True$.

2. The main conversion work – from the SMV format into the extended SYNTCOMP format – is done with scripts smvflatten and svmtoaig from the AIGER distribution [1]. The result of this step is an AIGER file that may contain invariant and justice signals, which is not supported by the current SYNTCOMP format. Thus the current synthesis tools from the competition cannot be used directly.
3. The file in the extended SYNTCOMP format is converted into the standard version (with the single output) using justice_2_safety.py. The conversion requires input positive integer k and is standard: $\mathsf{GF}\ just$ is replaced with $\mathsf{G}(just \vee \mathsf{X}\ just \vee \ldots \mathsf{X}^k\ just)$, where X^k means k repetitions of X.

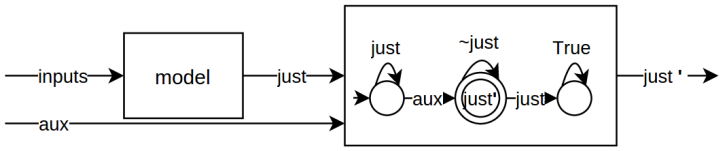
The result of this conversion is specification in either the standard or extended SYNTCOMP formats, and can be given to a synthesizer.

Converting models into AIGER

After the synthesizer produces a model, it can be turned into a benchmark in the standard AIGER format and then be fed to a model checker (e.g., one from the HWMCC competition):



If the input synthesis specification is in the standard SYNTCOMP format, then the model is also in the standard AIGER format and can be fed to a model checker directly. But in the case of the extended SYNTCOMP format we need to translate. Recall the semantics of our extended format (Equation 1): in our case a trace violating a liveness property would satisfy $\neg\,\mathsf{GF}\ just$, while the AIGER format has $\mathsf{GF}\ just'$. Thus, we convert the model into a model with signal $just'$ such that: if there is a trace that satisfies $\mathsf{GF}\ just'$ then it satisfies $\mathsf{FG}\neg just$. If denote the new model by M', and the original one by M, then: $M' \models \mathsf{EFG}\ just' \rightarrow M \not\models \mathsf{AGF}\ just$. The script synt_2_hwmcc.py does this by introducing a new input *aux* and attaching the automaton as shown below:



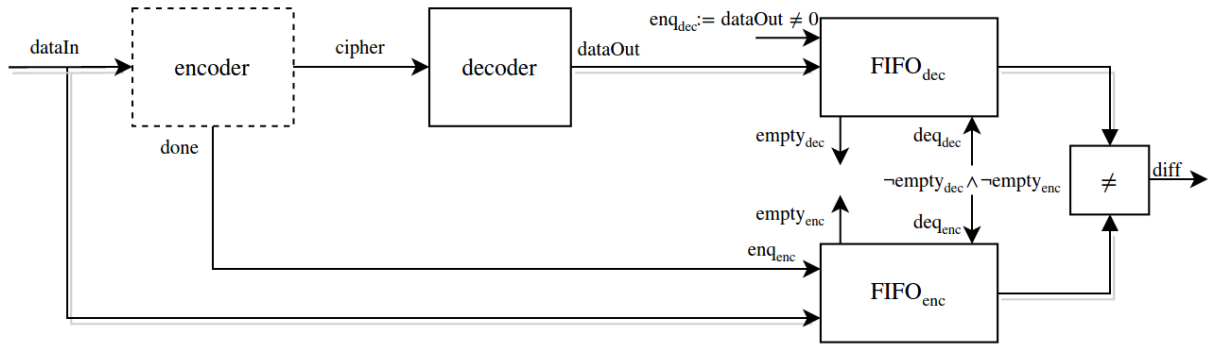
4 Example: Synthesizing a Huffman Encoder

This section demonstrates the use of the format and the framework. We implemented a simple synthesizer that solves Büchi games with invariants and safety objectives given in the extended SYNTCOMP format

described in Section 2. The results of the synthesis are then translated into the HWMCC format using script synt_2_hwmcc.py, and then model checked with IIMC [5].

We use the Huffman coding [8] to encode 26 English letters $A...Z$ and the space symbol into bit words of variable length (27 symbols in total). Let us assume that a Huffman decoder that decodes a stream of bits into letters is given [2] — the goal is to synthesize an encoder that works with the decoder.

The figure below shows the structure of the SMV specification of the synthesis task:



The dotted module (encoder) is to be synthesized, namely signals *cipher* and *done* (these signals are marked 'controllable' in the specification). The input is *dataIn* and has five bits of width, which is enough to encode 27 symbol: we use numbers 1..27 for encoding the symbols. The outputs of the encoder are boolean signals *cipher* and *done*; the intended meaning of *done* is "the last bit of the cipher is being sent now". The signal *cipher* is read by the decoder, which decodes the cipher and outputs it over *dataOut*; on successful decoding *dataOut* lasts for one tick, after which it is 0 again. The data-signal *dataOut* is then fed to the FIFO module $FIFO_{dec}$, and $FIFO_{enc}$ takes as input *dataIn*. FIFOs values are dequeued whenever they are not empty, and their values are compared. $FIFO_{enc}$ is enqueued whenever *done* is high, and $FIFO_{dec}$ – whenever *dataOut* encodes a letter. A FIFO gets blocked if we enqueue and not dequeue, and the FIFO is not empty currently (i.e., if $enq \wedge \neg deq \wedge empty$ holds).

All modules except dotted module *encoder* are given: FIFOs we coded manually (of size 1); the decoder is taken from the distribution of the model checker VIS [6].

In words, the specification is:

A1. assumption: "input *dataIn* is within range 1..27"

A2. assumption: "*dataIn* does not change until and including the moment when *done* is high"

G1. $\mathsf{G}(done \rightarrow \mathsf{X}\, enq_{dec})$ [3]

G2. $\mathsf{G}\neg diff$, i.e., if FIFOs are not empty, then they contain the same data

G3. liveness guarantee: $\mathsf{GF}\, done$

The specification in the SMV format is translated into the SYNTCOMP formats (standard safety and extended liveness) as described in Section 2. The semantics is as given in Equation 1 where: *bad* is the violation of any of the safety guarantees, *inv* is the truth of (A1) and (A2) so far, and $just = done$.

[2] Thus the decoder already has the letter frequencies built in.

[3] Strictly speaking this guarantee is not needed for the correct synthesis of the encoder, but without it the meaning of *done* may be different from the intended one ("the last cipher bit is being transferred").

Given the specification in the extended SYNTCOMP format, the synthesizer aisy.py synthesized the model in ≈ 2 minutes; the model has ≈ 130k new AND-gates [4]. The cipher synthesized is as expected (coincides with that of the Huffman decoder).

If we translate the specification into the k-safety variant with $k = 10$ (the minimal realizable), then aisy.py needs ≈ 4 minutes for the synthesis and the model has ≈ 120k AND-gates. We do not claim that in terms of efficiency the liveness specifications are superior to their safety variant – for this a more thorough research is needed. But the translation of liveness into safety requires a value of k as input: here we provided it manually, while in the general case its upper bound should be restricted and the permitted values should be iterated in some way.

Some final notes on the example. Initially, FIFOs implementations were non blocking, which permits the synthesizer to produce a cipher for a letter that is prefixed with ciphers of other letters (this version of the specification would compare only the last decoded letter). Also, with non-blocking FIFOs and without guarantee G2, the synthesizer produced a cipher that utilized the overflow in the state variable of the decoder. Hence in the general case the synthesized cipher may depend on in the implementation of the decoder and will not work with other implementations.

The benchmarks are available as a part of the conversion scripts distribution; aisy.py is available at `https://bitbucket.org/art_haali/aisy`.

5 Related Work

There are scripts and ways to create specification circuits in the SYNTCOMP format:

The script ltl2aig [10] takes as input specification in LTL format and signals partition and converts it into a circuit in the standard SYNTCOMP format. It does not use tools from the AIGER distribution [1] and supports all the routines natively. It also converts liveness properties into safety variants in the standard way. The limitation is that it does not allow the user to provide partial implementations.

The bundle ltl2smv[7] - smvflatten - smvtoaig [3] can translate SMV files with LTL properties embedded into AIGER format. The idea is:

1. smvflatten accepts a given SMV file with modules and variable types like range and enums, and translates it into boolean SMV file, preserving the original LTLSPEC section.
2. The result is sent to smvtoaig that translates LTLSPEC section into SMV module using ltl2smv, then joins the result, and translates it into AIGER circuit.

I.e, it does what we want but in the context of the model checking. For synthesis we also need:

- to provide the signals partition (into controllable and uncontrollable) – a minor issue, and
- to ensure there are no non-deterministic automata and thus no non-deterministic SMV modules produced at step (2) by ltl2smv. [5] One way to achieve this is to provide a custom implementation of ltl2smv. In hindsight, I think this might be a good way to go.

Finally, in the work in progress paper [12] the authors target a similar goal of providing a rich specification language that benefits from efficient synthesizers. In that work the authors automatically translate often used LTL patterns into the GR(1) fragment of LTL that has an efficient synthesis algorithm [11]. They do not allow for providing partial implementations.

[4] Recall that we synthesize a memory-less strategy, thus introduce only new AND-gates and no additional memory.

[5] This is because we cannot resolve non-determinism by adding the uncontrollable input: the synthesizer is aware of all circuit's signals, thus it may wait for the input to raise and then behave accordingly. I.e., we need to ensure that a system strategy is independent of the auxiliary signal – the partial information, which is not supported by the SYNTCOMP format.

6 Conclusions

In this paper we proposed a format to ease the specification task that allows the user to provide partial implementations, and we built the conversion scripts from the new format into the SYNTCOMP format. Both the specification format and the way we convert into the existing format are subject to discussion:

- Is there a more convenient format of specifications? Is SMV enough or Verilog should be used instead? Should we support GR(1)? Partial information?
- Is there a simpler way to convert from the new format into the SYNTCOMP format?

Acknowledgements. This paper would not be possible without numerous fruitful discussions with Robert Könighofer, Roderick Bloem, and Georg Hofferek. Many thanks to reviewers for valuable suggestions.

References

[1] Armin Biere: *AIGER format and toolbox*. Available at `http://fmv.jku.at/aiger/`.

[2] Armin Biere: *AIGER format version 20070427*. Available at `http://fmv.jku.at/aiger/FORMAT-20070427.pdf`.

[3] Armin Biere: smvflatten. Available at `http://fmv.jku.at/smvflatten/`.

[4] Armin Biere, Keijo Heljanko & Siert Wieringa (2011): *AIGER 1.9 and Beyond*. Available at `http://www.fmv.jku.at/hwmcc11/beyond1.pdf`.

[5] Aaron Bradley, Arlen Cox, Michael Dooley, Zyad Hassan, Fabio Somenzi & Yan Zhang: *IIMC*. Available at `http://ecee.colorado.edu/wpmu/iimc/`.

[6] RobertK. Brayton, GaryD. Hachtel, Alberto Sangiovanni-Vincentelli, Fabio Somenzi, Adnan Aziz, Szu-Tsung Cheng, Stephen Edwards, Sunil Khatri, Yuji Kukimoto, Abelardo Pardo, Shaz Qadeer, RajeevK. Ranjan, Shaker Sarwary, ThomasR. Staple, Gitanjali Swamy & Tiziano Villa (1996): *VIS: A system for verification and synthesis*. In: *CAV*, *LNCS* 1102, pp. 428–432, doi:10.1007/3-540-61474-5_95.

[7] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani & Armando Tacchella (2002): *NuSMV 2: An OpenSource Tool for Symbolic Model Checking*. In: *CAV*, *LNCS* 2404, pp. 359–364, doi:10.1007/3-540-45657-0_29.

[8] D.A. Huffman (1952): *A Method for the Construction of Minimum-Redundancy Codes*. *Proceedings of the IRE* 40(9), pp. 1098–1101, doi:10.1109/JRPROC.1952.273898.

[9] Swen Jacobs (2014): *Extended AIGER Format for Synthesis (v0.1)*.

[10] Guillermo A. Perez: ltl2aig. Available at `https://github.com/gaperez64/acacia_ltl2aig`.

[11] Nir Piterman, Amir Pnueli & Yaniv Saar (2006): *Synthesis of reactive (1) designs*. In: *Verification, Model Checking, and Abstract Interpretation*, Springer, pp. 364–380.

[12] Jan Oliver Ringert (2015): *Extensible Support for Specification Patterns in GR(1) Synthesis (Work in Progress)*. Young Researchers' Conference "Frontiers of Formal Methods". Available at `http://ffm2015.rwth-aachen.de/proceedings.php`.

[13] Yih-Kuen Tsay, Yu-Fang Chen, Ming-Hsien Tsai, Kang-Nien Wu & Wen-Chin Chan (2007): *GOAL: A graphical tool for manipulating Büchi automata and temporal formulae*. In: *TACAS*, Springer, pp. 466–471.

[14] M. Y. Vardi (2011): *The rise and fall of linear time logic*. 2nd Intl Symp. on Games, Automata, Logics and Formal Verification.

The complexity of approximations for epistemic synthesis

Xiaowei Huang
UNSW Australia

Ron van der Meyden
UNSW Australia

Epistemic protocol specifications allow programs, for settings in which multiple agents act with incomplete information, to be described in terms of how actions are related to what the agents know. They are a variant of the knowledge-based programs of Fagin et al [Distributed Computing, 1997], motivated by the complexity of synthesizing implementations in that framework. The paper proposes an approach to the synthesis of implementations of epistemic protocol specifications, that reduces the problem of finding an implementation to a sequence of model checking problems in approximations of the ultimate system being synthesized. A number of ways to construct such approximations is considered, and these are studied for the complexity of the associated model checking problems. The outcome of the study is the identification of the best approximations with the property of being PTIME implementable.

1 Introduction

Knowledge-based programs [9] are an abstract specification format for concurrent systems, in which the actions of an agent are conditional on formulas of the logic of knowledge [8]. This format allows the agent to be described in terms of what it must know in order to perform its actions, independently of how that knowledge is attained or concretely represented by the agent. This leads to implementations that are optimal in their use of the knowledge implicitly available in an agent's local state. The approach has been applied to problems including reliable message transmission [12], atomic commitment [11], fault-tolerant agreement [6], robot motion planning [4] and cache coherency [1].

The process of going from an abstract knowledge-based program to a concrete implementation is non-trivial, since it requires reasoning about all the ways that knowledge can be obtained, which can be quite subtle. Adding to the complexity, there is a circularity in that knowledge determines actions, which in turn affect the knowledge that an agent has. It is therefore highly desirable to be able to automate the process of implementation. Unfortunately, this is known to be an inherently complex problem: even deciding whether an implementation exists is intractable [9].

Sound local proposition epistemic specifications [7] are a generalization of knowledge-based programs proposed in part due to these complexity problems. These specifications require only *sufficient* conditions for knowledge, where knowledge-based programs require *necessary and sufficient conditions*. By allowing a larger space of potential implementations, this variant ensures that there always exists an implementation. However, some of these implementations are so trivial as to be uninteresting. In practice, one wants implementations in which agents make good use of their knowledge, so that the conditions under which they act closely approximate the necessary and sufficient conditions for knowledge. To date, a systematic approach to the identification of *good* implementations, and of automating the construction of such good implementations, has not been identified. This is the problem we address in the present paper. Ultimately, we seek an automated approach that is implementable in a way that scales to handling realistic examples. In this paper, we use a CTL basis for specifications, and use PTIME complexity of an associated model checking problem in an explicit state representation as a proxy for practical implementability.

To appear in EPTCS.

The contributions of the paper are two-fold: first, we present a general approach to the identification of good implementations, that extends the notion of sound local proposition epistemic specification by ordering the knowledge conditions to be synthesized, and then defining a way to construct implementations using a sequence of approximations to the final synthesized system, in which implementation choices for earlier knowledge conditions are fed back to improve the quality of approximation used to compute later implementation choices. This gives an intuitive approach to the construction of implementations, which we show by example to address some unintuitive aspects of the original knowledge-based program semantics. The approach is parametric in a choice of approximation scheme.

Second, we consider a range of possibilities for the approximation scheme to be used in the above ordered semantics, and evaluate the complexity of the synthesis computations associated with each approximation. The analysis leads to the identification of two orthogonal approximations that are optimal in their closeness to a knowledge-based program semantics, while remaining PTIME computable. This identifies the best prospects for future work on synthesis implementations.

The paper is structured as follows. Section 2 recalls basic definitions of temporal epistemic logic. Section 3 defines epistemic protocol specifications. In Section 4 we define the ordered semantics approximation approach for identification of good implementations. Section 5 defines a range of possible approximation schemes, which are then analyzed for complexity in Section 6. We discuss related work in Section 7 and conclude with a discussion of future work in Section 8.

2 A Semantic model for Knowledge and Time

In this section we lay out a general logical framework for agent knowledge, and describe how knowledge arises for agents that execute a concrete protocol in the context of some environment.

Let $Prop$ be a finite set of atomic propositions and Ags be a finite set of agents. The language CTL*K($Prop, Ags$) has the syntax:

$$\phi ::= \ p \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid X\phi \mid (\phi_1 U \phi_2) \mid A\phi \mid K_i\phi$$

where $p \in Prop$ and $i \in Ags$. This is CTL* plus the construct $K_i\phi$, which says that agent i knows that ϕ holds. We freely use standard operators that are definable in terms of the above, specifically $F\phi = \mathbf{true}U\phi$, $G\phi = \neg F\neg\phi$, $\phi_1 R\phi_2 = \neg((\neg\phi_1)U(\neg\phi_2))$, $E\phi = \neg A\neg\phi$. Our focus in this paper is on the fragment CTLK, in which the branching operators may occur only as $A\phi$ and $E\phi$, where ϕ is a formula in which the outermost operator is one of the temporal operators X, U, R, F or G. A further subfragment of this language CTLK$^+$, specified by the grammar

$$\phi ::= \ p \mid \neg p \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid AX\phi \mid AF\phi \mid AG\phi \mid A(\phi_1 U \phi_2) \mid A(\phi_1 R \phi_2) \mid K_i\phi$$

where $p \in Prop$ and $i \in Ags$. Intuitively, this is the sublanguage in which all occurrences of the operators A and K_i are in positive position.

To give semantics to all these languages it suffices to give semantics to CTL*K($Prop, Ags$). We do this using a variant of interpreted systems [8]. Let S be a set, which we call the set of global states. A *run* over S is a function $r : \mathbf{N} \to S$. A *point* is a pair (r, m) where r is a run and $m \in \mathbf{N}$. Given a set $\mathcal{R}$ of runs, we define $Points(\mathcal{R})$ to be the set of all points of runs $r \in \mathcal{R}$. An *interpreted system* for n agents is a tuple $\mathcal{I} = (\mathcal{R}, \sim, \pi)$, where $\mathcal{R}$ is a set of runs over S, the component $\sim$ is a collection $\{\sim_i\}_{i \in Ags}$, where for each $i \in Ags$, $\sim_i$ is an equivalence relation on $Points(\mathcal{R})$ (called agent i's *indistinguishability relation*) and $\pi : S \to \mathcal{P}(Prop)$ is an interpretation function. We say that a run r' is *equivalent to a run r up to time $m \in \mathbf{N}$* if $r'(k) = r(k)$ for $0 \leq k \leq m$.

We can define a general semantics of CTL*K(V,Ags) by means of a relation $\mathcal{I},(r,m) \models \phi$, where $\mathcal{I}$ is an intepreted system, (r,m) is a point of $\mathcal{I}$ and ϕ is a formula. This relation is defined inductively as follows:

- $\mathcal{I},(r,m) \models p$ if $p \in \pi(r(m))$, for $p \in Prop$;
- $\mathcal{I},(r,m) \models \neg\phi$ if not $\mathcal{I},(r,m) \models \phi$;
- $\mathcal{I},(r,m) \models \phi_1 \vee \phi_2$ if $\mathcal{I},(r,m) \models \phi_1$ or $\mathcal{I},(r,m) \models \phi_2$;
- $\mathcal{I},(r,m) \models A\phi$ if $\mathcal{I},(r',m) \models \phi$ for all runs $r' \in \mathcal{R}$ equivalent to r up to time m;
- $\mathcal{I},(r,m) \models X\phi$ if $\mathcal{I},(r,m+1) \models \phi$;
- $\mathcal{I},(r,m) \models \phi_1 U \phi_2$ if there exists $m' \geq m$ such that $\mathcal{I},(r,m') \models \phi_2$, and $\mathcal{I},(r,k) \models \phi_1$ for $m \leq k < m'$;
- $\mathcal{I},(r,m) \models K_i\phi$ if $\mathcal{I},(r',m') \models \phi$ for all points $(r',m') \sim_i (r,m)$ of $\mathcal{I}$.

For the knowledge operators, this semantics is essentially the same as the usual interpreted systems semantics. For the temporal operators, it corresponds to a semantics for branching time known as the *bundle semantics* [5, 22]. We write $\mathcal{I} \models \phi$ when $\mathcal{I},(r,0) \models \phi$ for all runs r of $\mathcal{I}$.

We are interested in systems in which each of the agents runs a protocol in which it chooses its actions based on local information, in the context of a larger environment. An *environment* for agents Ags is a tuple $E = \langle S, I, \{Acts_i\}_{i\in Ags}, \longrightarrow, \{O_i\}_{i\in Ags}, \pi\rangle$, where

1. S is a finite set of states,
2. I is a subset of S, representing the initial states,
3. for each agent i, component $Acts_i$ is a finite set of actions that may be performed by agent i; we define $Acts = \Pi_{i\in Ags} Acts_i$ to be the corresponding set of *joint actions*
4. $\longrightarrow \subseteq S \times Acts \times S$ is a transition relation, labelled by joint actions,
5. for each $i \in Ags$, component O_i is a mapping from S to some set O of observations,
6. $\pi : S \to \mathcal{P}(Prop)$ is an interpretation of some set of atomic propositions $Prop$.

Intuitively, a joint action $\mathbf{a}$ represents a choice of action $\mathbf{a}_i$ for each agent, performed simultaneously, and the transition relation resolves this into an effect on the state. We assume that $\longrightarrow$ is serial in the sense that for all $s \in S$ and $\mathbf{a} \in Acts$ there exists $t \in S$ such that $s \xrightarrow{\mathbf{a}} t$. We assume that $Acts_i$ always contains at least an action **skip**, and that for the joint action $\mathbf{a}$ with $\mathbf{a}_i =$ **skip** for all agents i, we have $s \xrightarrow{\mathbf{a}} t$ iff $s = t$. The set O of observations is an arbitrary set: for each agent i, we will be interested in the equivalence relation $s \sim_i t$ if $O_i(s) = O_i(t)$ induced by the observation function O_i rather than the actual values of O_i.

A proposition p is *local to agent i* in the enviroment E if it depends only on the agent's observation, in the sense that for all states s,t with $O_i(s) = O_i(t)$, we have $p \in \pi(s)$ iff $p \in \pi(t)$. We write $Prop_i$ for the set of propositions local to agent i. Intuitively, these are the propositions whose values the agent can always determine, based just on its observation. We similarly say that a boolean formula is local to agent i if it contains only propositions that are local to agent i. We assume that the set of local propositions is *complete* with respect to the observations, in that for each observation o there exists a local formula ϕ such that for all states s, we have $O_i(s) = o$ iff $\pi(s) \models \phi$. (This can be ensured by including a proposition p_o that is true at just states s with $O_i(s) = o$, or by including a proposition $v = c$ for each possible value c of each variable v making up agent i's observation.)

A *concrete protocol* for agent $i \in Ags$ in such an environment E is a Dijkstra style nondeterministic looping statement P_i of the form

$$\textbf{do } \phi_1 \to a_1 \ [] \ \ldots \ [] \ \phi_k \to a_k \textbf{ od} \tag{1}$$

where the a_j are actions in $Acts_i$ and the ϕ_j are boolean formulas local to agent i. Intuitively, this is a nonterminating program that is executed by the agent repeatedly checking which of the guards ϕ_j holds, and then nondeterministically performing one of the corresponding actions a_i. If none of the guards holds, then the action **skip** is performed. That is, implicitly, there is an additional clause $\neg\phi_1 \wedge \ldots \neg\phi_n \rightarrow$ **skip**. Without loss of generality, we may assume that the a_i are distinct. (We can always amalgamate two cases $\phi_1 \rightarrow a$ and $\phi_2 \rightarrow a$ with the same action a into a single case $\phi_1 \vee \phi_2 \rightarrow a$.) We say that action a_j is enabled in protocol P_i at state s if ϕ_j holds with respect to the assignment $\pi(s)$, and write $en(P_i, s)$ for the set of all actions enabled in protocol P_i at state s.

A *joint protocol* P is a collection $\{P_i\}_{i\in Ags}$ of protocols for the individual agents. A joint action $a \in Acts$ is *enabled by* P *at a state* s if $a_i \in en(P_i, s)$ for all $i \in Ags$. We write $en(P, s)$ for the set of all joint actions enabled by P at state s.

Given an environment $E = \langle S, I, \{Acts\}_{i\in Ags}, \longrightarrow, \{O_i\}_{i\in Ags}, \pi\rangle$ and a joint protocol P for the agents in E, we may construct an interpreted system $\mathcal{I}(E,P) = (\mathcal{R}(E,P), \sim, \pi)$ over global states S as follows. The set of runs $\mathcal{R}(E,P)$ consists of all runs $r : \mathbf{N} \rightarrow S$ such that $r(0) \in I$ and for all $n \in \mathbf{N}$ there exists $\mathbf{a} \in en(P, r(n))$ such that $r(n) \xrightarrow{\mathbf{a}} r(n+1)$. The component $\sim = \{\sim_i\}_{i\in Ags}$ is defined by $(r,m) \sim_i (r',m')$ if $O_i(r(m)) = O_i(r'(m'))$, i.e., two points are indistinguishable to agent i if it makes the same observation at the corresponding global states; this is known in the literature as the *observational* semantics for knowledge. The interpretation π in the interpreted system $\mathcal{I}(E,P)$ is identical to that in the environment E.

Note that in $\mathcal{I} = \mathcal{I}(E,P)$, the satisfaction of formulas of the form $K_i\phi$ in fact depends only on the observation $O_i(r(m))$. We therefore may write $\mathcal{I}, o \models K_i\phi$ for an observation value o to mean $\mathcal{I}, (r,m) \models K_i\phi$ for all points (r,m) of $\mathcal{I}$ with $O_i(r(m)) = o$.

3 Epistemic Protocol Specifications

Protocol templates generalize concrete protocols by introducing some variables that may be instantiated with local boolean formulas in order to obtain a concrete protocol. Formally, a *protocol template* for agent $i \in Ags$ is an expression in the same form as (1), except that the ϕ_j are now boolean expressions, not just over the local atomic propositions $Prop_i$, but may also contain boolean variables from an additional set X of *template variables*. We write $Vars(Prot_i)$ for the set of these additional boolean variables that occur in some ϕ_i.

An *epistemic protocol specification* is a tuple $\mathcal{S} = \langle Ags, E, \{\mathtt{P}_i\}_{i\in Ags}, \Phi\rangle$, consisting of a set of agents Ags, an environment E for Ags, a collection of protocol templates $\{\mathtt{P}_i\}_{i\in Ags}$ for environment E, and a collection of epistemic logic formulas Φ over the agents Ags and atomic propositions $X \cup Prop$. In this paper, we assume $\Phi \subseteq \mathrm{CTLK}(Ags, X \cup Prop)$. We require that $Vars(\mathtt{P}_i)$ and $Vars(\mathtt{P}_j)$ are disjoint when $i \neq j$.

Intuitively, the protocol templates in such a specification lay out the abstract structure of some concrete protocols, and the variables in X are "holes" that need to be filled in order to obtain a concrete protocol. The formulas in Φ state constraints on how the holes may be filled: it is required that these formulas be valid in the model that results from filling the holes.

To implement an epistemic protocol specification with respect to the observational semantics, we need to replace each template variable v in each agent i's protocol template by an expression over the agent's local variables, in such a way that the specification formulas are satisfied in the model resulting from executing the resulting standard program. We now formalize this semantics.

Let θ be a substitution mapping each template variable $x \in Vars(\mathtt{P}_i)$, for $i \in Ags$, to a boolean formula local to agent i. We may apply such a substitution to a protocol template $\mathtt{P}_i$ in the form (1) by applying θ

to each of the formulas ϕ_j, yielding

$$\textbf{do}\ \ \phi_1\theta \rightarrow a_1\ []\ \ldots\ []\ \phi_k\theta \rightarrow a_k\ \ \textbf{od}$$

which we write as $\mathtt{P}_i\theta$. Since the $\phi_j\theta$ contain only propositions in $Prop_i$, this is a concrete protocol for agent i. Consequently, we obtain a joint concrete protocol $\mathtt{P}\theta = \{\mathtt{P}_i\theta\}_{i\in Ags}$, which may be executed in the environment E, generating the system $\mathcal{I}(E,\mathtt{P}\theta)$. The substitution θ may also be applied to the specification formulas in Φ. Each $\phi \in \Phi$ is a formula over variables $X \cup Prop$, so $\phi\theta$ is a formula over variables $Prop$. We write $\Phi\theta$ for $\{\phi\theta \mid \phi \in \Phi\}$. We say that such a substitution θ provides an *implementation* of the epistemic protocol specification $\mathcal{S}$, provided $\mathcal{I}(E,\{\mathtt{P}_i\theta\}_{i\in Ags}) \models \Phi\theta$. The problem we study in this paper is the following: given an environment E and an epistemic protocol specification $\mathcal{S}$, synthesize an implementation θ.

Knowledge-based programs [8, 9] are a special case of epistemic protocol specifications. Essentially, knowledge-based programs are epistemic protocol specifications in which the set Φ is a collection of formulas of the form $AG(x \Leftrightarrow K_i\psi)$, with exactly one such formula for each agent $i \in Ags$ and each template variable $x \in Vars(\mathtt{P}_i)$. That is, each template variable is associated with a formula of the form $K_i\psi$, expressing some property of agent i's knowledge, and we require that the meaning of the template variable be equivalent to this property. The following example, an extension of an example from [4], illustrates the motivations for knowledge-based programs that have been advocated in the literature.

Example 1 *Two robots, A and B, sit on linear track with discretized positions* $0\ldots10$*. Initially A is at position* 0 *and B is at position* 10*. Their objective is to meet at a position at least* 2*, without colliding. Each robot is equipped with noisy position sensor, that gives at each moment of time a natural number value in the interval* $[0,\ldots,10]$*. (We consider various different sensor models below, each defined by a relationship between the sensor reading and the actual position.) The robots do not have a sensor for detecting each other's position. Each robot has an action* **Halt** *and an action* **Move***. The* **Halt** *action brings the robot to a stop at its current location, and it will not move again after this action has been performed. The* **Move** *action moves the robot in the direction that it is facing (right, i.e., from 0 to 10 for A, and left for B). However, the effects of this action are unreliable: when performed, the robot either stays at its current position or moves one step in the designated direction.*

Because of the nondeterminism in the sensor readings and the robot motion, it is a non-trivial matter to program the robots to achieve their goal. In particular, the programmer needs to reason about how the sensor readings are related to the actual positions, in view of the assumptions about the possible robot motions. However, there is a natural abstract description of the solution to the problem at the level of agent knowledge, which we may capture as a knowledge-based program as follows: A has the epistemic protocol specification

$$\begin{array}{ll}\mathtt{P}_A = & \textbf{do}\\ & \quad \neg x \rightarrow \textbf{Move}\\ & \quad []\ x\ \rightarrow \textbf{Halt}\\ & \textbf{od}\end{array}$$

$$AG(x \Leftrightarrow K_A(position_A \geq 2))$$

and B has the epistemic protocol specification

$$\begin{array}{ll} \mathtt{P}_B = & \textbf{do} \\ & \quad y \rightarrow \textbf{Move} \\ & \quad [] \neg y \rightarrow \textbf{Halt} \\ & \textbf{od} \end{array}$$

$$AG(y \Leftrightarrow K_B(\bigwedge_{p\in[0,\ldots 10]} position_B = p \Rightarrow AG(position_A < p-1)))$$

Intuitively, the specification for A says that A should move to the right until it knows that its position is at least 2. The specification for B says that B should move to the left so long as its knows that, if its current position is p, then A's position will always be to the left of the position $p-1$ *that a move might cause B to enter. If this does not hold then there could be a collision.*

One of the benefits of knowledge-based programs is that they can be shown to guarantee correctness properties of solutions for a problem independently of the way that knowledge is acquired and represented. This gives a desirable level of abstraction that enables a single knowledge level description to be used to generate multiple implementations that are tailored to different environments.

In the case of the above knowledge-based program, we note that it guarantees several properties independently of the details of the sensor model. Informally, since A halts only when it knows that its position is at least 2, and $K_A p \Rightarrow p$ *is a tautology of the logic of knowledge, its program ensures that when A halts, its position will be at least 2. Similarly, since B moves at most one position in any step, and moves only when it knows that moving to the position to its left will not cause a collision with A, a move by B will not be the cause of a collision. It remains to show that A does not cause a collision with B — this requires assumptions about A's sensor. (Note that if A is blind it never halts, and could collide with B even if B never moves, so assumptions are needed.) For termination, moreover, we require fairness assumptions about the way that A and B move (e.g., an action* **Move** *performed infinitely often eventually causes the position to change.).*

What implementations exist for the knowledge-based program depend on the assumptions we make about the error in the sensor readings. We assume that for each agent i, and possible sensor value v, there are propositions $sensor_i = v$, $sensor_i \geq v$, *and* $sensor_i \leq v$ *in* $Prop_i$, *with the obvious meaning. Suppose that we take the robots' position sensor to be free of error, i.e. for each agent i, we always have* $sensor_i = position_i$. *Then agent i always knows its exact position from its sensor value. In this case, the knowledge-based program has an implementation with* $\theta(x)$ *is* $sensor_A = 2$ *and* $\theta(y)$ *is* $sensor_B \geq 4$. *In this implementation, A halts at position 2 and B halts at position 3 (assuming that they reach these positions.)*

On the other hand, suppose that the sensor readings may be erroneous, with a maximal error of 1, i.e., when the robot's position is p, the sensor value is in $\{p-1, p, p+1\}$. *In this case, there exists an implementation* θ *in which* $\theta(x)$ *is* $sensor_A = 3 \vee sensor_A = 4 \vee sensor_A = 5$, *and* $\theta(y)$ *is* $sensor_B = 4 \vee sensor_B = 5 \vee sensor_B = 6$. *In this implementation, A moves until it gets a sensor reading in the set* $\{3,4,5\}$, *and then halts. The effect is that A halts at a location in the set* $\{2,3,4\}$*; which one depends on the pattern of sensor readings obtained. For example, the sequence* $(0,0),(1,1),(2,2),(3,2),(4,3)$ *of* $(position, sensor)$ *values leaves A at position 4, whereas the sequence* $(0,0),(1,1),(2,3)$ *leaves A at position* 2. *The effect of the choice of* $\theta(y)$ *is that B moves to the left and halts in one of the positions* $\{5,6,7\}$. *One run in which B halts at position* 5 *has* $(position, sensor)$ *values* $(10,10),(9,9),(8,8),(7,7),(6,7),(5,4)$. *A run in which B halts at position* 7 *is where these values are* $(10,10),(9,9),(8,8),(7,6)$. *Note that here the sensor reading 6 tells B that it is in the interval* $[5,7]$, *so it could be at* 5. *It is therefore not safe to move, since A might be at* 4.

P_i = **do**
  $start \wedge x_i \rightarrow \mathbf{c}$
 [] $start \wedge \neg x_i \rightarrow \mathbf{w}$
 [] $\neg start \rightarrow \mathbf{p}$
od

$AG(x_i \Leftrightarrow K_i AXw)$

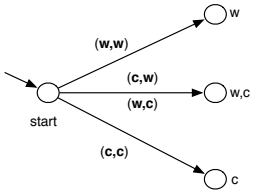


Figure 1: Knowledge-based program and environment

One of the advantages of the knowledge-based programs is that their implementations are optimal in the way that they use the information encoded in the agent's observations. For example, the program for A says that A should halt as soon *as it knows that it is in the goal region. In the case of the sensor with noise at most 1, the putative implementation for A given by* $\theta(x) = sensor_A \geq 4$ *would also ensure that A halts inside the goal region* [2, 10]*, but would not implement the knowledge-based program because there are situations (viz.* $sensor_A = 3$*), where A does not halt even though it knows that it is safe to halt.*

The semantics for knowledge-based programs results in implementations that are highly optimized in their use of information. Because knowledge for an implementation θ is computed in the system $\mathcal{I}(E, P\theta)$, agent's may reason with complete information about the implementation they are running in determining what information follows from their observations. This introduces a circularity that makes finding implementations of knowledge-based programs an inherently complex problem. Indeed, it also has the consequence that it is possible for a knowledge-based program to have no implementations. The following provides a simple example where this is the case. It also illustrates a somewhat counterintuitive aspect of knowledge-based programs, that we will argue is improved by our proposed ordered semantics for epistemic specifications below.

Example 2 *Alice and Bob have arranged to meet for a picnic. They are agreed that a picnic should have both wine and cheese, and each should bring one or the other. However, they did not think to coordinate in advance what each is bringing, and they are now not able to communicate, since Alice's phone is in the shop for repairs. They do know that each reasons as follows. Cheese being cheaper than wine, they prefer to bring cheese, and will do so if they know that there is already guaranteed to be wine. Otherwise, they will bring wine. This situation can be captured by the knowledge-based program (for each* $i \in \{A, B\}$*) and environment depicted in Figure 1. Here start is a proposition, local to both agents, that holds before the picnic (at time 0). We use* w, c *as propositions that hold if there is wine (respectively, cheese) in the picnic state (at time 1). Actions* $\mathbf{w}, \mathbf{c}, \mathbf{p}$ *represent bringing wine, bringing cheese, and picnicking, respectively. For any omitted joint actions* $\mathbf{a}$ *from a state s in the diagram, we assume an implicit self-loop* $s \xrightarrow{\mathbf{a}} s$*. We assume that for all states s and* $i \in \{A, B\}$*, we have* $O_i(s) = s$*, i.e., both agents have complete information about the current state.*

This epistemic specification has no implementations. Note that in any implementation, each agent i must choose either $\mathbf{w}$ *or* $\mathbf{c}$ *at the initial state. For each such selection, there is a unique successor state at time 1, so each implementation system* $\mathcal{I}(\mathrm{P}\theta, E)$ *has exactly one state at time 1. If this state satisfies w, then we have* $\mathcal{I}(\mathrm{P}\theta, E) \models K_i(AXw)$*, and this implies that both agents select action* $\mathbf{c}$ *at the start state. But then the state at time 1 does not satisfy w. Conversely, if the unique state at time* 1 *does not satisfy w, then* $\mathcal{I}(\mathrm{P}\theta, E) \models \neg K_i(AXw)$*, and this implies that both agents select action* $\mathbf{w}$ *at the start state, which produces a state at time 1 that satisfies w, also a contradiction. In either case, the assumption that we have an implementation results in a contradiction, so there are no implementations.* □

Testing whether there exists an implementation of a knowledge-based program when the temporal basis of the temporal epistemic logic used is the linear time logic LTL is PSPACE complete [9]. However, the primary source of the hardness here is that model checking LTL is already a PSPACE complete problem.

In the case of CTL as the temporal basis, where model checking can be done in PTIME, the problem of deciding the existence of an implementation of a given knowledge-based program in a given environment can be shown to be NP-complete. NP hardness follows from Theorem 5.4 in [9], which states that for *atemporal* knowledge-based programs, in which the knowledge formulas $K_i\phi$ used do not contain temporal operators, the complexity of determining the existence of an implementation is NP-complete. However, the construction in the proof in [9] requires both the environment and the knowledge-based program to vary. In practice, the size of the knowledge-based program is likely to be significantly smaller than the size of the environment, inasmuch as it is created by hand and effectively amounts to a form of specification. An alternate approach is to measure complexity as a function of the size of the environment for a fixed knowledge-based program. Even here, it turns out, the problem of deciding the existence of an implementation is NP-hard for very simple knowledge-based programs.

Theorem 1 *There exists a fixed atemporal knowledge-based program* P *for a single agent, such that the problem of deciding, given an environment E, whether* P *has an implementation in E, is NP-hard.*

The upper bound of NP for deciding the existence of implementations of knowledge-based programs is generalized by the following result for our more general notion of epistemic protocol specification.

Theorem 2 *Given an environment E and an epistemic protocol specification $\mathcal{S}$ expressed using CTLK, the complexity of determining the existence of an implementation for $\mathcal{S}$ in E is in NP.*

Theorem 2 assumes that the environment is presented by means of an explicit listing of its states and transitions. In practice, the inputs to the problem will be given in some format that makes their representation succinct, e.g., states will be represented as assignments to some set of variables, and boolean formulas will be used to represent the environment and protocol components. For this alternate input format, the problem of determining the existence of an implementation of a given epistemic protocol specification is NEXPTIME-complete [14].

Under either an implicit or explicit representation of environments, these results suggest that synthesis of implementations of general epistemic protocol specifications, and knowledge-based programs in particular, is unlikely to be practical. An implementation using symbolic techniques is presented in [14], but it works only on small examples and scales poorly (it requires the introduction of exponentially many fresh propositions before using BDD techniques; the number of propositions soon reaches the limit that can be handled efficiently by BDD packages.) In the following section, we consider a restricted class of specifications that weakens the notion of knowledge-based program in such a way that implementations can always be found, and focus on how to efficiently derive implementations that approximate the implementations of corresponding knowledge-based programs as closely as possible.

4 An Ordered Semantics

Sound local proposition epistemic protocol specifications are a generalization of knowledge-based programs, introduced in [7], with one of the motivations being that they provide a larger space of potential implementations, that may overcome the problem of the high complexity of finding an implementation. (There is the further motivation that the implementation of a knowledge-based program, when one exists,

itself may be intractable; e.g., it is shown in [19] that for perfect recall implementations of *atemporal* knowledge-based programs, deciding whether $K_i\phi$ holds at a given point of the implementation may be a PSPACE-complete problem. This specific motivation is less of concern for the observational case that we study in this paper.)

Formally, a *sound local proposition* epistemic protocol specification is one in which Φ is given by means of a function κ with domain $Vars(\mathrm{P})$, such that for each agent i and each template variable $x \in Vars(\mathrm{P}_i)$, the formula $\kappa(x)$ is of the form $K_i\psi$. The corresponding set of formulas for the epistemic protocol specification is $\Phi = \Phi_\kappa = \{AG(x \Rightarrow \kappa(x)) \mid x \in Vars(\mathrm{P})\}$.

As usual for epistemic protocol specifications, an implementation associates to each template variable a boolean formula local to the corresponding agent, such that the resulting system satisfies the specification Φ.[1] Thus, whereas a knowledge-based program requires that each knowledge formula in the program be implemented by a *necessary and sufficient* local formula, a sound local proposition specification requires only that the implementing local formula be *sufficient*.

It is argued in [7] that examples of knowledge-based programs can typically be weakened to sound local proposition specifications without loss of the desired *correctness* properties that hold of all implementations. However, implementations of knowledge-based programs may guarantee *optimality* properties that are not guaranteed by the corresponding sound local proposition specifications. For example, an implementation of a knowledge-based program that states "if $K_i\phi$ then do a" will be optimal in the sense that it ensures that the agent will do a *as soon as* it knows that ϕ holds. By contrast, an implementation that replaces $K_i\phi$ by a sufficient condition for this formula may perform a only much later, or even fail to do so, even if the knowledge necessary to do a is deducible from the agent's local state. (An example of such a situation is given in [1], which identifies a situation where a cache coherency protocol fails to act on knowledge that it has.)

Note that the substitution $\theta_\perp$, defined by $\theta_\perp(x) = \mathbf{false}$ for all template variables x, is *always* an implementation for a sound local proposition specification $\mathcal{S}$ in an environment E. It is therefore trivial to decide the existence of an implementation, and it is also trivial to produce a succinct representation of an implementation. Of course, an implementation of a program "if x then do a" that sets x to be **false** will never perform a, so this trivial implementation is generally not of much interest. What is more interesting is to find *good* implementations, that approximate the corresponding knowledge-based program implementations as closely as possible in order to behave as close to optimally as possible, while remaining tractable.

Consider the order on substitutions defined by $\theta \leq \theta'$ if for all variables x and states $s \in S$ of the environment we have $\pi(s) \models \theta(x) \Rightarrow \theta'(x)$. If both are implementations of $\mathcal{S}$ in E, we may find θ' preferable in that it provides weaker sufficient conditions (i.e., ones more often true) for the knowledge formulas $K_i\phi$ of interest. Pragmatically, if ϕ is a condition that an agent must know to be true before it can safely perform a certain action, the more often the sufficient condition $\theta(x)$ for $K_i\phi$ holds, the more often will the agent perform the action in the implementation. It is therefore reasonable to seek implementations that maximize θ with respect to the order $\leq$. The maximal sufficient condition for $K_i\phi$ is $K_i\phi$ itself, in the system $\mathcal{I}(E, P\theta)$ corresponding to an implementation θ, expressed as an equivalent local formula.[2]

The following result makes this statement precise:

[1] By the assumption of locality of $\theta(x)$, validity of $AG(\theta(x) \Rightarrow K_i\psi)$ in a system is equivalent to validity of $AG(\theta(x) \Rightarrow \psi)$, but we retain the epistemic form for emphasis and to maintain the connection to knowledge-based programs.

[2] The existence of such a formula follows from completeness of the set of local propositions. If we extend the propositions in an environment to include for each agent i and possible observation o of the agent, a proposition $p_{i,o}$ that holds at a state s iff $O_i(s) = o$, then the formula $\theta(x)$ such that $\mathcal{I} \models AG(\theta(x) \Leftrightarrow \kappa(x))$, where $\kappa(x) = K_i\phi$, can be constructed as $\bigvee\{p_{i,o} \mid o \in O_i(S),\ \mathcal{I}, o \models \kappa(x)\}$, and has size of order the number of observations.

Theorem 3 *Suppose that $\mathcal{S}$ is a sound local proposition epistemic protocol specification, and let $\mathcal{S}'$ be the knowledge-based program resulting from replacing each formula $AG(x \Rightarrow \kappa(x))$ in Φ by the formula $AG(x \Leftrightarrow \kappa(x))$. Then every implementation θ of $\mathcal{S}'$ is an implementation of $\mathcal{S}$.*

However, to have $\theta(x)$ equivalent to $K_i\phi$ in $\mathcal{I}(E,P\theta)$ would mean that θ implements a knowledge-based program. The complexity results of the previous section indicate that this is too strong a requirement, for practical purposes, since it is unlikely to be efficiently implementable. The compromise we explore in this paper is to require $\theta(x)$ to be equivalent to $K_i\phi$ not in the system $\mathcal{I}(E,P\theta)$ itself, but in another system that approximates $\mathcal{I}(E,P\theta)$. The basis for the correctness of this idea is the following lemma.

Lemma 1 *Suppose that $\mathcal{I} \subseteq \mathcal{I}'$, that r is a run of $\mathcal{I}$ and that ϕ is a formula in which knowledge operators and the branching operator A occur only in positive position. Then $\mathcal{I}', (r,m) \models \phi$ implies $\mathcal{I}, (r,m) \models \phi$.*

In particular, if, for a sound local proposition epistemic protocol specification $\mathcal{S}$, the formula $\kappa(x)$ associated to a template variable x is in CTLK$^+$, then this result applies to the formula $AG(x \Rightarrow \kappa(x))$ in Φ_κ, since this is also in CTLK$^+$. Suppose the system $\mathcal{I}'$ approximates the ultimate implementation $\mathcal{I}(E,P\theta)$ in the sense that $\mathcal{I}' \supseteq \mathcal{I}(E,P\theta)$. Let $\theta(x)$ be a local formula such that $\mathcal{I}' \models AG(\theta(x) \Leftrightarrow \kappa(x))$. Then also $\mathcal{I}' \models AG(\theta(x) \Rightarrow \kappa(x))$, hence, by Lemma 1, $\theta(x)$ will also satisfy the correctness condition $\mathcal{I}(E,P\theta) \models AG(\theta(x) \Rightarrow \kappa(x))$ necessary for θ to be an implementation of $\mathcal{S}$.

Our approach to constructing good implementations of $\mathcal{S}$ will be to compute local formulas $\theta(x)$ that are equivalent to $\kappa(x)$ in approximations $\mathcal{I}'$ of the ultimate implementation being constructed. We take this idea one step further. Suppose that we have used this technique to determine the value of $\theta(x)$ for some of the template variables x of $\mathcal{S}$. Then we have increased our information about the final implementation θ, so we are able to construct a *better* approximation $\mathcal{I}''$ to the final implementation $\mathcal{I}(E,P\theta)$, in the sense that $\mathcal{I}' \supseteq \mathcal{I}'' \supset \mathcal{I}(E,P\theta)$. Note that if $\mathcal{I}' \models AG(\phi' \Leftrightarrow \kappa(y))$ and $\mathcal{I}'' \models AG(\phi'' \Leftrightarrow \kappa(y))$, then it follows from $\mathcal{I}' \supseteq \mathcal{I}''$ that $\mathcal{I}'' \models AG(\phi' \Rightarrow \phi'')$. That is, ϕ'' is weaker than ϕ', and hence a better approximation to the knowledge condition $\kappa(y)$ in the ultimate implementation $\mathcal{I}(E,P\theta)$. Thus, by proceeding iteratively through the template variables, and improving the approximation as we construct a partial implementation, we are able to obtain better approximations to $\kappa(y)$ in $\mathcal{I}(E,P\theta)$ for later variables.

More precisely, suppose that we have a total pre-order on the set of all template variables $Vars(\mathrm{P}) = \cup_{i \in Ags} Vars(\mathrm{P}_i)$, i.e., a binary relation $\leq$ on this set that is transitive and satisfies $x \leq y \vee y \leq x$ for all $x, y \in Vars(\mathrm{P})$. Let this be represented by the sequence of subsets $X_1, \ldots, X_k$, where for $i \leq j$ and $x \in X_i$ and $y \in X_j$ we have $x < y$ if $i < j$ and $x \leq y \leq x$ if $i = j$. Suppose we have a sequence of interpreted systems $\mathcal{I}_0 \supseteq \ldots \supseteq \mathcal{I}_k$. Define a substitution θ to be *consistent* with this sequence if for all $i = 1 \ldots k$ and $x \in X_i$, we have $\mathcal{I}_{i-1} \models AG(\theta(x) \Leftrightarrow \kappa(x))$. That is, consistent substitutions associate to each template variable x a local formula that is equivalent to (not just sufficient for) $\kappa(x)$, but in an associated approximation system rather than in the final implementation.

Proposition 1 *Suppose that $\mathcal{I}_k$ is isomorphic to $\mathcal{I}(E,P\theta)$, and that for all $x \in Vars(\mathrm{P})$, the formula $\kappa(x)$ contains knowledge operators and the branching operator A only in positive position. Then θ implements the epistemic protocol specification $\langle Ags, E, P, \Phi_\kappa \rangle$.*

We will apply this result as follows: define an *approximation scheme* to be a mapping that, given an epistemic protocol specification $\mathcal{S} = \langle Ags, E, P, \Phi \rangle$ and a partial substitution θ for $\mathcal{S}$, yields a system $\mathcal{I}(\mathcal{S},\theta)$, satisfying the conditions

1. if $\theta \subseteq \theta'$ then $\mathcal{I}(\mathcal{S},\theta) \supseteq \mathcal{I}(\mathcal{S},\theta')$, and
2. if θ is total, then $\mathcal{I}(\mathcal{S},\theta)$ is isomorphic to $\mathcal{I}(E,P\theta)$.

Assume now that $\mathcal{S}$ is a sound local proposition specification based on the mapping κ. Given the ordering $\leq$ on $Vars(\mathtt{P})$, with the associated sequence of sets $X_1 \ldots X_k$, we define the sequence $\theta_0, \theta_1, \ldots, \theta_k$ inductively by $\theta_0 = \emptyset$ (the partial substitution that is nowhere defined), and θ_{j+1} to be the extension of θ_j obtained by defining, for $x \in X_{j+1}$, the value of $\theta_{j+1}(x)$ to be the local proposition ϕ such that $\mathcal{I}(\mathcal{S}, \theta_j) \models AG(\phi \Leftrightarrow \kappa(x))$. Plainly $\theta_0 \subseteq \theta_1 \subseteq \ldots \subseteq \theta_k$, so we have $\mathcal{I}(\mathcal{S}, \theta_0) \supseteq \mathcal{I}(\mathcal{S}, \theta_1) \supseteq \ldots \supseteq \mathcal{I}(\mathcal{S}, \theta'_k)$. It follows from the properties of the approximation scheme and Proposition 1 that the substitution θ_k is total and is an implementation of $\mathcal{S}$.

This idea leads to an extension of the idea of epistemic protocol specifications: we now consider specifications of the form $(\mathcal{S}, \leq)$, where $\mathcal{S}$ is a sound local proposition epistemic protocol specification, and $\leq$ is a total pre-order on the template variables of $\mathcal{S}$. Given an approximation scheme, the construction of the previous paragraph yields a unique implementation of $\mathcal{S}$. Intuitively, by specifying an order $\leq$, the programmer fixes the order in which implementations are synthesized for the template variables, and the approach guarantees that variables later in the order are synthesized using information about the values of variables earlier in the order.

5 A spectrum of approximations

It remains to determine which approximation scheme to use in the approach to constructing implementations described in the previous section. In this section, we consider a number of possibilities for the choice of approximation scheme. A number of criteria may be applied to the choice of approximation scheme. For example, since the programmer must select the order in which variables are synthesized, the approximation scheme should be simple enough to be comprehensible to the programmer, so that they may understand the consequences of their ordering decisions.

On the other hand, since synthesis is to be automated, we would like the computation of the values $\theta(x)$ to be efficient. This amounts to efficiency of the model checking problem $\mathcal{I}(\mathcal{S}, \theta') \models \kappa(x)$ for partial substitutions θ' and formulas $\kappa(x) \in \mathrm{CTLK}^+$. To analyze this complexity, we work below with a complexity measure that assumes explicit state representations of environments, but we look for cases where the model checking problem in the approximation systems is solvable in PTIME. We assume that the protocol template $\mathtt{P}$ and the formulas Φ in the epistemic protocol specification are fixed, and measure complexity as a function of the size of the environment E. This is because in practice, the size of the environment is likely to be the dominant factor in complexity.

One immediately obvious choice for the approximation scheme is to take the system $\mathcal{I}(\mathcal{S}, \theta)$, for a partial substitution θ, to be the union of all the systems $\mathcal{I}(E, P\theta')$, over all total substitutions θ' that extend the partial substitution θ. This turns out not to be a good choice (it is the intractable case $\mathcal{I}_{ii,ir,sc}$ below), so we consider a number of relaxations of this definition. The following abstract view of the situation provides a convenient format that unifies the definition of these relaxations.

Given an environment E with states S, define a *strategy* for E to be a function $\sigma : S^+ \to \mathcal{P}(S) \setminus \emptyset$ mapping each nonempty sequence of states to a set of possible successors. We require that for each $t \in \sigma(s_0 \ldots s_k)$ we have $s_k \xrightarrow{\mathbf{a}} t$ for some joint action $\mathbf{a}$. Given a set Σ of strategies, we can construct an interpreted system consisting of all runs consistent with some strategy in Σ. We encode the strategy into the run. We use the extended set of global states $S \times \Sigma$. We take $\mathcal{R}_\Sigma$ to be the set of all $r : \mathbf{N} \to S \times \Sigma$ such that there exists a strategy σ such that for all $n \in \mathbf{N}$ we have $r(n) = (s_n, \sigma)$, for some $s_n \in S$, and, we have $s_{n+1} \in \sigma(s_0 s_1 \ldots s_n)$ for all $n \in \mathbf{N}$. Intuitively, this is the set of all infinite runs, each using some fixed strategy in Σ, with the strategy encoded into the state. We define $\mathcal{I}(E, \Sigma) = (\mathcal{R}_\Sigma, \sim, \pi')$ where $\sim = \{\sim_i\}_{i \in Ags}$ is the relation on points of $\mathcal{R}_\Sigma$ defined by $(r, m) \sim_i (r', m')$ if, with $r(m) = (s, \sigma)$ and $r'(m') = (s', \sigma')$, we

have $O_i(s) = O_i(s')$. The interpretation π' on $S \times \Sigma$ is defined so that $\pi'(s,\sigma) = \pi(s)$, where $s \in S$, $\sigma \in \Sigma$ and π is the interpretation from E.

A *memory definition* is a collection of functions $\mu = \{\mu_i\}_{i \in Ags}$ with each μ_i having domain S^+. In particular, we work with the following memory definitions derived using the observation functions in the environment E:

- The *perfect information, perfect recall* definition $\mu^{pi,pr} = \{\mu_i^{pi,pr}\}_{i \in Ags}$ where $\mu_i^{pi,pr}(s_0 \dots s_k) = s_0 \dots s_k$
- The *perfect information, imperfect recall* definition $\mu^{pi,ir} = \{\mu_i^{pi,ir}\}_{i \in Ags}$ where $\mu_i^{pi,ir}(s_0 \dots s_k) = s_k$
- The *imperfect information, perfect recall* definition $\mu^{ii,pr} = \{\mu_i^{ii,pr}\}_{i \in Ags}$ where $\mu_i^{ii,pr}(s_0 \dots s_k) = O_i(s_0) \dots O_i(s_k)$
- The *imperfect information, imperfect recall* definition $\mu^{ii,ir} = \{\mu_i^{ii,ir}\}_{i \in Ags}$ where $\mu_i^{ii,ir}(s_0 \dots s_k) = O_i(s_k)$

A strategy *depends* on memory definition μ if there exist functions $F_i : range(\mu_i) \to \mathcal{P}(Acts_i)$ for $i \in Ags$ such that for all sequences $\rho = s_0 \dots s_k$, we have $t \in \sigma(s_0 \dots s_k)$ iff $s \xrightarrow{\mathbf{a}} t$ for some joint action $\mathbf{a}$ such that for all $i \in Ags$, we have $\mathbf{a}_i \in F_i(\mu_i(s_0 \dots s_k))$.

Let $\mathtt{P}$ be a joint protocol template and let θ be a partial substitution for $\mathtt{P}$. A strategy σ is *substitution consistent* with respect to $\mathtt{P}, \theta$ and a memory definition μ if σ depends on μ and for all sequences $s_0 \dots s_k$ there exists a substitution $\theta' \supseteq \theta$ mapping all the template variables of $\mathtt{P}$ undefined by θ to truth values, such that

$$\sigma(s_0 \dots s_k) = \{t \mid \text{there exists } \mathbf{a} \in en(P\theta', s_k),\ s_k \xrightarrow{\mathbf{a}} t\} \tag{2}$$

Note that since the choice of θ' is allowed to depend on $s_0 \dots s_k$, this does not imply that the set of possible successors states $\sigma(s_0 \dots s_k)$ depends only on the final state s_k; the reference to s_k in the right hand side of equation 2 is included just to allow the enabled actions to be determined in a way consistent with the substitution θ, which already associates some of the variables with predicates on the state s_k.

Example 3 *Consider the maximally nondeterministic, or* top*, strategy $\sigma_\top$, defined by $\sigma_\top(s_0 \dots s_k) = \{t \mid$ there exists $\mathbf{a} \in Acts, s_k \xrightarrow{\mathbf{a}} t\}$ for all $s_0 \dots s_k$. Intuitively, this strategy allows any action to be taken at any time. It is easily seen that $\sigma_\top$ depends on every memory definition μ. However, it is not in general substitution consistent, since there are protocol templates for which the set of enabled actions (and hence the transitions) depend on the substitution.*

Consider the protocol template $\mathtt{P} =$ **do** $x \to a$ [] $\neg x \to b$ **od** *for a single agent, in an environment with states $S = \{s_0, s_1, s_2\}$ and transitions $s_0 \xrightarrow{a} s_1$, $s_0 \xrightarrow{b} s_2$, $s_1 \xrightarrow{a,b} s_1$ and $s_2 \xrightarrow{a,b} s_2$. Let θ be the empty substitution. For all substitutions θ', $en(\mathtt{P}\theta', s_0)$ is either $\{a\}$ or $\{b\}$, so for the sequence s_0, the right hand side of equation (2) is equal to either $\{s_1\}$ or $\{s_2\}$. For the strategy $\sigma_\top$, we have $\sigma_\top(s_0) = \{s_1, s_2\}$. Hence this strategy is not substitution consistent in this environment.* □

We now obtain eight sets of strategies by choosing an information mode $a \in \{pi, ii\}$, a recall mode $b \in \{pr, ir\}$ and a selection $c \in \{sc, nsc\}$ to reflect a choice with respect to the requirement of substitution consistency. Formally, given a joint protocol template $\mathtt{P}$, a partial substitution θ for $\mathtt{P}$, and an environment E, we define $\Sigma^{a,b,c}(\mathtt{P}, \theta, E)$ to be the set of all strategies in E that depend on $\mu^{a,b}$, and that are substitution consistent with respect to P, θ and $\mu^{a,b}$ in the case $c = sc$.

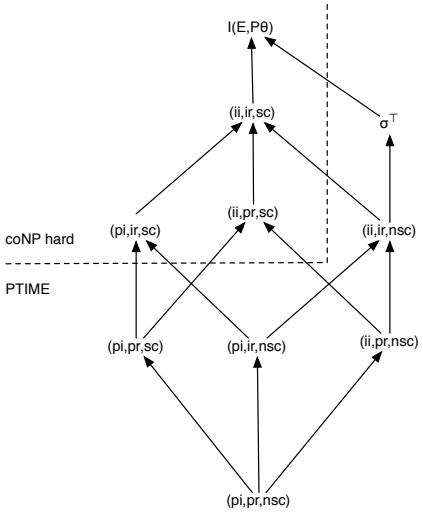


Figure 2: Lattice structure of the approximations

Corresponding to these eight sets of strategies, we obtain eight approximation schemes. Let $\mathcal{S}$ be an epistemic protocol specification with joint protocol template $\mathtt{P}$, and environment E. Given a partial substitution θ for $\mathtt{P}$, and a triple a,b,c, we define the system $\mathcal{I}_{a,b,c}(\mathcal{S},\theta)$ to be $\mathcal{I}(\Sigma^{a,b,c}(P,\theta,E),E)$.

Proposition 2 *For each information mode $a \in \{pi, ii\}$, a recall mode $b \in \{pr, ir\}$ and selection $c \in \{sc, nsc\}$, the mapping $\mathcal{I}_{a,b,c}$ is an approximation scheme.*

Additionally we have the approximation scheme $\mathcal{I}^{\top}(\mathcal{S},\theta)$ defined to be $\mathcal{I}(\{\sigma^{\top}_{E,P\theta}\},E)$, based on the top strategy in E relative to the protocol template $\mathtt{P}\theta$, which is defined by taking $\sigma^{\top}_{E,\mathtt{P}\theta}(s_0 \ldots s_k)$ to be the set of all states $t \in S$ such that there exists a joint action $a \in Acts$ such that for all $i \in Ags$, the protocol template $\mathtt{P}_i\theta$ contains a clause $\phi\theta \rightarrow a_i$ with $\phi\theta$ satisfiable relative to $\pi(s_k)$. (We note that here $\pi(s_k)$ provides the values of propositions *Prop* and we are asking for satisfiability for some assignment to the variables on which θ is undefined. Because we are interested in the case where $\mathtt{P}$, and hence ϕ, is fixed, this satisfiability test can be performed in PTIME as the environment varies.)

For reasons indicated in Example 3, the strategy $\sigma^{\top}_{E,P\theta}$ is not substitution-consistent. However, it is easily seen to depend only on the values $O_i(s_k)$, so we have $\sigma^{\top}_{E,P\theta} \in \Sigma^{ii,ir,nsc}$.

Figure 2 shows the lattice structure of the approximation schemes, with an edge from a scheme $\mathcal{I}$ to a scheme $\mathcal{I}'$ meaning that $\mathcal{I}'$ is a closer approximation to the final system $\mathcal{I}(E,\mathtt{P}\theta)$ synthesized, informally in the sense that $\mathcal{I}$ has more runs and more branches from any point than does $\mathcal{I}'$. (Generally, the relation is one of simple containment of the sets of runs, but in the case of edges involving $\mathcal{I}(E,\mathtt{P}\theta)$ and $\sigma^{\top}$, we need a notion of simulation to make this precise.)

Besides yielding an approach to the construction of implementations of epistemic protocol specifications, we note that our approach also overcomes the counterintuitive aspect of knowledge-based programs illustrated in Example 2.

Example 4 *Suppose that we replace the specification formulas $AG(x_i \Leftrightarrow K_iAXw)$ in Example 2 by the weaker form $AG(x_i \Rightarrow K_iAXw)$, and impose the ordering $x_A < x_B$ on the template variables. We compute the implementation obtained when we use $\mathcal{I}^{\top}$ as the approximation scheme. We take θ_0 to be the empty substitution. $\mathcal{I}(\{\sigma^{\top}_{E,P\theta_0}\},E)$ has all possible behaviours of the original environment, so at the start*

state, we have $\neg K_A(AXw)$. *It follows that substitution* θ_1, *which has domain* $\{x_A\}$ *assigns to* x_A *a local proposition that evaluates to* **false** *at the initial state. Hence,* $\mathtt{P}_A\theta_1$ *selects action* **w** *at the initial state. The effect of this is to delete the bottom transition from the state transition diagram for the environment in Figure 1. It follows that in* $\mathcal{I}(\{\sigma^{\top}_{E,P\theta_1}\}, E)$, *we have* $K_B(AXw)$ *at the initial state, so* $\theta_2(x_B)$ *evaluates to* **true** *at the initial state. This means that the final implementation* $\mathtt{P}\theta_2$ *is the protocol in which Alice brings wine and Bob brings cheese, leading to a successful picnic, by contrast with the knowledge-based program, which does not yield any solutions to their planning problem. (We remark that both Alice and Bob could compute this implementation independently, once given the ordering on the variables. They do not need to communicate during the computation of the implementation.)* □

We noted above in Theorem 3 that a sound local proposition specification that is obtained from a knowledge-based program includes amongst its implementations all the implementations of the knowledge-based program. The knowledge-based program, in effect, imposes additional optimality constraints on these implementations. Our ordered semantics aims to approximate these optimal implementations. It is therefore of interest to determine whether the ordered semantics for sound local proposition specifications can sometimes find such optimal implementations. Although it is not true in general, there are situations where the implementations obtained are indeed optimal. The following provides an example.

Example 5 *Consider the sound local proposition specification obtained from the knowledge-based program of Example 1 by replacing the* $\Leftrightarrow$ *operators in the formulas by* $\Rightarrow$. *That is, we take* Φ *to contain the formulas*

$$AG(x \Rightarrow K_A(position_A \geq 2))$$

and

$$AG(y \Rightarrow K_B(\bigwedge_{p\in[0,\ldots 10]} position_B = p \Rightarrow AG(position_A < p-1)))$$

We consider the setting where sensors readings are within 1 of the actual position. Suppose that we use $\mathcal{I}^{\top}$ *as the approximation scheme, and order the template variables using* $x < y$, *i.e., we synthesize a solution for A before synthesizing a solution for B (knowing what A is doing.) Then, for A, we construct* $\theta(x)$ *as the local proposition for A that satisfies*

$$AG(x \Leftrightarrow K_A(position_A \geq 2))$$

in a system where both A and B may choose either action **Move** *or* **Halt** *at any time. We obtain the substitution* θ_1 *where* $\theta_1(x)$ *is* $sensor_A \geq 3$, *which ensures that always* $position_A \leq 4$, *and in which A may halt at a position in the set* $\{2,3,4\}$. *In the next step, we synthesize* $\theta(y)$ *as the local proposition such that*

$$AG(y \Leftrightarrow K_B(\bigwedge_{p\in[0,\ldots 10]} position_B = p \Rightarrow AG(position_A < p-1)))$$

in the system where A runs $\mathtt{P}_A\theta_1$, *and where B may choose either action* **Move** *or* **Halt** *at any time. In this system, B knows that A's position is always at most* 4, *so it is safe for B to move if* $position_B \geq 6$. *Agent B knows that its position is at least 6 when it gets a sensor reading at least 7. Hence, we obtain the substitution* θ_2 *where* $\theta_2(y)$ *is* $sensor_B \geq 7$ *and* $\theta_2(x)$ *is* $sensor_A \geq 3$. *It can be verified that this substitution is in fact an implementation of the original knowledge-based program.*

6 Complexity of model checking in the approximations

To construct an implementation based on the extended epistemic protocol specification $(\mathcal{S}, \leq)$ using an approximation scheme $\mathcal{I}(\mathcal{S}, \theta)$, we need to perform model checking of formulas in CTLK$^+$ in the systems produced by the approximation scheme. We now consider the complexity of this problem for the approximation schemes introduced in the previous sections. We focus on the complexity of this problem with the protocol template fixed as we vary the size of the environment, for reasons explained above.

We say that the *environment-complexity* of an approximation scheme $\mathcal{I}(\mathcal{S}, \theta)$ is the maximal complexity of the problem of deciding $\mathcal{I}(\mathcal{S}, \theta), o \models \kappa(x)$ with all components fixed and only the environment E in $\mathcal{S}$ varying. More precisely, write $\mathcal{S}^- = \langle Ags, \mathtt{P}, \kappa \rangle$ for a tuple consisting of a set Ags of agents, a collection $\mathtt{P} = \{\mathtt{P}_i\}_{i \in Ags}$ of protocol templates for these agents, and a mapping κ associating, for each agent i, a formula $\kappa(x) = K_i\phi$ of CTLK$^+$ to each template variable x in $\mathtt{P}_i$. Given an environment E, write $\mathcal{S}^-(E)$ for the epistemic protocol specification $\langle Ags, E, \{\mathtt{P}_i\}_{i \in Ags}, \Phi_\kappa \rangle$ obtained from these components. Say that E *fits* a tuple $(\mathcal{S}^-, \theta, o, x)$ consisting of $\mathcal{S}^-$ as above, a substitution θ assigning a boolean formula to a subset of the template variables in $\mathtt{P}$, an observation o and a variable x, if E contains all actions used in $\mathtt{P}$, o is an observation in E of the agent i such that $\mathtt{P}_i$ contains x, and for each x such that $\theta(x)$ is defined, the formula $\theta(x)$ is local in E to the agent i such that $\mathtt{P}_i$ contains x. Given $\mathcal{S}^- = \langle Ags, \mathtt{P}, \kappa \rangle$ and θ, o and x, define $EC_{(\mathcal{S}^-, \theta, o, x)}$ to be the set

$$\{E \mid E \text{ fits } (\mathcal{S}^-, \theta, o, x) \text{ and } \mathcal{I}(\mathcal{S}^-(E), \theta), o \models \kappa(x)\} .$$

Then the environment-complexity of an approximation scheme $\mathcal{I}(\mathcal{S}, \theta)$ is the maximal complexity of the problem of deciding the sets $EC_{(\mathcal{S}^-, \theta, o, x)}$ over all choices of $\mathcal{S}^-$, θ, o and x .

We note that even though we have allowed perfect recall and/or perfect information in the strategy spaces used by the approximation, when we model check in the system generated by the approximation, knowledge operators are handled using the usual observational (imperfect recall, imperfect information) semantics. The stronger capabilities of the strategies are used to increase the size of the strategy space in order to weaken the approximation. (Model checking with respect to perfect recall, in particular, would *increase* the complexity of the model checking problem, whereas we are seeking to decrease its complexity.)

It turns out that several of the approximation schemes, that are closest to the final system synthesized (which would give the knowledge-based program semantics), share with the knowledge-based program semantics the disadvantage of being intractable. These are given in the following result.

Theorem 4 *The approximation schemes $\mathcal{I}_{ii,ir,sc}$, $\mathcal{I}_{ii,pr,sc}$, and $\mathcal{I}_{pi,ir,sc}$ have coNP-hard environment complexity, even for a single agent.*

Each of these intractable cases uses substitution consistent strategies and uses either imperfect recall or imperfect information. The proofs vary, but one of the key reasons for complexity in the imperfect recall cases is that the strategy must behave the same way each time it reaches a state. Intuitively, this means that we can encode existential choices from an NP hard problem using the behaviour of a strategy at a state in this case. In the case of $\mathcal{I}_{ii,pr,sc}$, we use obligations on multiple branches indistinguishable to the agent to force consistency of independent guesses representing the same existential choice. All the remaining approximation schemes, it turns out, are tractable:

Theorem 5 *The approximation schemes $\mathcal{I}^\top$, $\mathcal{I}_{ii,ir,nsc}$, $\mathcal{I}_{pi,pr,sc}$, $\mathcal{I}_{pi,ir,nsc}$, $\mathcal{I}_{ii,pr,nsc}$ and $\mathcal{I}_{pi,pr,nsc}$ have environment complexity in PTIME.*

The reasons are varied, but there are close connections to some known results. The scheme $\mathcal{I}^{\top}$ effectively builds a new finite state environment from the environment and protocol by allowing some transitions that would normally be disabled by the protocol, so its model checking problem reduces to an instance of CLTK model checking, which is in PTIME by a mild extension of the usual CTL model checking approach. It turns out, moreover, by simulation arguments, that for model checking CTLK^{+} formulas, the approximations $\mathcal{I}_{ii,ir,nsc}$ and $\mathcal{I}_{ii,pr,nsc}$ are equivalent to $\mathcal{I}^{\top}$, i.e., satisfy the same formulas at the same states, so the algorithm for $\mathcal{I}^{\top}$ also resolves these cases.

The cases $\mathcal{I}_{pi,pr,sc}$ and $\mathcal{I}_{pi,pr,nsc}$ are very close to the problem of *module checking* of universal CTL formulas, which is known to be in PTIME [16]. The proof technique here involves an emptiness check on a tree automaton representing the space of perfect information, perfect recall strategies (either substitution consistent or not required to be so), intersected with an automaton representing the complement of the formula. The cases $\mathcal{I}_{pi,pr,nsc}$ and $\mathcal{I}_{pi,ir,sc}$ can moreover be shown to be equivalent by means of simulation techniques, so the latter also falls into PTIME.

The demarcation between the PTIME and co-NP hard cases is depicted in Figure 2. This shows there are two best candidates for use as the approximation scheme underlying our synthesis approach. We desire an approximation scheme that is as close as possible to the knowledge-based program semantics, while remaining tractable. The diagram shows two orthogonal approximation schemes that are maximal amongst the PTIME cases, namely $\mathcal{I}^{\top}$ and $\mathcal{I}^{pi,pr,sc}$. The former generates a bushy approximation in that it relaxes substitution consistency. The latter remains close to the original protocol by using substitution consistent strategies, but at the cost of allowing perfect information, perfect recall strategies. It is not immediately clear what the impact of these differences will be with respect to the quality of the implementations synthesized using these schemes, and we leave this as a question for future work.

7 Related Work

Relatively little work has been done on automated synthesis of implementations of knowledge-based programs or of sound local proposition specifications, particularly with respect to the observational semantics we have studied in this paper. In addition to the works already cited above, some papers [18, 17, 20, 21, 3] have studied the complexity of synthesis with respect to specifications in temporal epistemic logic using the synchronous perfect recall semantics. A symbolic implementation for knowledge-based programs that run only a finitely bounded number of steps under a clock or perfect recall semantics for knowledge is developed in [13].

There also exists a line of work that is applying knowledge based approaches and model checking techniques to problems in discrete event control, e.g., [2, 10, 15]. In general, the focus of these works is more specific than ours (e.g., in restricting to synthesis for safety properties, rather than our quite general temporal epistemic specifications) but there is a similar use of monotonicity. It would be interesting to apply our techniques in this area and conduct a comparison of the results.

8 Conclusion

In this paper we have proposed an ordered semantics for sound local proposition epistemic protocol specifications, and analyzed the complexity of a model checking problem required to implement the approach, for a number of approximation schemes. This leads to the identification of two optimal approximation schemes, $\mathcal{I}^{\top}$ and $\mathcal{I}^{pi,pr,sc}$ with respect to which the model checking problem has PTIME complexity in an explicit state representation.

A number of further steps are required to obtain a practical framework for synthesis. Ultimately, we would like to be able to implement synthesis using symbolic techniques, so that it can also be practicably carried out for specifications in which the environment is given implicitly using program-like representations, rather than by means of an explicit enumeration of states. The complexity analysis in the present paper develops an initial understanding of the nature of the model checking problems that may be helpful in developing symbolic implementations. In the case of the approximation scheme $\mathcal{I}^{\top}$, in fact, the associated model checking problem amounts essentially to CTLK model checking in a transformed model, for which symbolic model checking techniques are well understood. In work in progress, we have developed an implementation of this case, and we will report on our experimental findings elsewhere.

In the case of the approximation $\mathcal{I}^{pi,pr,sc}$, the model checking problem is more akin to module checking, for which symbolic techniques are less well studied. This case represents an interesting question for future research, as does the question of how the implementations obtained in practice from these tractable approximations differ.

Our examples in this paper give some initial data points that suggest both that the ordered approach is able to construct natural implementations for the sound local proposition weakenings of knowledge-based programs that lack implementations, as well as implementations of such weakenings that are in fact implementations of the original knowledge-based program. More case studies are required to understand how general these phenomena are in practice. It would be interesting to find sufficient conditions under which the ordered approach is guaranteed to generate knowledge-based program implementations.

References

[1] K. Baukus & R. van der Meyden (2004): *A knowledge based analysis of cache coherence.* In: *Proc. 6th Int. Conf. on Formal Engineering Methods*, pp. 99–114.

[2] S. Bensalem, D. Peled & J. Sifakis (2010): *Knowledge Based Scheduling of Distributed Systems*. In: *Time for Verification, Essays in Memory of Amir Pnueli*, Springer LNCS 6200, pp. 26–41.

[3] R. Bozianu, C. Dima & E. Filiot (2014): *Safraless Synthesis for Epistemic Temporal Specifications*. In: *Proc. Int. Conf. on Computer Aided Verification*, pp. 441–456.

[4] R. I. Brafman, J-C. Latombe, Y. Moses & Y. Shoham (1997): *Applications of a logic of knowledge to motion planning under uncertainty*. *JACM* 44(5).

[5] J. Burgess (1979): *Logic and time*. *Journal of Symbolic Logic* 44, pp. 556–582.

[6] C. Dwork & Y. Moses (1990): *Knowledge and common knowledge in a Byzantine environment: crash failures*. *Information and Computation* 88(2), pp. 156–186.

[7] K. Engelhardt, R. van der Meyden & Y. Moses (1998): *Knowledge and the Logic of Local Propositions*. In: *Proc. Conf. Theoretical Aspects of Knowledge and Rationality*, pp. 29–41.

[8] R. Fagin, J. Halpern, Y. Moses & M. Vardi (1995): *Reasoning About Knowledge*. MIT Press.

[9] R. Fagin, J. Y. Halpern, Y. Moses & M. Y. Vardi (1997): *Knowledge-Based Programs*. *Distributed Computing* 10(4), pp. 199–225.

[10] Susanne Graf, Doron Peled & Sophie Quinton (2012): *Achieving distributed control through model checking*. *Formal Methods in System Design* 40(2), pp. 263–281. Available at `http://dx.doi.org/10.1007/s10703-011-0138-9`.

[11] V. Hadzilacos (1987): *A knowledge-theoretic analysis of atomic commitment protocols*. In: *PODS '87: Proc. 6th ACM Symp. on Principles of Database Systems*, pp. 129–134.

[12] J. Y. Halpern & L. D. Zuck (1992): *A little knowledge goes a long way: knowledge-based derivations and correctness proofs for a family of protocols*. *Journal of the ACM* 39(3), pp. 449–478.

[13] X. Huang & R. van der Meyden (2013): *Symbolic Synthesis of Knowledge-based Program Implementations with Synchronous Semantics*. In: *Proc. TARK*, pp. 121–130.

[14] X. Huang & R. van der Meyden (2014): *Symbolic Synthesis for Epistemic Specifications with Observational Semantics*. In: *Proc. Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, pp. 455–469.

[15] Gal Katz, Doron Peled & Sven Schewe (2011): *Synthesis of Distributed Control through Knowledge Accumulation*. In: *Proc. Int. Conf on Computer Aided Verification*, pp. 510–525.

[16] O. Kupferman, M. Y. Vardi & P. Wolper (2001): *Module Checking*. *Information and Computation* 164(2), pp. 322–344.

[17] R. van der Meyden (1996): *Constructing Finite State Implementations of Knowledge-Based Programs with Perfect Recall*. In: *Intelligent Agent Systems, Theoretical and Practical Issues*, LNCS, No. 1209, Springer, pp. 135–151.

[18] R. van der Meyden (1996): *Finite State Implementations of Knowledge-Based Programs*. In: *Proc. Conf. on Foundations of Software Technology and Theoretical Computer Science*, pp. 262–273.

[19] R. van der Meyden (1996): *Knowledge Based Programs: On the Complexity of Perfect Recall in Finite Environments*. In: *Proc. Conf. on Theoretical Aspects of Rationality and Knowledge*, pp. 31–49.

[20] R. van der Meyden & M. Y. Vardi (1998): *Synthesis from Knowledge-Based Specifications.* In: *Proc. CONCUR'98*, Springer LNCS 1466, pp. 34–49.

[21] R. van der Meyden & T. Wilke (2005): *Synthesis of Distributed Systems from Knowledge-Based Specifications*. In: *Proc. Int. Conf. on Concurrency Theory, CONCUR*, pp. 562–576.

[22] R. van der Meyden & K. Wong (2003): *Complete Axiomatizations for Reasoning about Knowledge and Branching Time*. *Studia Logica* 75(1), pp. 93–123.

[23] Moshe Y. Vardi & Pierre Wolper (1986): *Automata-Theoretic Techniques for Modal Logics of Programs*. *J. Comput. Syst. Sci.* 32(2), pp. 183–221.