TECHNISCHE UNIVERSITÄT MÜNCHEN
Lehrstuhl für Echtzeitsysteme und Robotik

An Implementation for Algorithmic Game Solving and its Applications in System Synthesis

Chih-Hong Cheng

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Helmut Seidl

Prüfer der Dissertation:
1. Univ.-Prof. Dr. Alois Knoll
2. Univ.-Prof. Dr. Dr. h.c. Javier Esparza

Die Dissertation wurde am 22.08.2011 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 17.03.2022 angenommen.

Zusammenfassung

Die vorliegende Arbeit beschäftigt sich mit der praktischen Anwendung von algorithmischer Spieletheorie. Spieletheorie ist ein aktiver Forschungsbereich in der theoretischen Informatik und im Bereich formaler Methoden. Die Arbeit stellt das Werkzeug GAVS+ vor, das verschiedenste Arten von Spielen und Gewinnbedingungen, die eine praktische Relevanz haben, umfasst. GAVS+ is das erste Werkzeug, das die verschiedenen Ansätze in dem Bereich, umfassend unterstützt. Auf Basis von dem Werkzeug wird das formale Fundament der Spieltheorie mit einem praktischen Anwendungsfeld, der Synthese, verbunden.

Das erste Anwendungsfeld des Ansatzes ist die Synthese einer Konfiguration, die die Sicherheit und Verklemmungsfreiheit von Systemen aus interagierenden, nicht veränderbaren Komponenten sicherstellt. Konkret werden die Prioritäten von Anwendungen in dem Werkzeug BIP synthetisiert. Der Ansatz erweitert damit existierende Werkzeuge, wie D-Finder, die zwar mögliche Verklemmungen in BIP-Programmen erkennen können, aber nicht automatisch eine korrekte Lösung berechnen können.

Das zweite Anwendungsfeld des Ansatzes ist die Anwendung der Spieletheorie zur Synthese von Plänen. Hierzu wurde GAVS+ um eine Unterstützung von einer modifizierten Version von STRIPS/PDDL erweitert. Die Modifikation betraf dabei die Erweiterung um einen zweiten Spieler, der beispielsweise Störungen oder Fehler in der Systemumgebung repräsentieren kann.

Als letztes Anwendungsfeld wurde die Synthese von fehlertoleranten Systemen durch den Einsatz von Spieletheorie untersucht. Der Ansatz basiert dabei auf drei Phasen: der Zeitabstraktion im ersten Schritt, der Synthese geeigneter Fehlertoleranzmechanismen und schlieslich der Wiederherstellung der Zeitinformationen.

Zusammenfassend beweist diese Arbeit, dass Spieletheorie ein sehr mächtiges Werkzeug für Syntheseprobleme in verteilten, eingebetteten Systemen darstellt. Auch wenn es theoretischer Sicht selbst einfachste Probleme unentscheidbar sind, können die Ansäetze durch Beschränkung des Lösungsraums praktisch angewandt werden. Die Arbeit identifiziert deshalb erste Ansäetze wie der Lösungsraum reduziert werden kann.

Abstract

This thesis describes efforts in bringing algorithmic game solving from theoretical results towards concrete applications. First, an implementation of a game-solving library is presented. The library supports several game types with different winning conditions. Concerning applications, we first consider the problem of risk avoidance and deadlock prevention in component-based systems. A technique called priority synthesis is presented, which enables to automatically generate stateless precedence over actions to avoid risk and deadlock states. The second application is related to behavioral-level synthesis. We extend PDDL to include game aspects, develop algorithmic methods to speed up synthesis, and present a case study in synthesizing controllers for FESTO MPS systems. Third, we investigate how HW/SW level fault-tolerant synthesis can be combined with games. Lastly, to study synthesis in distributed systems, we present initial investigations to compute resource-bounded strategies.

Acknowledgements

First of all, I want to thank my supervisor, Prof. Alois Knoll, for providing me the opportunity to prepare this thesis and for supporting my freedom in research. I am very thankful to Prof. Javier Esparza for accepting to be my second reviewer. His opinions are constructive since my early years in doctoral studies. Many thanks to members in the PUMA doctoral program, the chair of Embedded Systems and Robotics at the TU München, and Fortiss GmbH for the pleasant atmosphere in research.

I am also fortunate to have many external collaborators who stimulate my research spirit. They are Dr. Harald Ruess (Fortiss), Dr. Barbara Jobstmann (Verimag), Prof. Saddek Bensalem (Verimag), Dr. Michael Luttenberger (TUM), Dr. Christian Buckl (Fortiss), Mr. Michael Geisinger (Fortiss), Dr. Rongjie Yan (ISCAS), and Dr. Yu-Fang Chen (Academia Sinica). Dr. Ruess is an important mentor of mine who gives me a flavor of research. My visits to Verimag were wonderful experiences, and many thanks to Barbara and Prof. Bensalem. My knowledge in formal methods came from Prof. Farn Wang (NTU), and I learned embedded systems under Prof. Edward A. Lee (UC Berkeley). They were my supervisors during my master studies, and they are continuously supportive since then.

I also thank my parents for the wonderful family education and all the opportunities they offered me. My wife Tzuchen is very supportive concerning my academic careers. I thank her for her understanding and love.

Finally, my best satisfaction during my doctoral studies is to know Jesus Christ and accept him as my savor for the rest of my life. This thesis, although far from perfect, should be devoted to the Lord.

Contents

List of Figures

CHAPTER 1

Introduction

Contents

1.1 Background

The quest of designing systems satisfying the specification is a fundamental issue in the research of computer science. *Verification* is amongst one of the prevailing techniques, where the goal is to check the correctness of *designed systems* under given mathematical specifications. Techniques for verification include model checking, type analysis, theorem proving and many others. However, when a system is diagnosed as incorrect by verification, it remains to be designers' responsibilities to perform suitable modifications such that its operation satisfies the specification.

The motivation of this thesis originates from an alternative approach for increasing the quality of systems called *synthesis*, which refers to a paradigm that automatically creates a system satisfying a given mathematical specification. As the process of creating systems from specification requires no human-intervention, it is a very appealing approach. We target on the analysis of models containing *controllable and uncontrollable actions*. Reflecting these models to system design, we can interpret the controllable portion as all allowable action rules specified (e.g., for the design of an elevator system, it corresponds to all allowable moves of the elevator), and interpret the uncontrollable portion as the *behavior of the environment* (e.g., demand of users for the elevator). The goal of our synthesis problem is to generate programs (either centralized or distributed) restricting / orchestrating controllable actions such that regardless of actions taken by the environment, the behavior of the constrained model satisfies the specification. Note that it is also possible that the environment is not included as part of the model; for this the concept of synthesis still applies.

With the above restriction, it is natural to connect our synthesis framework with *games*. Overall, games provide (i) an easy way to specify pre-defined behaviors for components in concrete applications, (ii) an intuitive mathematical model for synthesis, and (iii) natural translation between applications and synthesis models. Games of two categories are considered.

- (Two-player game) When the environment is contained as part of the model description, we interpret the synthesis problem as a two-player (or a two-party) game between the **system** and the **environment**. The synthesized result (controller program) should achieve goal-oriented behavior regardless of moves performed by the environment. We may view the environment as *malicious*, i.e., it performs all worst case scenarios to achieve conflicting goals opposite to goals of the system. In games such as chess, this idea of opposite goals is similar: a player should create a strategy (set of allowable moves) to win against his opponent.
- (One-player game) Similarly, one-player game can be used when the environment is not included as part of the model. For example, sliding puzzles (e.g., rush hour[1]) are games of this kind.

With the above analogy, it can be observed that automatic methods for synthesis are closely related to techniques in *algorithmic game solving*, which is a discipline for studying efficient methods to solve games. Except explicitly mentioned, we refer games in this thesis as *infinite games* used in automata theory and theoretical computer science [GTW02]. Infinite games are games where non-terminating (infinite) behavior is the key of consideration; the game itself can be played on a finite or infinite game graph. Notice that for algorithmic game solving, it contains another branch on solving games in economics and on making strategic decisions. The branch above mentioned is not considered within the scope of this thesis; for details, we refer interested readers to [NRTV07].

1.2 Algorithmic Game Solving from Theory to Applications in Synthesis

For algorithmic game solving, despite fruitful results in theory, our main focus is to bridge the gap between theories and applications in synthesis. However, we find it very difficult to find appropriate connections to concrete application domains, if our starting point is a fixed result in theory. Therefore, we try to use the methodology of *application-driven research* used in the Parallel Computing Laboratory (PARLAB), EECS, University of California Berkeley. The following is summarized from a talk[2] as the goal statement of parallel programming in PARLAB [ABC$^+$06]:

[1] `http://en.wikipedia.org/wiki/Rush_Hour_(board_game)`
[2] `http://parlab.eecs.berkeley.edu/wiki/_media/berkeleyview2.ppt`

> For conventional wisdom in the research of computer science, computer scientists solve individual parallel problems with clever language feature, new compiler pass, or novel hardware widget (e.g., SIMD). Then scientists push (or foist) CS nuggets/solutions on users. The problem is that it is difficult for users to use proper solutions. An alternative approach is to work with domain experts developing compelling applications to understand requirements. Based on these applications or requirements, provide HW/SW infrastructure necessary to build, compose, and understand parallel software written in multiple languages.

Applying this concept to synthesis, we provoke to first study concrete problems. Then we extract their underlying models and combine them with existing techniques in games. Concerning novelty of our application-driven methodology, it mainly comes from two aspects.

- Proposing new algorithms which glue concrete applications and games.
- From the need of applications, proposing new models or new algorithms currently absent in the research of games.

To achieve this goal, we find it important to *first* understand games with a broad knowledge, followed by creating a framework which summarizes all types of games having potentials to be applied in practice. For this, we consider a concrete, mature and open-source *implementation* for games as foundational. It offers a platform for the communication between game theorists and practitioners. This is especially important for synthesis, as its goal is to assist engineers (who mostly have little knowledge on formal methods) to facilitate their design processes.

1.3 Main Contributions of this Thesis

The result of this thesis is the effort to connect algorithmic game solving from theories to applications. Main contributions are summarized as follows.

- We create the tool GAVS+ [CBLK10, CKLB11] which collects most types of games having practical interests with various winning conditions. We start with two-player turn-based games [CBLK10], and then we expand its spectrum concerning concurrency, infinite state, distributivity, probability [CKLB11], and lastly, games of imperfect information and others. GAVS+ is the first library which summarizes existing efforts in such a comprehensive way. With the library in hand, we start bridging games in theory and concrete applications in synthesis; new algorithms are introduced when required.
- The first application under investigation is the safety problem (including deadlock prevention) in interaction systems constructed by components; we use the Behavior-Interaction-Priority (BIP) modeling language [BBS06] developed at the Verimag Laboratory[3] as an example. Tools based on verification (e.g., D-finder [BBNS09]) are capable of deadlock finding for BIP systems while the bur-

[3]`http://www-verimag.imag.fr`

den of deadlock removal remains. By understanding the problem and combining concepts in games, we create a technique called **priority synthesis**, which results in a tool targeted for automatic supervisory control [CBJ$^+$11]; it automatically adds static priorities over interactions to restrict the system behavior for safety. We formulate priority synthesis for BIP systems using the automata-theoretic framework proposed by Ramadge and Wonham [RW89]. We focus on the hardness of synthesizing priorities and show that finding a supervisor based on priorities that ensures deadlock freedom of the supervised system is NP-complete [CJBK11]. We describe the underlying algorithms and methods to perform synthesis on complex systems [CBC$^+$11].

- We study how to increase the use of **behavioral-level synthesis**. We argue that an easy-to-use modeling language is required rather than existing approach in LTL synthesis. For this we have implemented in GAVS+ novel features such that the user can now process and synthesize planning (game) problems described in the established STRIPS/PDDL language. It is achieved by introducing a slight extension which allows specifying a second player [CKLB11]. We also use program optimization (transformation) techniques such that a smaller (w.r.t. size) game is brought to the synthesis engine as input. The developed technique is evaluated on the FESTO modular production systems [CGR$^+$12b, CGR$^+$12a].
- For the third application, we study how **HW/SW level fault-tolerant synthesis** can be combined with games [CRBK11, CBK10]. Under strict assumptions, we are able to complete the whole flow, which is based on (a) timing-abstraction to convert systems to games, (b) game solving to select appropriate fault-tolerant mechanisms, and (c) constraint resolution to restore timing information on concrete hardware. This first result is implemented in a software prototype.
- Finally, during our investigation, we found distributed games extremely powerful to capture synthesis problems in embedded systems. Unfortunately, in theory even simple reachability or safety conditions are undecidable for such games. Even so, methodologies of finding strategies with bounded resource are important for practical applications. Our initial investigations to compute **resource-bounded strategies** over such games constitute the last part of the thesis.

We relate three synthesis techniques from three perspectives.

- **(Methodology)** Synthesis can be categorized into two sub categories: *complete* synthesis and *partial* synthesis. Complete synthesis is to synthesize the full functionality, while partial synthesis is applied when system is nearly complete but requires to increase features or remove unwanted behaviors. Overall, behavioral-level synthesis studies complete synthesis, while priority synthesis (restriction-based) and fault-tolerant synthesis (addition-based) are partial techniques.
- **(Working models: level of abstraction)** Behavioral-level synthesis uses PDDL as its description language and is intended to generate abstract plans. Contrarily, our fault-tolerant synthesis works on the process level and contains hardware models in order to perform synthesis fitting timing constraints. Priority synthesis works on a level of abstraction in between: It contains components, and these components are implicitly related to dedicated hardware platforms.

- **(Distributivity)** Behavioral-level synthesis generates monolithic controllers, while fault-tolerant synthesis on HW/SW level generates distributed controllers. Priority synthesis generates results in between: the created priorities can either be implemented as a centralized controller or refined as a distributed controller.

1.4 Structure of this Thesis

The thesis is composed of eight parts discussing different aspects, where most of the chapters have been published or are currently under review for conference proceedings. After the introduction, the thesis continues in Chapter 2 with the concept of games and descriptions of divergent game types having potential for practical use in synthesis. For each type of the game, an outline of basic algorithms for game solving is listed. Therefore, Chapter 2 can be viewed as the basis of this thesis.

- Chapter 3 presents the tool GAVS+. In this chapter, we focus on (i) how games can be constructed using the graphical user interface (GUI) of GAVS+, (ii) how visualization of strategies or intermediate results of computation is presented, and (iii) the underlying software architecture as guidance for extending new algorithms or new game types[4]. Contents of this chapter are mainly in [CBLK10, CKLB11].
- Chapter 4 describes the use and the underlying algorithms of VISSBIP, a tool for constructing simple interaction systems and importantly, automatically synthesizing priorities over actions to achieve system safety. For the ease of understanding, we introduce the model of computation called (synchronous) BIP [BBS06] before describing algorithmic issues. Contents of this chapter are mainly in [CBC$^+$11, CBJ$^+$11, CJBK11].
- Chapter 5 presents the connection between games and the planning domain. In the AI community, the PDDL language is used as the standard language to describe behavioral-level actions for robotics. We first give a brief description on PDDL (restricted to Level 1 only), then describe the effort to represent the second player (e.g., it can be used to model faults or environment uncertainties). With this feature, it is possible to perform behavioral-level synthesis. Contents of this chapter are in [CKLB11, CGR$^+$12b, CGR$^+$12a].
- Chapter 6 presents fault-tolerant synthesis in embedded systems. We first describe starting models for synthesis and (implementable) assumptions used in the framework. Then we present algorithms to (i) generate games from models and (ii) translate from the result of synthesis back to the model, including required modifications. A simple instruction on our prototype tool GECKO is also listed. Contents of this chapter are mainly in [CRBK11, CBK10].
- Chapter 7 describes three heuristic algorithms to find controller strategies for distributed games. These heuristics arise from the design intention how engineers construct systems out of components to achieve goal-oriented behavior. Results concerning reachability games are within [CRBK11].

[4]The tool is available at `http://www6.in.tum.de/~chengch/gavs/`

- Chapter 8 gives a summary of the main results of this thesis. In addition, several starting points for future / ongoing research are identified.

Related work is listed distributively in each chapter.

CHAPTER 2

Games for Synthesis: a Very Short Introduction

Abstract

We give concise[1] descriptions on games for synthesis. We start with two-player, turn-based games played on finite graphs, and then expand the spectrum concerning concurrency, infinite state, distributivity, probability, games of imperfect information, and time[2]. Figure 2.1 provides a summary on games described in this chapter and illustrates their relations. Further game extensions by combining multiple extension criteria (e.g., stochastic pushdown games) are left to readers.

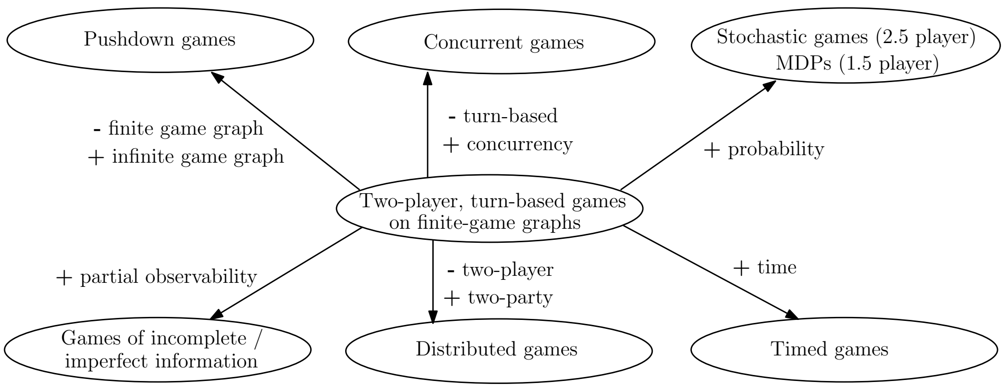


Figure 2.1: A spectrum of games for synthesis and their relations.

[1]This chapter is intended to serve as a short tutorial for a quick grasp on games. Therefore, it only contains required definitions to state algorithmic concepts. Yet examples are offered for the ease of understanding. To collect results and algorithms with full detail is beyond the scope of the thesis. Even for the newly published book [AG11] (available from January 2011), it is unable to cover every aspect mentioned in this chapter.

[2]Direction of our expansion is based on reviewing existing work on games.

Contents

2.1 Two-player, Turn-based Games over Finite Arenas

2.1.1 Definition

In this section, we define basic two-player, turn-based games on finite game graphs. Contents of this section are reorganized from Chapter 2 of the book [GTW02]. We start with the definition of *arena* (game graph), then proceed with *plays* on the arena, *winning conditions*, and *strategies*.

2.1.1.1 Arena, play, and game

An *arena* or a *game graph* is a directed graph $A = (V_0 \uplus V_1, E)$ whose *nodes* (or *locations*) are partitioned into two classes V_0 and V_1. We only consider the case of two participants in the following and call them **player 0 (system)** and **player 1 (environment)** for simplicity. A *play* starting from node v_0 is simply a maximal path $\pi = v_0 v_1 \dots$ in G where we assume that player i determines the *move* $(v_k, v_{k+1}) \in E$ if $v_k \in V_i$, $i = 0, 1$. With Occ(π) (Inf(π)) we denote the set of nodes visited (visited infinitely often) by a play π.

Let A be an arena defined above, refer $Win \subseteq (V_0 \uplus V_1)^\omega$ to be the *winning set* over arena A. Then define a **game** G as the pair (A, Win). A play $\pi = v_0 v_1 \dots$ is won by player 0 if (i) it is an infinite play and $\pi \in Win$ or (ii) it is a finite play $\pi = v_0 v_1 \dots v_k$, $v_k \in V_1$ and player 1 is unable to proceed a move (i.e., deadlock for player 1).

2.1.1.2 Winning conditions over the game graph

In this thesis, the winning set of a given game is defined using the notion of *winning condition*. Given game graph $A = (V_0 \uplus V_1, E)$,

- the **reachability condition** is defined by $Win = \{v_0 v_1 \dots \in (V_0 \uplus V_1)^\omega \mid \text{Occ}(v_0 v_1 \dots) \cap V_{goal} \neq \emptyset\}$, where $V_{goal} \subseteq V_0 \uplus V_1$ is the set of *goal states*.
- the **co-reachability condition** is defined by $Win = \{v_0 v_1 \dots \in (V_0 \uplus V_1)^\omega \mid \text{Occ}(v_0 v_1 \dots) \cap V_{risk} = \emptyset\}$, where $V_{risk} \subseteq V_0 \uplus V_1$ is the set of *risk states*. Sometimes co-reachability conditions are called **safety** conditions.

- the **Büchi condition** is defined by $Win = \{v_0v_1 \ldots \in (V_0 \uplus V_1)^\omega \mid \text{Inf}(v_0v_1 \ldots) \cap V_{goal} \neq \emptyset\}$, where V_{goal} is the set of goal states in $V_0 \uplus V_1$.
- the **Muller condition** is defined by $Acc = \{v_0v_1 \ldots \in (V_0 \uplus V_1)^\omega \mid \text{Inf}(v_0v_1 \ldots) \in \mathcal{F}\}$, where $\mathcal{F} \subseteq \mathbf{2}^{V_0 \uplus V_1}$, and $\mathbf{2}^{V_0 \uplus V_1}$ is the powerset of $V_0 \uplus V_1$.
- the **Staiger-Wagner condition** is defined by $Acc = \{v_0v_1 \ldots \in (V_0 \uplus V_1)^\omega \mid \text{Occ}(v_0v_1 \ldots) \in \mathcal{F}\}$, where $\mathcal{F} \subseteq \mathbf{2}^{V_0 \uplus V_1}$, and $\mathbf{2}^{V_0 \uplus V_1}$ is the powerset of $V_0 \uplus V_1$.
- the **Streett condition** is defined by $Win = \{v_0v_1 \ldots \in (V_0 \uplus V_1)^\omega \mid \forall i \in \{1, \ldots, m\} : \text{Inf}(v_0v_1 \ldots) \cap F_i \neq \emptyset \Rightarrow \text{Inf}(v_0v_1 \ldots) \cap E_i \neq \emptyset\}$, where $\forall i \in \{1, \ldots, m\} : E_i, F_i \subseteq V_0 \uplus V_1$.

2.1.1.3 Parity and weak-parity conditions based on coloring

Here we describe the *coloring function* over vertices in the game arena, such that the winning condition can also be defined using the associated color of a play. Given an arena $A = (V_0 \uplus V_1, E)$, define the coloring function $c : V_0 \uplus V_1 \rightarrow C$, where $C \subset \mathbb{N}_0$ is a *finite* set of nonnegative integers. For a vertex v, define its color as $c(v)$; for a play $\pi = v_0v_1 \ldots$, let $c(\pi) = c(v_0)c(v_1) \ldots$ be the coloring. With $\text{Occ}(c(\pi))$ ($\text{Inf}(c(\pi))$) we denote the set of colors visited (visited infinitely often) by a play π. Given game graph $A = (V_0 \uplus V_1, E)$ with the coloring function c,

- the **weak parity condition** is defined by $Acc = \{v_0v_1 \ldots \in (V_0 \uplus V_1)^\omega \mid \texttt{max}(\text{Occ}(c(v_0)c(v_1) \ldots)) \texttt{ mod } 2 = 0\}$.
- the **parity condition** is defined by $Acc = \{v_0v_1 \ldots \in (V_0 \uplus V_1)^\omega \mid \texttt{max}(\text{Inf}(c(v_0)c(v_1) \ldots)) \texttt{ mod } 2 = 0\}$.

2.1.1.4 Strategies

Given a game $G = ((V_0 \uplus V_1, E), Win)$, a **strategy** for player 0 starting at vertex set V_{start} is a partial function $f : (V_0 \uplus V_1)^* \times V_0 \rightarrow V_0 \uplus V_1$ such that for all play prefix $v_0 \ldots v_k$, $v_0 \in V_{start}$, f assigns a successor vertex v_{k+1} where $(v_k, v_{k+1}) \in E$, if $v_k \in V_0$ and a successor exists. For player 0, play $\pi = v_0v_1 \ldots$ follows strategy f if $\forall v_i \in V_0 : v_{i+1} = f(v_0 \ldots v_i)$. We call f a *winning strategy* for player 0 if all plays following the strategy while starting from V_{start} is contained in Win. Lastly, we refer a positional (memoryless) strategy for player 0 as the function $f : V_0 \rightarrow V_0 \uplus V_1$ such that the history of a play is not considered when deciding the next move. The above definitions on strategies can be applied analogously for player 1.

A node v is *won* by player i if player i can always choose his moves in a way that he wins any resulting play starting from v; the *winning region* for i, i.e., the sets of nodes won by player i are denoted by Win_i ($i \in \{0, 1\}$).

Before stepping further, it is worth mentioning the *determinacy result*, which states that given a game graph with the above winning conditions, Win_0 and Win_1 form a partition over $V_0 \uplus V_1$. For details on determinacy results, we refer readers to the discussion in Chapter 6 of [GTW02].

2.1.1.5 Example

Here we give a simple example[3] intended to illustrate all concepts mentioned above. Consider the arena $A = (V_0 \uplus V_1, E)$ in Figure 2.2. In this thesis, we use rectangles to represent player-1 vertices and circles for player-0 vertices.

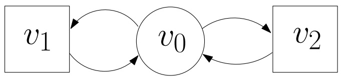


Figure 2.2: A simple finite game graph.

- $V_0 = \{v_0\}$, $V_1 = \{v_1, v_2\}$.
- $E = \{(v_0, v_1), (v_0, v_2), (v_1, v_0), (v_2, v_0)\}$.

We outline some winning conditions and possible winning strategies.

- (Reachability) Define V_{goal} to be $\{v_1\}$, i.e., the goal is to reach vertex V_1. Player 0 has a positional winning strategy from $\{v_0, v_1, v_2\}$ by applying the edge (v_0, v_1).
- (Safety) Define V_{risk} to be $\{v_2\}$, i.e., the goal is to never reach vertex V_2. Player 0 has a positional winning strategy from $\{v_0, v_1\}$ by applying the edge (v_0, v_1); player 1 has a positional winning strategy from $\{v_2\}$ by applying the edge (v_2, v_0).
- (Muller) Define $\mathcal{F}$ to be $\{\{v_0, v_1, v_2\}\}$, i.e., the goal is to reach all vertices infinitely often. It can be checked that player 0 does not have any positional winning strategy. However, he has winning strategies using memory to win from $\{v_0, v_1, v_2\}$. E.g., *if the previous location is v_1 then go to v_2; if the previous location is v_2 then go to v_1*.
- (Streett) Define $E_1 = \{v_2\}$ and $F_1 = \{v_1\}$, i.e., if a play reaches v_1 infinitely often, then it should also reach v_2 infinitely often. For player 0 positional strategy suffices: on v_0 move to v_2.

2.1.2 Algorithms

In this section, we summarize existing algorithmic results. The focus is on (i) attractor computation, (ii) techniques for parity game solving, (iii) game reduction, and (iv) symbolic encoding for computing attractors. Our selection of algorithms is based on the implementation in GAVS+.

2.1.2.1 Attractor

We recall the definition of **attractor**, a term which is later used in the implementation of the solvers: for $i \in \{0, 1\}$ and $X \subseteq V_0 \uplus V_1$, the map $\mathsf{attr}_i(X)$ is defined by

$$\mathsf{attr}_i(X) := X \cup \{v \in V_i \mid vE \cap X \neq \emptyset\} \cup \{v \in V_{1-i} \mid \emptyset \neq vE \subseteq X\},$$

[3]This example is from the lecture in *Summer School of Logic and Learning 2009* taught by Dr. Sophie Pinchinat (`http://videolectures.net`).

i.e., $\mathsf{attr}_i(X)$ extends X by all those nodes from which either player i can move to X within one step or player $1-i$ cannot prevent to move within the next step. (vE denotes the set of successors of v.) Then $\mathsf{Attr}_i(X) := \bigcup_{k\in\mathbb{N}} \mathsf{attr}_i^k(X)$ contains all nodes from which player i can force any play to visit the set X.

With the definition, for reachability games we can decide the winning region Win_0 by computing $\mathsf{Attr}_0(V_{goal})$; for safety games derive Win_1 by computing $\mathsf{Attr}_1(V_{risk})$. Without loss of generality, here we assume that no deadlock occurs. Otherwise,

- for reachability games, we add all player-1 vertices without outgoing edges as goal states.
- for safety games, we add all player-0 vertices without outgoing edges as risk states.

[Example: computing the attractor] For the reachability game defined previously in Section 2.1.1.5, $\mathsf{attr}_0^1(\{v_1\}) = \{v_0, v_1\}$ and $\mathsf{Attr}_0(\{v_1\}) = \mathsf{attr}_0^2(\{v_1\}) = \{v_0, v_1, v_2\}$.

For Büchi games, the method of finding the winning region needs to first computing the *recurrence region*. Given a set of goal states V_{goal}, the recurrence region is defined as the set of states $V_{recur} \subseteq V_{goal}$ where there exists strategy to ensure that vertices in V_{recur} can be reached repeatedly. Computing the recurrence region applies similar concepts as the attractor; details are omitted here. When the recurrence region is computed, then by computing $\mathsf{Attr}_0(V_{recur})$ we derive Win_0.

2.1.2.2 Reduction

In this section, we outline reduction techniques which perform transformation between games having different winning conditions. Precisely, given two games $G = ((V, E), Win)$ and $G' = ((V', E'), Win')$, a *game reduction* is a function $\gamma : V \to V'$ such that player 0 wins G from vertex $v \in V$ iff player 0 wins G' from vertex $\gamma(v) \in V'$.

- **(Muller game)** The algorithm performs reductions from Muller games to parity games using the latest appearance record (LAR). Intuitively, LAR records the permutation of all vertices in a Muller game: when a move (v_1, v_2) is taken in the Muller game, in the reduced parity game the permutation (between two corresponding vertices) changes by moving v_2 to the front of permutation and right-shifting the original permutation (which starts with v_1). The coloring is based on evaluating the prefix of the permutation from v_2 to the first position: if the set of vertices from this prefix is an element of $\mathcal{F}$ then it is labeled as even.
- **(Streett game)** The algorithm performs reductions from Streett games to parity games using the index appearance record (IAR). Intuitively, given the specification $(E_1, F_1), \ldots, (E_k, F_k)$ in a Streett game, IAR also updates the permutation when executing a move, and keeps track of the visited E-sets and F-sets in the reduced game.

The arena of the reduced parity game can be interpreted as the strategy automaton of the Muller or the Streett game. Notice that for Muller or Streett games, in general the exponential blow-up of the memory size caused by the permutation of vertices is unavoidable.

2.1.2.3 Parity game solving using strategy improvement

Parity games, as shown in the previous section, offer a unified foundation for game solving. We summarize properties of parity games.

- Parity games are determined, and positional strategy suffices [EJ91].
- Parity game can be solved in $\texttt{NP} \cap \texttt{coNP}$.

Currently, techniques of solving parity games are based on a methodology called *strategy improvement* [VJ00]. Under this setting, each strategy is evaluated (under a certain criterion) such that the set of all strategies forms a partial order. Starting from an initial (arbitrary) strategy, player 0 and player 1 perform alternating iterations to improve their strategy evaluation by modifying their strategies; the process ends until no improvement can be made on both sides.

The algorithm proposed in [VJ00] is a *global* method: it decides for every node v, whether v belongs to Win_0. Contrarily, *local* methods are methods deciding whether a given node $v \in V_0 \uplus V_1$ belongs to Win_0. Local methods (e.g., algorithms in [FL10]) can be used in many applications where deciding whether an initial location belongs to Win_0 is the only concern, e.g., model checking branching time logics and the satisfiability problem for modal μ-calculus. The main idea of local strategy improvement in [FL10] is to expand the game graph on-the-fly: if a subgraph is sufficient to define the winner of a given initial node, then there is no need to explore others.

2.1.2.4 Symbolic computation for attractors using BDDs

Lastly, we outline how games can be solved symbolically with *Binary Decision Diagrams* (BDD) [Bry86]. In GAVS+, game engines which are implemented symbolically[4] include winning conditions for reachability, safety, Büchi, weak-parity, and Staiger-Wagner.

- For all games (except Staiger-Wagner conditions), locations can be encoded as bit vectors using binary encoding. Therefore, a total number of $2\lceil \log_2(|V_0 \uplus V_1|)\rceil$ BDD variables is used. The factor of 2 is used for the *primed* version of the location.
- For Staiger-Wagner games, as the strategy of winning requires memory exponential to the size of the vertex (a strategy need to keep track of all visited vertices), for each location, two BDD variables are used for memory update (together with the primed version).
- The attractor computation (similarly the recurrence region for Büchi games and coloring in weak-parity games) can be computed efficiently using BDD. During the computation, the list of strategy edges chosen (for player-0 in reachability, Büchi, weak-parity games) should also be recorded. Further details are omitted.

2.1.2.5 Symbolic techniques for finding strategies using SAT

To compute strategies for reachability games, except using BDD-based approach, Madhusudan, Nam, and Alur also proposed a *bounded-witness algorithm* [AMN05] for solv-

[4]The implementation of GAVS+ is based on JDD [jdd], a Java-based BDD package.

ing reachability games using SAT, a natural generalization of bounded model checking [CBRZ01]. Notice that in their formulation, the game starts in the initial state but in every step, the system and the environment pick a move simultaneously and the state evolves according to this choice [AMN05]. Without mentioning details, we comment that their formulation can be translated to the standard notation similar to ours by creating a bipartite arena[5].

Based on their experiments, the witness algorithm is not as efficient as the BDD-based approach. Nevertheless, with some modifications, we find it useful and create a modification for solving distributed games. The witness algorithm will be discussed in Chapter 6 together with distributed games.

2.2 Two-player Games over Pushdown Game Graphs

In this section, we consider two-player, turn-based games played over pushdown systems. A pushdown automaton extends a nondeterministic automaton where a (last-in-first-out) stack is equipped. Transitions may also define update rules for the stack. A pushdown system naturally models a program where recursion is used. *Pushdown games* are extensions combining games and pushdown systems. One (potential) application for pushdown games is automatic repair of recursive programs, a generalization of Boolean program repair [JGB05].

To our knowledge, there are two representations to describe pushdown games, namely (i) representation based on rewrite rules and (ii) representation based on recursive automata. These two formulations are equivalent concerning expressiveness. In the thesis we use (i), i,e., we follow the formulation in the work of Cachat [Cac02, Cac03b].

2.2.1 Definition

2.2.1.1 Arena (APDS), play, and game

An *alternating pushdown system* (APDS) is a tuple $\mathcal{A} = (V_0 \uplus V_1, \Gamma, \Delta)$, where (i) $V_0 \uplus V_1$ is the partition of locations for player 0 and 1, (ii) Γ is the set of *stack alphabets*, and (iii) $\Delta \subseteq (V_0 \uplus V_1 \times \Gamma) \times (V_0 \uplus V_1 \times \Gamma^*)$ is the set of *rewrite rules*. A *configuration* of an APDS is represented as vw, where $v \in V_0 \uplus V_1$ is the current location and $w \in \Gamma^*$ is the current stack content. The arena in an APDS is the set of all configurations together with transitions between configurations. We use $vw \xrightarrow{\delta} v'w'$ to represent the change of configuration from vw to $v'w'$ using the rewrite rule δ. An infinite *play* starting from a given configuration v_0w_0, where $v_0 \in V_0 \uplus V_1$ and $w_0 \in \Gamma^*$ is simply a maximal path $\pi = (v_0w_0)(v_1w_1)\dots$ in G such that $\forall i$ where $v_i \in V_0 \uplus V_1 \ \wedge \ w_i \in \Gamma^*$, $\exists \delta \in \Delta : v_iw_i \xrightarrow{\delta} v_{i+1}w_{i+1}$. For the definition of finite plays, it follows analogously. In a play, we assume that player i determines the move $v_kw_k \xrightarrow{\delta} v_{k+1}w_{k+1}$ if $v_k \in V_i$, $i \in \{0, 1\}$.

[5]Also, the formulation of reachability game in [AMN05] matches the case of *sure-winning* in concurrent reachability games [DAHK07] (see Section 2.3 for complete definition). The algorithm of computing sure-winning strategies uses precisely the concept of attractor.

In this thesis *normalization* over rewrite rules is used, i.e., each rule $\delta \in \Delta$ can only be one of the following three types (we use the symbol $\hookrightarrow$ to represent the update of a rule).

- (Decrease stack size by 1) $\delta := \langle v, \gamma \rangle \hookrightarrow \langle v', \varepsilon \rangle$
- (Keep stack size the same) $\delta := \langle v, \gamma \rangle \hookrightarrow \langle v', \gamma' \rangle$
- (Increase stack size by 1) $\delta := \langle v, \gamma \rangle \hookrightarrow \langle v', \gamma'\gamma'' \rangle$

where $v, v' \in V_0 \uplus V_1$, $\gamma, \gamma', \gamma'' \in \Gamma$.

2.2.1.2 Winning conditions over APDS

Here we outline three winning conditions for player 0, namely reachability, Büchi, and parity, following the constraints mentioned in [Cac02, Cac03b] for implemention.

- **(Reachability)** Reachability condition is defined via the set of goal configurations. As the size of the pushdown game graph (the number of configurations) is infinite, to compute the attractor of reachability conditions, in [Cac02] the set of goal configurations is further constrained to be a *regular* set, i.e., it can be represented by a finite automaton.
- **(Büchi)** Following the constraint for reachability conditions, for Büchi conditions the set of goal configurations should be regular. Furthermore, in our implementation we use the set of the goal configurations with *simplest form* $R = V_{goal}\Gamma^*$. It is indicated in [Cac02] that for each APDS $\mathcal{A}$ having a regular set Reg of configurations as a goal set, it can be translated to another APDS $\mathcal{A} \times A_{Reg}$ having the simplest form as the goal set, where A_{Reg} is the automaton recognizing Reg.
- **(Parity)** Similar to parity conditions over finite game graphs, given an APDS define its coloring function $c : V_0 \uplus V_1 \rightarrow C$, where C is a finite set of nonnegative integers. Given an APDS $(V_0 \uplus V_1, \Gamma, \Delta)$ with coloring function c, a configuration vw, where $v \in V_0 \uplus V_1$ and $w \in \Gamma$, has color $c(v)$.

2.2.1.3 Example

Here we give a simple example $\mathcal{A} = (V_0 \uplus V_1, \Gamma, \Delta)$, where

- $V_0 = \{S_0, S_2\}$
- $V_1 = \{S_1\}$
- $\Gamma = \{a, b, c\}$
- $\Delta = \{\langle S_0, b \rangle \hookrightarrow \langle S_1 \rangle, \langle S_0, b \rangle \hookrightarrow \langle S_1, a \rangle, \langle S_1, a \rangle \hookrightarrow \langle S_2, ab \rangle,$
  $\langle S_1, a \rangle \hookrightarrow \langle S_2, a \rangle, \langle S_2, a \rangle \hookrightarrow \langle S_2, ab \rangle, \langle S_2, a \rangle \hookrightarrow \langle S_2, a \rangle, \langle S_1, b \rangle \hookrightarrow \langle S_b, b \rangle\}$.

This example will be used in Section 2.2.3 when we revisit the saturation algorithm proposed in [Cac02, Cac03b].

2.2.2 Algorithms

We summarize algorithms for pushdown games from [Cac02, Cac03b]. For reachability and Büchi games, we describe (i) algorithms computing the winning region, and (ii) how a strategy from an (accepting) initial configuration is extracted.

2.2.2.1 Reachability Games

For reachability analysis, the algorithm [Cac02] computing the winning region of a given pushdown game is a generalization of the *saturation method* [EHRS00], which is used for model-checking pushdown systems (PDS). A PDS can be viewed as an APDS where $V_1 = \emptyset$. In [EHRS00], the set of goal configurations is represented as an finite automaton, called *$\mathcal{P}$-automaton*. Notice that in a $\mathcal{P}$-automaton, for each state representing the location of the pushdown system, initially it is not the destination of any edge. Then the saturation method adds edges to the $\mathcal{P}$-automaton with the following rule[6].

> If there exists $\langle v, \gamma \rangle \hookrightarrow \langle v', \gamma' \rangle$ ($v, v' \in V_0$, $\gamma, \gamma' \in \Gamma$), and if transition (v', q) with label γ' is in the $\mathcal{P}$-automaton then add a transition (v, q) with transition label γ to the $\mathcal{P}$-automaton. (*)

The whole process ends until no edge can be further added. To check if a given configuration pw is contained in the $\mathcal{P}$-automaton (membership problem), a backward analysis from the accepting state is performed.

For reachability games, the set of all accepting configurations can also be computed analogously. Initially, the (alternating) $\mathcal{P}$-automaton represents the set of goal configurations. Then the saturation method proceeds as follows.

- For $v \in V_1$, instead of adding an edge, a *multi-edge* is added. A multi-edge is an edge having one source vertex but multiple destination vertices. Adding a multi-edge represents all possibilities induced by player 1: for all rules having the precondition $\langle v, \gamma \rangle$, it is the choice of player 1 to freely decide which rule to use. Thus the multi-edge can be viewed as an $\forall$-transition.
- For $v \in V_0$, adding an edge follows the same saturation method for PDS.

To generate winning strategies (i.e., how to apply rewrite rules), two methods are proposed in [Cac02].

- **(Positional min-rank strategy)** To apply positional min-rank strategy, during the construction of the $\mathcal{P}$-automaton, it is required to augment costs over each added edge. For example, in (*), if the cost of (v', γ', q) is i, then the cost of (v, γ, q) equals $1 + i$. When player 0 needs to decide the applied rewrite rule over configuration vw, he performs a linear search over the P-automaton augmented with costs from vertex v sequentially with alphabets in w. For the path which has the least sum of cost from initial state to final state, the algorithm suggests the rewrite rule based on the first edge of the path.
- **(Pushdown strategy)** A pushdown strategy refers to the strategy where a push-

[6]Here we only list the case where the stack size remains the same after the rule is applied; for other two cases similar arguments follow.

down stack is equipped for player 0 during the play to select the rewrite rule. Given an initial configuration, generate the initial stack content during the test of membership; as the test of membership is performed backwards, the stack content is filled with information over the selected edges of the $\mathcal{P}$-automaton. During a play, for player 0, select the rewrite rule based on the stack content. Also, update the stack based on the rule selected by player 0 or by player 1.

(Remark over two strategies [Cac02]) Positional strategies require linear execution time in each step, and for strategies with pushdown memory each step can be executed in constant time.

2.2.2.2 Büchi Games

We give an outline how to compute winning regions for Büchi games; for details we refer readers to [Cac02]. For Büchi games, the requirement of recurrence condition (visiting locations repeatedly) is implemented by multiple copies of the set of locations in the $\mathcal{P}$-automaton; recall that in reachability games only one copy is required. During the i^{th} recurrence round, for each location $v \in V_0 \uplus V_1$, the algorithm introduces vertex v_i in the $\mathcal{P}$-automaton (representing the i^{th} copy), add an ε-transition (v_i, v_{i-1}) to the $i - 1^{th}$ copy, and apply the saturation algorithm. However, to achieve termination, at the end of the round, the algorithm proceeds with two operations.

- For each transition going to vertices representing the $i - 1^{th}$ copy, modify the transition to redirect it to the i^{th} copy; this is defined in [Cac02] as the *projection* operator.
- Check whether the i^{th} copy and the $i - 1^{th}$ copy are having same outgoing transitions, where the concept of "sameness" is defined in [Cac02] via the *contraction* operator. If the answer is positive, then terminate and report the set of winning configurations.

To create the winning strategy, it follows similar approaches for reachability games.

2.2.2.3 Parity Games

We outline two methods for solving parity games.

1. **(Reduction method [Cac03b])** This method performs a translation from a pushdown parity game to a finite game with parity condition. The translation ensures that running the parity game over the finite game graph simulates the execution in the corresponding pushdown game. As a finite game graph is unable to store the content of the stack (which is infinite), it applies the concept of *subgame* - when a push operation occurs in the APDS, in the reduced finite game player 0 needs to make a guess: the guess answers "after popping the stack content, what is the maximum color encountered between the push and the pop process". Player 1 then can challenge or agree with the guess of player 0.
    - If player 1 challenges, then the game enters a subgame while the goal is to check (i) the claim by player 0 when the pop-of-stack occurs and (ii) if the content is never popped, then player 0 again needs to win the parity con-

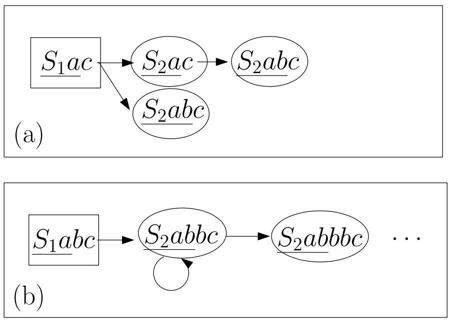


Figure 2.3: Run graphs over initial conditions S_1ac (a) and S_1abc (b).

dition. Notice that here the algorithm applies the concept of *summary*, i.e., finite representation which summarizes important information of the visited configuration. In this method, the summary is a subset of locations together with their colors: when player 1 challenges, the claim made by player 0 is stored as a summary and the play continues.

- If player 1 accepts, then the game visits a node having the guessed color, and the play continues.

The drawback of this method is that creating the reduction generates an immediate exponential blowup on the resulting parity game; in our preliminary evaluation, even for simple pushdown games with 3 locations, 3 colors, 2 alphabets (including a bottom-of-stack alphabet) and 8 rules, the reduction algorithm creates a finite pushdown game having roughly 350 vertices.

2. **(Saturation method [HO09])** Here we omit technical details, but mention that the saturation method in [HO09] further extends the algorithm in [Cac02] used for solving Büchi games. By defining the winning region of a pushdown parity game using modal μ-calculus [Wal96], the algorithm iteratively performs expansions (for least fixpoints) and contractions (for greatest fixpoints). Termination is achieved using the projection operator (the concept is described previously in Section 2.2.2.2); when the algorithm stops, the winning region is precisely recognized. Using saturation methods can avoid an immediate explosion, compared to the first method.

2.2.3 Cachat's Symbolic Algorithm [Cac03a, Cac02] Revisited

In this section, we briefly describe additional remarks over Cachat's original saturation method in order to construct the $\mathcal{P}$-automaton correctly using examples, as we find algorithm in [Cac03a, Cac02] lack of details.

We consider the APDS described in Section 2.2.1.3. Given the set of goal configurations as $\{S_2abc\}$, we consider two cases with two initial configurations.

1. If the initial configuration be S_1ac, then player 0 is **able** to win the game, as shown in Figure 2.3a.

2. If the initial configuration be S_1abc, then player 0 is **unable** to win the game, as shown in Figure 2.3b.

Based on the descriptions, given APDS $\mathcal{A}$ with the set of goal configurations as $\{S_2abc\}$, detailed construction steps for the $\mathcal{P}$-automaton are shown in Figure 2.4. Edges having more than one arrows are *multi-edges*, e.g., the edge $S_1 \rightarrow \{1, 2\}$ in Figure 2.4c. The saturation method terminates at step (d).

In [Cac03a, Cac02], the correctness claim for the saturation algorithm is based on the definition of attractor, as shown in Figure 2.5. The definition is correct by intuition.

In Figure 2.5, observe the definition of $Attr_0^{i+1}$. For a player 1 configuration pw to be added to the attractor, it should be the case that for all possible rewrite rules γ applicable on p, after player 1 applies γ on pw and create new configuration qv ($pw \hookrightarrow_\gamma qv$), qv should be in $Attr_0^i$.

Back to the example, we may compute the attractor (based on the definition in the proof) as follows:

- $Attr_0^0 = \{S_2abc\}$ (goal configuration only), which is similar to Figure 2.4a.
- $Attr_0^1 = \{S_2abc, S_2ac\}$, which is similar to Figure 2.4b.
- Now consider configuration S_1ac. $Attr_0^2$ should contain S_1ac:
  1. For the rewrite rule $\langle S_1, a\rangle \hookrightarrow \langle S_2, ab\rangle$, the result after rewriting, i.e., S_2abc is in $Attr_0^1$.
  2. For the rewrite rule $\langle S_1, a\rangle \hookrightarrow \langle S_2, a\rangle$, the result after rewriting, i.e., S_2ac is in $Attr_0^1$.

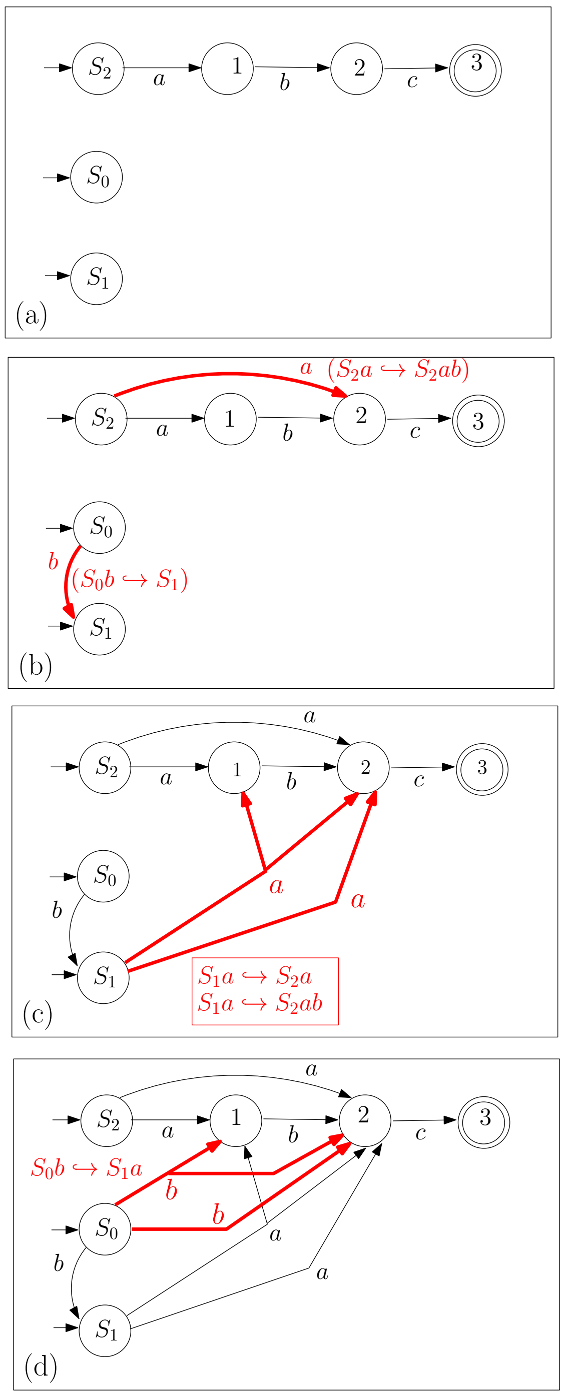
Therefore, for step (c), we should construct *two* multi-edges $(S_1, \{2, 1\})$ and $(S_1, \{2, 2\})$, as shown in Figure 2.4[7]

When the saturation method terminates, the next step is to use **membership algorithm** to test whether a given initial configuration is within the language of the $\mathcal{P}$-automaton. The snapshot of the membership algorithm is in Figure 2.6. Following the construction, it can be observed that now the membership algorithm is able to differentiate between two initial configurations S_1ac (accepting) S_1abc (non-accepting).

2.3 Games of Concurrency

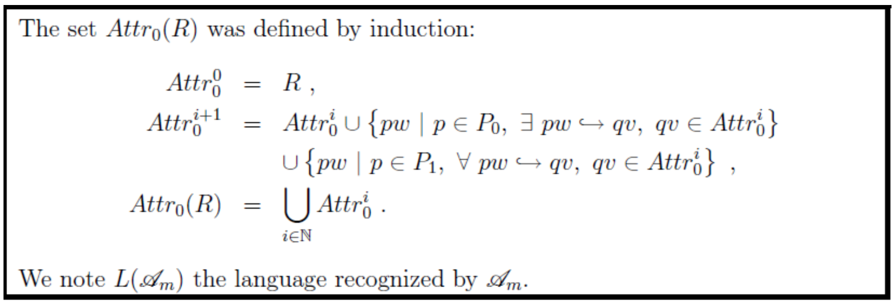
In previous sections, games are *turn-based*, i.e., a location in an arena is either played by player 0 or played by player 1, but not both. In this section, we consider situations where both players make their choices simultaneously, and the combined choice decides the next location. Games of this type are called *concurrent games*. Here we only focus on generating strategies for reachability conditions. Definitions and algorithms in this section are summarized from [DAHK07].

[7]From this example, we conclude an implicit algorithmic detail in Algorithm 1 of [Cac02]: we have to apply the rule of player-1 to *any possible combination of valid rule choices* for each successor. Thanks to Dr. Michael Luttenberger for the discussion and confirmation.



(a)
(b)
a $(S_2a \hookrightarrow S_2ab)$
b $(S_0b \hookrightarrow S_1)$
(c)
$S_1a \hookrightarrow S_2a$
$S_1a \hookrightarrow S_2ab$
(d)
$S_0b \hookrightarrow S_1a$

Figure 2.4: Detailed step on computing the $\mathcal{P}$-automaton using Cachat's algorithm.

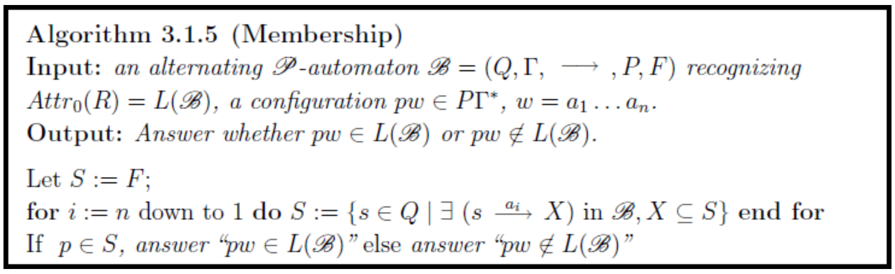


The set $Attr_0(R)$ was defined by induction:

$$\begin{aligned} Attr_0^0 &= R\ , \\ Attr_0^{i+1} &= Attr_0^i \cup \{pw \mid p \in P_0,\ \exists\ pw \hookrightarrow qv,\ qv \in Attr_0^i\} \\ &\quad \cup \{pw \mid p \in P_1,\ \forall\ pw \hookrightarrow qv,\ qv \in Attr_0^i\}\ , \\ Attr_0(R) &= \bigcup_{i \in \mathbb{N}} Attr_0^i\ . \end{aligned}$$

We note $L(\mathscr{A}_m)$ the language recognized by $\mathscr{A}_m$.

Figure 2.5: Snapshot for the definition of attractor in the ICALP paper [Cac02].



Algorithm 3.1.5 (Membership)
Input: *an alternating $\mathscr{P}$-automaton $\mathscr{B} = (Q, \Gamma, \longrightarrow, P, F)$ recognizing $Attr_0(R) = L(\mathscr{B})$, a configuration $pw \in P\Gamma^*$, $w = a_1 \ldots a_n$.*
Output: *Answer whether $pw \in L(\mathscr{B})$ or $pw \notin L(\mathscr{B})$.*

```
Let $S := F$;
for $i := n$ down to 1 do $S := \{s \in Q \mid \exists\ (s \xrightarrow{a_i} X)$ in $\mathscr{B}, X \subseteq S\}$ end for
If $p \in S$, answer "$pw \in L(\mathscr{B})$" else answer "$pw \notin L(\mathscr{B})$"
```

Figure 2.6: Snapshot of the membership algorithm in the ICALP paper [Cac02].

2.3.1 Definition

2.3.1.1 Arena, play, and game

A *concurrent arena* is a tuple $A = (V, \delta, \Sigma_0, \Sigma_1)$, where V is the set of *locations*, $\Sigma_i, i \in \{0, 1\}$ is the set of *action symbols* for player i, and $\delta : V \times \Sigma_0 \times \Sigma_1 \to V$ is the *transition function*[8]. A *play* starting from node v_0 is simply a maximal path $\pi = v_0 v_1 \ldots$ in A, where from location v_i, the *move* (v_i, v_{k+1}) is decided by $\delta(v_k, \sigma_0, \sigma_1)$ - player 0 chooses $\sigma_0 \in \Sigma_0$ and independently player 1 chooses $\sigma_1 \in \Sigma_1$. Given concurrent arena A, a concurrent reachability game is created by defining the set of *goal locations* $V_{goal} \subseteq V$.

2.3.1.2 Winning locations

Given a concurrent reachability game (A, V_{goal}), in [DAHK07] the author defines three types of winning locations for player 0.

- **(Sure winning)** A location v is *sure winning* iff player 0 has a strategy to reach V_{goal} from v, regardless of the strategy made by player 1.
- **(Almost-sure winning)** A location v is *almost-sure winning* iff player 0 has a strat-

[8]Notice that compared to the definition in [DAHK07], our definition requires that for every location, actions for player 0 and player 1 are completely defined. Our definition is simpler, and for reachability conditions, we can translate the definition in [DAHK07] to ours: First, if $\sigma_0 \in \Sigma_0$ is undefined at v, then $\forall \sigma_1 \in \Sigma_1$ add transition from (v, σ_0, σ_1) to a new self-absorbing sink location. Second, if $\sigma_1 \in \Sigma_1$ is undefined at v, then add transition from (v, σ_0, σ_1), where σ_0 is defined in v, to any goal location.

egy to reach V_{goal} from v with probability equal to 1, regardless of the strategy made by player 1.

- **(Limit-sure winning)** A location v is *limit-sure winning* iff for all $\varepsilon > 0$, player 0 has a strategy to reach V_{goal} from v with probability greater than $1 - \varepsilon$, regardless of the strategy made by player 1.

2.3.1.3 Strategy

For concurrent reachability games, randomized strategies (strategies with probability) are more powerful than deterministic strategies, contrary to the situation in turn-based games. We explain this idea using the game specified in Figure 2.7, where player 0 and 1 are playing rock-paper-scissors. Assume that the game starts with the initial position of draw (S_{draw}), a play proceeds and two players make their choices simultaneously. It can be observed that if player 0 randomly chooses his move, then the probability of endlessly staying in S_{draw} is 0. In other words, starting with S_{draw}, player 0 has a randomized strategy to reach $\{S_{win_0}, S_{win_1}\}$ with probability equal to 1 (i.e., almost-sure winning). Notice that all deterministic strategies are incomparable to the randomized strategy mentioned above, i.e., for each deterministic strategy of player-0, there exists a counter-strategy for player 1 to confine the play staying in S_{draw}.

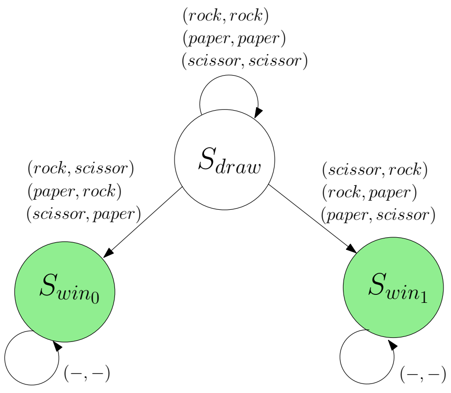


Figure 2.7: A concurrent reachability game (rock-paper-scissor), where $(-,-)$ represents all possible combinations over actions.

Formally, for current games, a strategy for player i is a function $f_i : V^+ \rightarrow \mathcal{D}(\Sigma_i)$, where V^+ represents the history of a game, and $\mathcal{D}(\Sigma_i)$ is the set of all *probability distributions* over action symbols in Σ_i. For a probability distribution $d \in \mathcal{D}$ over Σ_i, it assigns $\sigma_i \in \Sigma_i$ a value $d(\sigma_i)$, and $\sum_{\sigma_i \in \Sigma_i} d(\sigma_i) = 1$.

2.3.1.4 Example: Rock-paper-scissor game

We formulate the concurrent arena $A = (V, \delta, \Sigma_0, \Sigma_1)$ in Figure 2.7.

- $V = \{S_{draw}, S_{win_0}, S_{win_1}\}$.
- $\Sigma_0 = \Sigma_1 = \{rock, paper, scissor\}$.

- $\delta = \{\,(S_{draw}, rock, rock, S_{draw}), (S_{draw}, paper, paper, S_{draw}), (S_{draw}, scissor, scissor, S_{draw}),$ $(S_{draw}, rock, scissor, S_{win_0}), (S_{draw}, paper, rock, S_{win_0}), (S_{draw}, scissor, paper, S_{win_0}),$ $(S_{draw}, scissor, rock, S_{win_1}), (S_{draw}, rock, paper, S_{win_1}), (S_{draw}, paper, scissor, S_{win_1})\}$ $\bigcup_{\sigma_0\in\Sigma_0,\sigma_1\in\Sigma_1}\{(S_{win_0}, \sigma_0, \sigma_1, S_{win_0}), (S_{win_1}, \sigma_0, \sigma_1, S_{win_1})\}$.

We summarize the winning locations of player 0 for the concurrent reachability game $G = (A, \{S_{win_0}, S_{win_1}\})$.

- Locations S_{win_0} and S_{win_1} are both sure, almost-sure, and limit-sure winning.
- Location S_{draw} is almost-sure and limit-sure winning.

2.3.2 Algorithms

In this section, we give conceptual ideas on how to compute the winning region and derive winning strategies for sure, almost-sure, and limit-sure reachability winning conditions.

2.3.2.1 Sure reachability

For sure reachability, the computation follows the concept of *attractor* for turn-based reachability games. Given arena A and the set of locations $X \subseteq V$, the map attr_{CRG} is defined by

$$\mathsf{attr}_{CRG}(X) := X \cup \{v \in V \mid \exists\sigma_0 \in \Sigma_0 \cdot \forall\sigma_1 \in \Sigma_1 : \delta(v, \sigma_0, \sigma_1) \in X\}$$

$\mathsf{attr}_{CRG}(X)$ extends X by all those nodes from which player 0 can move to X within one step, regardless of actions taken by player 1. Then the sure winning region $\mathsf{SureWin}_0(V_{goal})$ can be computed symbolically by $\bigcup_{k\in\mathbb{N}} \mathsf{attr}^k_{CRG}(V_{goal})$.

Consider the rock-paper-scissor example in Figure 2.7. In location S_{draw}, for all actions of player 0 (e.g., $rock$), there exists a counter-action for player 1 (e.g., $rock$) to stop reaching $\{S_{win_0}, S_{win_1}\}$. Therefore, S_{draw} is not sure winning.

2.3.2.2 Almost-sure reachability

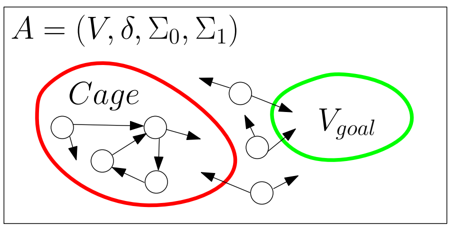


Figure 2.8: The concept of cage in computing almost-sure reachability set.

Here we use Figure 2.8 to help understanding the idea of computing the almost-sure winning region.

- Initially, consider the *cage* created by player 1, defined as the set of vertices $C_0 \subseteq V \setminus V_{goal}$ (vertices *not-safe-escape*) where player 1 can (i) perform restrictions such that for all possible plays starting from $v \in C_0$, the set of visited locations are within C_0 and (ii) ensure to never reach V_{goal}. To reach the goal state with probability equal to 1, player 0 must avoid entering C_0 in all costs. For this, the algorithm generates (i) a restricted vertex set $U_0 \subseteq V \setminus C$ where player 0 can constrain the play of not entering C_0 and (ii) a restricted action set $\Sigma_{0_0} \subseteq \Sigma_0$ that player 0 can use.
- As player 0 now has fewer choices on actions, player 1 may again create a new cage $C_1 \subseteq U_0 \setminus V_{goal}$ to constrain the play. After player 1 creates the new cage, player 0 must respond and create U_1 and Σ_{0_1} respectively. The process continues until step k, where player 1 is unable to constrain $U_k = U_{k-1} \subseteq U_{k-2} \subseteq \ldots \subseteq U_0$. Then U_k is the set of almost-sure winning locations. Concerning the winning strategy, player 0 should play with actions in Σ_{0_k} uniformly at random.

Consider the rock-paper-scissor example in Figure 2.7. As player 1 is unable to constrain the play in $\{S_{draw}\}$ (e.g., for player 1's action $scissor$, there exists action $rock$ of player 0 that enables to escape from $\{S_{draw}\}$), $C_0 = \emptyset$. Therefore, player 0 can win by playing at S_{draw} with actions in $\{rock, paper, scissor\}$ uniformly at random.

2.3.2.3 Limit-sure reachability

Back to the algorithm of computing winning regions for almost-sure reachability, at the i^{th} iteration, C_i is the cage where player 0 should escape and U_i is the region where player 0 should stay. In almost-sure winning, player 0 does not risk to escape from leaving U_i. For limit-sure reachability, it allows to risk from retreating U_i.

Instead of computing C_i (not-safe-escape locations) at each round in the case of almost-sure, the limit-sure algorithm now computes C'_i, the set of (not-limit-escape locations) for each iteration. To detect whether a location v is limit-escape, given two sets $U \subseteq V$, $C \subseteq U$ of locations, a subroutine computes two sequences of sets of actions $\mathcal{A}_0, \mathcal{A}_1, \ldots$ (each is a subset of Σ_0) and $\mathcal{B}_0, \mathcal{B}_1, \ldots$ (each is a subset of Σ_1) until step i, where $\mathcal{A}_{i-1} = \mathcal{A}_i$ and $\mathcal{B}_{i-1} = \mathcal{B}_i$. Here we explain the algorithm using an imaginary situation where saturation is reached under three iterations, i.e., $A_1 = A_2$ and $B_1 = B_2$. The location is limit-escape if $B_1 = B_2 = \Sigma_1$.

1. Originally, $\mathcal{A}_0$ is as the subset of actions in Σ_0 where player 0 can use them and stay safe within U, regardless of actions by player 1. $\mathcal{B}_0$ is the subset of actions in Σ_1 where there exists an action in $\mathcal{A}_0$ that escapes from C with non-zero one-round probability and with zero risk of capture (i.e., outside U).
2. Compute $\mathcal{A}_1$, which is a subset of actions in Σ_0 that either
   - an action $\sigma_0 \in \Sigma_0$ is also in $\mathcal{A}_0$, or
   - an action $\sigma_0 \in \Sigma_0$ incurs risk (i.e., outside U) by combining actions with $\mathcal{B}_0$.
3. Compute $\mathcal{B}_1$, which is a subset of actions in Σ_1 where there exists an action in $\mathcal{A}_1$ that escapes from C with positive probability.
4. Continue the process until saturation; here compute $\mathcal{A}_2$ and $\mathcal{B}_2$.

To derive the strategy, given $\varepsilon > 0$, in this example we should set the probability distribution d, such that an action in $\mathcal{A}_1 \setminus \mathcal{A}_0$ is with probability ε, and an action in $\mathcal{A}_0$ is with probability $\frac{1-|\mathcal{A}_1 \setminus \mathcal{A}_0|\varepsilon}{|\mathcal{A}_0|}$. Notice that for action $\sigma \in \mathcal{A}_1 \setminus \mathcal{A}_0$ (action to escape but suffer from risk) and $\sigma' \in \mathcal{A}_0$ (action to stay safe), $\lim_{\varepsilon \to 0} \frac{d(\sigma)}{d(\sigma')} = 0$. We have the following observation.

- For actions in $\mathcal{A}_0$, it does not run the risk.
- For action $\sigma \in \mathcal{A}_1 \setminus \mathcal{A}_0$, it jumps outside C, while it incurs risk with maximum probability $d(\sigma) = \varepsilon$.
- Therefore, when $\varepsilon \to 0$, for the randomized strategy, the chance of risk approaches 0, and the chance of staying safe approaches 1. However, we can observe that if a location v is limit-sure but not almost-sure winning, the expected time to successfully escape (reach the goal) can be unbounded.

2.4 Games of Imperfect / Incomplete Information

In this section, we discuss an extension for two-player, turn-based games where *imperfect information* is imposed on player 0. Intuitively, imperfect information refers to the phenomenon that player 0 should make decision with the constraint that he is unable to precisely know his position: instead of position, an *observation* is assigned to player 0. A motivating example for games of imperfect information can be found when synthesizing a controller connected to a digital sensor which reads an analog signal. As no concrete analog value is offered to the controller, the value obtained from the digital sensor (with finite precision) can be viewed as an observation.

We summarize the fixed-point theory for solving games of imperfect information proposed by De Wulf, Doyen, and Raskin [DWDR06], where later in Chapter 7 we utilize this theory and propose an algorithm for distributed games. To connect the result with distributed games, we adapt a different formulation than the one used in [DWDR06].

2.4.1 Definition

A *game of imperfect information* [Rei84, DWDR06] extends from a two-player, turn-based game graph $A = (V_0 \uplus V_1, E)$, where player 0 is unaware of his position with absolute precision. The imprecision is defined by an *observation set* (Obs, γ) where $\gamma : \mathsf{Obs} \to 2^{V_0}$ such that $\forall v \in V_0 \cdot \exists \mathsf{obs} \in \mathsf{Obs} : v \in \gamma(\mathsf{obs})$ (Obs is a finite set of *identifiers*). During a play, when reaching a vertex $v \in V_0$, an arbitrary observation accompanied with v will be assigned to player 0; he is only aware of the observation but not the location. As the location is not known with precision, the successors are imprecise as well. Thus edges for player 0 are labeled with elements in a set Σ of *actions*. A strategy for player 0 should be *observation-based*: it means that the strategy is a function $f : \mathsf{Obs}^+ \to \Sigma$ from the history of observations to actions. For all player 1's edges, a unique label u is used.

2.4.1.1 Example

We give a simple example to explain the concept. For Figure 2.9, consider the corresponding arena of imperfect information $A = ((V_0 \uplus V_1, E), (Obs, \gamma), \Sigma)$.

- $V_0 = \{v_0, v_2, v_4, v_{risk}\}$, $V_1 = \{v_1, v_3, v_{freeze}, v_{burn}\}$.
- $\Sigma = \{heat, cool\}$.
- $E = \{(v_0, heat, v_1), (v_0, cool, v_{freeze}), (v_1, u, v_0), (v_1, u, v_2), (v_2, cool, v_1), (v_2, heat, v_3), (v_3, u, v_2), (v_3, u, v_4), (v_4, cool, v_3), (v_4, heat, v_{burn}), (v_{burn}, u, v_{risk}), (v_{freeze}, u, v_{risk})\}$.
- Obs $= \{cold, hot, danger\}$.
- $\gamma(cold) = \{v_0, v_2\}, \gamma(hot) = \{v_2, v_4\}, \gamma(danger) = \{v_{risk}\}$.

Assume that a game starts with the initial location v_2. Based on the received observation (in Figure 2.9 a value within the set $\{\mathbf{cold}, \mathbf{hot}\}$; remember the application concerning the reading of a digital sensor), the controller should decide either to heat up ($heat$) or to cool down ($cool$) the system. The goal of the game is to create an observation-based strategy such that the system never reaches the risk state v_{risk}.

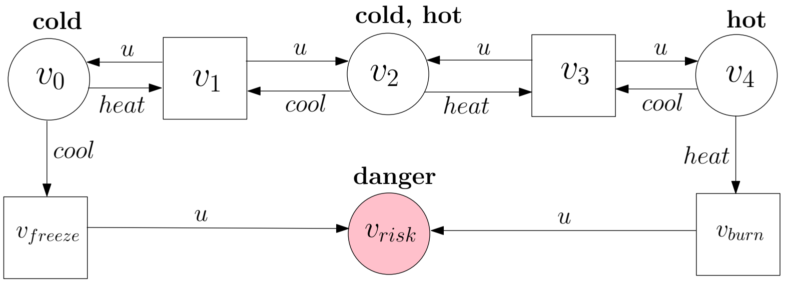


Figure 2.9: A game of imperfect information for temperature control.

2.4.2 Algorithms

Here we reuse the formulation in [DWDR06] to form the lattice of antichains of set of states, where a state corresponds to the location of a distributed game [DWDR06].

Let S be set of states. Let $q, q' \in 2^{2^S}$, and define $q \sqsubseteq q'$ iff $\forall s \in q : \exists s' \in q' : s \subseteq s'$. A set $s \subseteq S$ is *dominated* in q iff $\exists s' \in q : s \subset s'$, and define the set of dominated elements of q as $Dom(q)$. Lastly, donote $\lceil q \rceil$ to be $q \setminus Dom(q)$. $\langle L, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top \rangle$ forms a complete lattice [DWDR06], where

- L as the set $\{\lceil q \rceil \mid q \in 2^{2^S}\}$.
- For $Q \subseteq L$, $\bigsqcap Q = \lceil \{\bigcap_{q \in Q} s_q \mid s_q \in q\} \rceil$ is the greatest lower bound for Q.
- For $Q \subseteq L$, $\bigsqcup Q = \lceil \{s \mid \exists q \in Q : s \in q\} \rceil$ is the least upper bound for Q.
- $\bot = \emptyset$, $\top = \{S\}$.

Without mentioning further details, we summarize how to use the operator CPre [DWDR06] over an antichain of set of states q to derive the set of Controllable Predecessors; by iterating backwards from the set of all locations until saturation, an observation-based strategy can be established. It is not difficult to observe that finding strategies for safety games amounts to finding strategies for continuous control: In a safety game, if a risk location is in V_0, we can remove all of its outgoing edges; is a risk location is in V_1, we can connect it to a new vertex $v' \in V_0$ where v' has no outgoing edges.

Define $Enabled(\sigma)$ to be the set of locations where an edge labeled σ is possible as an outgoing edge, and $Post_\sigma(S)$ (similarly $Post_e(S)$) be the set of locations by taking the edges labeled by σ (with transition e) from a set of locations S. Assume that the game graph is bipartite. Then, $\mathsf{CPre}(q) := \lceil\{s \subseteq V_0 \mid \exists\sigma \in \Sigma \cdot \forall \mathsf{obs} \in \mathsf{Obs} \cdot \exists s' \in q : s \subseteq Enabled(\sigma) \wedge \bigcup_{e\in E_1} Post_e(Post_\sigma(s)) \cap \gamma(\mathsf{obs}) \subseteq s')\}\rceil$. Intuitively, CPre computes a set of player locations, such that by playing a common move σ, for all successor locations after the environment moves, player can decide the set of locations it belongs using the observation.

[Example: computing the observation-based strategy using the operator CPre]

For the game described in Section 2.4.1.1, we show intermediate steps when computing the fix-point for the set of controllable predecessors $\mathsf{CPre}^*(V_0 \uplus V_1)$.

- $S_1 = \mathsf{CPre}(\{V_0 \uplus V_1\}) = \{\{v_0, v_2, v_4\}_{heat}\}$; during the computation we use $heat$ for the term σ in the computation of CPre.
- $S_2 = \mathsf{CPre}(S_1) = \{\{v_0, v_2\}_{heat}, \{v_2, v_4\}_{cool}\}$.
- $S_3 = \mathsf{CPre}(S_2) = S_2$, and the fixed point has reached. Here we illustrate detailed steps for the element $\{v_0, v_2\}_{heat}$.
    1. The action $\sigma = heat$ can be enabled at all locations in $\{v_0, v_2\}$.
    2. $Post_\sigma(\{v_0, v_2\}) = \{v_1, v_3\}$.
    3. $Post_{e\in E_1}(\{v_1, v_3\}) = \{v_0, v_2, v_4\}$. Now consider all observations.
        - $\bigcup_{e\in E_1} Post_e(\{v_1, v_3\}) \cap \gamma(cold) = \{v_0, v_2\}$. Let $s' = \{v_0, v_2\}$ then the condition holds.
        - $\bigcup_{e\in E_1} Post_e(\{v_1, v_3\}) \cap \gamma(hot) = \{v_2, v_4\}$. Let $s' = \{v_2, v_4\}$ then the condition holds.
        - $\bigcup_{e\in E_1} Post_e(\{v_1, v_3\}) \cap \gamma(danger) = \emptyset$. Let $s' = \{v_0, v_2\}$ then the condition holds.

When $\mathsf{CPre}^*(V_0 \uplus V_1)$ is computed, if in the game, the starting location is v_2 (with two possible observations), the constructed strategy automaton can be found in Figure 2.10 (for details concerning the definition of the strategy automaton, we refer readers to Chapter 7 of the thesis or the paper [DWDR06]). The strategy automaton starts with vertex $init$. Initially at v_2, if player 0 receives the observation **cold**, he updates the strategy automaton by moving to the vertex $heat$ (using the edge $init \xrightarrow{\textbf{cold}} heat$), and uses $heat$ as the performed action.

2.5 Distributed Games

We describe distributed games using the definition of Mohalik and Walukiewicz [MW03]. In this model there are no explicit means of interaction among processes as such in-

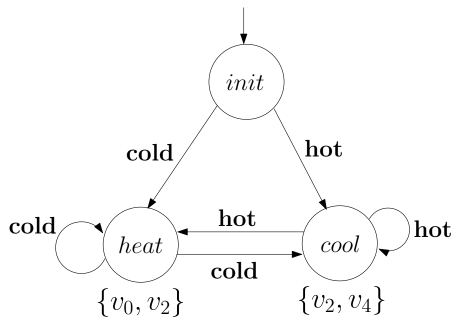


Figure 2.10: An observation-based strategy automaton for the game in Figure 2.9.

teraction must take place through the environment. Moreover, each player has only a local view of the global system state, whereas the (hostile) environment has access to the global history. Distributed games are rich enough to model various variations of distributed synthesis problems proposed in the literature [MW03].

2.5.1 Definition

2.5.1.1 Local Games

Here we perform a simple renaming over two-player, turn-based finite games defined previously to define local arena. A *local game graph* or *local arena* is a directed graph $G = (V_s \uplus V_e, E)$ whose nodes are partitioned into two classes V_s and V_e. All definitions are similar to con Section 2.1, where we use $V_0 \uplus V_1$, and here we use $V_s \uplus V_e$. Two participants, player 0 and player 1 are now called **player** (for player 0) and **environment** (for player 1). We also use E_s (E_e) to represent the set of player (environment) edges in E.

2.5.1.2 Distributed games

Definition 1. *For all $i \in \{1, \ldots, n\}$, let $G = (V_{si} \uplus V_{ei}, E_i)$ be a game graph with the restriction that it is bipartite. A* distributed game *$\mathcal{G}$ is of the form $(\mathcal{V}_s \uplus \mathcal{V}_e, \mathcal{E}, Acc)$*

- *$\mathcal{V}_e = V_{e1} \times \ldots \times V_{en}$ is the set of environment vertices.*
- *$\mathcal{V}_s = (V_{s1} \uplus V_{e1}) \times \ldots \times (V_{sn} \uplus V_{en}) \setminus \mathcal{V}_e$ is the set of player vertices.*
- *Let $(x_1, \ldots, x_n), (x'_1, \ldots, x'_n) \in \mathcal{V}_s \uplus \mathcal{V}_e$, then $\mathcal{E}$ satisfies:*
  - *If $(x_1, \ldots, x_n) \in \mathcal{V}_s$, $((x_1, \ldots, x_n), (x'_1, \ldots, x'_n)) \in \mathcal{E}$ if and only if $\forall i.(x_i \in V_{si} \rightarrow (x_i, x'_i) \in E_i) \wedge \forall j.\ (x_j \in V_{1_j} \rightarrow x_j = x'_j)$.*
  - *For $(x_1, \ldots, x_n) \in \mathcal{V}_e$, if $((x_1, \ldots, x_n), (x'_1, \ldots, x'_n)) \in \mathcal{E}$, then for every x_i, either $x_i = x'_i$ or $x'_i \in V_{si}$, and moreover $(x_1, \ldots, x_n) \neq (x'_1, \ldots, x'_n)$*[9]

[9]Another definition is to also add that all local moves of the environment should be explicitly listed in the local game; this is not required, as mentioned in the paper [MW03]. Thus for the ease of read-

- *$Acc \subseteq (\mathcal{V}_s \uplus \mathcal{V}_e)^\omega$ is the winning condition*[10].

Notice that there is an asymmetry in the definition of environment's and player's moves. In a move from player's to environment's position, all components which are player's position must change. In the move from environment's to player's, all components are environment's position but only some of them need to change.

In a distributed game $\mathcal{G} = (\mathcal{V}_s \uplus \mathcal{V}_e, \mathcal{E}, Acc)$, a play is defined analogously as defined in local games: a *play* starting from node v_0 is a maximal path $\pi = v_0 v_1 \ldots$ in $\mathcal{G}$ where player determines the *move* $(v_k, v_{k+1}) \in \mathcal{E}$ if $v_k \in \mathcal{V}_s$; the environment decides when $v_k \in \mathcal{V}_e$. For a vertex $x = (x_1, \ldots, x_n)$, we use the function $proj(x, i)$ to retrieve the i-th component x_i, and use $proj(X, i)$ to retrieve the i-th component for a set of vertices X. For simplicity, denote $\pi_{\leq j}$ as $v_0 v_1 \ldots v_j$ and use $proj(\pi_{\leq j}, i)$ for $proj(v_0, i) \ldots proj(v_j, i)$, i.e., a sequence from $\pi_{\leq j}$ by projecting over i^{th} element.

A *distributed strategy* of a distributed game for player is a tuple of functions $\xi = \langle f_1, \ldots, f_n \rangle$, where each function $f_i : (V_{si} \uplus V_{ei})^* \times V_{si} \rightarrow (V_{si} \uplus V_{ei})$ is a local strategy for G_i based on its observable history of local game i and current position of local game i. A distributed strategy is *positional* if
$f_i : V_{si} \rightarrow V_{si} \uplus V_{ei}$, i.e., the update of location depends only on the current position of local game. Contrarily, for environment a strategy is a function $f : (\mathcal{V}_s \uplus \mathcal{V}_e)^+ \rightarrow (\mathcal{V}_s \uplus \mathcal{V}_e)$ that assigns each play prefix $v_0 \ldots v_k$ a vertex v_{k+1} where $(v_k, v_{k+1}) \in \mathcal{E}$. The formulation of distributed games models the asymmetry between the environment (full observability) and the a set of local controllers (partial observability).

Definition 2. *A distributed game $\mathcal{G} = (\mathcal{V}_s \uplus \mathcal{V}_e, \mathcal{E}, Acc)$ is for player winning by a distributed strategy $\xi = \langle f_1, \ldots, f_n \rangle$ over initial states $\mathcal{V}_{ini} \in \mathcal{V}_e$, if for every play $\pi = v_0 v_1 v_2, \ldots$ where $v_0 \in \mathcal{V}_{ini}$, player wins π following his own strategy (regardless of strategies of environment), i.e.,*

- *$\pi \in Acc$.*
- *$\forall i \in \mathbb{N}_0.\ (v_i \in \mathcal{V}_s \rightarrow (\forall j \in \{1, \ldots, n\}.(proj(v_i, j) \in V_{sj} \rightarrow proj(v_{i+1}, j) = f_j(proj(\pi_{\leq i}, j)))$.*

Definition 3. *Given distributed game graph $(\mathcal{V}_s \uplus \mathcal{V}_e, \mathcal{E})$,*

- *the reachability condition is defined by $Acc = \{v_0 v_1 \ldots \in (\mathcal{V}_s \uplus \mathcal{V}_e)^\omega \mid$ $Occ(v_0 v_1 \ldots) \cap \mathcal{V}_{goal} \neq \emptyset\}$, where $\mathcal{V}_{goal}$ is the set of goal states in $\mathcal{V}_s \uplus \mathcal{V}_e$.*
- *the safety (co-reachability) condition is defined by $Acc = \{v_0 v_1 \ldots \in (\mathcal{V}_s \uplus \mathcal{V}_e)^\omega \mid$ $Occ(v_0 v_1 \ldots) \cap \mathcal{V}_{risk} = \emptyset\}$, where $\mathcal{V}_{risk}$ is the set of risk states in $\mathcal{V}_s \uplus \mathcal{V}_e$.*
- *the Büchi condition is defined by $Acc = \{v_0 v_1 \ldots \in (\mathcal{V}_s \uplus \mathcal{V}_e)^\omega \mid$ $Inf(v_0 v_1 \ldots) \cap \mathcal{V}_{goal} \neq \emptyset\}$, where $\mathcal{V}_{goal}$ is the set of goal states in $\mathcal{V}_s \uplus \mathcal{V}_e$.*

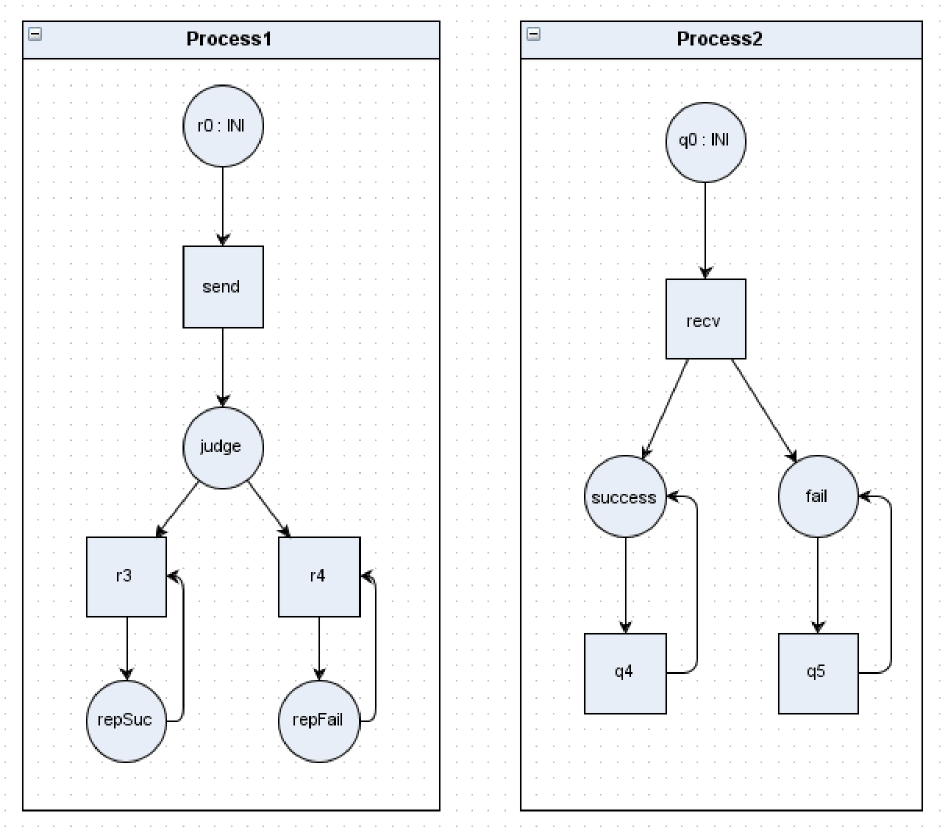


Figure 2.11: Two local games modeling unreliable network transmission.

2.5.1.3 Example: Unreliable network protocols

To indicate the applicability of distributed games, we use a simple example (modeled using the GAVS+ tool) to illustrate the communication of two processes over an unreliable network, as shown in Figure 2.11. `Process1` and `Process2` are modeled as local games with initial location `r0` and `q0`; these vertices are labeled with "`: INI`". For the distributed game, we define the set of environment moves using 6 transitions, i.e.,
$\{((\texttt{send}, \texttt{recv}), (\texttt{judge}, \texttt{success})), ((\texttt{send}, \texttt{recv}), (\texttt{judge}, \texttt{fail}))$
$((\texttt{r3}, \texttt{q4}), (\texttt{repSuc}, \texttt{success})), ((\texttt{r3}, \texttt{q5}), (\texttt{repSuc}, \texttt{fail})),$
$((\texttt{r4}, \texttt{q4}), (\texttt{repFail}, \texttt{success})), ((\texttt{r4}, \texttt{q5}), (\texttt{repFail}, \texttt{fail}))\}$.

We give an intuitive explanation over the meaning of the game with an assumption that the environment acts as an unreliable network.

1. The game starts at $(\texttt{r0}, \texttt{q0})$, meaning that two processes are preparing to communicate. For each local game, perform its only transition and move jointly to state $(\texttt{send}, \texttt{recv})$.
2. At $(\texttt{send}, \texttt{recv})$, the environment has two choices.

ing, we only construct environment moves in the distributed game and avoid drawing complicated environment edges in the local game.

[10]In this thesis we also use $\mathcal{G}$ as the identifier for the distributed game graph $(\mathcal{V}_s \uplus \mathcal{V}_e, \mathcal{E})$.

  - Use edge $((\texttt{send}, \texttt{recv}), (\texttt{judge}, \texttt{success}))$. This is interpreted that the network forwards the message successfully from `Process1` to `Process2`.
  - Use edge $((\texttt{send}, \texttt{recv}), (\texttt{judge}, \texttt{fail}))$. This is interpreted that the network discards the message; `Process2` does not receive it from `Process1`.
3. Now the move of `Process2` is deterministic, so we consider only `Process1` at location `judge`.
  - If `Process1` chooses its local edge $(\texttt{judge}, \texttt{r3})$, then it later reaches state `repSuc` (report success). This is interpreted as follows: `Process1` judges that the message is sent *successfully*.
  - If `Process1` chooses its local edge $(\texttt{judge}, \texttt{r4})$, then it later reaches state `repFail` (report failure). This is interpreted as follows: `Process1` judges that the message is sent *unsuccessfully*.

Therefore, provided the set of goal states $V_{goal} = \{(\texttt{repSuc}, \texttt{success}), (\texttt{repFail}, \texttt{fail})\}$, having a distributed strategy of the game means that `Process1` is guaranteed to have correct knowledge on whether the message is sent successfully without the ability to peer the state of `Process2`.

2.5.2 Algorithms

As the problem of finding strategies for distributed games is *undecidable* even when restricted to safety, reachability, or Büchi winning conditions [Jan07], in this thesis we focus on incomplete methods. Discussions on incomplete algorithms are postponed to Chapter 6 and Chapter 7.

2.6 Other Games Having Practical Interests

2.6.1 Games with Probability

In this section, we consider extensions when uncertainty is introduced in the game. *Markov decision processes (MDPs, $1\frac{1}{2}$-player games)* [WW89] are optimization models for decision making under stochastic environments used in economics and machine learning. Similarly, *stochastic ($2\frac{1}{2}$-player) games* [Sha53] are games consisting of controllable, uncontrollable, and probabilistic vertices. Although in GAVS+, the graph drawing framework supports general stochastic games, currently we focus on simple stochastic games (SSG) [Con93]; many complicated games can be reduced to SSGs or solved by algorithms similar to algorithms solving SSGs. For MDP and SSG, descriptions in this thesis are mainly summarized from two review papers [WW89, Con93].

2.6.1.1 Simple Stochastic Games

An arena of a *simple stochastic game* (SSG) is a tuple $A = (V_0 \uplus V_1 \uplus V_{\frac{1}{2}} \uplus \{sink_0, sink_1\}, E)$, where V_0 and V_1 are defined similarly as in two-player, turn-based games, $V_{\frac{1}{2}}$ is the set of *stochastic locations*, and $sink_0$ and $sink_1$ are two special vertices, called *0-sink* and *1-sink*. E is the set of transitions between locations. For each location $v \in V_0 \uplus V_1 \uplus V_{\frac{1}{2}}$, it has two outgoing transitions. However, $sink_0$ and $sink_1$ do not have any outgoing transitions. When a location $v \in V_{\frac{1}{2}}$ is visited, the successor is decided uniformly at random between destinations of two outgoing edges.

The problem of SSG is as follows. Given an arena A and an initial location $v_{ini} \in V_0 \uplus V_1 \uplus V_{\frac{1}{2}}$, decide whether there exists a strategy for player 0 to reach $sink_0$ with probability no less than $\frac{1}{2}$. Condon has shown that the problem is in $\texttt{NP} \cap \texttt{coNP}$ [Con92]; it is an open problem whether the problem is in $\texttt{P}$.

[Example] An example of SSG can be found in Figure 2.12, where

- $V_0 = \{v_0\}, V_1 = \{v_1\}, V_2 = \{v_2\}$.
- $E = \{(v_0, v_1), (v_0, sink_1), (v_1, sink_0), (v_1, v_2), (v_2, sink_0), (v_2, sink_1)\}$.

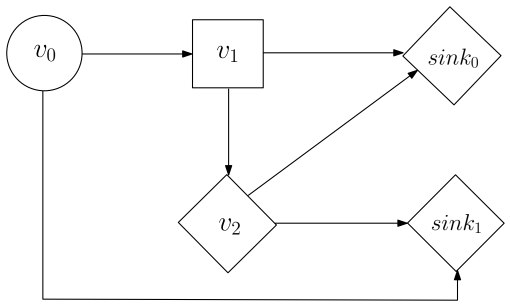


Figure 2.12: A simple stochastic game.

In general, methods of solving simple stochastic games fall within three categories, namely (i) value iteration, (ii) policy iteration, and (iii) reduction to quadratic programming with non-convex objective functions [Con93]. Here we omit the latter and only introduce (i) and (ii), as the first two methods are also applicable when solving MDPs.

- **(Value Iteration)** The method of value iteration, first proposed by Shapley [Sha53], is a method of continuously updating the value of each location (called *vector*) based on the values of its children. This method converges to the optimal value in the limit, but may take exponential time to even get constant factor of this value [Con93]. To initiate the iteration process, it is required to start with an *initial feasible vector*; due to space limit we do not define it here and refer readers to [Con93, Sha53] for details. In our implementation, the process of finding an initial feasible vector is achieved by solving a linear constraint system.
- **(Policy Iteration)** The method of policy iteration, first proposed by Hoffman and Karp [HK66], starts with an arbitrary strategy for player 0 (also for player 1). Analogous to the process in solving parity conditions for two-player, turn-based games (see Section 2.1.2.3 for the concept of strategy improvement), player 0 and player 1 then iteratively update their strategies based on the currently evaluated value until no change can be made for both players.

2.6.1.2 Markov Decision Processes

An arena of a *Markov decision process* (MDP) is a tuple $A = (V_0 \uplus V_{\frac{1}{2}}, E, \gamma)$, where V_0 is the set of *states* (player 0 locations), $V_{\frac{1}{2}}$ is the set of *actions* (stochastic locations), and $E \subseteq (V_0 \times V_{\frac{1}{2}}) \uplus (V_{\frac{1}{2}} \times \mathbb{Q} \times \mathbb{Q} \times V_0)$ is the set of *transitions*. For an edge $e = (v, p, r, v') \in E$, p is the *probability* and r is the *immediate reward*. For convenience, we denote $p(e)$, $r(e)$ to be the probability and reward associated with transition e. For transitions, an additional constraint is imposed as follows: $\forall v \in V_{\frac{1}{2}} : \sum_{(v,p,r,v') \in E} p = 1$, meaning that starting from a stochastic vertex, the sum of probabilities associated with outgoing edges equals to 1. Lastly, $\gamma \in [0, 1)$ is the *discount value*.

A strategy $f : V_0 \rightarrow V_{\frac{1}{2}}$ assigns each state an action to execute. Given the discount value γ, starting with an initial vertex, a run following the strategy is an infinite *tree* $\mathcal{T}_f$ (i.e., an unfolding of the MDP). In $\mathcal{T}_f$, a *path* $p = v_0 \xrightarrow{e_0} v_1 \xrightarrow{e_1} \ldots$, where (i) $i \texttt{ mod } 2 = 0 \rightarrow v_i \in V_0$, (ii) $i \texttt{ mod } 2 = 1 \rightarrow v_i \in V_{\frac{1}{2}}$ and (iii) $\forall i : e_i \in E$, can be assigned with the *cumulative discounted reward* $\sum_{i=0}^{\infty} \gamma^i p(e_{2i+1}) r(e_{2i+1})$. For player 0, the goal is find an optimal strategy f_{opt} to maximize the cumulative discounted reward over the sum of all paths in the corresponding tree $\mathcal{T}_{f_{opt}}$.

[Example] Consider the Markov decision process in Figure 2.13. It models a robot having two battery levels `high` and `low`, and contains actions including `waiting`, `searching`, and battery `recharging`[11]. When executing an action, the robot changes the battery level with certain probability and receives the reward. E.g., for action $search_H$, it moves to location $high$ with probability 0.4 and generates an immediate reward of value 6.

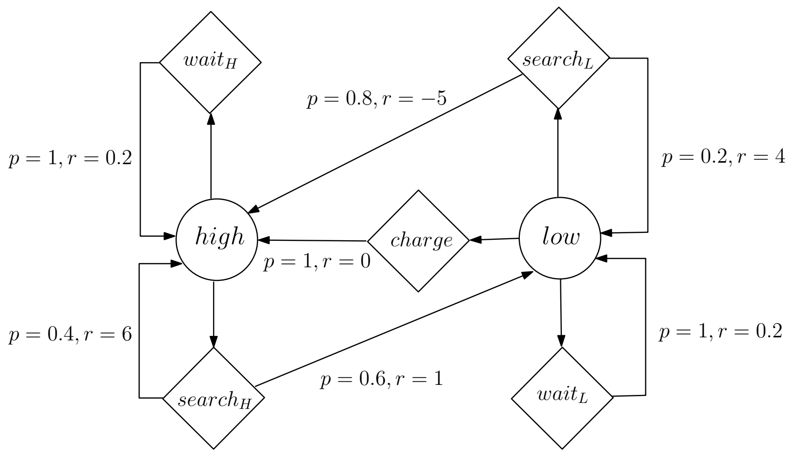


Figure 2.13: A simple Markov decision process for robot control.

For algorithms computing optimal strategies for MDPs, for infinite horizon with discount value, known methods include value iteration, policy iteration, and linear programming. For finite horizon, algorithms are mainly based on dynamic programming. For further details, we refer readers to the review paper [WW89].

[11]The example is adapted from the example in `http://www.cse.lehigh.edu/~munoz/CSE335/`.

2.6.2 Timed Games

Lastly, we indicate an important extension called *timed games* [MPS95, BCFL04], combining *timed automata* [AD94] and games. As for timed games, a mature research tool called UPPAAL-Tiga [BCD^{+}07a] is available (although the tool is not released as open-source software), in our proposed framework we do not endeavor implementing such games.

We remark on the decidability result for solving timed games, using the example of safety. In [MPS95], solving a game is similar to the formulation for games of imperfect information in [DWDR06]: the algorithm continuously computes the *controllable predecessor*. Nevertheless, to derive decidability results, the computation must guarantee convergence, i.e., a fixed point must be reached. Thanks to the property of timed automata where *quotient construction* is possible [AD94], for each quotient, its controllable predecessor is also a finite union of quotients. Further discussion of such games is omitted, and we refer interested readers to [MPS95] for the formulation, to [BCFL04] for efficient on-the-fly algorithms, and to [BCD^{+}07a] for the UPPAAL-Tiga tool.

CHAPTER 3

GAVS+: An Open Platform for the Research of Algorithmic Game Solving

Contents

3.1 GAVS: An Earlier Version

Reasoning on the properties of computer systems can often be reduced to deciding the winner of a game played on a finite graph. In this section, we present GAVS[1], an open source tool (an earlier version of GAVS+) which allows to visualize and solve some of the most common two-player games encountered in theoretical computer science, amongst others reachability, Büchi and parity games. Our focus is on two-player, turn-based games on finite graphs. During the introduction, we also explain the underlying software architecture for the tool; GAVS+ is constructed on top of the software architecture.

The importance of these games results from the reduction of different questions regarding the analysis of computer systems to two-player games played on finite graphs. For example, liveness (safety) properties can easily be paraphrased as a game play on the control-flow graph of a finite program: will the system always (never) visit a given state no matter how the user might interact with the system? The resulting games are

[1]Short for "**G**ame **A**rena **V**isualization and **S**ynthesis".

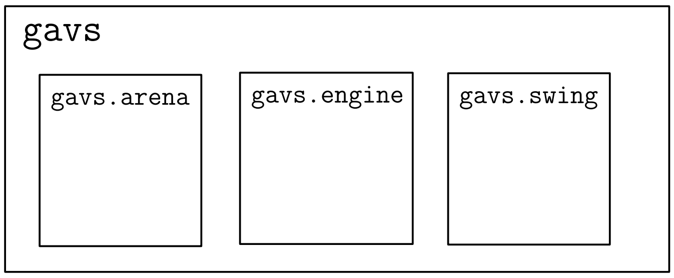


Figure 3.1: A coarse overview on software packages in GAVS.

usually called (co-)reachability if only finite program runs are considered. Similarly, one obtains (co-)Büchi games when considering infinite runs. Another well-known example is the class of parity games which correspond to the model-checking problem of μ-calculus. Advantages of the game theoretic reformulation of these analysis problems are the easier accessibility and the broadened audience.

The main goal of GAVS is to further enhance these advantages by providing educational institutions with a graphical tool for both constructing game graphs and also visualizing standard algorithms for solving them step-by-step. Still, symbolic methods, where applicable, have been used in the implementation in order to ensure scalability.

3.1.1 Software Architecture

GAVS consists of three major parts: (a) a graphical user interface (GUI), (b) synthesis engines for different winning conditions, and (c) a two-way translation function between graphical representations and internal formats acceptable by the engine. Figure 3.1 illustrates the software package (only at the coarsest level).

- The package `gavs.arena` contains data structures for internal representation of games.
- The package `gavs.swing` contains the front-end GUI and the translation function. For the front-end, we use the library JGraphX [jgr] for the construction of GUI.
- The package `gavs.engine` contains algorithms which compute strategies and winning regions for games. Different synthesis engines are implemented in GAVS as back-end. In the following, we give a brief description and categorize them based on the algorithmic methodologies applied:
  - SYMBOLIC TECHNIQUES: Algorithms of this type are implemented using JDD [jdd], a Java-based BDD package. The supported winning conditions include reachability, safety, Büchi, weak-parity, and Staiger-Wagner.
  - EXPLICIT STATE OPERATING TECHNIQUES: Algorithms of this type are implemented based on direct operations over the graph structure.
    * **Parity game** In addition to an inefficient version which enumerates all possibilities using BDDs, we have implemented the discrete strategy

improvement algorithm adapted from [VJ00]; the algorithm allows the number of nodes/vertices to exceed the number of colors in a game.

– REDUCTION TECHNIQUES: Algorithms of these games are graph transformations to other games with different types of winning conditions.
  * **Muller game** The algorithm performs reductions to parity games using the latest appearance record (LAR) [GTW02].
  * **Streett game** The algorithm performs reductions to parity games using the index appearance record (IAR) [GTW02].

We outline the underlying workflow after synthesis options are selected from the menu, offering a guideline how to extend GAVS with new solvers/game types.

For each engine, a unique action defined in `swing/EditorAction.java` is invoked, where an object of the data type `mxGraph` (the predefined data structure for a graph in JGraphX) is retrieved. As (a) for the execution of synthesis algorithms, information concerning visual representation is irrelevant, and (b) the retrieval of vertices and edges in `mxGraph` requires complicated function calls, we regard this object not suitable to be manipulated directly. Thus we translate it to a simpler graph structure (defined in `arena/GameArena.java`), which offers a simple entry for users to extend GAVS with new algorithms.

For symbolic algorithms, our simple graph structure is further translated into logic formulae using BDDs. Under this setting, vertices are encoded using numbers in binary format, ensuring that no additional BDD variable is introduced. After the algorithm is executed, GAVS contains mechanisms to interpret the BDD tree, annotate the result of synthesis on the original model, and visualize the new model on the GUI. To redirect the result of synthesis back to the GUI, a map structure with the type `HashMap<String, HashSet<String>>` is required, where the key is the source vertex and the value set contains destination vertices, describing the edges that should be labeled by GAVS. For explicit state operating algorithms, mechanisms follow analogously.

3.1.2 Example: Working with GAVS

Due to page limits, we give two small yet representative examples on using GAVS. We refer the reader to the software homepage for a full-blown tutorial, and more and larger examples.

3.1.2.1 Example: Safety Games

We give a brief description of how to use GAVS for constructing a safety game and solving it step-by-step with the assist of fig. 3.2.

In the first step, the user constructs the game graph by simply drawing it using the graphical interface, similar to Figure 3.2-a: the states V_1 are called plant states and are of *rectangular shape*, while the states V_0 are called control states and are of *circular shape*.

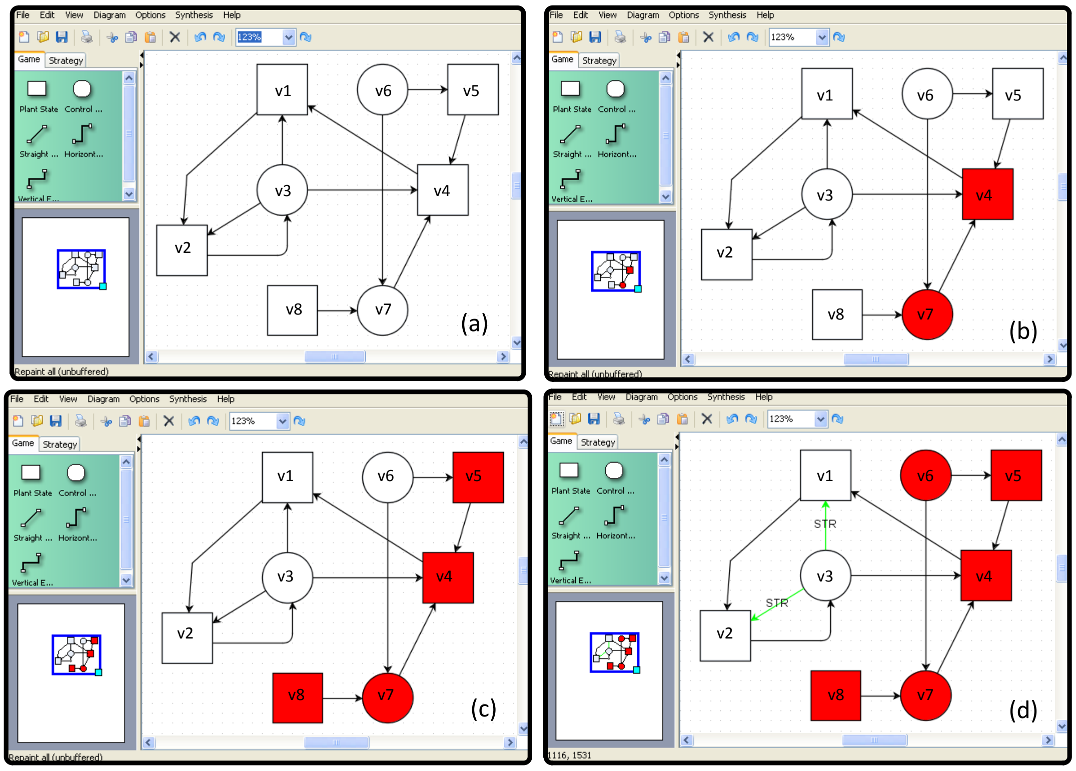


Figure 3.2: An example for constructing and executing a safety game.

Next, the user specifies the target nodes F, i.e., the nodes which player 0 tries to avoid. GAVS supports both graphical and textual methods[2]. In Figure 3.2-b, states v_4 and v_7 are painted by the user with red color, offering an graphical description of risk states.

Finally, GAVS can be used to either compute the winning set Win_1 of player 1 immediately or to guide the user through the computation of Win_1 step-by-step. In Figure 3.2, two intermediate steps are shown; in Figure 3.2-c and Figure 3.2-d, the set of red states represents $\mathsf{attr}_1^1(\{v_4, v_7\})$ and $\mathsf{attr}_1^2(\{v_4, v_7\})$, respectively. For games with positional strategies, the result of synthesis will be shown automatically on the graph with edges labeled with "$\mathtt{STR}$". In Figure 3.2, when safety game is played, edges (v_3, v_2) and (v_3, v_1) are highlighted as safe transitions.

3.1.2.2 Example: Muller Games

For Muller and Streett games, instead of generating strategies directly, game reductions are implemented for clearer understanding regarding the meaning of strategies. This is due to the fact that the generated FSM strategies for Muller and Streett games require

[2]Graphical specification is only available with reachability, safety, and Büchi winning conditions; for weak-parity and parity games, colors of vertices can also be labeled directly on the game graph.

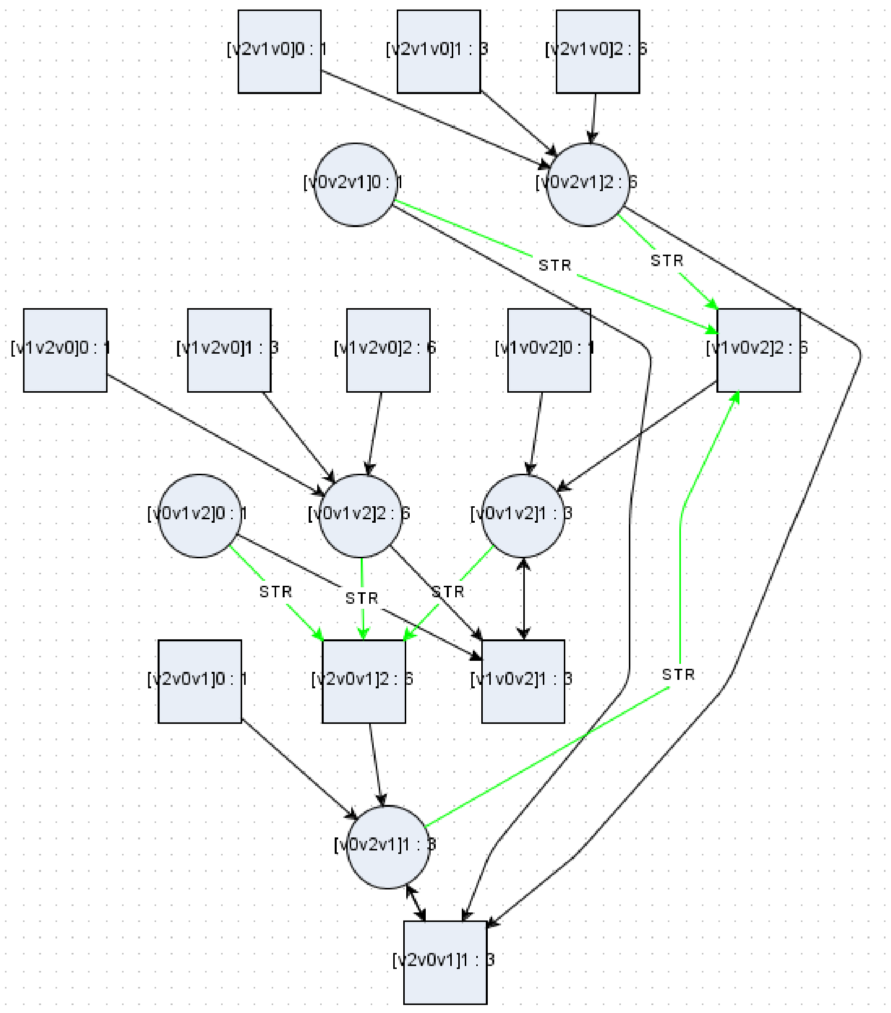


Figure 3.3: The synthesized result in the form of the parity game in GAVS.

memory which is in the worst case factorial to the number of states. It can be difficult for users to comprehend.

To indicate how Muller game reduction is applied in GAVS, we revisit again Figure 2.2 in Section 2.1.1.5. A Muller game is defined by using the arena and setting $\mathcal{F}$ to be $\{\{v_0, v_1, v_2\}\}$, i.e., the goal is to reach all vertices infinitely often. It can be checked that player 0 does not have any positional winning strategy. However, he has winning strategies using memory to win from $\{v_0, v_1, v_2\}$. E.g., *if the previous location is v_1 then go to v_2; if the previous location is v_2 then go to v_1.*

The reduced parity game generated by GAVS is shown in Figure 3.3, where each vertex is of the format `"[Vertex Permutation]LAR index : Color"`. The user then can directly invoke the parity game engine and observe the created strategy. By interpreting the strategy using LAR, it is clear that for player 0, the generated strategy is to follow the strategy described above. E.g.,

- Starting with initial vertex v_0, v_0 should move to v_1, as indicated in the reduced parity game, the strategy moves from vertex `[v0v2v1]0:1` to vertex `[v1v0v2]2:6`.
- Then v_1 moves back to v_0. In the reduced game, move from vertex `[v1v0v2]2:6` to vertex `[v0v1v2]1:3`.
- Now at v_0, player 0 should move to v_2, as indicated in the reduced parity game, the strategy moves from vertex `[v0v1v2]1:3` to vertex `[v2v0v1]2:6`.
- The play continues, and the highest color visited infinitely often is 6.

3.2 GAVS+: An Open Platform for the Research of Algorithmic Game Solving

We now present a major revision of the tool GAVS, i.e., the GAVS+ tool[3]. The number of supported games has been greatly extended and now encompasses in addition many classes important for the design and analysis of programs, e.g., it now allows to explore concurrent / probabilistic / distributed games, games played on pushdown graphs, and games of imperfect information. To our knowledge, GAVS+ is the first tool that supports such a comprehensive set of different games.

GAVS+ has three main goals:

- support of game types currently under active research; many times an implementation of a solver is only hard to come by, or only partial implementations exist.
- support of different input and output formats in order to allow for both interoperability with other tools and for easy access of existing collections of models, examples, and test cases in concrete application domains;
- ease of use by a unified graphical user interface (GUI) which allows to graphically specify the game and explore the computed solution.

The last requirement is partially fulfilled by the previous version of GAVS+: the GUI allows to visualize two-player, turn-based games on finite graph, solve the game, and store intermediate results in order to visualize the algorithms step-by-step. This also makes it a very useful tool for teaching these algorithms. In this section, we focus on the first requirement; the second requirement is discussed in Chapter 5.

3.2.1 Supported Games in GAVS+

For a complete overview on all supported games we refer the reader to Figure 3.4, here, we only give a very brief recap on newly added games.

- *Concurrent games* [DAHK07] are used to capture the condition when the next location is based on the combined decision simultaneously made by control and environment. When considering randomized strategies in reachability games (CRG),

[3]GAVS+ is a shortcut for **G**ame **A**rena **V**isualization and **S**ynthesis; the symbol **+** stands for an enhanced version. It is available at **`http://www6.in.tum.de/~chengch/gavs`**.

Game type (visualization)	Implemented algorithms
Fundamental game	Symbolic: Reachability, Safety, Büchi, Weak-parity, Staiger-Wagner
	Explicit state: Parity (global discrete strategy improvement, local strategy improvement)
	Reduction: Muller, Streett
Concurrent game	Sure reachability, Almost-sure reachability, Limit-sure reachability
Game of imperfect information	Safety using antichain
Pushdown game‡	Reachability (positional min-rank strategy, PDS strategy), Büchi (positional min-rank strategy), Parity (reduction)
Distributed game	Reachability (bounded distributed positional strategy for player-0), safety (projection, risk partition)
Markov decision process	Policy iteration, Value iteration, Linear programming (LP)
Simple stochastic game	Shapley (value iteration), Hoffman-Karp (policy iteration)

Figure 3.4: Game types and implemented algorithms in GAVS+, where "‡" indicates that visualization is currently not available.

efficient algorithms to compute sure, almost-sure, and limit-sure winning regions are available [DAHK07].

- *Stochastic games* model systems with uncertainty. The classical model of *Markov decision process (MDP, $1\frac{1}{2}$-player game)* [WW89] is widely used in economics and machine learning and considers a single player who has to work against an environment exhibiting random behavior. Adding an opponent to MDPs, one obtains *stochastic ($2\frac{1}{2}$-player) games* [Sha53]. Currently we focus on the subclass of simple stochastic games (SSG) [Con93]; many complicated games can be reduced to SSGs or solved by algorithms similar to algorithms solving SSG. For implemented algorithms for MDP and SSG, we refer readers to two survey papers [WW89, Con93] for details.
- *Games on pushdown graphs* (APDS) arise naturally when recursive programs are considered. Symbolic algorithms exist for reachability and Büchi winning conditions [Cac03a], and for parity conditions, a reduction[4] to two-player, finite-state parity games based on summarization is possible [Cac03a].
- *Distributed games* [MW03] are games formulating multiple processes with no interactions among themselves but only with the environment. Generating strategies for such a game is very useful for distributed systems, as a strategy facilitates orchestration of interacting components. Although the problem is undecidable in general [MW03], finding a distributed positional strategy for player-0 of such a system ($PositionalDG_0$), if it exists, is a practical problem. As $PositionalDG_0$ is NP-complete for reachability games [CRBK11], we modify the SAT-based witness algorithms in [AMN05] and implement a distributed version [CRBK11] for

[4]Currently, as the algorithm introduces an immediate exponential blowup in the graph, it is difficult to solve the game using the built-in algorithm specified in [VJ00].

bounded reachability games. Contents of distributed games are be described in later chapters.

3.2.2 Working with GAVS+

3.2.2.1 Example: Pushdown Games (APDSs)

For games with infinite states, our interest is in games played over push-down graphs (APDS), a natural extension when recursion is considered. Consider the following simple example:

- $V_0 = \{P_0\}$
- $V_1 = \{P_1\}$
- $\Gamma = \{a\}$
- $\Delta = \{\langle P_0, a\rangle \hookrightarrow \langle P_0\rangle, \langle P_0, a\rangle \hookrightarrow \langle P_0, aa\rangle, \langle P_1, a\rangle \hookrightarrow \langle P_0\rangle, \langle P_1, a\rangle \hookrightarrow \langle P_0, a\rangle.$

Assume that the initial configuration is P_1aa and the set of goal configuration is $\{P_0aa\}$. Then Figure 3.5 indicates the textual input format for GAVS+ to process the pushdown game[5][6].

```
## Comments used in the pds file (example1.pds)
P0_STATE = {P0}
P1_STATE = {P1}
ALPHABET = {a}
RULE = {P0 a -> P0; P0 a -> P0 a a; P1 a -> P0; P1 a -> P0 a}
INIT = {P1 a a}
GOAL = {P0 a a}
```

Figure 3.5: A simple APDS.

- To solve the pushdown game for the reachability criteria, on the menu bar execute `GAVS+ -> Pushdown Game -> Reachability Game (from APDS)`, similar to Figure 3.6.
- Once when the strategy is found, GAVS+ offers an option for interactive simulation: the user can act as the role of player-1 (environment) by selecting the rewriting rule, while GAVS+ updates the internal data structure and outputs the next move for player-0 (control). The supported strategy includes:
  1. Positional min-rank strategy, and
  2. PDS strategy.
- For Büchi games, only positional min-rank strategy simulation is offered.

[5]This example can be found in the GAVS+ software package with file name `GAVS_plus_testcase/APDS/example1.pds`.

[6]Currently no visualization of APDS is possible. We will use recursive games for the visualization in our future version.

- In our implementation, we follow the algorithm by T. Cachat in [Cac02], meaning that for games satisfying the Büchi condition, currently the engine solves a restricted form (although it is equivalent to the general form), as indicated in Section 2.2.2.2.
- Under this restriction, the goal configuration should be of the type $V_{goal} \cdot \Sigma^*$, where $V_{goal} \subseteq V_0 \uplus V_1$ is the set of all configurations which has states starting with the set of locations V_{goal}. In the textual representation, e.g., with goal location $P \in V_0 \uplus V_1$, the specification should be represented as `{P}`.

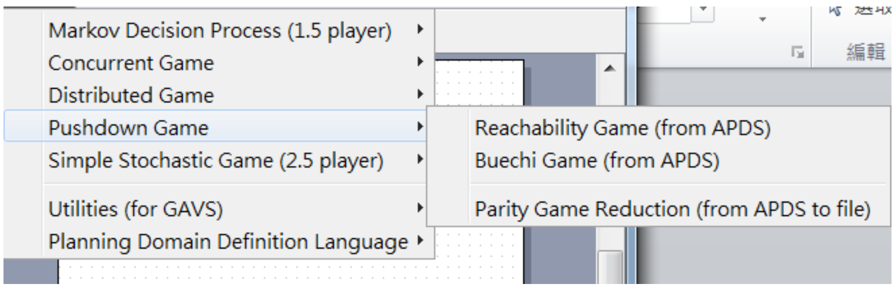
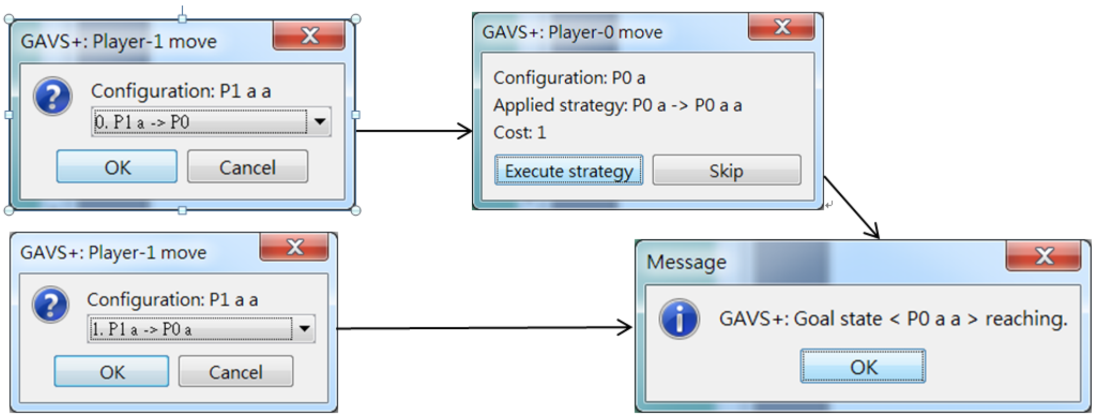

Markov Decision Process (1.5 player)
Concurrent Game
Distributed Game
Pushdown Game
Simple Stochastic Game (2.5 player)
Utilities (for GAVS)
Planning Domain Definition Language
Reachability Game (from APDS)
Buechi Game (from APDS)
Parity Game Reduction (from APDS to file)
編輯

Figure 3.6: The menu bar for solving APDS.

For the game in Figure 3.5, the screenshot of interactive execution (including all choices available by player-1) is shown in Figure 3.7.

- Result: In this example, the interactive simulation starts from selecting the interactive simulation type. Here we select with the option "`Positional Min-rank Strategy`".
- The user is now playing the role of player-1 (spoiler) and selects the move based on his wish. In Figure 3.7, for configuration P_1aa, two rewrite rules $\langle P_1, a\rangle \hookrightarrow \langle P_0\rangle, \langle P_1, a\rangle \hookrightarrow \langle P_0, a\rangle$ can be selected.
- Once when GAVS+ receives the move selected by the user, it performs the update based on the Positional min-rank strategy (for positional min-rank strategy, the



Configuration: P1 a a
0. P1 a -> P0

Configuration: P0 a
Applied strategy: P0 a -> P0 a a
Cost: 1

Configuration: P1 a a
1. P1 a -> P0 a

GAVS+: Goal state < P0 a a > reaching.

Figure 3.7: The tree of complete interactive simulation for APDS in Figure 3.5.

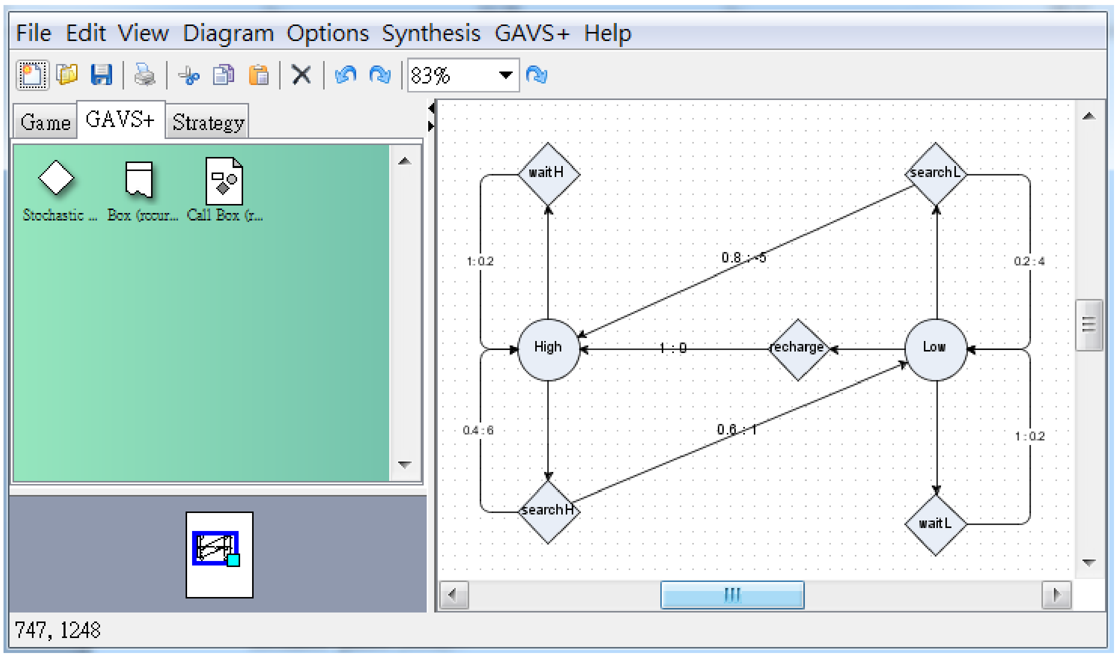


Figure 3.8: Constructing MDPs using the diamond vertices.

cost to reach the goal is computed and listed on the message box). For the above example, when the user selects rule $\langle P_1, a\rangle \hookrightarrow \langle P_0, a\rangle$, the configuration changes to P_0a. GAVS+ pops out the window as an indication of the next move ($\langle P_0, a\rangle \hookrightarrow \langle P_0, aa\rangle$). Finally, the goal state is reached.

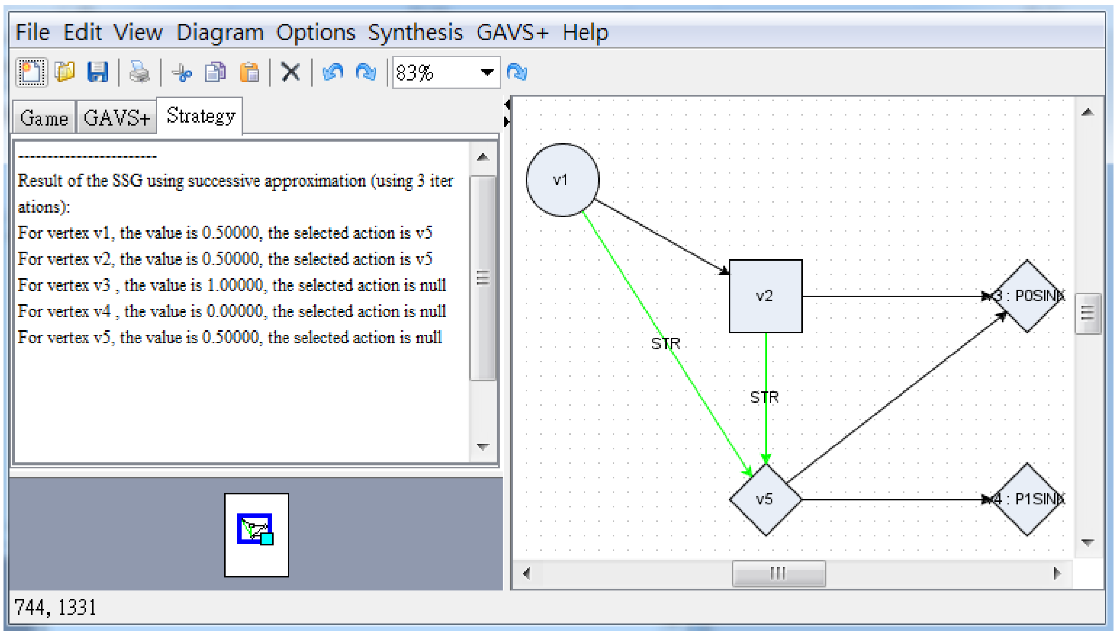
3.2.2.2 Example: Visualization and synthesis of MDPs

To visualize Markov decision processes, it is required to create actions (stochastic vertices), which can be found in the "**GAVS+**" panel (diamond shape, see Figure 3.8).

For the MDP described in Section 2.6 (Figure 2.13[7]), Figure 3.8 illustrates its modeling in GAVS+. For example, in action `searchH`, the label "`0.4:6`" means that selecting action `searchH` from location `High` may later go to location `High` with probability 0.4, and generate reward of value 6.

- The goal of a MDP is to generate a strategy which optimizes the reward. Note that a discount value between the interval $[0, 1)$ is specified to avoid generating infinite reward.
- Once when the game graph is constructed, to generate the strategy, currently in GAVS+, we have implemented three algorithms for solving MDPs:
  1. Value iteration (with visualization of intermediate steps). Execute `GAVS+ -> Markov Decision Process -> Infinite Horizon Discounted -> Value Iteration`.

[7] This example can be found in the GAVS+ software package with file name `GAVS_plus_Testcase/MDP/MDP.mxe`.



```
-----------------------
Result of the SSG using successive approximation (using 3 iter
ations):
For vertex v1, the value is 0.50000, the selected action is v5
For vertex v2, the value is 0.50000, the selected action is v5
For vertex v3 , the value is 1.00000, the selected action is null
For vertex v4 , the value is 0.00000, the selected action is null
For vertex v5, the value is 0.50000, the selected action is null
```

Figure 3.9: The SSG in Section 2.6 with labeled strategies.

  - For value iteration, the algorithm stops when the calculated value is very close to the previous calculated value (difference < 0.0001); as the source code is fully available, users can modify this value freely.

2. Policy iteration (with visualization of intermediate steps).
3. Linear programming. It is implemented using the `SimplexSolver` in Apache Common Math library [apa].

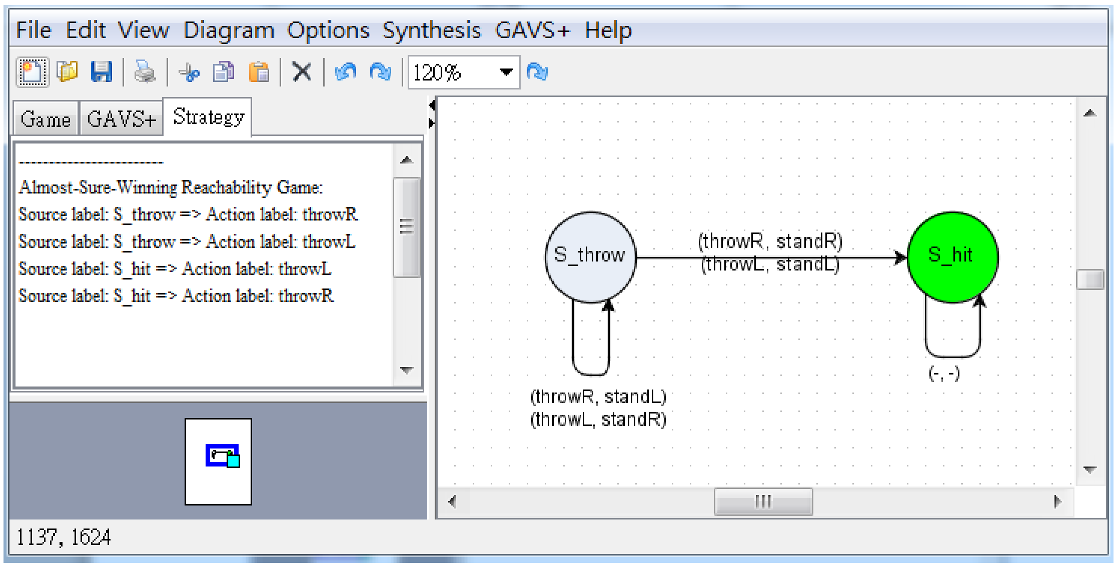
- Notice that compared to other games types (e.g., games implemented in GAVS) where no information is stored on the edge label, for MDPs edges are labeled with probability and rewards. Therefore clearing the strategy label should not remove all contents on every edge. For this special purpose, on the menu bar select `GAVS+ -> MDP -> Clear strategy labels (for MDP)`.

3.2.2.3 Example: Visualization and synthesis of SSGs

For simple stochastic games, construction techniques can be applied similar to the construction of MDPs. The only remark is that to differentiate $sink_0$ from other vertices, the user needs to annotate a vertex with text "`:P0SINK`" (similarly label $sink_1$ with text "`:P1SINK`"), similar to the labeling of color in two-player, turn-based games when solving parity games.

Figure 3.9 shows resulting SSG constructed using GAVS+ in Section 2.6 (Figure 2.12), where the optimal response for control and plant is highlighted in green[8].

[8]This example can be found in the GAVS+ software package with file name `GAVS_plus_Testcase/SSG/Sample.mxe`.



```
------------------------
Almost-Sure-Winning Reachability Game:
Source label: S_throw => Action label: throwR
Source label: S_throw => Action label: throwL
Source label: S_hit => Action label: throwL
Source label: S_hit => Action label: throwR
```

Figure 3.10: The concurrent reachability game (left-or-right) described in [DAHK07] and the generated strategy (almost-sure winning).

3.2.2.4 Example: Visualization and synthesis of concurrent reachability games

Concurrent games are used to capture the condition where the control and environment simultaneously select their moves, and the next location is based on the combined decision. The synthesis engine implemented in GAVS+ is able to solve the following winning conditions in a concurrent game:

1. Sure reachability winning.
2. Almost-sure reachability winning.
3. Limit-sure reachability winning.

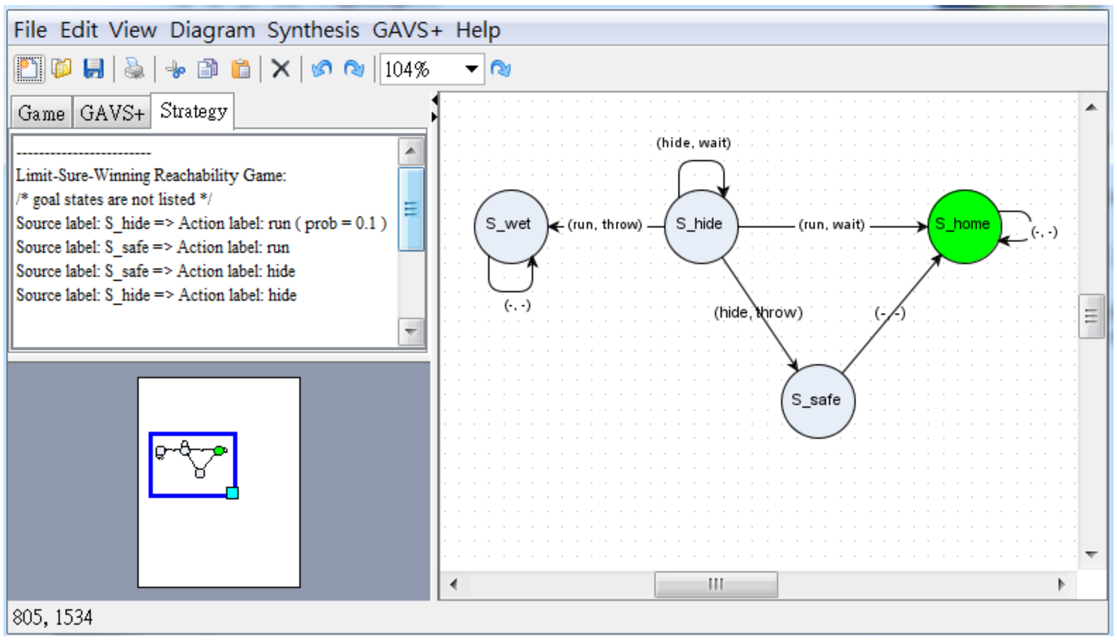
[Almost-sure winning] Figure 3.10 illustrates the *left-or-right game* example from [DAHK07][9], where the scenario is as follows: In a play, player-0 continuously throws a snow ball on the left or right window (with action symbol `throwL` and `throwR`), and player-1 shows up each time on either the left or the right window (with action symbol `standL` and `standR`). Choices of player-0 and player-1 are made simultaneously. For player-0 to hit player-1, there are two possibilities, i.e.,

- Player-0 throws the ball to the left and player-1 stands on the left, or
- player-0 throws the ball to the right and player-1 stands on the right.

Instructions executing the engine and the generated results are as follows.

- *(Construction)* For the edge labeling, `(act1, act2)` means that player-0 uses action `act1` and player-1 selects `act2`. The symbol `(-,-)` is designed for user convenience; the engine will generate all possible combinations of action pairs in its

[9]This example can be found in the GAVS+ software package with file name `GAVS_plus_Testcase/CRG/LeftOrRightGame.mxe`.



```
------------------------
Limit-Sure-Winning Reachability Game:
/* goal states are not listed */
Source label: S_hide => Action label: run ( prob = 0.1 )
Source label: S_safe => Action label: run
Source label: S_safe => Action label: hide
Source label: S_hide => Action label: hide
```

Figure 3.11: The concurrent reachability game (hide-and-run) described in [DAHK07] and the generated strategy (limit-sure winning).

internal representation. Lastly, for concurrent reachability games, a user should use graph labeling (mark the target state in green) to create the specification.

- *(Strategy Generation)* To generate the strategy, execute `GAVS+ -> Concurrent Game -> Almost-sure Reachability Winning`.
- *(Strategy Interpretation)* For this game, player-0 can hit player-1 (reaches state `S_hit`) with probability 1 from `S_throw`, i.e., player-0 is *almost-sure winning* at `S_throw`. On the strategy panel, the engine prints out the almost-sure winning region. We observe that for `S_throw`, both actions `throwL` and `throwR` are listed. This means that player-0 should play a random strategy which executes `throwL` with probability 0.5 and executes `throwR` with probability 0.5.

[Limit-sure winning] Figure 3.11 illustrates the *hide-or-run game* example from [DAHK07][10], where the scenario is as follows: Player-1 has only one snow ball, and he can decide at each iteration either to throw the ball or to wait (with action symbol `throw` and `wait`). Player-0 is initially hiding behind a wall (i.e., the games starts with location `S_hide`); he can decide to run or to continue hiding (with action symbol `run` and `hide`). We enumerate all situations at `S_hide`:

- When player-0 runs and player-1 throws the snow ball, player-0 gets wet (i.e., the play reaches `S_wet`), which is undesirable.
- When player-0 runs and player-1 waits, player-0 can reach home (i.e., the play reaches `S_home`).

[10]This example can be found in the GAVS+ software package with file name `GAVS_plus_Testcase/CRG/HideOrRunGame.mxe`.

- When player-0 hides and player-1 waits, continue next round (i.e., the play stays in `S_hide`).
- When player-0 hides and player-1 throws the snow ball, player-0 can be safe (i.e., reach `S_safe`), as player-1 only has one snow ball.

For this game, player-0 has a *limit-sure winning* strategy at location `S_hide`, and an *almost-sure winning* strategy at location `S_safe`[11].

- For limit-sure winning strategies, the user should specify the ε value for limit-sure winning, which is offered by GAVS+ using an additional dialog. Here assume that ε is set to 0.1 by the user.
- For the generated strategy, given a winning position:
    1. If a strategy is labeled with probability value, then it is executed based on the probability value. Indicated on the strategy panel of Figure 3.11, the randomized strategy for state `S_hide` is to perform action `run` with probability 0.1 and action `hide` with probability $(1 - 0.1)$.
    2. Otherwise, all other strategies should perform uniformly at random with the remaining probability from (1). For example, at position `S_safe`, player-0 should play actions `hide` and `run` with equal probability.
- Notice that for limit-sure winning, currently due to our algorithm implementation, if a state is a goal state, it will **not** be listed/reported on the strategy panel.

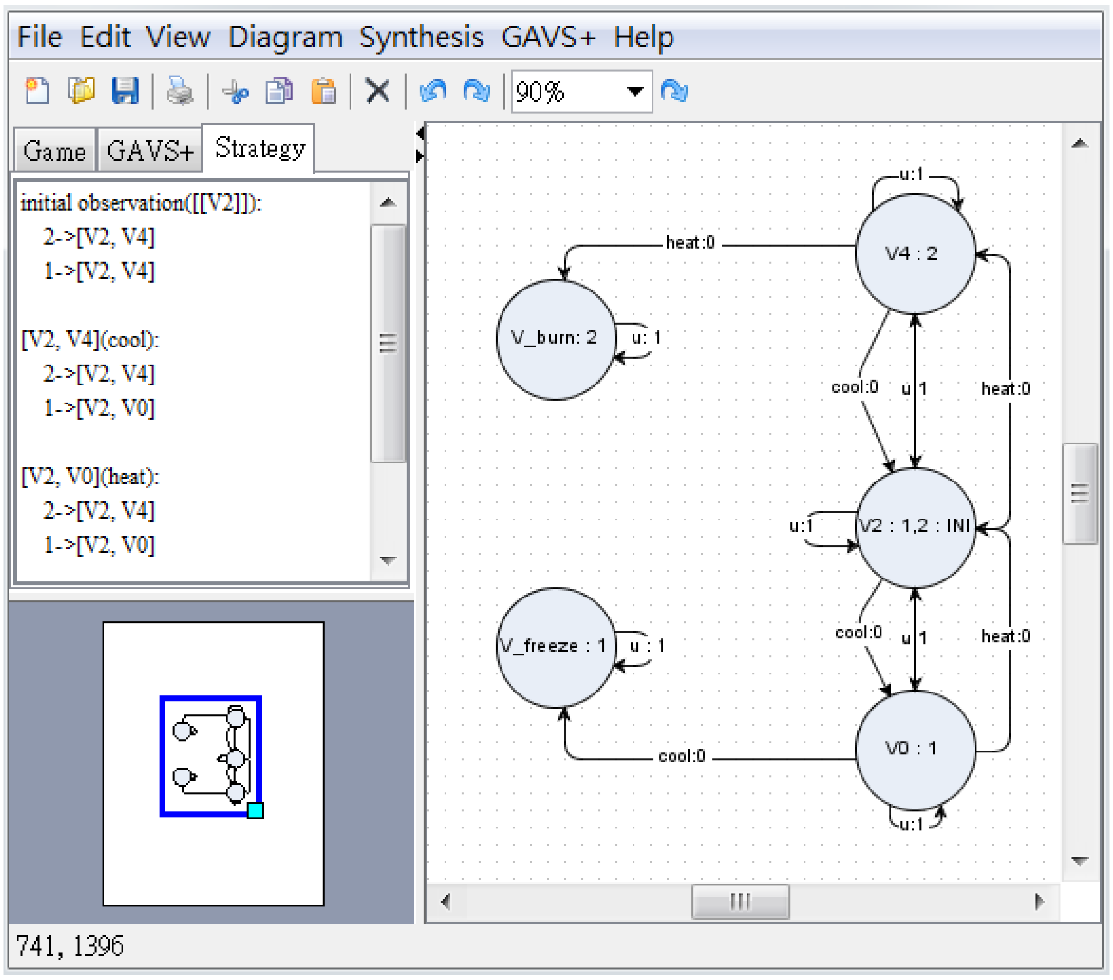
3.2.2.5 Example: Visualization and synthesis of games of imperfect information

We revisit the game of imperfect information in Section 2.4 (Figure 2.9). In the implementation of GAVS+, we follow the formulation by De Wulf, Doyen, and Raskin [DWDR06][12]. Therefore, the game graph looks slightly different:

- Each node is labeled with a set of observation identifiers. For example, the vertex `V_burn:2` has its name as `V_burn` and observation identifier `2`.
    - Following Figure 2.9, we interpret observation `2` as `hot`, and observation `1` as `cold`.
    - Location `V2` is labeled with "`:INI`", indicating that it is the initial location.
- Following the formulation in [DWDR06], actions are split into *controllable and uncontrollable actions*.
    - In Figure 3.12, the self-loop of `V_burn` is labeled with "`u:1`", where `u` is the action symbol, and `1` indicates that it is an uncontrollable action. Similarly, for the edge labeled "`heat:0`", it is a controllable action with action symbol `heat`.

[11]Precisely, at `S_safe`, player-0 has a sure-winning strategy. However, when invoking the limit-sure engine, it only generates limit-sure and almost-sure strategies.

[12]In Section 2.4, we use an alternative formulation, as we need to connect games of imperfect information with distributed games.



```
initial observation([[V2]]):
  2->[V2, V4]
  1->[V2, V4]

[V2, V4](cool):
  2->[V2, V4]
  1->[V2, V0]

[V2, V0](heat):
  2->[V2, V4]
  1->[V2, V0]
```

741, 1396

Figure 3.12: Simple temperature control in Figure 2.10 modeled using GAVS+.

In [DWDR06], the execution proceeds by first receiving an observation. Then player-0 chooses an controllable action, followed by the environment who arbitrarily executes an uncontrollable action. It is not difficult to observe that starting with vertex labeled `V2` with two possible observations `1` and `2` (i.e., `cold` and `hot`), Figure 3.12 models the same temperature control system as Figure 2.9. At locations `V_burn` and `V_freeze`, player-0 is unable to perform a move, and thus are considered as risk states.

On the strategy panel in Figure 3.12, we can examine the observation-based strategy (a finite state machine) generated by GAVS+.

- Initially, starting with vertex `V2`, when the observation returns `1` or `2`, the text "`1->[V2,V4]`" indicates that player-0 is certain to be in the vertex set {`V2`, `V4`}.
- Then "`[V2, V4](cool)`" indicates that as player-0 is certain to be in the vertex set {`V2`, `V4`}, he should play with the action symbol `cool`. If player-0 continues the play by following the strategy, locations `V_burn` and `V_freeze` are never reached.

3.2.3 Invoking GAVS+ on the console using standardized input format

GAVS+ offers interfaces to invoke the engine from the console without using the GUI. To observe engines supporting the console mode, on the console, execute "`java -jar GAVS+ -help`", then the set of available parameters for console execution will be listed. Details are omitted here.

3.3 Related Work

We give a brief recap on recent implementations in games, including GIST (for probabilistic games) [CHJR10], Alpaga (for parity games of imperfect information) [BCDW$^+$09], UPPAAL-Tiga (for timed games) [BCD$^+$07a]. Also after our implementation and publication, we find a tool called GASt (short for **G**ames, **A**utomata and **St**rategies) [ATW06][13] which also contains implementations for two-player finite games. From automata theories, an implementation called GOAL [TCT$^+$08] is available for the research of omega automata and temporal logic. The development of the GOAL tool motivates our development of GAVS+.

Compared to tools mentioned above, the uniqueness of our tool relies on the fulfillment of three features.

- The graphical user interface of GAVS+ enables users to construct the game with intuition.
- GAVS+ supports game solving under a broad spectrum. Admittedly not all algorithms are optimized, but for algorithms implemented with BDDs, they scale nicely to relatively large examples.
- The open-source feature (GPLv3) of GAVS+ makes it easily accessible for interested researchers to modify the tool and integrate new features or algorithms.

3.4 Summary

In this section, we present GAVS+, containing a solver library for games together with a unified front-end GUI. The tool targets to serve as an open platform for the research community in algorithmic game solving. In addition, It is meant to serve as a playground for researchers thinking of mapping interesting problems to game solving. We list some used applications.

- We have collaborated with RWTH Aachen University to implement game solvers which translates Muller game solving to safety game solving.[14].
- In Chapter 4, VISSBIP, a tool to model BIP systems and perform priority synthesis, is extended from GAVS+.

[13]We thank Dr. Barbara Jobstmann for her notification of the GASt tool.

[14]For details concerning the underlying technique, we refer interested readers to [NRZ11].

CHAPTER 4

Application A. Priority Synthesis: Theories, Algorithms, and Tools

Abstract

In this chapter, we present a concept called *priority synthesis*, which aims to automatically generate a set of priorities such that the system constrained by the synthesized priorities satisfies a given *safety property* or *deadlock freedom*. We formulate the problem, explain the underlying complexity results, give algorithms for synthesis, and lastly, propose extensions for more complex systems.

We formulate priority synthesis for BIP systems using the automata-theoretic framework proposed by Ramadge and Wonham [RW89] (a direct modeling using game semantics is also possible). In this framework, priority synthesis results in searching for a supervisor from the restricted class of supervisors, in which each is solidly expressible using priorities. While priority-based supervisors are easier to use, e.g., they support the construction of distributed protocols, they are harder to compute. We show that finding a supervisor based on priorities that ensures deadlock freedom of the supervised system is NP-complete in the size of the system. As priority synthesis is expensive and we seek for strategies which largely maintain the original behavior, we present a counter-example guided interactive synthesis (CEGIS) framework combing the concept of fault-localization (using safety-game) and fault-repair (using SAT for conflict resolution). For complex systems, we propose three complementary methods as preprocessing steps for priority synthesis, namely (a) data abstraction to reduce component complexities, (b) alphabet abstraction and ♯-deadlock to ignore components, and (c) automated assumption learning for compositional priority synthesis.

For implementation, we have created VISSBIP, a software tool for visualizing and automatically orchestrating component-based systems consisting of a set of components and their possible interactions. The graphical interface of VISSBIP allows the user to interactively construct BIP models [BBS06], from which executable code (C/C++) is generated. It also checks if the system satisfies a given safety property. If the check

fails, the tool automatically generates additional priorities on the interactions that ensure the desired property.

Contents

4.1 Introduction

Priorities define *stateless-precedence relations between actions* available in component-based systems. They can be used to restrict the behavior of a system in order to avoid undesired states. They are particularly useful to avoid deadlock states (i.e., states in which all action are disabled), because they do not introduce new deadlock states and therefore avoid creating new undesired states. Furthermore, due to their stateless property and the fact that they operate on the interface of a component, they can be relatively easy implemented in a distributed setting [GPQ10, BBQ11].

In this chapter, we present a concept called *priority synthesis* (first proposed in our tool paper [CBJ^{+}11]), which aims to automatically generate a set of priorities such that the system constrained by the synthesized priorities satisfies a given *safety property* or *deadlock freedom*. We formulate the problem, explain the underlying complexity results, give algorithms for synthesis, and lastly, propose extensions for more complex systems.

For theory, we formulate priority synthesis under BIP systems using an automata-theoretic framework similar to [RW89]. Then, we focus on the *hardness of synthesizing priorities*, which constitutes one of our main contributions. We prove that, given a labeled transition system, finding a set of priorities that ensures safety and deadlock freedom is NP-complete in the size of the system. Our result is in contrast to the work in [RW89], where a general (monolithic) supervisor, which is usually difficult to distribute, can be found in polynomial-time in the size of the system. Our priority-based supervisors are easier to distribute but harder to compute. Contents concerning theoretical results can be found in the Appendix.

As priority synthesis is expensive, we present a CEGIS-based search framework for priority synthesis, which mimics the process of *fault-localization* and *fault-repair* (Section 4.4). Intuitively, a state is a fault location if it is the latest point from which there is a way to avoid a failure, i.e., there exists (i) an outgoing action that leads to an *attracted state*, a state from which all paths unavoidably reach a bad state, and (ii) there exists an alternative action that avoids entering any of the attracted states. We compute fault locations using the algorithm for *safety games*. Given a set of fault locations, priority synthesis is achieved via fault-repair: an algorithm resolves potential conflicts in priorities generated via fault-localization and finds a satisfying subset of priorities as a solution for synthesis. Our symbolic encodings on the system, together with the new variable ordering heuristic and other optimizations, help to solve problems much more efficiently compared to our preliminary implementation in [CBJ$^+$11]. Furthermore, it allows us to integrate an adversary environment model similar to the setting in Ramadge and Wonham's controller synthesis framework [RW89].

Abstraction or compositional techniques are widely used in verification of infinite state or complex systems for safety properties but *not all* techniques ensure that synthesizing an abstract system for deadlock-freeness guarantees deadlock-freeness in the concrete system (Section 4.5). Therefore, it is important to find appropriate techniques to assist synthesis on complex problems. We first revisit *data abstraction* (Section 4.5.1) for data domain such that priority synthesis works on an abstract system composed by components abstracted component-wise [BBSN08]. Second, we present a technique called *alphabet-abstraction* (Section 4.5.2), handling complexities induced by the composition of components. Lastly, for behavioral-safety properties (not applicable for deadlock-avoidance), we develop theoretical results for *compositional priority synthesis* and utilize automata-learning [Ang87] to automatically perform such task (Section 4.6).

We implemented the presented algorithms (except connection with the data abstraction module in D-Finder [BGL$^+$11]) in the VISSBIP[1] tool and performed experiments to evaluate them (Section 4.7). VISSBIP is an open-source tool which enables users to construct, analyze, and synthesize component-based systems. Our examples show that the process using fault-localization and fault-repair generates priorities that are highly desirable. Alphabet abstraction enables us to scale to arbitrary large problems. We also present a model for distributed communication. In this example, the priorities synthesized by our engine are completely local (i.e., each priority involves two local actions within a component). Therefore, they can be translated directly to distributed control. We summarize related work and conclude with an algorithmic flow in Section 4.8 and 4.9.

4.2 Introduction to the Behavior-Interaction-Priority (BIP) Framework

The Behavior-Interaction-Priority (BIP) framework[2] provides a rigorous component-based design flow for heterogeneous systems. Component-based systems can be modeled using three ingredients: (a) *Behaviors*, which define for each basic component a finite set of labeled transitions (i.e., an automaton), (b) *Interactions*, which define syn-

[1]VISSBIP is a shortcut for **Vi**sualization and **S**ynthesis of **S**imple **BIP** models.
It is available at **`http://www6.in.tum.de/~chengch/vissbip`**.

[2]http://www-verimag.imag.fr/Rigorous-Design-of-Component-Based.html?lang=en

chronizations between two or more transitions of different components, and (c) *Priorities*, which are used to choose between possible interactions. A detailed description of the BIP language can be found in [BBS06].

In the BIP framework [BBS06], the user writes a model using a programming language based on the Behavior-Interaction-Priority principle. Using the BIP tool-set, this model can be compiled to run on a dedicated hardware platforms. The core of the execution is the *BIP engine*, which decides which interactions are executed and ensures that the execution follows the semantics. The interactions and priorities are used to ensure global properties of the systems. For instance, a commonly seen problem is mutual exclusion, i.e., two components should avoid being in two dedicated states at the same time. Intuitively, we can enforce this property by requiring that interactions that exit one of the dedicated states have higher priority than interactions that enter the states.

4.3 Component-based Modeling and Priority Synthesis

4.3.1 Behavioral-Interaction-Priority Framework

To simplify the explanations, we focus on *simple* systems, i.e., systems without hierarchies and finite data types. Intuitively, a simple BIP system consists of a set of automata (extended with data) that synchronize on joint labels.

Definition 4 (BIP System)**.** *We define a (simple BIP) system as a tuple $\mathcal{S} = (C, \Sigma, \mathcal{P})$, where*

- *Σ is a finite set of* ***events*** *or interaction labels, called* ***interaction alphabet****,*
- *$C = \bigcup_{i=1}^{m} C_i$ is a finite set of* ***components****. Each component C_i is a transition system extended with data. Formally, C_i is a tuple $(L_i, V_i, \Sigma_i, T_i, l_i^0, e_i^0)$:*
  - *$L_i = \{l_{i_1}, \ldots, l_{i_n}\}$ is a finite set of* control locations.
  - *$V_i = \{v_{i_1}, \ldots, v_{i_p}\}$ is a finite set of* (local) variables *with a finite domain. Wlog we assume that the domain is the Boolean domain $\mathbb{B} = \{\texttt{True}, \texttt{False}\}$. We use $|V_i|$ to denote the number of variables used in C_i. An* evaluation (or assignment) *of the variables in V_i is a functions $e : V_i \rightarrow \mathbb{B}$ mapping every variable to a value in the domain. We use $\mathcal{E}(V_i)$ to denote the set of all evaluations over the variables V_i. Given a Boolean formula $f \in \mathcal{B}(V_i)$ over the variables in V_i and an evaluation $e \in \mathcal{E}(V_i)$, we use $f(e)$ to refer to the truth value of f under the evaluation e.*
  - *$\Sigma_i \subseteq \Sigma$ is a subset of interaction labels used in C_i.*
  - *T_i is the set of* transitions. *A transition $t_i \in T_i$ is of the form (l, g, σ, f, l'), where $l, l' \in L_i$ are the* source and destination location, *$g \in \mathcal{B}(V_i)$ is called the* guard *and is a Boolean formula over the variables V_i. $\sigma \in \Sigma_i$ is an interaction label (specifying the event triggering the transition), and $f : V_i \rightarrow \mathcal{B}(V_i)$ is the* update function *mapping every variable to a Boolean formula encoding the change of its value.*
  - *$l_i^0 \in L_i$ is the* initial location *and $e_i^0 \in \mathcal{E}(V_i)$ is the initial evaluation of the variables.*
- *$\mathcal{P}$ is a finite set of interaction pairs (called* ***priorities****) defining a relation $\prec\ \subseteq \Sigma \times \Sigma$*

between the interaction labels. We require that $\prec$ is (1) transitive and (2) non-reflexive (i.e., there are no circular dependencies) [GS03]. For $(\sigma_1, \sigma_2) \in \mathcal{P}$, we sometimes write $\sigma_1 \prec \sigma_2$ to highlight the property of priority.

Definition 5 (Configuration)**.** *Given a system $\mathcal{S}$, a* configuration (or state) *c is a tuple $(l_1, e_1, \ldots, l_m, e_m)$ with $l_i \in L_i$ and $e_i \in \mathcal{E}(V_i)$ for all $i \in \{1, \ldots, m\}$. We use $\mathcal{C}_\mathcal{S}$ to denote the set of all reachable configurations. The configuration $(l^0_1, e^0_1, \ldots, l^0_m, e^0_m)$ is called the* initial configuration *of $\mathcal{S}$ and is denoted by c^0.*

Definition 6 (Enabled Interactions)**.** *Given a system $\mathcal{S}$ and a configuration $c = (l_1, e_1, \ldots, l_m, e_m)$, we say an interaction $\sigma \in \Sigma$ is **enabled (in** c**)**, if the following conditions hold:*

1. *(Joint participation) $\forall i \in \{1, \ldots, m\}$, if $\sigma \in \Sigma_i$, then $\exists g_i, f_i, l'_i$ such that $(l_i, g_i, \sigma, f_i, l'_i) \in T_i$ and $g_i(e_i) =$ `True`.*
2. *(No higher priorities enabled) For all other interaction $\bar{\sigma} \in \Sigma$ satisfying joint participation (i.e., $\forall i \in \{1, \ldots, m\}$, if $\bar{\sigma} \in \Sigma_i$, then $\exists(l_i, \bar{g}_i, \bar{\sigma}, \bar{f}_i, \bar{l}'_i) \in T_i$ such that $\bar{g}_i(e_i) =$ `True`), $(\sigma, \bar{\sigma}) \notin \mathcal{P}$ holds.*

Definition 7 (Behavior)**.** *Given a system $\mathcal{S}$, two configurations $c = (l_1, e_1, \ldots, l_m, e_m)$, $c' = (l'_1, e'_1, \ldots, l'_m, e'_m)$, and an interaction $\sigma \in \Sigma$ enabled in c, we say c'* is a σ-successor (configuration) of *c, denoted $c \xrightarrow{\sigma} c'$, if the following two conditions hold for all components $C_i = (L_i, V_i, \Sigma_i, T_i, l^0_i, e^0_i)$:*

- *(Update for participated components) If $\sigma \in \Sigma_i$, then there exists a transition $(l_i, g_i, \sigma, f_i, l'_i) \in T_i$ such that $g_i(e_i) =$ `True` and for all variables $v \in V_i$, $e'_i = f_i(v)(e_i)$.*
- *(Stutter for idle components) Otherwise, $l'_i = l_i$ and $e'_i = e_i$.*

Given two configurations c and c', we say c' is reachable from *c with the interaction sequence $w = \sigma_1 \ldots \sigma_k$, denoted $c \xrightarrow{w} c'$, if there exist configurations $c_0, \ldots, c_k$ such that (i) $c_0 = c$, (ii) $c_k = c'$, and (iii) for all $i : 0 \leq i < k$, $c_i \xrightarrow{\sigma_{i+1}} c_{i+1}$. We denote the set of all configuration of $\mathcal{S}$ reachable from the initial configuration c^0 by $\mathcal{R}_\mathcal{S}$. The* language *of a system $\mathcal{S}$, denoted $\mathcal{L}(\mathcal{S})$, is the set $\{w \in \Sigma^* \mid \exists c' \in \mathcal{R}_\mathcal{S}$ such that $c^0 \xrightarrow{w} c'\}$. Note that $\mathcal{L}(\mathcal{S})$ describes the behavior of $\mathcal{S}$, starting from the initial configuration c^0.*

In this chapter, we adapt the following simplifications:

- We do not consider uncontrollable events (of the environment), since the BIP language is currently not supporting them. However, our framework would allow us to do so. More precisely, we solve priority synthesis using a game-theoretic version of controller synthesis [RW89], in which uncontrollability can be modeled. Furthermore, since we consider only safety properties, our algorithms can be easily adapted to handle uncontrollable events.
- We do not consider data transfer during the interaction, as it is merely syntactic rewriting over variables between different components. However, in our implementation, simple data transfer is supported.

4.3.2 Priority Synthesis for Safety and Deadlock Freedom

Definition 8 (Risk-Configuration/Deadlock Safety). *Given a system $\mathcal{S} = (C, \Sigma, \mathcal{P})$ and the set of* risk configuration $\mathcal{C}_{risk} \subseteq \mathcal{C}_{\mathcal{S}}$ *(also called* bad states*), the system is* ***safe*** *if the following conditions hold. (A system that is not safe is called* ***unsafe****.)*

- **(Deadlock-free)** $\forall c \in \mathcal{R}_{\mathcal{S}}, \exists \sigma \in \Sigma, \exists c' \in \mathcal{R}_{\mathcal{S}} : c \xrightarrow{\sigma} c'$
- **(Risk-state-free)** $\mathcal{C}_{risk} \cap \mathcal{R}_{\mathcal{S}} = \emptyset$.

Definition 9 (Priority Synthesis). *Given a system $\mathcal{S} = (C, \Sigma, \mathcal{P})$, and the set of risk configuration $\mathcal{C}_{risk} \subseteq \mathcal{C}_{\mathcal{S}}$, priority synthesis searches for a set of priorities $\mathcal{P}_{+}$ such that*

- *For $\mathcal{P} \cup \mathcal{P}_{+}$, the defined relation $\prec_{\mathcal{P} \cup \mathcal{P}_{+}} \subseteq \Sigma \times \Sigma$ is also (1) transitive and (2) non-reflexive.*
- $(C, \Sigma, \mathcal{P} \cup \mathcal{P}_{+})$ *is safe.*

Given a system $\mathcal{S}$, we define the size of $\mathcal{S}$ as the size of the product graph induced by $\mathcal{S}$, i.e, $|Q| + |\Sigma| + |\delta|$, where Q is the set of vertices in the graph. Then, we have the following result.

Theorem 1 (Hardness of priority synthesis [CJBK11]). *Given a system $\mathcal{S} = (C, \Sigma, \mathcal{P})$, finding a set $\mathcal{P}_{+}$ of priorities such that $(C, \Sigma, \mathcal{P} \cup \mathcal{P}_{+})$ is safe is NP-complete in the size of $\mathcal{S}$.*

Proof. We leave the proof as well as discussions concerning its relations to the work by Ramadge and Wonham [RW89] to the appendix. □

We briefly mention the definition of **behavioral safety**, which is a powerful notion to capture erroneous behavioral-patterns for the system under design.

Definition 10 (Behavioral Safety). *Given a system $\mathcal{S} = (C, \Sigma, \mathcal{P})$ and a regular language $\mathcal{L}_{\neg P} \subseteq \Sigma^*$ called the* risk specification*, the system is* ***B-safe*** *if $\mathcal{L}(\mathcal{S}) \cap \mathcal{L}_{\neg P} = \emptyset$. A system that is not B-safe is called* ***B-unsafe****.*

It is well-known that the problem of asking for behavioral safety can be reduced to the problem of risk-state freeness. More precisely, since $\mathcal{L}_{\neg P}$ can be represented by a finite automaton $\mathcal{A}_{\neg P}$ (the monitor), priority synthesis for behavioral safety can be reduced to priority synthesis in the synchronous product of the system $\mathcal{S}$ and $\mathcal{A}_{\neg P}$ with the goal to avoid any product state that has a final state of $\mathcal{A}_{\neg P}$ in the second component.

4.4 A Framework of Priority Synthesis based on Fault-Localization and Fault-Repair

In this section, we describe our symbolic encoding scheme, followed by presenting our priority synthesis mechanism using a fault-localization and repair approach.

4.4.1 System Encoding

Our symbolic encoding is inspired by the execution semantics of the BIP engine, which during execution, selects one of the enabled interactions and executes the interaction. In our engine, we mimic the process and create a two-stage transition: For each iteration,

- (Stage 0) The *environment* raises all enabled interactions.
- (Stage 1) Based on the raised interactions, the *controller* selects one enabled interaction (if there exists one) while respecting the priority, and updates the state based on the enabled interaction.

Given a system $\mathcal{S} = (C, \Sigma, \mathcal{P})$, we use the following sets of Boolean variables to encode $\mathcal{S}$:

- $\{stg, stg'\}$ is the *stage indicator* and its primed version.
- $\bigcup_{\sigma\in\Sigma}\{\sigma, \sigma'\}$ are the variables representing interactions and their primed version. We use the same letter for an interaction and the corresponding variable, because there is a one-to-one correspondence between them.
- $\bigcup_{i=1\dots m} Y_i \cup Y_i'$, where $Y_i = \{y_{i1}, \dots, y_{ik}\}$ and $Y_i' = \{y_{i1}', \dots, y_{ik}'\}$ are the variables and their primed version, respectively, used to encode the locations L_i. (We use a binary encoding, i.e., $k = \lceil log|L_i|\rceil$). Given a location $l \in L_i$, we use $enc(l)$ and $enc'(l)$ to refer to the encoding of l using Y_i and Y_i', respectively.
- $\bigcup_{i=1\dots m}\bigcup_{v\in V_i}\{v, v'\}$ are the variables of the components and their primed version.

Algorithm 1: Generate Stage-0 transitions

```
input  : System $\mathcal{S} = (C, \Sigma, \mathcal{P})$
output: Stage-0 transition predicate $\mathcal{T}_{stage_0}$
begin
    for $\sigma \in \Sigma$ do
1       let predicate $P_\sigma$ := True
    for $\sigma \in \Sigma$ do
        for $i = \{1, \dots, m\}$ do
2           if $\sigma \in \Sigma_i$ then $P_\sigma := P_\sigma \wedge \bigvee_{(l,g,\sigma,f,l')\in T_i}(enc(l) \wedge g)$
    let predicate $\mathcal{T}_{stage_0} := stg \wedge \neg stg'$
    for $\sigma \in \Sigma$ do
3       $\mathcal{T}_{stage_0} := T_{stage_0} \wedge (\sigma' \leftrightarrow P_\sigma)$
    for $i = \{1, \dots, m\}$ do
4       $\mathcal{T}_{stage_0} := T_{stage_0} \wedge \bigwedge_{y\in Y_i} y \leftrightarrow y' \wedge \bigwedge_{v\in V_i} v \leftrightarrow v'$
    return $\mathcal{T}_{stage_0}$
```

We use Algorithm 1 and 2 to create transition predicates $\mathcal{T}_{stage_0}$ and $\mathcal{T}_{stage_1}$ for Stage 0 and 1, respectively. Note that $\mathcal{T}_{stage_0}$ and $\mathcal{T}_{stage_1}$ can be merged but we keep them

Algorithm 2: Generate Stage-1 transitions

```
input  : System $\mathcal{S} = (C, \Sigma, \mathcal{P})$
output: Stage-1 transition predicate $\mathcal{T}_{stage_1}$
begin
    let predicate $\mathcal{T}_{stage_1}$ := False
    for $\sigma \in \Sigma$ do
        let predicate $T_\sigma := \neg stg \wedge stg'$
        for $i = \{1, \dots, m\}$ do
            if $\sigma \in \Sigma_i$ then
1               $T_\sigma := T_\sigma \wedge \bigvee_{(l,g,\sigma,f,l') \in T_i} (enc(l) \wedge g \wedge \sigma \wedge \sigma' \wedge enc'(l') \wedge \bigwedge_{v \in V_i} v' \leftrightarrow f(v))$
        for $\sigma' \in \Sigma, \sigma' \neq \sigma$ do
2           $T_\sigma := T_\sigma \wedge \sigma' =$ False
        for $i = \{1, \dots, m\}$ do
3           if $\sigma \notin \Sigma_i$ then $T_\sigma := T_\sigma \wedge \bigwedge_{y \in Y_i} y \leftrightarrow y' \wedge \bigwedge_{v \in V_i} v \leftrightarrow v'$
        $\mathcal{T}_{stage_1} := \mathcal{T}_{stage_1} \vee T_\sigma$
    for $\sigma_1 \prec \sigma_2 \in \mathcal{P}$ do
4       $\mathcal{T}_{stage_1} := \mathcal{T}_{stage_1} \wedge ((\sigma_1 \wedge \sigma_2) \rightarrow \neg \sigma_1')$
    return $\mathcal{T}_{stage_1}$
```

separately, in order to (1) have an easy and direct way to synthesize priorities, (2) allow expressing the freedom of the environment, and (3) follow the semantics of the BIP engine.

- In Algorithm 1, Line 2 computes for each interaction σ the predicate P_σ representing all the configurations in which σ is enabled in the current configuration. In Line 3, starting from the first interaction, $\mathcal{T}_{stage_0}$ is continuously refined by conjoining $\sigma' \leftrightarrow P_\sigma$ for each interaction σ, i.e., the variables σ' is true if and only if the interaction σ is enabled. Finally, Line 4 ensures that the system configuration does not change in stage 0.
- In Algorithm 2, Line 1, 2, 3 are used to create the transition in which interaction σ is executed (Line 2 ensures that only σ is executed; Line 3 ensures the stuttering move of unparticipated components). Given a priority $\sigma_1 \prec \sigma_2$, in configurations in which σ_1 and σ_2 are both enabled (i.e., $\sigma_1 \wedge \sigma_2$ holds), the conjunction with Line 4 removes the possibility to execute σ_1 when σ_2 is also available.

4.4.2 Step A. Finding Fix Candidates using Fault-localization

Synthesizing a set of priorities to make the system safe can be done in various ways, and we use Figure 4.1 to illustrate our underlying idea. Consider a system starting from state c_1. It has two risk configurations c_6 and c_7. In order to avoid risk using priorities, one method is to work on the initial configuration, i.e., to use the set of priorities $\{e \prec$

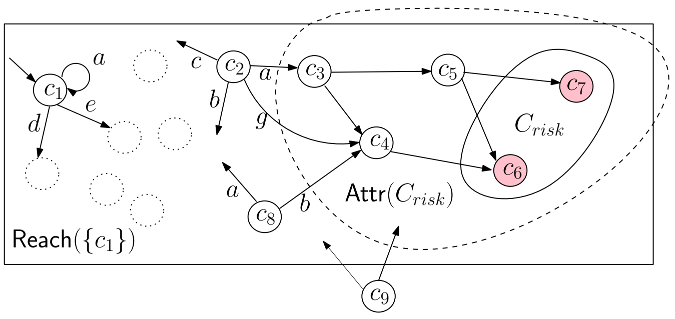


Figure 4.1: Locating fix candidates.

$a, d \prec a\}$. Nevertheless, it can be observed that the synthesized result is not very desirable, as the behavior of the system has been greatly restricted.

Alternatively, our methodology works *backwards* from the set of risk states and finds states which is able to *escape from risk*. In Figure 4.1, as states c_3, c_4, c_5 unavoidably enter a risk state, they are within the *risk-attractor* ($\mathsf{Attr}(\mathcal{C}_{risk})$). For state c_2, c_8, and c_9, there exists an interaction which avoids risk. Thus, if a set of priorities $\mathcal{P}_+$ can ensure that from c_2, c_8, and c_9, the system can not enter the attractor, then $\mathcal{P}_+$ is the result of synthesis. Furthermore, as c_9 is not within the set of reachable states from the initial configuration ($\mathsf{Reach}(\{c_1\})$ in Figure 4.1), then it can be eliminated without consideration. We call $\{c_2, c_8\}$ a **fault-set**, meaning that an erroneous interaction can be taken to reach the risk-attractor.

Under our formulation, we can directly utilize the result of **algorithmic game solving** [GTW02] to compute the fault-set. Algorithm 3 explains the underlying computation: For conciseness, we use $\exists\Xi$ ($\exists\Xi'$) to represent existential quantification over all umprimed (primed) variables used in the system encoding. Also, we use the operator `SUBS`(X, Ξ, Ξ') for variable swap (substitution) from unprimed to primed variables in X: the `SUBS` operator is common in most BDD packages.

- In the beginning, we create P_{ini} for initial configuration, P_{dead} for deadlock (no interaction is enabled), and P_{risk} for risk configurations.
- In Part A, adding a stage-0 configuration can be computed similar to adding the environment state in a safety game. In a safety game, for an environment configuration to be added, there exists a transition which leads to the attractor.
- In Part A, adding a stage-1 configuration follows the intuition described earlier. In a safety game, for a control configuration c to be added, all outgoing transitions of c should lead to the attractor. This is captured by the set difference operation `PointTo` $\setminus$ `Escape` in Line 5.
- In Part B, Line 7 creates the transition predicate entering the attractor. Line 8 creates predicate `OutsideAttr` representing the set of stage-1 configuration outside the attractor. In Line 9, by conjuncting with `OutsideAttr` we ensure that the algorithm does not return a transition within the attractor.
- Part C removes transitions whose source is not within the set of reachable states.

Algorithm 3: Fault-localization

```
input  : System $\mathcal{S} = (C, \Sigma, \mathcal{P})$, $\mathcal{T}_{stage_0}$, $\mathcal{T}_{stage_1}$
output: $\mathcal{T}_f \subseteq \mathcal{T}_{stage_1}$ as the set of stage-1 transitions starting from the fault-set but
        entering the risk attractor
begin
    let $P_{ini} := stg \wedge \bigwedge_{i=1\ldots m}(enc(l_i^0) \wedge \bigwedge_{v \in V_i} v \leftrightarrow e_i^0(v))$
    let $P_{dead} := \neg stg \wedge \bigwedge_{\sigma \in \Sigma} \neg\sigma$
    let $P_{risk} := \neg stg \wedge \bigvee_{(l_1,e_1,\ldots,l_m,e_m) \in \mathcal{C}_{risk}} (enc(l_1) \wedge \bigwedge_{v \in V_1} v \leftrightarrow e_1(v) \wedge \ldots$
    $enc(l_m) \wedge \bigwedge_{v \in V_m} v \leftrightarrow e_m(v))$

    // Part A: solve safety game
    let $\text{Attr}_{pre} := P_{dead} \vee P_{risk}$, $\text{Attr}_{post} :=$ False
1   while True do
        // add stage-0 (environment) configurations
2       $\text{Attr}_{post,0} := \exists \Xi' : (\mathcal{T}_{stage_0} \wedge \texttt{SUBS}((\exists \Xi' : \text{Attr}_{pre}), \Xi, \Xi'))$
        // add stage-1 (system) configurations
3       let PointTo $:= \exists \Xi' : (\mathcal{T}_{stage_1} \wedge \texttt{SUBS}((\exists \Xi' : \text{Attr}_{pre}), \Xi, \Xi'))$
4       let Escape $:= \exists \Xi' : (\mathcal{T}_{stage_1} \wedge \texttt{SUBS}((\exists \Xi' : \neg\text{Attr}_{pre}), \Xi, \Xi'))$
5       $\text{Attr}_{post,1} :=$ PointTo $\setminus$ Escape
6       $\text{Attr}_{post} := \text{Attr}_{pre} \vee \text{Attr}_{post,0} \vee \text{Attr}_{post,1}$          // Union the result
        if $Attr_{pre} \leftrightarrow Attr_{post}$ then break // Break when the image saturates
        else $\text{Attr}_{pre} := \text{Attr}_{post}$

    // Part B: extract $\mathcal{T}_f$
7   PointTo $:= \mathcal{T}_{stage_1} \wedge \texttt{SUBS}((\exists \Xi' : \text{Attr}_{pre}), \Xi, \Xi'))$
8   OutsideAttr $:= \neg\text{Attr}_{pre} \wedge (\exists \Xi' : \mathcal{T}_{stage_1})$
9   $\mathcal{T}_f :=$ PointTo $\wedge$ OutsideAttr

    // Part C: eliminate unused transition using reachable
       states
    let $\text{reach}_{pre} := P_{ini}$, $\text{reach}_{post} :=$ False
10  while True do
        $\text{reach}_{post} := \text{reach}_{pre} \vee \texttt{SUBS}(\exists \Xi : (\text{reach}_{pre} \wedge (\mathcal{T}_{stage_0} \vee \mathcal{T}_{stage_1})), \Xi', \Xi)$
        if $\text{reach}_{pre} \leftrightarrow \text{reach}_{post}$ then break          // Break when the image
        saturates
        else $\text{reach}_{pre} := \text{reach}_{post}$
11  return $\mathcal{T}_f \wedge \text{reach}_{post}$
```

4.4.3 Step B. Priority Synthesis via Conflict Resolution - from Stateful to Stateless

Due to our system encoding, in Algorithm 3, the return value $\mathcal{T}_f$ contains not only the risk interaction but also all possible interactions simultaneously available. Recall Figure 4.1, $\mathcal{T}_f$ returns three transitions, and we can extract **priority candidates** from each transition.

- On c_2, a enters the risk-attractor, while b, g, c are also available. We have the following candidates $\{a \prec b, a \prec g, a \prec c\}$.
- On c_2, g enters the risk-attractor, while a, b, c are also available. We have the following candidates $\{g \prec b, g \prec c, g \prec a\}$[3].
- On c_8, b enters the risk-attractor, while a is also available. We have the following candidate $b \prec a$.

From these candidates, we can perform **conflict resolution** and generate a set of priorities that ensures avoiding the attractor. For example, $\{a \prec c, g \prec a, b \prec a\}$ is a set of satisfying priorities to ensure safety. Note that the set $\{a \prec b, g \prec b, b \prec a\}$ is not a legal priority set, because it creates circular dependencies. In our implementation, conflict resolution is performed using SAT solvers: In the SAT problem, any priority $\sigma_1 \prec \sigma_2$ is presented as a Boolean variable $\underline{\sigma_1 \prec \sigma_2}$, which can be set to `True` or `False`. If the generated SAT problem is satisfiable, for all variables $\underline{\sigma_1 \prec \sigma_2}$ which is evaluated to `True`, we add priority $\sigma_1 \prec \sigma_2$ to $\mathcal{P}_+$. The synthesis engine creates four types of clauses.

1. **[Priority candidates]** For each edge $t \in \mathcal{T}_f$ which enters the risk attractor using σ and having $\sigma_1, \ldots, \sigma_e$ available actions (excluding σ), create clause $(\bigvee_{i=1\ldots e} \underline{\sigma \prec \sigma_i})$[4].
2. **[Existing priorities]** For each priority $\sigma \prec \sigma' \in \mathcal{P}$, create clause $(\underline{\sigma \prec \sigma'})$.
3. **[Non-reflective]** For each interaction σ used in (1) and (2), create clause $(\neg\underline{\sigma \prec \sigma})$.
4. **[Transitive]** For any three interactions $\sigma_1, \sigma_2, \sigma_3$ used in (1) and (2), create clause $((\underline{\sigma_1 \prec \sigma_2} \wedge \underline{\sigma_2 \prec \sigma_3}) \Rightarrow \underline{\sigma_1 \prec \sigma_3})$.

When the problem is satisfiable, we only output the set of priorities within the priority candidates (as non-reflective and transitive clauses are inferred properties). Admittedly, here we still solve an NP-complete problem. Nevertheless,

- The number of interactions involved in the fault-set can be much smaller than Σ.
- As the translation does not involve complicated encoding, we observe from our experiment that solving the SAT problem does not occupy a large portion (less than 20% for all benchmarks) of the total execution time.
- The whole approach tries to establish a barrier to avoid entering bad states, mimicking the pervasive strategy.

[3]Notice that at least one candidate is a true candidate for risk-escape. Otherwise, during the attractor computation, c_2 will be included within the attractor.

[4]In implementation, Algorithm 3 works symbolically on BDDs and proceeds on **cubes** of the risk-edges (a cube contains a set of states having the same enabled interactions and the same risk interaction), hence it avoids enumerating edges state-by-state.

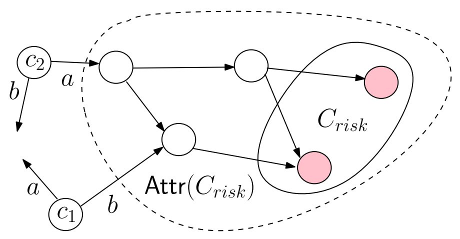


Figure 4.2: A simple scenario where conflicts are unavoidable on the fault-set.

4.4.4 Optimization

Currently, we use the following optimization techniques compared to the preliminary implementation of [CBJ$^+$11].

4.4.4.1 Handling Unsatisfiability

In the resolution scheme in Section 4.4.3, when the generated SAT problem is unsatisfiable, we can redo the process by moving some states in the fault-set to the attractor. This procedure is implemented by selecting a subset of priority candidates and annotate to the original system. We call this process **priority-repushing**. E.g., consider the system $\mathcal{S} = (C, \Sigma, \mathcal{P})$ in Figure 4.2. The fault-set $\{c_1, c_2\}$ is unable to resolve the conflict: For c_1 the priority candidate is $a \prec b$, and for c_2 the priority candidate is $b \prec a$. When we redo the analysis with $\mathcal{S} = (C, \Sigma, \mathcal{P} \cup \{a \prec b\})$, this time c_2 will be in the attractor, as now c_2 must respect the priority and is unable to escape using a. Currently in our implementation, we supports the repushing under fixed depth to increase the possibility of finding a fix.

4.4.4.2 Initial Variable Ordering: Modified FORCE Heuristics

As we use BDDs to compute the risk-attractor, a good initial variable ordering can greatly influence the total required time solving the game. Although finding an optimal initial variable ordering is known to be NP-complete [THY93], many heuristics can be applied to find a good yet non-optimal ordering[5]. The basic idea of these heuristics is to group variables close if they participate in the same transition [CGP99]; experiences have shown that this creates a BDD diagram of smaller size. Thus our goal is to find a heuristic algorithm which can be computed efficiently while createing a good ordering.

We adapt the concept in the FORCE heuristic [AMS03]. Although the purpose of the FORCE heuristic is to work on SAT problems, we find the concept very beneficial in our problem setting. We explain the concept of FORCE based on the example in [AMS03], and refer interested readers to the paper [AMS03] for full details.

Given a CNF formula $C = c_1 \wedge c_2 \wedge c_3$, where $c_1 = (a \vee c), c_2 = (a \vee d), c_3 = (b \vee d)$.

[5]Also, dynamic variable ordering, a technique which changes the variable ordering at run-time, can be beneficial when no good variable ordering is known [CGP99]

- Consider a variable ordering $\langle a, b, c, d\rangle$. For this ordering, we try to evaluate it by considering the sum of the *span*. A span is the maximum distance between any two variables within the same clause. For c_1, under the ordering the span equals 2; for c_2 the span equals 3, and the sum of the span equals 7.
- Consider another variable ordering $\langle c, a, d, b\rangle$. Then the sum of span equals 3. Thus we consider that $\langle c, a, d, b\rangle$ is superior than $\langle a, b, c, d\rangle$.
- The purpose of the FORCE heuristic is to reduce the sum of such span. In the CNF example, the name of the heuristics suggests that a conceptual force representing each clause is grouping variables used within the clause.

Back to priority synthesis, consider the set of components $\bigcup_{i=1}^{n} C_i$ together with interaction labels Σ. We may similarly compute the sum of all spans, where now a span is *the maximum distance between any two components participating the same interaction* $\sigma \in \Sigma$. Precisely, we analogize clauses and variables in the original FORCE heuristic with interaction symbols and components. Therefore, we regard the FORCE heuristics equally applicable to create a better initial variable ordering for priority synthesis.

[Algorithm Sketch] Our modified FORCE heuristics is as follows.

1. Create an initial order of vertices composed from a set of components $\bigcup_{i=1}^{n} C_i$ and interactions $\sigma \in \Sigma$. Here we allow the user to provide an initial variable ordering, such that the FORCE heuristic can be applied more efficiently.
2. Repeat for limited time or until the span stops decreasing:
   - Create an empty list.
   - For each interaction label $\sigma \in \Sigma$, derive its center of gravity $COG(\sigma)$ by computing the *average position* of all participated components. Use the average position as its value. Add the interaction with the value to the list.
   - For each component C_i, compute its value by $\frac{\sum_{\sigma \in Sigma_i} COG(\sigma)}{|\Sigma_i|}$. Add the component with the value to the list.
   - Sort the list based on the value. The resulting list is considered as a new variable ordering. Compute the new span and compare with the span from the previous ordering.

4.4.4.3 Dense Variable Encoding

The encoding in Section 4.4.1 is *dense* compared to the encoding in [CBJ$^+$11]. In [CBJ$^+$11], for each component C_i participating interaction σ, one separate variable σ_i is used. Then a joint action is done by an `AND` operation over all variables, i.e., $\bigwedge_i \sigma_i$. This eases the construction process but makes BDD-based game solving very inefficient: For a system $\mathcal{S}$, let $\Sigma_{use1} \subseteq \Sigma$ be the set of interactions where only one component participates within. Then the encoding in [CBJ$^+$11] uses at least $2|\Sigma \setminus \Sigma_{use1}|$ more BDD variables than the dense encoding.

4.4.4.4 Safety Engine Speedup

Lastly, as our created game graph is *bipartite*, Algorithm 3 can be refined to work on two separate images of stage-0 and stage-1, such that line 2 and line {3,4} are executed in alternation.

4.5 Handling Complexities

In verification, it is standard to use *abstraction* and *modularity* to reduce the complexity of the analyzed systems. Abstraction is also useful in synthesis. However, note that if an abstract system is deadlock-free, it does not imply that the concrete system is as well. E.g., in Figure 4.3, the system composed by C_1 and C_2 contains deadlock (assume both a and b needs to be paired to be executed). However, when we over-approximate C_1 to an abstract system C_1^{α}, a system composed by C_1^{α} and C_2 is deadlock free. On the other hand, deadlock-freeness of an under-approximation also does not imply deadlock-freeness of a concrete system. An obvious example can be obtained by under-approximating the system C_1 in Figure 4.3 to an abstract system C_1^{β}. Again, the composition of C_1^{β} and C_2 is deadlock-free, while the concrete system is not.

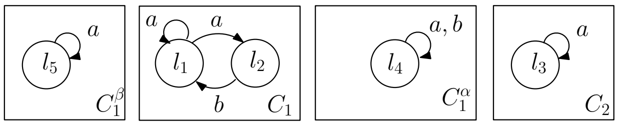


Figure 4.3: A scenario where the concrete system contains deadlock, but the abstract system is deadlock free.

In the following, we propose three techniques.

4.5.1 Data abstraction

Data abstraction techniques presented in the previous work [BBSN08] and implemented in the D-Finder tool kit [BGL$^+$11] are *deadlock preserving*, i.e., synthesizing the abstract system to be deadlock free ensures that the concrete system is also deadlock free. Basically, the method works on an abstract system composed by components abstracted component-wise from concrete components. For example, if an abstraction preserves all control variables (i.e., all control variables are mapped by identity) and the mapping between the concrete and abstract system is precise with respect to all guards and updates (for control variables) on all transitions, then it is deadlock preserving. For further details, we refer interested readers to [BBSN08, BGL$^+$11].

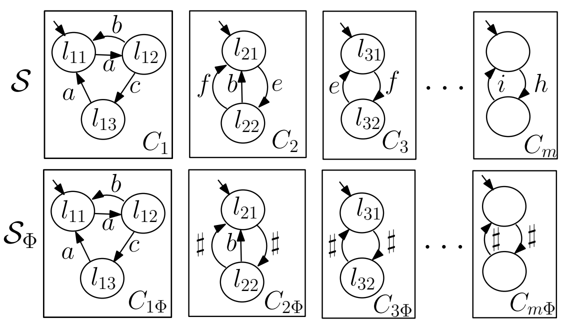


Figure 4.4: A system $\mathcal{S}$ and its $\sharp$-abstract system $\mathcal{S}_\Phi$, where $\Sigma_\Phi = \Sigma \setminus \{a, b, c\}$.

4.5.2 Alphabet abstraction

Second, we present *alphabet abstraction*, targeting to synthesize priorities to avoid deadlock (but also applicable for risk-freeness with extensions). The underlying intuition is to abstract concrete behavior of components out of concern.

Definition 11 (Alphabet Transformer). *Given a set Σ of interaction alphabet. Let $\Sigma_\Phi \subseteq \Sigma$ be **abstract alphabet**. Define $\alpha : \Sigma \rightarrow (\Sigma \setminus \Sigma_\Phi) \cup \{\sharp\}$ as the alphabet transformer, such that for $\sigma \in \Sigma$,*

- *If $\sigma \in \Sigma_\Phi$, then $\alpha(\sigma) := \sharp$.*
- *Otherwise, $\alpha(\sigma) := \sigma$.*

Definition 12 (Alphabet Abstraction: Syntax). *Given a system $\mathcal{S} = (C, \Sigma, \mathcal{P})$ and abstract alphabet $\Sigma_\Phi \subseteq \Sigma$, define the **$\sharp$-abstract system** $\mathcal{S}_\Phi$ to be $(C_\Phi, (\Sigma \setminus \Sigma_\Phi) \cup \{\sharp\}, \mathcal{P}_\Phi)$, where*

- *$C_\Phi = \bigcup_{i=1\ldots m} C_{i\Phi}$, where $C_{i\Phi} = (L_i, V_i, \Sigma_{i\Phi}, T_{i\Phi}, l_i^0, e_i^0)$ changes from C_i by **syntactically** replacing every occurrence of $\sigma \in \Sigma_i$ to $\alpha(\sigma)$.*
- *$\mathcal{P} = \bigcup_{i=1\ldots k} \sigma_i \prec \sigma_i'$ changes to $\mathcal{P}_\Phi = \bigcup_{i=1\ldots k} \alpha(\sigma_i) \prec \alpha(\sigma_i')$, and the relation defined by $\mathcal{P}_\Phi$ should be transitive and nonreflexive.*

The definition for a configuration (state) of a $\sharp$-abstract system follows Definition 2. Denote the set of all configuration of $\mathcal{S}_\Phi$ reachable from c_0 as $\mathcal{C}_{\mathcal{S}_\Phi}$. The update of configuration for an interaction $\sigma \in \Sigma \setminus \Sigma_\Phi$ follows Definition 3. The only difference is within the semantics of the $\sharp$-interaction.

Definition 13 (Alphabet Abstraction: Semantics for $\sharp$-interaction). *Given a configuration $c = (l_1, v_1, \ldots, l_m, v_m)$, the $\sharp$-interaction is **enabled** if the following conditions hold.*

1. *(≥ 1 participants) **Exists** $i \in \{1, \ldots, m\}$ where $\sharp \in \Sigma_{i\Phi}$, $\exists t_i = (l_i, g_i, \sharp, f_i, l_i') \in T_{i\Phi}$ such that $g(v_i) = \texttt{True}$.*
2. *(No higher priorities enabled) There exists no other interaction $\sigma_\flat \in \Sigma, (\sharp, \sigma_\flat) \in \mathcal{P}_\Phi$ such that $\forall i \in \{1, \ldots, m\}$ where $\sigma_\flat \in \Sigma_i$, $\exists t_{i\flat} = (l_i, g_{i\flat}, \sigma_{i\flat}, f_{i\flat}, l_i'') \in T_i$, $g_{i\flat}(v_i) = \texttt{True}$.*

Then for a configuration $c = (l_1, v_1, \ldots, l_m, v_m)$, the configuration after taking an enabled $\sharp$-interaction changes to $c^\flat = (l_1^\flat, v_1^\flat, \ldots, l_m^\flat, v_m^\flat)$:

- *(**May-update** for participated components) If $\sharp \in \Sigma_i$, then for transition $t_i = (l_i, g_i, \sharp, f_i, l_i') \in T_{i\Phi}$ such that $g_i(v_i) = \texttt{True}$, either*

1. *$l_i^\flat = l_i'$, $v_i^\flat = f_i(v_i)$, or*
2. *$l_i^\flat = l_i$, $v_i^\flat = v_i$.*

Furthermore, at least one component updates (i.e., select option 1).

- *(Stutter for unparticipated components) If $\sharp \notin \Sigma_i$, $l_i^\flat = l_i$, $v_i^\flat = v_i$.*

Lastly, the behavior of a $\sharp$-abstract system follows Definition 4. In summary, the above definitions indicate that in a $\sharp$-abstract system, any local transitions having alphabet symbols within Σ_Φ can be executed in isolation or jointly. Thus, we have the following result.

Lemma 1. *Given a system $\mathcal{S}$ and its $\sharp$-abstract system $\mathcal{S}_\Phi$, define $\mathcal{R}_\mathcal{S}$ ($\mathcal{R}_{\mathcal{S}_\Phi}$) be the reachable states of system $\mathcal{S}$ (corresponding $\sharp$-abstract system) from from the initial configuration c^0. Then $\mathcal{R}_\mathcal{S} \subseteq \mathcal{R}_{\mathcal{S}_\Phi}$.*

Proof. Result from the comparison between Definition 6 and 13. □

As alphabet abstraction looses the execution condition by overlooking paired interactions, a $\sharp$-abstract system is deadlock-free does not imply that the concrete system is deadlock free. E.g., consider a system $\mathcal{S}'$ composed only by C_2 and C_3 in Figure 4.4. When $\Phi = \Sigma \setminus \{b\}$, its $\sharp$-abstract system $\mathcal{S}'_\Phi$ is shown below. In $\mathcal{S}'$, when C_2 is at location l_{21} and C_3 is at location l_{31}, interaction e and f are disabled, meaning that there exists a deadlock from the initial configuration. Nevertheless, in $\mathcal{S}'_\Phi$, as the $\sharp$-interaction is always enabled, it is deadlock free.

In the following, we strengthen the deadlock condition by the notion of **$\sharp$-deadlock**. Intuitively, a configuration is $\sharp$-deadlocked, if it is deadlocked, or the only interaction available is the $\sharp$-interaction.

Definition 14 ($\sharp$-deadlock)**.** *Given a $\sharp$-abstract system $\mathcal{S}_\Phi$, a configuration $c \in \mathcal{C}_{\mathcal{S}_\Phi}$ is $\sharp$-deadlocked, if $\nexists \sigma \in \Sigma \setminus \Sigma_\Phi, c' \in \mathcal{C}_{\mathcal{S}_\Phi}$ such that $c \xrightarrow{\sigma} c'$.*

In other words, a configuration c of $\mathcal{S}_\Phi$ is $\sharp$-deadlocked implies that all interactions labeled with $\Sigma \setminus \Sigma_\Phi$ are disabled at c.

Lemma 2. *Given a system $\mathcal{S}$ and its $\sharp$-abstract system $\mathcal{S}_\Phi$, define $\mathcal{D}$ as the set of deadlock states reachable from the initial state in $\mathcal{S}$, and $\mathcal{D}^\sharp$ as the set of $\sharp$-deadlock states reachable from the initial state in $\mathcal{S}_\Phi$. Then $\mathcal{D} \subseteq \mathcal{D}^\sharp$.*

Proof. Consider a deadlock state $c \in \mathcal{D}$.

1. Based on Lemma 1, c is also in $\mathcal{R}_{\mathcal{S}_\Phi}$.
2. In $\mathcal{S}$, as $c \in \mathcal{D}$, all interactions are disabled in c. Then correspondingly in $\mathcal{S}_\Phi$, for state c, any interaction $\sigma \in \Sigma \setminus \Sigma_\Phi$ is also disabled. Therefore, c is $\sharp$-deadlocked.

Based on 1 and 2, $c \in \mathcal{D}^\sharp$. Thus $\mathcal{D} \subseteq \mathcal{D}^\sharp$. □

Theorem 2. *Given a system $\mathcal{S}$ and its $\sharp$-abstract system $\mathcal{S}_\Phi$, if $\mathcal{S}_\Phi$ is $\sharp$-deadlock-free, then $\mathcal{S}$ is deadlock-free.*

Proof. As $\mathcal{S}_\Phi$ is $\sharp$-deadlock-free, we have $\mathcal{R}_{\mathcal{S}_\Phi} \cap \mathcal{D}^\sharp = \emptyset$. According to Lemma 1 and 2, we have $\mathcal{R}_\mathcal{S} \subseteq \mathcal{R}_{\mathcal{S}_\Phi}$ and $\mathcal{D} \subseteq \mathcal{D}^\sharp$. Hence $\mathcal{R}_\mathcal{S} \cap \mathcal{D} = \emptyset$, implying that $\mathcal{S}$ is deadlock-free. □

(Algorithmic issues) Based on the above results, the use of alphabet abstraction and the notion of $\sharp$-deadlock offers a methodology for priority synthesis working on abstraction. Detailed steps are presented as follows.

1. Given a system $\mathcal{S}$, create its $\sharp$-abstract system $\mathcal{S}_\Phi$ by a user-defined $\Sigma_\Phi \subseteq \Sigma$. In our implementation, we let users select a subset of components $C_{s_1}, \ldots, C_{s_k} \in C$, and generate $\Sigma_\Phi = \Sigma \setminus (\Sigma_{s_1} \cup \ldots \cup \Sigma_{s_k})$.
   - E.g., consider system $\mathcal{S}$ in Figure 4.4 and its $\sharp$-abstract system $\mathcal{S}_\Phi$. The abstraction is done by looking at C_1 and maintaining $\Sigma_1 = \{a, b, c\}$.
   - When a system contains no variables, the algorithm proceeds by eliminateing components whose interaction are completely in the abstract alphabet. In Figure 4.4, as for $i = \{3 \ldots m\}$, $\Sigma_{i\Phi} = \{\sharp\}$, it is sufficient to eliminate all of them during the system encoding process.
2. If $\mathcal{S}_\Phi$ contains $\sharp$-deadlock states, we could obtain a $\sharp$-deadlock-free system by synthesizing a set of priorities $\mathcal{P}_+$, where the defined relation $\prec_+ \subseteq ((\Sigma \setminus \Sigma_\Phi) \cup \{\sharp\}) \times (\Sigma \setminus \Sigma_\Phi)$ using techniques presented in Section 4.4.
   - In the system encoding, the predicate $P_{\sharp dead}$ for $\sharp$-deadlock is defined as $stg = \texttt{False} \wedge \bigwedge_{\sigma \in \Sigma \setminus \Sigma_\Phi} \sigma = \texttt{False}$.
   - If the synthesized priority is having the form $\sharp \prec \sigma$, then translate it into a set of priorities $\bigcup_{\sigma' \in \Sigma_\Phi} \sigma' \prec \sigma$.

4.6 Assume-guarantee Based Priority Synthesis

We use an assume-guarantee based compositional synthesis algorithm for behavior safety. Given a system $\mathcal{S} = (C_1 \cup C_2, \Sigma, \mathcal{P})$ and a risk specification described by a *deterministic finite state automaton* R, where $\mathcal{L}(R) \subseteq \Sigma^*$. We use $|\mathcal{S}|$ to denote the size of $\mathcal{S}$ and $|R|$ to denote the number of states of R. The synthesis task is to find a set of priority rules $\mathcal{P}_+$ such that adding $\mathcal{P}_+$ to the system $\mathcal{S}$ can make it B-Safe with respect to the risk specification $\mathcal{L}(R)$. This can be done using an *assume-guarantee* rule that we will describe in the next paragraph.

We first define some notations needed for the rule. The system $\mathcal{S}_+ = (C_1 \cup C_2, \Sigma, \mathcal{P} \cup \mathcal{P}_+)$ is obtained by adding priority rules $\mathcal{P}_+$ to the system $\mathcal{S}$. We use $\mathcal{S}_1 = (C_1, \Sigma, \mathcal{P} \cap \Sigma \times \Sigma_1)$ and $\mathcal{S}_2 = (C_2, \Sigma, \mathcal{P} \cap \Sigma \times \Sigma_2)$ to denote two sub-systems of $\mathcal{S}$. We further partition the alphabet Σ into three parts Σ_{12}, Σ_1, and Σ_2, where Σ_{12} is the set of interactions appear both in the sets of components C_1 and C_2 (in words, the shared alphabet of C_1 and C_2), Σ_i is the set of interactions appear only in the set of components C_i (in words, the local alphabet of C_i) for $i = 1, 2$. Also, we require that the decomposition of the system must satisfy that $\mathcal{P} \subseteq \Sigma \times (\Sigma_1 \cup \Sigma_2)$, which means that we do not allow a shared interaction to have a higher priority than any other interaction. This is **required** for the soundness proof of the assume-guarantee rule, as we also explained later that we will **immediately lose soundness by relaxing this restriction**. For $i = 1, 2$, the system

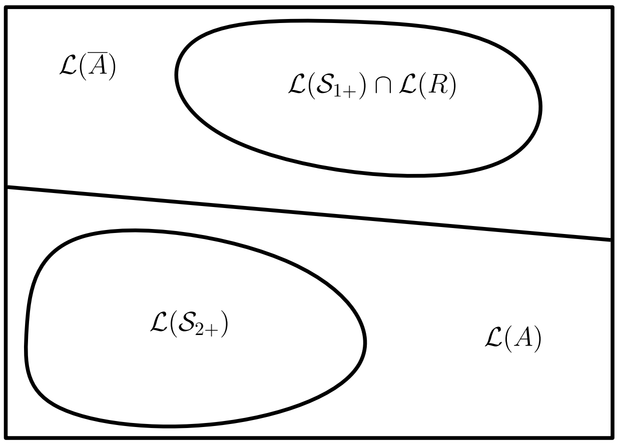

$\mathcal{L}(\overline{A})$
$\mathcal{L}(\mathcal{S}_{1+}) \cap \mathcal{L}(R)$
$\mathcal{L}(\mathcal{S}_{2+})$
$\mathcal{L}(A)$

Figure 4.5: The relation between the languages.

$\mathcal{S}_{i+} = (C_i \cup \{d_i\}, \Sigma, (\mathcal{P} \cap \Sigma \times \Sigma_i) \cup \mathcal{P}_i)$ is obtained by (1) adding priority rules $\mathcal{P}_i \subseteq \Sigma \times \Sigma_i$ to $\mathcal{S}_i$ and, (2) in order to simulate stuttering transitions, adding a component d_i that contains only one location with self-loop transitions labeled with symbols in Σ_{3-i} (the local alphabet of the other set of components). Then the following assume-guarantee rule can be used to decompose the synthesis task into two smaller sub-tasks:

$$\begin{array}{rcll} \mathcal{L}(\mathcal{S}_{1+}) \cap \mathcal{L}(R) \cap \mathcal{L}(A) & = & \emptyset & (a) \\ \mathcal{L}(\mathcal{S}_{2+}) \cap \mathcal{L}(\overline{A}) & = & \emptyset & (b) \\ \hline \mathcal{L}(\mathcal{S}_{+}) \cap \mathcal{L}(R) & = & \emptyset & (c) \end{array}$$

The above assume-guarantee rule says that $\mathcal{S}_+$ is B-Safe with respect to $\mathcal{L}(R)$ iff there exists an assumption automaton A such that (1) $\mathcal{S}_{1+}$ is B-Safe with respect to $\mathcal{L}(R) \cap \mathcal{L}(A)$ and (2) $\mathcal{S}_{2+}$ is B-Safe with respect to $\mathcal{L}(\overline{A})$, where $\overline{A}$ is the complement of A, $\mathcal{P}_+ = \mathcal{P}_1 \cup \mathcal{P}_2$ and no conflict in $\mathcal{P}_1$ and $\mathcal{P}_2$. In the following, we prove the above assume-guarantee rule is both sound and complete. Nevertheless, it is unsound for deadlock freeness. An example can be found at the beginning of Section 4.5.

Theorem 3 (Soundness). *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two non-conflicting priority rules, A be the assumption automaton, R be the risk specification automaton, $\mathcal{S}_{1+} = (C_1 \cup \{d_1\}, \Sigma, (\mathcal{P} \cap \Sigma \times \Sigma_1) \cup \mathcal{P}_1)$, and $\mathcal{S}_{2+} = (C_2 \cup \{d_2\}, \Sigma, (\mathcal{P} \cap \Sigma \times \Sigma_2) \cup \mathcal{P}_2)$, where $\mathcal{P}_i \subseteq \Sigma \times \Sigma_i$ for $i = 1, 2$ and $\mathcal{P} \subseteq \Sigma \times (\Sigma_1 \cup \Sigma_2)$. If $\mathcal{L}(\mathcal{S}_{1+}) \cap \mathcal{L}(R) \cap \mathcal{L}(A) = \emptyset$ and $\mathcal{L}(\mathcal{S}_{2+}) \cap \mathcal{L}(\overline{A}) = \emptyset$. The priority rule $\mathcal{P}_1 \cup \mathcal{P}_2$ ensures that the system $\mathcal{S} = (C_1 \cup C_2, \Sigma, \mathcal{P})$ is B-Safe with respect to R.*

Proof. First, from $\mathcal{L}(\mathcal{S}_{1+}) \cap \mathcal{L}(R) \cap \mathcal{L}(A) = \emptyset$ and $\mathcal{L}(\mathcal{S}_{2+}) \cap \mathcal{L}(\overline{A}) = \emptyset$, we can obtain the relation between those languages described in Figure 4.5. From the figure, one can see that the two languages $\mathcal{L}(\mathcal{S}_{1+}) \cap \mathcal{L}(R)$ and $\mathcal{L}(\mathcal{S}_{2+})$ are disjoint. This follows that $\mathcal{L}(\mathcal{S}_{1+}) \cap \mathcal{L}(R) \cap \mathcal{L}(\mathcal{S}_{2+}) = \emptyset$. By Lemma 3, we have $\mathcal{L}(\mathcal{S}_{+}) \cap \mathcal{L}(R) \subseteq \mathcal{L}(\mathcal{S}_{1+}) \cap \mathcal{L}(\mathcal{S}_{2+}) \cap \mathcal{L}(R) = \emptyset$. Hence the set of priorities $\mathcal{P}_1 \cup \mathcal{P}_2$ ensures that $\mathcal{S}$ is B-Safe with respect to R. $\square$

Lemma 3 (Composition). *Let $\mathcal{S}_1 = (C_1 \cup \{d_1\}, \Sigma, \mathcal{P}_1)$, and $\mathcal{S}_2 = (C_2 \cup \{d_2\}, \Sigma, \mathcal{P}_2)$, and $\mathcal{S}_{1+2} = (C_1 \cup C_2, \Sigma, \mathcal{P}_1 \cup \mathcal{P}_2)$ be three systems, where $\mathcal{P}_i \subseteq \Sigma \times \Sigma_i$ for $i = 1, 2$. We have $\mathcal{L}(\mathcal{S}_{1+2}) \subseteq \mathcal{L}(\mathcal{S}_1) \cap \mathcal{L}(\mathcal{S}_2)$.*

Proof. For a word $w = \sigma_1, \ldots, \sigma_n \in \mathcal{L}(\mathcal{S}_{1+2})$, we consider inductively from the first interaction. If σ_1 is enabled in the initial configuration $(l_1, v_1, \ldots, l_n, v_n, \ldots l_m, v_m)$ of $\mathcal{S}_{1+2}$, then according to Definition 6, we have (1) if σ_1 is in the interaction alphabet of component $c_i \in C_1 \cup C_2$, then there exist a transition $(l_i, g_i, \sigma_1, f_i, l'_i)$ in c_i such that $g_i(v_i) = \texttt{True}$ and (2) there exists no transition $(l_i, g_i, \sigma', f_i, l'_i)$ in components of C_1 and C_2 such that $g_i(v_i) = \texttt{True}$ and $(\sigma_1, \sigma') \in \mathcal{P}_1 \cup \mathcal{P}_2$.

We want to show that σ_1 is also enabled in the initial configuration of $\mathcal{S}_1$. In order to do this, we have to prove (1) components in $C_1 \cup \{d_1\}$ can move with σ_1 and (2) there exists no transition $(l_i, g_i, \sigma', f_i, l'_i)$ in $C_1 \cup \{d_i\}$ such that $g_i(v_i) = \texttt{True}$, l_i is an initial location, and $(\sigma_1, \sigma') \in \mathcal{P}_1$.

- For (1), we consider the following cases: (a) If $\sigma_1 \in \Sigma_{12}$, components of C_1 can move with σ_1 and d_1 can move with σ_1 via a self-loop transition. (b) If $\sigma_1 \in \Sigma_1$, components of C_1 can move with σ_1 and it is not an interaction of d_1. (c) If $\sigma_1 \in \Sigma_2$, it is not an interaction of C_1 and d_1 can move with σ_1 via a self-loop transition. Therefore, components in $C_1 \cup \{d_1\}$ can move with σ_1.
- For (2), first, it is not possible to have such a transition in any component of C_1 by the definition of $\mathcal{S}_{1+2}$ and Definition 6. Then, if the transition is in d_i, we have $\sigma' \in \Sigma_2$ and it follows that $(\sigma, \sigma') \notin \mathcal{P}_1 \subseteq \Sigma \times \Sigma_1$.

By the above arguments for (1) and (2), σ_1 is enabled in the initial configuration of $\mathcal{S}_1$. By a similar argument, σ_1 is also enabled in the initial configuration of $\mathcal{S}_2$.

The inductive step can be proved using the same argument. Thus $w \in \mathcal{L}(\mathcal{S}_1)$ and $w \in \mathcal{L}(\mathcal{S}_2)$. It follows that $\mathcal{L}(\mathcal{S}_{1+2}) \subseteq \mathcal{L}(\mathcal{S}_1) \cap \mathcal{L}(\mathcal{S}_2)$. □

Theorem 4 (Completeness). *Let $\mathcal{S}_+ = (C, \Sigma, \mathcal{P} \cup \mathcal{P}_+)$ be a system and R be the risk specification automaton. If $\mathcal{L}(\mathcal{S}_+) \cap \mathcal{L}(R) = \emptyset$, then there exists an assumption automaton A, system components C_1 and C_2 such that $C = C_1 \cup C_2$, $C_1 \cap C_2 = \emptyset$, and two non-conflicting priority rules $\mathcal{P}_1 \subseteq \Sigma \times \Sigma_1$ and $\mathcal{P}_2 \subseteq \Sigma \times \Sigma_2$ such that $\mathcal{L}(C_1 \cup \{d_1\}, \Sigma, \mathcal{P} \cup \mathcal{P}_1) \cap \mathcal{L}(R) \cap \mathcal{L}(A) = \emptyset$, $\mathcal{L}(C_2 \cup \{d_2\}, \Sigma, \mathcal{P} \cup \mathcal{P}_2) \cap \mathcal{L}(\overline{A}) = \emptyset$, and $\mathcal{P}_+ = \mathcal{P}_1 \cup \mathcal{P}_2$.*

Proof. Can be proved by taking $C_1 = C$, $C_2 = \emptyset$, A as an automaton that recognizes Σ^*, $\mathcal{P}_1 = \mathcal{P}_+$, and $\mathcal{P}_2 = \emptyset$. □

Below we give an example that if we allow the priority $\mathcal{P}$ to be any relation between the interactions, then the assume-guarantee rule we used is unsound. The key is that Lemma 3 will no longer be valid with the relaxed constraints to the priority. In Figure 4.6, both C_1 and C_2 has only one components, $\Sigma_1 = \emptyset$, $\Sigma_2 = \{c\}$, and $\Sigma_{12} = \{a, b\}$. Assume that we have the priority rule $\mathcal{P} = \{b \prec a\}$ in $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}$. Then we get $\mathcal{L}(\mathcal{S}_1) = \{a\}$, $\mathcal{L}(\mathcal{S}_2) = \{b + ca\}$, which implies $\mathcal{L}(\mathcal{S}_1) \cap \mathcal{L}(\mathcal{S}_2) = \emptyset$. However, $\mathcal{L}(\mathcal{S}) = \{b\}$. Then we found a counterexample for Lemma 3. This produces a counterexample of the soundness of the assume-guarantee rule. With a risk specification $\mathcal{L}(R) = \{b\}$, an assumption automaton $\mathcal{L}(A) = \Sigma^*$, and priorities $\mathcal{P} = \mathcal{P}_1 = \mathcal{P}_2 = \{b \prec a\}$, the subtasks

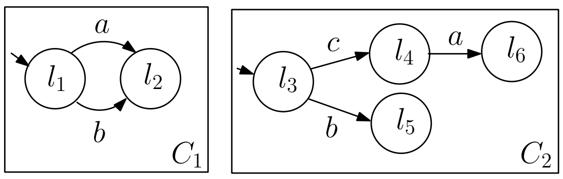


Figure 4.6: A counterexample when we allow a shared interaction to have higher priority than others.

of the assume-guarantee rule can be proved to be B-Safe. However, the system $\mathcal{S}$ is not B-Safe with respect to $\mathcal{L}(R)$. The reason why Σ_{12} can not be placed on the right-hand side of $\mathcal{P}$, $\mathcal{P}_1$, and $\mathcal{P}_2$ is because even in the subsystem a shared interaction can block other interactions successfully, when composing two systems together, it may no longer block other interactions (as now they need to be paired).

Notice that (1) the complexity of a synthesis task is NP-complete in the number of states in the risk specification automaton product with the size of the system and (2) $|\mathcal{S}|$ is approximately equals to $|\mathcal{S}_1| \times |\mathcal{S}_2|$[6]. Consider the case that one decomposes the synthesis task of $\mathcal{S}$ with respect to $\mathcal{L}(R)$ into two subtasks using the above assume-guarantee rule. The complexity original synthesis task is NP-complete in $|\mathcal{S}| \times |R|$ and the complexity of the two sub-tasks are $|\mathcal{S}_1| \times |R| \times |A|$ and $|\mathcal{S}_2| \times |A|$[7], respectively. Therefore, if one managed to find a small assumption automaton A for the assume-guarantee rule, the complexity of synthesis can be greatly reduced. We propose to use the machine learning algorithm L* [Ang87] to automatically find a small automaton that is suitable for compositional synthesis. Next, we will first briefly describe the L* algorithm and then explain how to use it for compositional synthesis.

The L* algorithm works iteratively to find a minimal deterministic automaton recognizing a target regular language U. It assumes a *teacher* that answers two types of queries: (a) *membership queries* on a string w, where the teacher returns *true* if w is in U and *false* otherwise, (b) *equivalence queries* on an automaton A, where the teacher returns *true* if $\mathcal{L}(A) = U$, otherwise it returns *false* together with a counterexample string in the difference of $\mathcal{L}(A)$ and U. In the i-th iteration of the algorithm, the L* algorithm acquires information of U by posing membership queries and guess a candidate automaton A_i. The correctness of the A_i is then verified using an equivalence query. If A_i is not a correct automaton (i.e., $\mathcal{L}(A) \neq U$), the counterexample returned from the teacher will be used to refine the conjecture automaton of the $(i+1)$-th iteration. The learning algorithm is guaranteed to converge to the minimal deterministic finite state automaton of U in a polynomial number of iterations[8]. Also the sizes of conjecture automata increase strictly monotonically with respect to the number of iterations (i.e., $|A_{i+1}| > |A_i|$ for all $i > 0$).

The flow of our compositional synthesis is in Figure 4.7. Our idea of compositional synthesis via learning is the following. We use the notations $\mathcal{S}_i^+$ to denote the system

[6]This is true only if the size of the alphabet is much smaller than the number of reachable configurations.

[7]Since A is deterministic, the sizes of A and its complement $\overline{A}$ are identical.

[8]In the size of the minimal deterministic finite state automaton of U and the longest counterexample returned from the teacher.

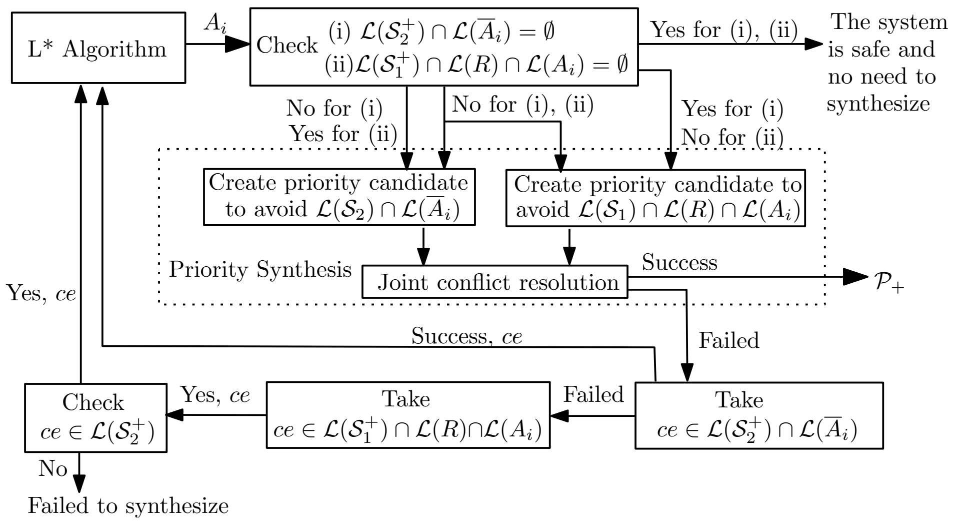

L* Algorithm
A_i
Check (i) $\mathcal{L}(\mathcal{S}_2^+) \cap \mathcal{L}(\overline{A_i}) = \emptyset$ (ii) $\mathcal{L}(\mathcal{S}_1^+) \cap \mathcal{L}(R) \cap \mathcal{L}(A_i) = \emptyset$
Yes for (i), (ii)
The system is safe and no need to synthesize
No for (i) Yes for (ii)
No for (i), (ii)
Yes for (i) No for (ii)
Create priority candidate to avoid $\mathcal{L}(\mathcal{S}_2) \cap \mathcal{L}(\overline{A_i})$
Create priority candidate to avoid $\mathcal{L}(\mathcal{S}_1) \cap \mathcal{L}(R) \cap \mathcal{L}(A_i)$
Priority Synthesis
Joint conflict resolution
Success
$\mathcal{P}_+$
Yes, ce
Success, ce
Failed
Check $ce \in \mathcal{L}(\mathcal{S}_2^+)$
Yes, ce
Take $ce \in \mathcal{L}(\mathcal{S}_1^+) \cap \mathcal{L}(R) \cap \mathcal{L}(A_i)$
Failed
Take $ce \in \mathcal{L}(\mathcal{S}_2^+) \cap \mathcal{L}(\overline{A_i})$
No
Failed to synthesize

Figure 4.7: The flow of the assume-guarantee priority synthesis.

$\mathcal{S}_i$ equipped with a stuttering component. First we use L* to learn the language $\mathcal{L}(\mathcal{S}_2^+)$. Since the transition system induced from the system $\mathcal{S}_2^+$ has finitely many states, one can see that $\mathcal{L}(\mathcal{S}_2^+)$ is regular. For a membership query on a word w, our algorithm simulates it symbolically on $\mathcal{S}_2^+$ to see if it is in $\mathcal{L}(\mathcal{S}_2^+)$. Once the L* algorithm poses an equivalence query on a deterministic finite automaton A_i, our algorithm tests conditions $\mathcal{L}(\mathcal{S}_1^+) \cap \mathcal{L}(R) \cap \mathcal{L}(A_i) = \emptyset$ and $\mathcal{L}(\mathcal{S}_2^+) \cap \mathcal{L}(\overline{A_i}) = \emptyset$ one after another. So far, our algorithm looks very similar to the compositional verification algorithm proposed in [CGP03]. There are a few possible outcomes of the above test

1. Both condition holds and we proved the system is B-Safe with respect to $\mathcal{L}(R)$ and no synthesis is needed.
2. At least one of the two conditions does not hold. In such case, we try to synthesize priority rules to make the system B-Safe (see the details below).
3. If the algorithm fails to find usable priority rules, we have two cases:
   a) The algorithm obtains a counterexample string ce in $\mathcal{L}(\mathcal{S}_1^+) \cap \mathcal{L}(R) \setminus \mathcal{L}(\overline{A_i})$ from the first condition. This case is more complicated. We have to further test if $ce \in \mathcal{L}(\mathcal{S}_2^+)$. A negative answer implies that ce is in $\mathcal{L}(A_i) \setminus \mathcal{L}(\mathcal{S}_2^+)$. This follows that ce can be used by L* to refine the next conjecture. Otherwise, our algorithm terminates and reports not able to synthesize priority rules.
   b) The algorithm obtains a counterexample string ce in $\mathcal{L}(\mathcal{S}_2^+) \setminus \mathcal{L}(A_i)$ from the second condition, in such case, ce can be used by L* to refine the next conjecture.

The deterministic finite state automata R, A_i, and also its complement $\overline{A_i}$ can be treated as components without data and can be easily encoded symbolically using the approach in Section 4.4.1. Also the two conditions can be tested using standard symbolic reachability algorithms.

Compositional Synthesis

Recall that our goal is to find a set of suitable priority rules via a small automaton A_i. Therefore, before using the ce to refine and obtain the next conjecture A_{i+1}, we first attempt to synthesis priority rules using A_i as the assumption automaton. Synthesis algorithms in previous sections can then be applied separately to the system composed of $\{\mathcal{S}_1^+, R, A_i\}$ and the system composed of $\{\mathcal{S}_2^+, \overline{A_i}\}$ to obtain two non-conflicting priority rules $\mathcal{P}_{1i} \subseteq (\Sigma_1 \cup \Sigma_{12}) \times \Sigma_1$ and $\mathcal{P}_{2i} \subseteq (\Sigma_2 \cup \Sigma_{12}) \times \Sigma_2$. Then $\mathcal{P}_{1i} \cup \mathcal{P}_{2i}$ is the desired priority for $\mathcal{S}$ to be B-Safe with respect to R. To be more specific, we first compute the CNF formulae f_1 and f_2 (that encode all possible priority rules that are *local*, i.e., we remove all non-local priority candidates) of the two systems separately using the algorithms in Section 4.4, and then check satisfiability of $f_1 \wedge f_2$. The priority rules $\mathcal{P}_{1i}$ and $\mathcal{P}_{2i}$ can be derived from the satisfying assignment of $f_1 \wedge f_2$.

4.7 Evaluation: The VISSBIP toolkit

We implemented the presented algorithms (except connection the data abstraction module in D-Finder [BGL$^+$11]) in the VISSBIP[9] tool and performed experiments to evaluate them. In this section, we first illustrate how to use VISSBIP to construct and synthesize systems, followed by presenting our experimental results.

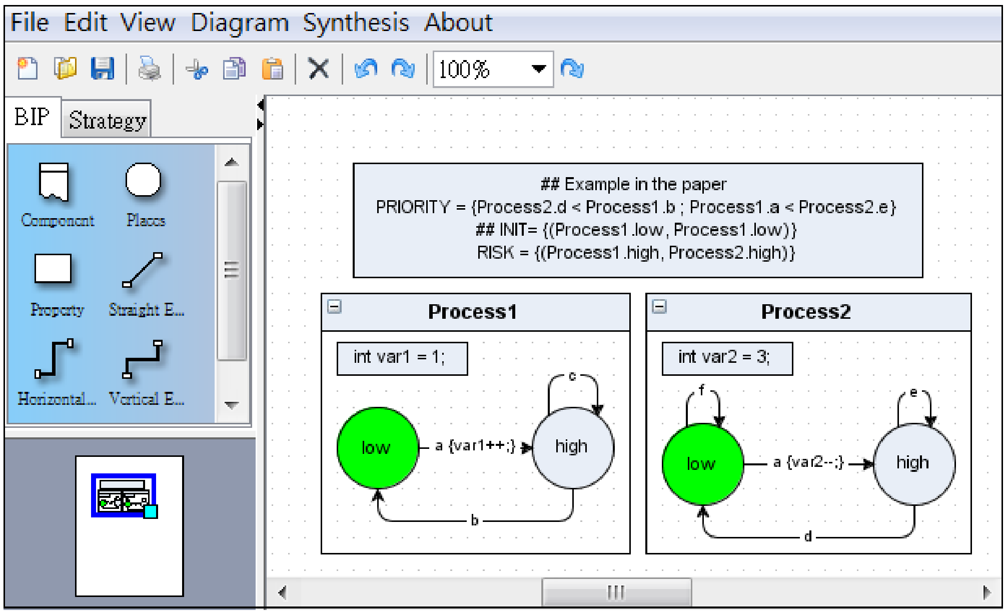
4.7.1 Visualizing Simple Interaction Systems

The user can construct a system using the drag-and-drop functionality of VISSBIP's graphical user interface shown in Figure 4.8. BIP objects (components, places, properties, and edges) can be simply dragged from the menu on the left to the drawing window on the right.

We use the system shown in Figure 4.8 to illustrate how a system is represented. The system consists of two components (`Process1` and `Process2`) represented as boxes. Each component has two places (`high` and `low`) and a local variable (`var1` and `var2`, respectively). A place (also called location) is represented by a circle. A green circle indicates that this place is an initial location of a behavioral component. E.g., place `low` is marked as initial in `Process1`. Squares denotes variables definitions and their initialization within a component. E.g., `var1` and `var2` are both initialized to 1. Edges between two locations represent *transitions.* Each transition is of the format `{precondition} port-name {postcondition}`. E.g., the transition of `Process1` from place `low` to `high` is labeled with port name `a` and upon its execution the value of `var1` is increased by 1. For simplicity we use **port-name bindings** to construct interactions between components, i.e., that transitions using the same port name are automatically grouped to a single interaction and are executed jointly[10]. In the following, we refer to an interaction by its port name. Finally, addi-

[9]Available for download at `http://www6.in.tum.de/~chengch/vissbip`

[10]It is possible to pass data through an interaction. The user specifies the data flow associated to an interaction in the same way she describes priorities. For details, we refer readers to the manual



```
## Example in the paper
PRIORITY = {Process2.d < Process1.b ; Process1.a < Process2.e}
## INIT= {(Process1.low, Process1.low)}
RISK = {(Process1.high, Process2.high)}
```

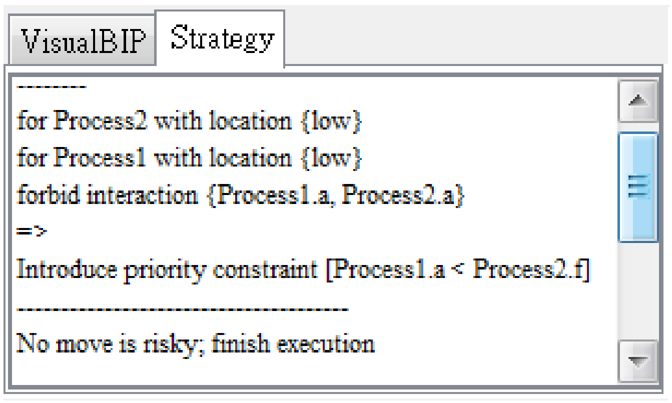
Figure 4.8: Model construction using VISSBIP.

tional squares outside of any component, are used to define system properties such as priorities over interactions and winning conditions (for synthesis or verification). In particular, we use the keyword `PRIORITY` to state priorities. E.g., the statement `Process2.d < Process1.b` means that whenever interactions `b` and `d` are available, the BIP engine always executes `b`. The keyword `RISK` is used to state risk conditions. E.g., the condition `RISK = {(Process1.high, Process2.high)}` states that the combined location pair `(Process1.high, Process2.high)` is never reached. Apart from the stated conditions, we also implicitly require that the system is deadlock-free, i.e., at anytime, at least one interaction is enabled. When only deadlock avoidance is required, the keyword `DEADLOCK` can be used instead. Lines started with `##` are comments.

4.7.2 Safety synthesis by adding global priorities: examples

The user can invoke the synthesis engine on a system like to one shown in Figure 4.8. The engine responds in one of the following three ways: (1) It reports that no additional priorities are required. (2) It returns a set priority constraints that ensure the stated property. (3) It states that no solution based on priorities can be found by the engine.

Figure 4.9 shows the *Strategy* panel of VISSBIP, which displays the results obtained by invoking the synthesis engine on the example of Figure 4.8. Recall, that in the example, we stated that the combined location pair `(Process1.high, Process2.high)` is never reached. The engine reports that the priority constraint `Process1.a < Process2.f` should be added. Note that if the system is in state (`Process1.low, Process2.low`), then the interaction `Process1.a` (which is a joint action from Process 1 and Process 2) would immediately leads to a risk state (a state sat-



```
--------
for Process2 with location {low}
for Process1 with location {low}
forbid interaction {Process1.a, Process2.a}
=>
Introduce priority constraint [Process1.a < Process2.f]
----------------------------------------
No move is risky; finish execution
```

Figure 4.9: The automatically synthesized priority for the model in Figure 4.8.

isfying the risk condition). This can be avoided by executing `Process2.f` first. The new priority ensures that interaction `f` is executed forever, which is also deadlock-free.

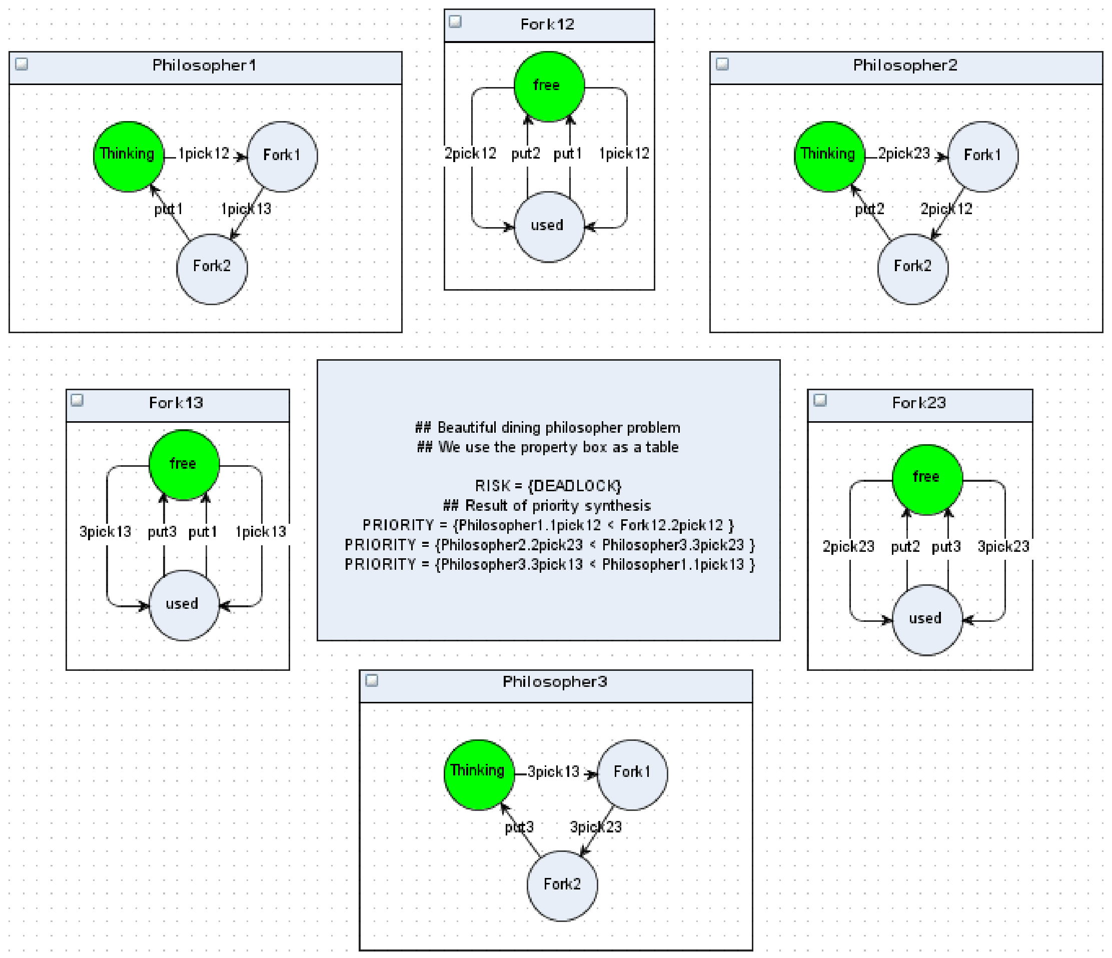
4.7.3 Example: Dining Philosophers Problem Revisited

We continue with the second example: How can priority synthesis be used as a technique to automatically generate solutions for the dining philosophers problem - a classic problem concerning resource contention among multiple processes.

The system modeled using VISSBIP in Figure 4.10 represents three philosophers sitting around a table (the property square), where one fork is located between two philosophers. As modeled in Figure 4.10, each philosopher always takes his/her left fork first. Once he has the left fork, he takes his right fork. It can be observed that philosophers may create deadlock when each one is holding a fork and is waiting for others to release the fork.

VISSBIP generates three priorities, which can be viewed as politeness rules to yield others using the fork: *At any instance, each philosopher should wait to take his left fork if he found that the philosopher to his left is able to take the fork between them.* With some pondering, it can be proved that the three priority rules specified in Figure 4.10 are sufficient to avoid deadlock.

This example indicates the potential of priority synthesis for resource protection problems. For the set of priority fixes generated by VISSBIP (like the rules for philosophers), it can be further refined as distributed / centralized protocols to orchestrate components achieving goal-oriented behavior.



```
## Beautiful dining philosopher problem
## We use the property box as a table

RISK = {DEADLOCK}
## Result of priority synthesis
PRIORITY = {Philosopher1.1pick12 < Fork12.2pick12 }
PRIORITY = {Philosopher2.2pick23 < Philosopher3.3pick23 }
PRIORITY = {Philosopher3.3pick13 < Philosopher1.1pick13 }
```

Figure 4.10: The dining philosopher problem modeled using VISSBIP.

4.7.4 Evaluation

To observe how our algorithm scales, in Table 5.7 we summarize results of synthesizing priorities for the dining philosophers problem[11]. Our preliminary result in [CBJ$^+$11] fails to synthesize priorities when the number of philosophers is greater than 15 (i.e., a total of 30 components), while currently we are able to solve problems of 50 within reasonable time. By analyzing the bottleneck, we found that 50% of the execution time are used to construct clauses for transitive closure, which can be easily parallelized. Also the synthesized result (i) does not starve any philosopher and (ii) ensures that each philosopher only needs to observe his left and right philosopher, making the resulting priority very desirable. Contrarily, it is possible to select a subset of components and ask to synthesize priorities for deadlock freedom using alphabet abstraction. The execution time using alphabet abstraction depends on the number of selected components; in our case we select 4 components thus is executed extremely fast. Of course,

[11]Evaluated under Intel 2.93GHz CPU with 2048Mb RAM for JVM.

Table 4.1: Experimental results

	Time (seconds)				# of BDD variables			
Problem	NFM[1]	Opt.[2]	Ord.[3]	Abs.[4]	NFM	Opt.	Ord.	Abs.
Phil. 10	0.813	0.303	0.291	0.208	202	122	122	38
Phil. 20	-	86.646	0.755	0.1	-	242	242	38
Phil. 25	-	-	1.407	0.193	-	-	302	38
Phil. 30	-	-	3.74	0.235	-	-	362	38
Phil. 35	-	-	5.913	0.185	-	-	422	38
Phil. 40	-	-	10.21	0.197	-	-	482	38
Phil. 45	-	-	18.344	0.231	-	-	542	38
Phil. 50	-	-	30.384	0.288	-	-	602	38
DPU v1	5.335	0.299	x	x	168	116	x	x
DPU v2	4.174	0.537	1.134[R]	x	168	116	116[R]	x
Traffic	x	x	0.651	x	x	x	272	x

[1] Engine based on [CBJ+11].
[2] Dense variable encoding
[3] Opt. + initial variable ordering
[4] Alphabet abstraction
[R] Priority repushing
[x] Not evaluated
[-] Timeout / not evaluated

the synthesized result is not very satisfactory, as it starves certain philosopher. Nevertheless, this is unavoidable when overlooking interactions done by other philosophers. Except the traditional dining philosophers problem, we have also evaluated on (i) a BIP model (5 components) for data processing in digital communication (DPU; See Appendix 4.10.2 for description), (ii) a simplified protocol of automatic traffic control (Traffic), and (iii) an antenna model for the Dala robot. The antenna model has 25 components, and the engine is able to synthesize priorities to avoid deadlock within 30 seconds. Our preliminary evaluation on compositional priority synthesis is in Appendix 4.10.3.

4.8 Related Work

For deadlock detection, well-known model checking tools such as SPIN [Hol04] and NuSMV [CCGR99] support deadlock detection by given certain formulas to specify the property. D-Finder [BGL+11] applies compositional and incremental methods to compute invariants for an over-approximation of reachable states to verify deadlock-freedom automatically. Nevertheless, all the above tools do not provide any deadlock avoidance strategies when real deadlocks are detected.

Synthesizing priorities is subsumed by the framework of controller synthesis proposed by Ramadge and Wohnham [RW89], where the authors proposed an automata-theoretical approach to restrict the behavior of the system (the modeling of environment is also possible). Essentially, when the environment is modeled, the framework computes the risk attractor and creates a centralized controller. Similar results using centralized control can be dated back from [BHG$^+$93] to the recent work by Autili et al [AINT07] (the SYNTHESIS tool). Nevertheless, the centralized coordinator forms a major bottleneck for system execution. Transforming a centralized controller to distributed controllers is difficult, as within a centralized controller, the execution of a local interaction of a component might need to consider the configuration of all other components.

Priorities, as they are stateless, can be distributed much easier for performance and concurrency. E.g., the synthesized result of dining philosophers problem indicates that each philosopher only needs to watch his left and right philosophers without considering all others. We can continue with known results from the work of Graf et al. [GPQ10] to distribute priorities, or partition the set of priorities to multiple controllers under layered structure to increase concurrency (see work by Bonakdarpour et al. [BBQ11]). Our algorithm can be viewed as a step forward from centralized controllers to distributed controllers, as architectural constraints (i.e., visibility of other components) can be encoded during the creation of priority candidates. Therefore, we consider the recent work of Abujarad et al.[ABK09] closest to ours, where they proceeds by performing distributed synthesis (known to be undecidable [PR90]) directly. In their model, they take into account the environment (which they refer it as faults), and consider handling deadlock states by either adding mechanisms to recover from them or preventing the system to reach it. It is difficult to compare two approaches directly, but we give hints concerning performance measure: (i) Our methodology and implementation works on game concept, so the complexity of introducing the environment does not change. (ii) In [ABK09], for a problem of 10^{33} states, under 8-thread parallelization, the total execution time is 3837 seconds, while resolving the deadlock of the 50 dining philosophers problem (a problem of 10^{38} states) is solved within 31 seconds using our monolithic engine.

Lastly, the research of deadlock detection and mechanisms of deadlock avoidance is an important topic within the community of Petri nets (see survey paper [LZW08] for details). Concerning synthesis, some theoretical results are available, e.g., [IMA02], but efficient implementation efforts are, to our knowledge, lacking.

4.9 Summary

In this chapter, we explain the underlying algorithm for priority synthesis and propose extensions to synthesize priorities for more complex systems. Figure 4.11 illustrates a potential flow of priority synthesis. A system can be first processed using data abstraction to create models suitable for our analysis framework. Besides the monolithic engine, two complementary techniques are available to further reduce the complexity of problem under analysis. Due to the stateless property and the fact that they preserve deadlock-freedom, priorities can be relatively easy implemented in a distributed setting. Concretizing the concept, we implement VISSBIP for constructing simple BIP systems together with a technique called priority synthesis, which automatically adds priorities over interactions to maintain system safety and deadlock-freedom.

Lastly, as priority synthesis is essence a method of *synthesizing simple component glues*

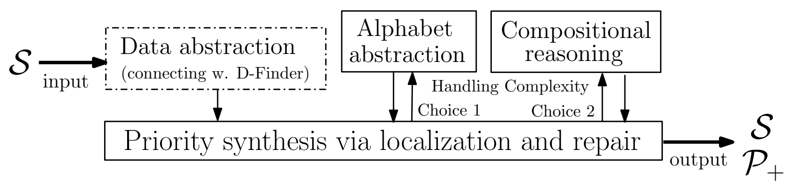


Figure 4.11: The framework of priority synthesis presented in this chapter, where the connection with the D-Finder tool [BGL+11] is left for future work.

for conflict resolution, under suitable modifications, our technique is applicable for multicore/manycore systems working on task models for resource protection, which is our next step.

4.10 Appendix

4.10.1 On the Hardness of Priority Synthesis

In this section, we formulate priority synthesis for BIP systems using the automata-theoretic framework proposed by Ramadge and Wonham [RW89]. In this framework, priority synthesis results in searching for a supervisor from the restricted class of supervisors, in which each is solidly expressible using priorities. While priority-based supervisors are easier to use, e.g., they support the construction of distributed protocols, they are harder to compute. Then we focus on the hardness of synthesizing priorities and show that finding a supervisor based on priorities that ensures deadlock freedom of the supervised system is NP-complete.

4.10.1.1 Formulating BIP Models and Priority Synthesis based on Transition Systems

We first translate simple BIP models (without variables) into automata, i.e., the logical discrete-event system (DES) model in [RW89]. Given a simple BIP model, we can always construct the transition system representing the asynchronous product of its components. We follow the definitions in [RW89] to simplify a comparison between priority synthesis and the controller synthesis technique.

Definition 15 (Transition System)**.** *We define a transition system (called a logical DES model or generator in [RW89]) as a tuple $G = (Q, \Sigma, q_0, \delta)$, where*

- *Q is a finite set of* states,
- *Σ is a finite set of* event or interaction labels, *called* interaction alphabet,
- *q_0 is the* initial state, *i.e., $q_0 \in Q$,*
- *$\delta : Q \times \Sigma \to Q \cup \{\perp\}$ is a* transition function *mapping a state and an interaction label to a successor state or a distinguished symbol $\perp$ that indicates that the given state and interaction pair has no successor. If $\delta(q, \sigma) = \perp$ for some $q \in Q$ and $\sigma \in \Sigma$, then we say $\delta(q, \sigma)$ is* undefined. *We slightly abuse the notation and extend δ to sequences of interactions in the usual way, i.e., $\delta(q, \epsilon) = q$ and $\delta(q, w\sigma) = \delta(\delta(q, w), \sigma)$ with $w \in \Sigma^*$ and $\sigma \in \Sigma$.*

Denote the size of the transition system to be $|Q| + |\Sigma| + |\delta|$.

Figure 4.12 illustrates the transition system for the BIP model in Figure 4.8. Transitions in dashed lines are blocked by the priorities. Note that for the formulation in [RW89], a logical DES model is able to further partition Σ into Σ_c (controllable input) and Σ_u (uncontrollable input), i.e., a transition system can also model a game. For systems translated from BIP models the partition is not required. However, our hardness result of cause applies to the alphabet-partitioned setting as well. We define the **run of** G **on a word** $w = w_0 \ldots w_n \in \Sigma^*$ as the finite sequence of states $q_0 q_1 ... q_{n+1}$ such that for all $i, 0 \leq i \leq n$, $\delta(q_i, w_i) = q_{i+1}$. Note that if $\delta(q_i, w_i)$ is undefined for some i, then there exists no run of G on w. A state $q \in Q$ with no outgoing transitions, i.e., $\forall \sigma \in \Sigma, \delta(q, \sigma) = \perp$, is called **deadlock state**. A system G has a **deadlock** if there exists

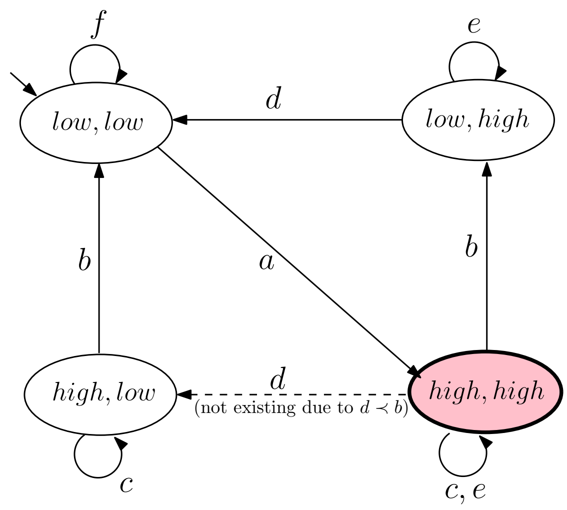


Figure 4.12: The transition system for the BIP model (without variables) in Figure 4.8.

a word w such that the run $q_0 \dots q_{|w|}$ of G on w ends in a deadlock state, i.e., $q_{|w|}$ is a deadlock state.

We now define the concept of **supervisor**, i.e., machinery that controls the execution of the system by suppressing transitions.

Definition 16 (Supervisor). *Given $G = (Q, \Sigma, q_0, \delta)$, a supervisor for G is a function $\mathcal{C} : Q \times \Sigma \to \{\texttt{True}, \texttt{False}\}$. The transition system $G_{\mathcal{C}}$ obtained from G* **under the supervision of** *$\mathcal{C}$ is defined as follows: $G_{\mathcal{C}} = (Q, \Sigma, q_0, \delta_{\mathcal{C}})$ with $\delta_{\mathcal{C}}(q, \sigma) = \delta(q, \sigma) \neq \bot$, if $\mathcal{C}(q, \sigma) = \texttt{True}$, and $\delta_{\mathcal{C}}(q, \sigma) = \bot$ otherwise.*

Definition 17. *Given $G = (Q, \Sigma, q_0, \delta)$, a* ***zero-effect supervisor*** *$\mathcal{C}_\emptyset$ is a supervisor that disables all undefined interactions, i.e., interactions leading to $\bot$. Formally, for all states $q \in Q$ and interactions $\sigma \in \Sigma$, $\mathcal{C}_\emptyset(q, \sigma) = \texttt{False}$ iff $\delta(q, \sigma) = \bot$. Note that $\mathcal{C}_\emptyset$ has no effect on G, i.e., $G_{\mathcal{C}_\emptyset} = G$.*

Given a transition system, adding priorities to the system can be viewed as masking some transitions. The masking can be formulated using supervisors.

Definition 18 (Priorities). *Given an interaction alphabet Σ,* a set of priorities *$\mathcal{P}$ is a finite set of interaction pairs defining a relation $\prec \subseteq \Sigma \times \Sigma$ between the interactions. We called a priority set* legal*, if the relation $\prec$ is (1) transitive and (2) non-reflexive (i.e., there are no circular dependencies) [GS03].*

We are only interested in legal sets, as a supervisor from a non-legal set of priorities may induce more deadlocks over the existing system. Note that given an arbitrary set, we can easily check if there exists a corresponding legal set.

Definition 19 (Priority Supervisor). *Given a transition system $G = (Q, \Sigma, q_0, \delta)$ and a legal priority set $\mathcal{P} = \bigcup_{i=0}^{n} \sigma_i \prec \sigma_i'$*[12] *with $\sigma_i, \sigma_i' \in \Sigma$, we define the corresponding supervisor $\mathcal{C}_{\mathcal{P}}$ inductively over the number of priority pairs as follows:*

[12]We write $\sigma_i \prec \sigma_i'$ instead of (σ_i, σ_i') to emphasize that priorities are not symmetric.

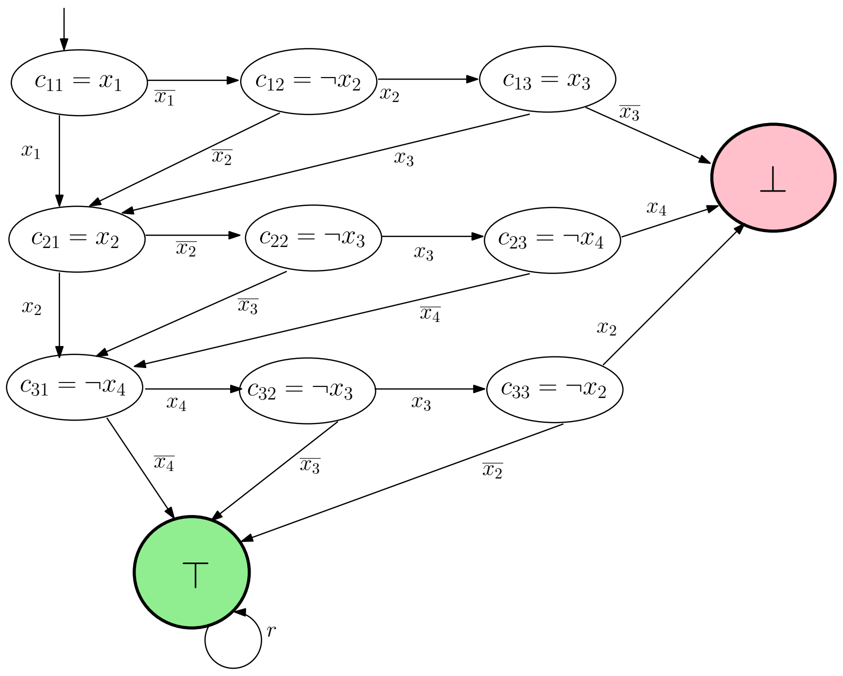


Figure 4.13: The reduced system from the 3SAT instance $\phi = c_1 \wedge c_2 \wedge c_3$, where $c_1 := (x_1 \vee \neg x_2 \vee x_3), c_2 := (x_2 \vee \neg x_3 \vee \neg x_4), c_3 := (\neg x_4 \vee \neg x_3 \vee \neg x_2)$.

- *Base case:* $\mathcal{C}_{\mathcal{P}} = \mathcal{C}_{\emptyset}$, *if* $\mathcal{P} = \{\}$
- *Inductive step: Let* $\mathcal{P}' = \mathcal{P} \cup \{\sigma_k \prec \sigma'_k\}$, *then for all state* $q \in Q$, *if* $\mathcal{C}_{\mathcal{P}}(q, \sigma_k) = \mathcal{C}_{\mathcal{P}}(q, \sigma'_k) = \texttt{True}$, *then* $\mathcal{C}_{\mathcal{P}'}(q, \sigma'_k) = \texttt{False}$ *and for all interactions* $\sigma \neq \sigma'_k$: $\mathcal{C}_{\mathcal{P}'}(q, \sigma) = \mathcal{C}_{\mathcal{P}}(q, \sigma)$, *otherwise for all* $\sigma \in \Sigma : \mathcal{C}_{\mathcal{P}'}(q, \sigma) = \mathcal{C}_{\mathcal{P}}(q, \sigma)$.

Definition 20 (Safety)**.** *Given a transition system* $G = (Q, \Sigma, q_0, \delta)$ *and the set of risk states* $Q_{risk} \subseteq Q$, *the system is* ***safe*** *if the following conditions holds.*

- **(Risk-free)** $\forall w \in \Sigma^*$, *if* $\delta(q_0, w) \neq \perp$, *then* $\delta(q_0, w) \notin Q_{risk}$
- **(Deadlock-free)** $\forall w \in \Sigma^*, \exists \sigma \in \Sigma$ *s.t. if* $\delta(q_0, w) \neq \perp$, *then* $\delta(q_0, w\sigma) \neq \perp$.

A system that is not safe is called ***unsafe****.*

Note that by removing all outgoing transitions for risk states every risk state is also a deadlock state. Therefore, risk-freeness reduces to deadlock-freeness and there is no need to handled it separately.

Definition 21 (Priority Synthesis)**.** *Given a transition system* $G = (Q, \Sigma, q_0, \delta)$, *and the set of risk states* $Q_{risk} \subseteq Q$, *priority synthesis searches for a set of priorities* $\mathcal{P}$ *such that* G *supervised by* $\mathcal{C}_{\mathcal{P}}$ *is safe.*

4.10.1.2 Priority Synthesis is NP-Complete

We now state the main result, i.e., the problem of priority synthesis is NP-complete.

Theorem 5. *Given a transition system $G = (Q, \Sigma, q_0, \delta)$, finding a set $\mathcal{P}$ of priorities such that G under $\mathcal{C}_\mathcal{P}$ is safe is NP-complete in the size of G.*

Proof. Given a set of a priorities $\mathcal{P}$, checking if $G_{\mathcal{C}_\mathcal{P}}$ is safe can be done in polynomial time by a simple graph search in $G_{\mathcal{C}_\mathcal{P}}$ for reachable states that have no outgoing edges. Therefore, the problem is in NP.

For the NP-hardness, we give a polynomial-time reduction from Boolean 3-Satisfiability (3-SAT) to Priority Synthesis. Consider a 3-SAT formula ϕ with the set of variables $X = \{x_1, \ldots, x_n\}$ and the set of clauses $C = \{c_1, \ldots, c_m\}$, where each clause c_i consists of the literals c_{i1}, c_{i2}, and c_{i3}. We construct a transition system $G_\phi = (Q, \Sigma, q_0, \delta)$ using Algorithm 4. The transition system has one state for each literal c_{ji} and two designated states $\top$ and $\bot$, indicating if an assignment satisfies or does not satisfy the formula. For each variable x, the alphabet Σ of G includes two interactions x_i and $\overline{x_i}$ indicating if x is set to true or false, respectively. The transition system consists of m layers. Each layer corresponds to one clause. The transitions allows one to move from layer i to the layer $i + 1$ iff the corresponding clause is satisfied. E.g., consider the 3SAT formula $\phi = c_1 \wedge c_2 \wedge c_3$ with $c_1 := (x_1 \vee \neg x_2 \vee x_3), c_2 := (x_2 \vee \neg x_3 \vee \neg x_4), c_3 := (\neg x_4 \vee \neg x_3 \vee \neg x_2)$, Figure 4.13 shows the corresponding transition system.

We prove that ϕ is satisfiable iff there exists a set of priorities $\mathcal{P}$ such that G_ϕ supervised by $\mathcal{C}_\mathcal{P}$ is safe, i.e., in G_ϕ supervised by $\mathcal{C}_\mathcal{P}$ the state $\bot$ is unreachable.

($\rightarrow$) Assume that ϕ is satisfiable, and let $v : X \rightarrow \{0, 1\}$ be a satisfying assignment. Then, we create the priority set $\mathcal{P}$ as follows:

$$\mathcal{P} := \{\overline{x} \prec x \mid v(x) = 1\} \cup \{x \prec \overline{x} \mid v(x) = 0\}$$

E.g., consider the example in Figure 4.13, a satisfying assignment for ϕ is $v(x_1) = 1$ and $v(x_2) = v(x_3) = v(x_4) = 0$, then we obtain $\mathcal{P} = \{\overline{x_1} \prec x_1, x_2 \prec \overline{x_2}, x_3 \prec \overline{x_3}, x_4 \prec \overline{x_4}\}$.

Recall that G_ϕ under $\mathcal{C}_\mathcal{P}$ is safe iff it never reaches the state $\bot$. In G_ϕ, we can only reach the state $\bot$, if the priorities allows us, in some layer i, to move from c_{i1} to c_{i2} to c_{i3} and from there to $\bot$. This path corresponds to an unsatisfied clause. Since the priorities are generated from a satisfying assignment, in which all clauses are satisfied, there is no layer in which we can move from c_{i1} to $\bot$.

($\leftarrow$) For the other direction, consider a set of priorities $\mathcal{P}$. Let $\mathcal{P}'$ be the set of all priorities in $\mathcal{P}$ that refer to the same variable, i.e., $\mathcal{P}' = \{p \prec q \in \mathcal{P} \mid \exists x \in X : (p = x \wedge q = \overline{x}) \vee (p = \overline{x} \wedge q = x)\}$. Since $\mathcal{P}$ is a valid set of priorities (no circular dependencies), the transition system G_ϕ has the same set of reachable states under $\mathcal{C}_\mathcal{P}$ and under $\mathcal{C}_{\mathcal{P}'}$. There, the state $\bot$ is also avoided with using the set $\mathcal{P}'$. Given $\mathcal{P}'$, we construct a corresponding satisfying assignment as follows:

$$v(x) = \begin{cases} 0 & x \prec \overline{x} \in \mathcal{P}' \\ 1 & \overline{x} \prec x \in \mathcal{P}' \\ 0 & \text{otherwise.} \end{cases}$$

Algorithm 4: Transition System Construction Algorithm

```
Data: 3SAT Boolean formula $\phi$ with $n$ variables and $m$ clauses
Result: Transition System $G_\phi = (Q, \Sigma, q_0, \delta)$
begin
    $Q := \{\top, \bot\}$
    for clause $c_i = (c_{i1} \vee c_{i2} \vee c_{i3})$, $i = 1, \ldots, m$ do
        $Q := Q \cup \{c_{i1}, c_{i2}, c_{i3}\}$
    $\Sigma = \bigcup_{i=1\ldots n} \{x_i, \overline{x_i}\} \cup \{r\}$
    for clause $c_i = (c_{i1} \vee c_{i2} \vee c_{i3})$ with variables $x_{i1}, x_{i2}, x_{i3}$, $i = 1, \ldots, m$ do
        if $i \neq m$ then
            /* Connect the truth assignment to state $c_{(i+1)1}$ */
            if $x_{i1}$ appears positive in $c_{i1}$ then
                $\delta(c_{i1}, x_{i1}) := c_{(i+1)1}; \delta(c_{i1}, \overline{x_{i1}}) := c_{i2}$
            else
                $\delta(c_{i1}, \overline{x_{i1}}) := c_{(i+1)1}; \delta(c_{i1}, x_{i1}) := c_{i2}$
            if $x_{i2}$ appears positive in $c_{i2}$ then
                $\delta(c_{i2}, x_{i2}) := c_{(i+1)1}; \delta(c_{i2}, \overline{x_{i2}}) := c_{i3}$
            else
                $\delta(c_{i2}, \overline{x_{i2}}) := c_{(i+1)1}; \delta(c_{i2}, x_{i2}) := c_{i3}$
            if $x_{i3}$ appears positive in $c_{i3}$ then
                $\delta(c_{i3}, x_{i3}) := c_{(i+1)1}$; $\delta(c_{i3}, \overline{x_{i3}}) := \bot$
            else
                $\delta(c_{i2}, \overline{x_{i3}}) := c_{(i+1)1}$; $\delta(c_{i3}, x_{i3}) := \bot$
        else
            /* Connect the truth assignment to $\top$ */
            if $x_{i1}$ appears positive in $c_{i1}$ then
                $\delta(c_{i1}, x_{i1}) := \top; \delta(c_{i1}, \overline{x_{i1}}) := c_{i2}$
            else
                $\delta(c_{i1}, \overline{x_{i1}}) := \top; \delta(c_{i1}, x_{i1}) := c_{i2}$
            if $x_{i2}$ appears positive in $c_{i2}$ then
                $\delta(c_{i2}, x_{i2}) := \top; \delta(c_{i2}, \overline{x_{i2}}) := c_{i3}$
            else
                $\delta(c_{i2}, \overline{x_{i2}}) := \top; \delta(c_{i2}, x_{i2}) := c_{i3}$
            if $x_{i3}$ appears positive in $c_{i3}$ then
                $\delta(c_{i3}, x_{i3}) := \top$; $\delta(c_{i3}, \overline{x_{i3}}) := \bot$
            else
                $\delta(c_{i2}, \overline{x_{i3}}) := \top$; $\delta(c_{i3}, x_{i3}) := \bot$
    $\delta(\top, r) := \top$
    $q_0 := c_{11}$
    return $(Q, \Sigma, q_0, \delta)$
```

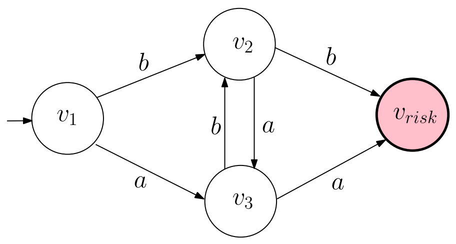


Figure 4.14: An example where priority synthesis is unable to find a set of priorities.

The size the transition system G_ϕ is polynomial in n and m. In particular, the transition system G_ϕ has $3 \cdot m + 2$ states, $2 \cdot n + 1$ interaction letters, and $2 \cdot 3 \cdot m + 1$ transitions. $\square$

4.10.1.3 Discussion

The framework of priority systems in [GS03, BBS06, CBJ$^+$11] offers a methodology to incrementally construct a system satisfying safety properties while maintaining deadlock freedom. In this thesis, we use an automata-theoretic approach to formulate the problem of priority synthesis, followed by giving an NP-completeness proof. We conclude that, although using priorities to control the system has several benefits, the price to take is the hardness of an automatic method which finds appropriate priorities. Also, based on the formulation, it is not difficult to show that it is possible to find a supervisor in the framework of Ramadge and Wonham [RW89] while priority synthesis is unable to find one. This is because priorities are stateless properties, and sometimes to achieve safety, executing interactions conditionally based on states is required. E.g., for the transition system in Figure 4.14, applying priority $a \prec b$ or $b \prec a$ is unable to ensure system safety, but there exists a supervisor (for safety) which disables b at state v_2 and a at v_3.

4.10.2 Data Processing Units in Digital Communication

In digital communication, to increase the reliability of data processing units (DPUs), one common technique is to use multiple data sampling. We have used VISSBIP to model the components and synchronization for a simplified DPU. In the model, two interrupts (`SynchInt` and `SerialInt` respectively) are invoked sequentially by a `Master` to read the data from a `Sensor`. The `Master` may miss any of the two interrupts. Therefore, `SerialInt` records whether the interrupt from `SynchInt` is lost in the same cycle. If it is missed, `SerialInt` will assume that the two interrupts have read the same value in the two continuous cycles. According to the values read from the two continuous cycle, `Master` calculates the result. In case that the interrupt from `SerialInt` is missing in the second cycle or both interrupts are missing in the first cycle, `Master` will not calculate anything. Ideally, the calculation result from `Master` should be the same as what is computed in `SerialInt`. The mismatch will lead to global deadlocks.

The synthesis of VISSBIP focuses on the deadlock-freedom property. First, we have selected the non-optimized engine. VISSBIP reports that it fails to generate priority rules to avoid deadlock, in 4.174 seconds with 168 BDD variables. Then we have selected the optimized engine and obtained the same result in 0.537 seconds with 116 BDD variables. The reason of the failure is that two contradictory priority rules are collected in the synthesis. Finally, we have allowed the engine to randomly select a priority between the contradicts (priority-repushing). A successful priority is finally reported in 1.134 seconds to avoid global deadlocks in the DPU case study.

4.10.3 Compositional Priority Synthesis: A Preliminary Evaluation

Lastly, we conduct preliminary evaluations on compositional synthesis using dining philosophers problem. Due to our system encoding, when decomposing the philosophers problem to two subproblems of equal size, compare the subproblem to the original problem, the number of BDD variables used in the encoding is only 22.5% less. This is because the saving is only by replacing component construction with the assumption; for interactions, they are all kept in the encoding of the subsystem. Therefore, if the problem size is not big enough, the total execution time for compositional synthesis is not superior than than monolithic method, as the time spent on inappropriate assumptions can be very costly. Still, we envision this methodology more applicable for larger examples, and it should be more applicable when the size of alphabet is small (but with lots of components).

CHAPTER 5

Application B. Requirements and Optimizations for Software Controller Synthesis - Extending PDDL with Game Semantics

Abstract

In this chapter, we study how to bring software controller synthesis to the level which is easy for engineers to use. As existing methodologies in LTL synthesis for reactive system design are not directly applicable for synthesizing component-based systems, we alternatively extend the behavioral description language PDDL with game concepts. Although the use of such high-level language eases modeling and creates understandable code, the size of the created game can be excessively large. We view and adapt program optimization techniques as the key weapon for solving games locally, creating drastic performance gain. Results are implemented as an extension of the GAVS+ tool (called MGSyn), and in our evaluation, we have created automatic translation from the generated control strategy to executable control code for a demonstrator of FESTO modular production system.

Contents

5.1 Introduction

The motivation of this chapter starts with the goal of letting end-users perform controller synthesis to the level which is light-weight for them to describe the system while synthesizing the result within reasonable time. For this to be possible we focus on building easy-to-use tools. During our tool construction, we have encountered the following challenges.

- **(Challenge A: Ease of modeling)** Our first task is to let engineers be comfortable to describe a system setup, i.e., from a user's perspective, to describe a system under an appropriate level of abstraction. E.g, a user may define basic abilities of each component as atomic actions separately (for component reuse). Then based on individual problems, the user specifies topologies how systems are interconnected from components, together with the specification[1]. For the above requirements, we consider existing approaches in LTL synthesis in reactive system design not directly applicable, as for such input language it is more hardware-oriented, i.e., the user specifies input and output variables and their relation as a whole. Therefore, we extend PDDL [GAI^{+}98, FL03] (the *de-facto* language in AI planning) under the game-theoretic concept, enabling users to model a system with parameterized control actions as well as the behavior of the environment. Also, we enable users to specify logic specifications formally using PDDL-like formats.
- **(Challenge B: Efficient game solving)** After the first step, our engine performs translation and solves a game under various winning conditions (e.g., reachability, Buchi, Generalized Reactivity (1) conditions [PPS06]) to synthesize high-level control actions; from these actions, executable code for dedicated platform is created by automatic refinement. Although high-level language facilitates the use of modeling, the size of the created arena is always formidably high. This lowers the speed of game solving drastically and hinders its applicability to complex systems.

  In this work, we do not propose new algorithms to solve games (as winning conditions mentioned above are well studied). Instead, we argue that efficient synthesis relies on performing game-solving algorithms on a subarena of interest (based on the specification). We demonstrate that *techniques within program optimization under the context of controller synthesis may be viewed as weapons for local game solving*, leading to powerful performance gain in overall synthesis time. Our methodology, as it is based on static checking, can be extended to existing framework in LTL synthesis as preprocessing techniques.
- **(Challenge C: Ease of refinement)** Due to our language construct, the generated strategy can already be presented in a user-understandable way where refinement is likely. However, we also observe that (1) many applications are code

[1]We refer readers to Section 5.5 for an example how atomic actions are extracted from the FESTO modular production system.

blocks chained together and executed repeatedly, and (2) it is difficult to understand the strategy in reactive form, as programmers are used to sequential and forward reasoning (while attractor is computed backwards). Therefore, for reachability games, we further sequentialize the generated strategy based on extracting witness from the attractor computation, such that users can freely choose between sequentialized-reactive or reactive code.

For evaluation, we have extended our open-source tool GAVS+ which enables users to easily construct and to automatically synthesize Java-like controller programs (with support of various winning conditions), such that further refinement to platform specific code is likely.

- From the planning community, we have reused and modified examples from the PDDL4J library [PDD], the MBP tool [BCP$^+$01], as well as benchmarks from the international planning competition. The main purpose is to obtain an idea concerning our preprocessing scheme in subarena creation. We try to apply it on basic planning problems where no environment move is modeled, and compare with other PDDL-based planners. Despite that our program is implemented in Java, it competes with them (implemented in C++, including those which uses CUDD library) concerning the overall planning time on selected benchmarks. In addition, our game setting enables us to create controllers for complex conditions than these planners.
- From the verification community, we have reused examples from the paper by Piterman, Pnueli, and Saár [PPS06]. Although our implementation is not as efficient as other synthesis tools such as Anzu [JGWB07] or Unbeast [Ehl11], thanks to the parameterized feature of actions, the generated strategy can be tailored by our algorithm for the ease of understanding rather than merely outputting bit patterns. We also report examples in (1) SW synthesis for FESTO MPS system prototyping and (2) SW synthesis for model-train controller prototyping. E.g., for FESTO MPS model domain, without any optimization the generated strategy has more than 8000 lines of code, while optimization enables to create highly understandable sequentialized-reactive code with < 500 lines. For FESTO MPS, for educational purposes we have also created scripts supporting automatic translation from our synthesized behavioral-level code to executable.

The rest of the chapter is presented as follows. Section 5.2 reviews PDDL concerning its syntax and semantics, followed by our extension to allow the description of a second player. Section 5.3 describes the underlying translation algorithm which creates symbolic encoding for linking to game engines. We briefly summarize GR(1) condition and its algorithm in Section 5.3.2. For postprocessing and serialize reactive programs (under reachability condition), we outline its algorithm in Section 5.3.3. Section 5.4 describes our preprocessing scheme based on program optimization. Section 5.5 describes our implementation and collects the experimental result. We briefly conclude in Section 5.6.

(Motivating scenario) Throughout this chapter, we consider a classical scenario in artificial intelligence, where the goal is to develop intelligence for a (robot) monkey to accomplish certain tasks. Figure 5.1 illustrates the experiment setup. There are four

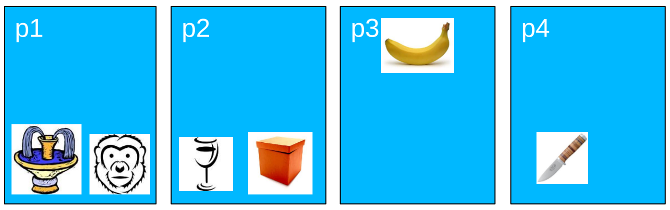


Figure 5.1: An illustration for the monkey experiment in AI.

positions p1, p2, p3 and p4 where a box, a knife, a fountain, a glass, as well as the monkey can be placed arbitrarily based on the initial configuration. The banana is hanging on the top of one location. The goal of the monkey may vary from simple reachability (e.g., to drink water) to complicated ones involving subgoal achievement (e.g., to repeatedly drink water and eat banana). Assume the goal of the monkey is to get the banana hanging at high place, the monkey shall grasp the knife, push the box to dedicated positions, climb up the box, and finally, cut the banana with the knife. We refer the above sequence of actions as a *plan* for the task.

5.2 PDDL and its Extension for Games

5.2.1 PDDL: Syntaxes and Semantics

As the entire PDDL language is too broad for processing, we focus on a small portion of PDDL (which we refer it as *core PDDL*) which is commonly seen from our collected examples[2]. Table 5.1 summarizes the result, where requirements represent subsets of features categorized in PDDL. For the ease of explanation, we omit typing in all of our definitions (while it is implemented). For details concerning the PDDL language, we refer interested readers to [GAI+98, FL03] for a full-blown manual.

In general, an instance under planning consists of two parts [GAI+98].

- A ***domain*** contains parameterized system descriptions, including constants, predicates and actions (action schemata).
- A ***problem*** specifies what planners intend to solve. It contains objects, the initial configuration and the goal specification.

Definition 22 (PDDL domain: Syntax [GAI+98])**.** *Define the Extended BNF[3] for a domain in (core) PDDL by contents in Figure 5.2. Lines starting with characters "* ; *" are comments.*

[Example] Figure 5.3 contains the domain description for the monkey experiment.

- We define 6 **constants**, namely monkey, box, knife, bananas, waterfounntain and glass.

[2]PDDL is designed with the anticipation that only a few planners will handle the entire PDDL language [GAI+98].

[3]In Extended BNF form [GAI+98], symbol "*" represents *zero or more elements*, symbol "+" represents *at least one element*, symbol "|" represents or, and objects enclosed in square brackets are optional fields.

Table 5.1: Requirements supported in our implementation. Descriptions of each requirement are from the PDDL tutorial [GAI+98] and the PDDL4J library [PDD]

Requirement	Description	Supported?
`:strips`	Basic STRIPS-style.	Supported[1]
`:negative-preconditions`	Allows `not` in goal and preconditions descriptions.	Supported
`:disjunctive-preconditions`	Allows `or` in goal and preconditions descriptions.	Supported
`:equality`	Supports = as built-in predicate.	Supported
`:conditional-effects`	Allows `when` in action effects	Supported
`:typing`	Allows type names in declaration of variables.	Supported
`:safety`	Allows `:safety` conditions for a domain.	not supported in PDDL4J[3]

[1] Currently quantification over objects is not supported for simplicity issues. Users may rewrite the quantification by enumerating concrete objects in the problem. Also, we only support predicates with two parameters.
[2] We use (and extend) PDDL4J as our front-end language parser.
[3] In our implementation the user may specify safety constraints.

- We define **predicates**, which are operators or functions that return a value that is either `true` or `false` based on its parameters. For example, `location ?x` takes an object (using one variable `?x`) an returns the truth assignment.
- We define **action schemata** for the domain. For example, consider the action `GRAB-BANANAS`.
  - It uses one *parameter* `?y`.
  - To execute the action, it should satisfy the *precondition* where the monkey is at location parameterized by `?y` (`location ?y`), has the knife (`hasknife`) and stands on the box (`onbox ?y`). Also the banana should be at location `?y` (`at bananas ?y`).
  - The *effect* turns the predicate `hasbananas` to `true`.

Definition 23 (PDDL problem: Syntax [GAI+98])**.** *Define the Extended BNF for a problem in (core) PDDL using contents in Figure 5.4 (for items defined previously in Definition 22, it is not defined repeatedly). We use the `<goal>` field to store reachability, safety and Büchi conditions, and for generalized reactivity conditions they are offered to the synthesis engine as additional input.*

[Example] Figure 5.5 contains the problem description of the monkey experiment.

- The problem `pb1` is connected to the `monkey` domain in Figure 5.3.
- Objects including 4 locations (`p1,p2,p3,p4`) are created. All 4 constants specified in the domain are used in the problem.
- The *initial configuration* specifies a scenario that the monkey is in `p1`, standing on the floor without having a knife or a glass.
- The target of fetching the banana is described in the *goal statement* "`:goal (and (hasbananas))`", i.e., the purpose of the planning problem is to reach the configuration where banana is fetched.

```
<domain>          := (define (domain <name>)
                      [<extension-def>]
                      [<require-def>]
                      [<constants-def>]
                      [<predicate-def>]
                      <action-def>*)
<extension-def>  := (:extends <domain-name>)
<require-def>    := (:requirements <require-key>+)
<constants-def>  := (:constants <name>+)
<predicates-def> := (:predicates <atomic formula skeleton>+)
<atomic formula skeleton> := (<predicate> <variable>*)
<predicate>      := <name>
<variable>       := ?<name>
<name>           := identifier
<require-key>    := Follow Table 5.1

<action-def>     := (:action <action functor>
                             :parameters ( <variable>* )
                             <action-def body>)
<action functor> := <name>
<action-def body> := :precondition <goal description> :effect <effect>
<goal description> := (and <goal description>*) | (not <goal description>)
                      | (not <goal description>) | <literal(term)>
<literal(t)>     := <atomic formula(t)>
                    | (not <atomic formula(t)>)
<atomic formula(t)> := (<predicate> t*)
<term>           := <name> | <variable>
<effect>         := (and <effect>*) | <atomic formula(term)>
                    | (not <atomic formula(term)>) | (when <goal description> <effect>)
```

Figure 5.2: Extended BNF for the domain in PDDL (partially modified from [GAI+98]).

5.2.1.1 Configuration and Semantics

It is not difficult to observe that a problem under a given domain can be translated to a transition system with a given initial state (all variables not appearing in the description of `:init` are considered as `false` in initial state) and a set of goal states (all variables not appearing in the description of `:goal` are considered as `don't-cares`). Therefore, the configuration and semantics follow the standard definition of automata.

5.2.2 Extension to Games

We focus on extensions to two-player, turn-based games over finite arena, while extensions to other game types are left as future work. To fix our notation we refer player-0 as the system (with controllable moves) and player-1 as the environment (with uncontrollable moves). Our syntactic extension is achieved with two additional steps.

1. Introduce a binary predicate `P0TRAN` on each action in the domain file to partition player-0 and player-1 states and transitions. For a given configuration, when `P0TRAN` is evaluated to true, it is a system (player-0) configuration; otherwise it represents an environment (player-1) configuration.
2. On the requirement field (`:requirements`), introduce a new requirement "`:game`". Also, "`:negative-preconditions`" should be added for game solv-

```
(define (domain monkey)
  (:requirements :strips)
  (:constants monkey box knife bananas waterfountain glass)
  (:predicates (location ?x) (on-floor) (at ?m ?x) (hasknife) (onbox ?x)
            (hasbananas) (hasglass) (haswater))
  ;; movement and climbing
  (:action GO-TO
     :parameters (?x ?y)
     :precondition (and (location ?x) (location ?y) (on-floor)
                  (at monkey ?y))
     :effect (and (at monkey ?x) (not (at monkey ?y))))
  (:action GRAB-BANANAS
     :parameters (?y)
     :precondition (and (location ?y) (hasknife)
                 (at bananas ?y) (onbox ?y))
     :effect (hasbananas))
  (:action CLIMB
     ...)
  (:action PUSH-BOX
     ...)
  (:action GET-KNIFE
     ...)
  (:action PICK-GLASS
     ...)
  (:action GET-WATER
     ...)
)
```

Figure 5.3: The domain of the monkey experiment described using PDDL.

```
<problem>          := (define (problem <name>)
                              (:domain <name>)
                     [<require-def>]
                     [<object declaration>]
                     [<init>]
                     <goal>)
<object declaration>  := (:objects <name>+)
<init>             := (:init <literal(name)>+)
<goal>             := (:goal <goal description>)
```

Figure 5.4: Extended BNF for the problem in PDDL (partially modified from [GAI+98]).

```
(define (problem pb1)
    (:domain monkey)
    (:objects p1 p2 p3 p4 bananas monkey box knife waterfountain glass)
    (:init (location p1) (location p2) (location p3) (location p4)
        (at monkey p1) (on-floor) (at box p2) (at bananas p3) (at knife p4)
        (at waterfountain p1) (at glass p2)
    )
    (:goal (and (hasbananas)))
)
```

Figure 5.5: The problem instance of the monkey experiment described using PDDL.

ing. This is because for environment moves, the precondition always comes with `(not P0TRAN)`.

[Example: Monkey-Experimenter Game] We consider a scenario extension where *the banana can be brought to other places by the experimenter*. However, the maximum number of movement done by the experimenter is limited to 1. Also, we assume that the model operates under a turn-based scenario, i.e., the monkey and the experimenter perform their moves in alternation. The above scenario can be naturally translated to a two-player, turn-based game played on finite game graphs.

For the scenario of monkey experiment, let player-0 refer to the monkey and player-1 refer to the experimenter. Figure 5.6 describes the domain description of the game (some actions are omitted).

- For the introduced predicate, except `P0TRAN` which is introduced for state partitioning, another predicate `banana-moved` is introduced to constrain the number of disturbance moves (which should be at most 1) done by the experimenter.
- For existing actions (e.g., `GRAB-BANANAS`) the precondition statement should conjunct with predicate "`P0TRAN`", indicating that it is a player-0 transition. Also, the effect statement should conjunct with "`not (P0TRAN)`", indicating that the next move belongs to the experimenter.
- Two additional actions `TAKE-BANANAS` and `STAY-BANANAS` are actions belonging to player-1. For action `TAKE-BANANAS`, it constrains the maximum number of movements to 1 using the predicate `banana-moved`.

5.3 Algorithms for Symbolic Game Creation, Game Solving, and Strategy Creation

5.3.1 Outline of Synthesis

We first summarize the conceptual flow in our implementation when processing planning/game problems using Figure 5.7. The engine first combines an input instance from the domain and the problem (domain-problem binding), then solves it with the appropriate synthesis algorithms based on the specification. If it is impossible to satisfy the specification, the engine reports a negative result. Otherwise, for planning it reports an action sequence; for games it returns a strategy automaton (finite state machine) in Java-like formats. Here we outline the underlying steps for the synthesis framework:

1. Based on the number of objects and predicates used in the problem description, declare corresponding BDD variables to represent sets of states/transitions in the symbolic form. Each (BDD) variable is a predicate where each parameter is concretized with an object from the domain. Details of our preprocessing (optimization) steps are described in Section 5.4.
2. Create the set of transitions in symbolic form. In the implementation, for each action, the algorithm first assigns an object from the domain for each parameter. Then it recursively traverses the abstract syntax tree (AST) of the precondition and postcondition to build up the formula. Also, during the traversal of the AST

```
  (define (domain monkey)
  (:requirements :strips :negative-preconditions :game)
  (:constants monkey box knife bananas waterfountain glass)
  (:predicates (location ?x) (on-floor) (at ?m ?x) (hasknife) (onbox ?x)
            (hasbananas) (hasglass) (haswater) (P0TRAN) (banana-moved))

  ...

  (:action GRAB-BANANAS
     :parameters (?y)
     :precondition (and (P0TRAN) (location ?y) (hasknife)
                                 (at bananas ?y) (onbox ?y))
     :effect (and (not (P0TRAN)) (hasbananas)) )

  ...

  ;; experimenter takes banana
  (:action TAKE-BANANAS
      :parameters (?x ?y)
      :precondition (and (not (P0TRAN)) (not (banana-moved))
                         (location ?x) (location ?y) (at bananas ?y))
      :effect (and  (P0TRAN) (at bananas ?x) (not (at bananas ?y))
                     (banana-moved)))
  ;; experimenter does nothing
  (:action STAY-BANANAS
          :parameters ()
          :precondition (and (not (P0TRAN)) )
          :effect (and  (P0TRAN)) )
)
```

Figure 5.6: The domain of monkey experiments when the experimenter can move the banana once.

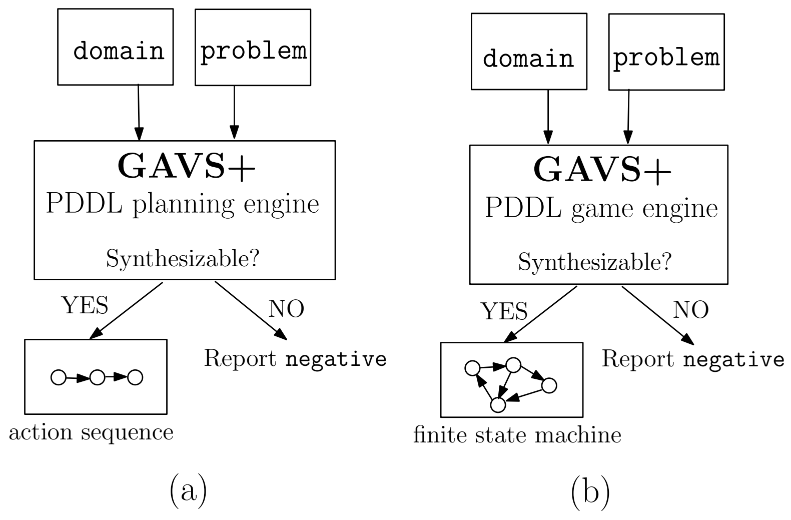


Figure 5.7: The conceptual flow executing GAVS+ for (a) planning problems and (b) game problems.

of the postcondition, it explicitly records variables being updated. After traversal, for variable v which is not updated, conjunct the generated transition by $v \leftrightarrow v'$ to ensure that the value of variable v remains unchanged (v' stands for the primed version).

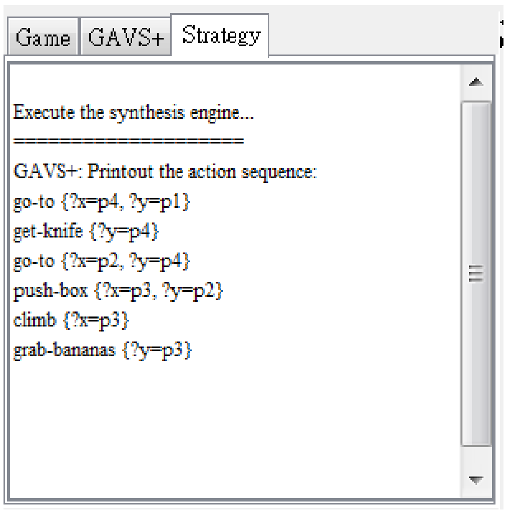
Detailed steps are as follows.

a) Perform symbolic encoding to create the set of transitions for player-1.

b) Create a list L, where for each element in the list, it is an player-0 action whose parameters are concretized by the domain of objects. E.g., in the monkey example, the symbolic representation of `GRAB-BANANAS(p1)` is stored as an element in the list. List L is used to interpret the resulting strategy in later stages[4]. The set of transitions for player-0 amounts to the disjunction over elements in list L.

3. Execute the engine selected by the user.
   - For planning (reachability), a forward symbolic reachability engine is invoked. During computation, $Reach_i$, the image of symbolic step i is stored in a list as intermediate results. At step k, if the intersection between $Reach_k$ and the set of goal states $Goal$ is nonempty, perform backward analysis from $Reach_k \cap Goal$ to generate the sequence of witness $\delta_1 \ldots \delta_k$.
     a) During the backward analysis, starting from $Wit_0 = Reach_k \cap Goal$, we need to generate Wit_i from Wit_{i-1}. This is done by first computing $Pre(Wit_{i-1})$, the preimage of Wit_{i-1}. Then $Wit_i = Reach_{k-i} \cap Pre(Wit_{i-1})$.
     b) Create δ_{k-i} by selecting action δ in L such that $\delta \cap Wit_{k-i+1} \cap Wit'_{k-i} \neq \emptyset$ (Wit'_{k-i} is the primed version of Wit_{k-i}).
     c) Reset Wit_{k-i+1} to by the result of performing existential quantification over our all primed variables in $\delta_{k-i} \cap Wit_{k-i+1}$.
     d) Continue step (a), (b), and (c) if initial state is not in Wit_{k-i+1}.
   - For reachability games, a backward attractor computation engine is invoked. During the attractor computation, continuously record the set of transitions T which leads to the goal state. When the initial state is contained in the attractor, immediately stop the computation, and intersect T with each element δ (concretized action) in L to derive the precondition to perform action δ. Lastly, use built-in functions in GAVS+ to parse the precondition and create Java-like statements. Details are explained in Section 5.3.3.
   - For safety and Büchi games, creating the strategy follows the description in Chapter 2.
   - We have also created a winning condition combining reachability and safety. For this criterion, the engine first applies safety game to created the set of all

[4]When viewing the PDDL engine as LTL synthesis tools, this step brings difference to others such as Unbeast [Ehl11], where in these tools only bit-patterns can be represented as output. With such list L, we can conjunct the generated strategy with concretized actions (both represented in BDDs) in L to derive precise preconditions for executing each action. This makes our generated strategy easier to understand.



```
Execute the synthesis engine...
====================
GAVS+: Printout the action sequence:
go-to {?x=p4, ?y=p1}
get-knife {?y=p4}
go-to {?x=p2, ?y=p4}
push-box {?x=p3, ?y=p2}
climb {?x=p3}
grab-bananas {?y=p3}
```

Figure 5.8: The generated plan for the monkey experiment by GAVS+.

```
(:action CLIMB_DOWN
      :parameters (?x)
      :precondition (and (P0TRAN) (not (on-floor)) (location ?x)
                    (onbox ?x) (at box ?x) (at monkey ?x))
   :effect (and (not (P0TRAN)) (on-floor) (not (onbox ?x))))
```

Figure 5.9: The newly introduced action for a monkey to climb down.

safe transitions δ_{safe} that avoids the set of bad states, followed by computing the attractor to goal states using δ_{safe} as player-0 moves.

- Contents concerning GR(1) are described in Section 5.3.2.

[Example: Planning] Under the problem setting, the result of planning by executing GAVS+ is indicated on Figure 5.8, matching our expectations. In fact, it can be observed that the synthesis engine contained in GAVS+ returns the shortest action sequence.

[Example: Monkey-Experimenter Game] Under this problem setting, GAVS+ reports a *negative* result: it is impossible to have a strategy which guarantees that the monkey can get the banana from the initial state. The reason comes from the situation when the monkey has climbed up on the box, i.e., `onbox` is `true`. Under this setting, when the experimenter moves the banana, the monkey is unable to proceed further, as no action of climbing down is offered in the domain[5].

[Fix of domain] Consider an additional `CLIMB-DOWN` action is offered in the domain, similar to the action in Figure 5.9[6]. By executing the action the monkey can step down from the box (i.e., `not (onbox ?x)`) and again stand on the floor (`on-floor`). With

[5]This example can be found in the GAVS+ software package with the folder `GAVS_plus_Testcase/PDDL/synthesis/monkey/`

[6]This example can be found in the GAVS+ software package with the folder `GAVS_plus_Testcase/PDDL/synthesis/monkeyUpAndDown/`

this new action, GAVS+ is able to create a strategy, which is a finite-state machine (FSM) outputted to a separate file using Java-like formats.

5.3.2 Solving Games under Generalized Reactivity(1) Conditions

Currently for game solving in PDDL, reachability, safety, Büchi, and Generalized Reactivity(1) (GR(1)) conditions are supported. For the first three conditions, contents of solving such games are listed in Chapter 2. In the following, we introduce the last winning condition and sketch its algorithm.

Synthesizing full LTL specifications is known to be 2EXPTIME-complete (in terms of the size of the specification) by the work of Pnueli and Ronser [PR89]. However, based on the experience of Piterman, Pnueli and Saár [PPS06], many specifications in hardware design can be summarized using the following form

$$(\Box\Diamond p_1 \wedge \ldots \wedge \Box\Diamond p_m) \rightarrow (\Box\Diamond q_1 \wedge \ldots \wedge \Box\Diamond q_n)$$

where "$\Box$" and "$\Diamond$" are temporal operators with intuitive meaning of "*always*" and "*in the future*"[7], p_i, q_j are boolean combinations over atomic propositions. Such specification, called Generalized Reactivity[1] conditions, can be efficiently solved in time cubic to the size of specification.

[Algorithm Sketch] Computing the winning region for such conditions requires three layers of nested-fixpoint computation (as GR(1) can be represented as μ-calculus [Koz83] with alternation depth 3).

- The outer fixpoint characterizes the strategy change from moving from q_j to $q_{(j+1)\%n}$ and so on (when q_j is reached, then the system shall try to visit $q_{(j+1)\%n}$).
- The middle fixpoint characterizes the winning region of visiting q_j.
- The inner fixpoint characterizes situations when trying to visit q_j, it is possible to perform alternative moves, i.e., to loop within any $\neg p_i$ infinitely when (then the specification trivially holds).

It is not difficult to observe that the resulting strategy should be a layered state machine, as it contains functionalities of mode change (i.e., change from reaching q_j to reaching $q_{(j+1)\%n}$).

5.3.3 Generating Strategies

We briefly remark on how strategies in game solving are interpreted from BDD to output; we only list out reachability games while other winning conditions are omitted. Let set T be the set of synthesized control transitions leading to the goal. In Section 5.3.1, we store a list L of set of transitions where each element represents a transition concretized with parameters. For each δ in L,

- Intersect with T. If the result is not empty, then perform existential quantification over primed variables to derive the precondition c when δ should be executed.

[7]Therefore, $\Box\Diamond$ represents *infinitely often*.

- Printout c as a union of cubes, and interpret the precondition based on each element of the cube.

To create sequentialized-reactive code (see introduction for motivation), here we assume that any execution of a plant transition shall immediately be followed by a control transition, and the initial state is a control location. Then the algorithm proceeds as follows.

1. Starting with an initial state S_0, intersect S_0 with T to derive the set of transitions T_0 which steps closer to the goal state. Interpret T_0 (using techniques above) as step $i = 0$ of the controller.
2. From T_0, create the set of states S_1 which is the successor state of T_0 and is outside of the goal state $Goal$. Partition S_1 to be $S_{1sys} \uplus S_{1env}$ by whether a state belongs to control (S_{1sys}) or to environment (S_{1env}). From S_{1env}, let environment perform all possible moves to $S_{1env \rightarrow sys}$, again the set of control states.
3. Repeat the first step where replace S_0 by $S_{1sys} \cup S_{1env \rightarrow sys} \setminus Goal$ and increment i. If such a set is empty, then stop the execution, as no remaining states are required to step closer to goal.
4. Here we assume that initial state is a control state. If the initial state is a plant state, then first perform all possible environment moves to reach a set of control states S'_0, then continue with (1) via replacing S_0 by S'_0.

5.4 Program Optimization for Local Game Solving

Concerning automatic game creation, a standard translation from a system description to the corresponding game creates a symbolic encoding of an arena with the number of locations (and thus the number of BDD-variables) formidably high. Denote P_i to be the set of predicates of arity i, and assume i ranges from 0 to 2 (which is applicable for most examples). Furthermore, assume that for all predicates, during the domain-problem binding, all parameters map to the same domain of constants C. Then when symbolically encoding the arena, the number of BDD variables used equals $2(|P_0| + |P_1||C| + |P_2||C|^2)$; the constant factor of 2 is used for a variable and its primed version. With such construction, we are solving a game with an arena of size $2^{|P_0|+|P_1||C|+|P_2||C|^2}$.

For example, consider the Hanoi tower domain (from the PDDL4J library [PDD]) in Figure 5.10, it contains three predicates `clear`, `on`, and `smaller`. Under a problem of 3 pegs and 8 disks (i.e., in PDDL form `(:objects peg1 peg2 peg3 d1 d2 d3 d4 d5 d6 d7 d8)`), the number of variables during the BDD construction equals $2(11 + 11^2 + 11^2) = 506$. Although use of typing (i.e., separate between pegs and disks) helps to alleviate the number of used BDD variables, we still need general variable-reduction techniques that is applicable for most examples.

To increase the speed of synthesis (game solving), our idea is to construct and solve the game on a subarena of high interest (based on the specification). In the following, we use program optimization (static analysis over the domain-problem binding) to achieve this goal. Formal proofs concerning correctness are not explicitly listed. Nevertheless,

```
(define (domain hanoi)
        (:requirements :strips)
        (:predicates
            (clear ?x)
            (on ?x ?y)
            (smaller ?x ?y)
        )
    (:action MOVE
        :parameters (?disc ?from ?to)
        :precondition (and (smaller ?to ?disc)
                (on ?disc ?from) (clear ?disc)(clear ?to))
        :effect  (and (clear ?from) (on ?disc ?to)
                (not (on ?disc ?from))(not (clear ?to)))
    )
)
```

Figure 5.10: Hanoi tower domain described using PDDL.

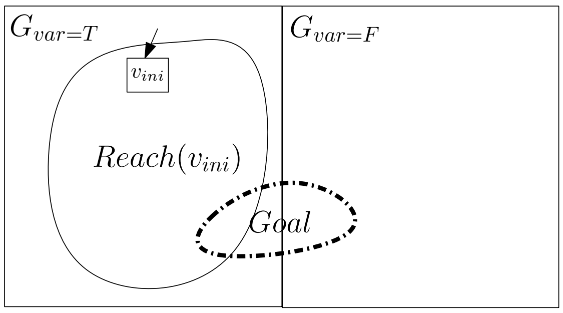


Figure 5.11: The effect of constant replacement in game solving.

we point out that the underlying correctness claim relies on the concept of **invariants**, i.e., given property p, if

- p holds on the initial configuration, and
- p holds on every transition,

then for the set of all reachable states, property p holds.

5.4.1 Optimization A: Constant Replacement

The idea behind constant replacement is to find a set of variables (predicates whose parameters are concretized) whose value never changes during the synthesis process. With such knowledge, we may omit declaring BDD variables for them and replace them by `true` or `false` during the creation of symbolic transitions instead.

Figure 5.11 illustrates how constant replacement helps to solve a game locally. Let v_{ini} be the initial location of the arena $G_{var=T} \cup G_{var=F}$. As for the set of all reachable locations $Reach(v_{ini})$, variable var is always evaluated to `true`, then the set of all reachable locations are within the subarena $G_{var=T}$ of G. Therefore, as we are not interested in $G_{var=F}$ (as it is never reachable), we only need to encode arena $G_{var=T}$, and replace every occurrence of var by `true` during our symbolic encoding. In other words, find-

```
(define (problem pb1)
    (:domain hanoi)
    (:requirements :strips)
    (:objects peg1 peg2 peg3 d1 d2 d3)
    (:init
        (smaller peg1 d1) (smaller peg1 d2) (smaller peg1 d3)
        (smaller peg2 d1) (smaller peg2 d2) (smaller peg2 d3)
        (smaller peg3 d1) (smaller peg3 d2) (smaller peg3 d3)
        (smaller d2 d1) (smaller d3 d1) (smaller d3 d2)
        (clear peg2)
        (clear peg3)
        (clear d1)
        (on d3 peg1)
        (on d2 d3)
        (on d1 d2)
    )
    (:goal  (and (on d3 peg3) (on d2 d3) (on d1 d2))
    )
)
```

Figure 5.12: Hanoi tower problem (3 disks and 3 pegs) described using PDDL.

ing a strategy starting from v_{ini} in the whole arena amounts to finding a strategy in $G_{var=T}$.

Consider again the problem of Hanoi tower (Figure 5.12), the predicate `smaller` is used to describe the relation between (a) peg and disks and (b) two disks. Also from the action schema in Figure 5.10, predicate `smaller`, once concretized, never changes its value (e.g., `(smaller peg1 d1)` remains `true`). Therefore, we may safely ignore the construction of `smaller` and replace the occurrence in concretized transitions by its initial value. E.g., in Figure 5.12 during the symbolic transition construction, when encountering `(smaller peg1 d1)` in the precondition, return `true`. Under a problem of 3 pegs and 8 disks, the number of variables during the BDD construction drastically changes from 506 to $2(11 + 11^2) = 264$, leading to a 48% spare of declared variables.

[Algorithm Sketch] Our implemented algorithm is as follows. Given a predicate $p(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n$ are of domain C. If p never occurs in any updates (i.e., postcondition) of an action schema, then omit creating the set of variables $\{p(c_1, \ldots, c_n) | c_1, \ldots, c_n \in C\}$. Furthermore,

- If $p(c_1, \ldots, c_n)$ occurs positively in the initial configuration of the problem, then during the creation of symbolic transitions, replace every occurrence of $p(c_1, \ldots, c_n)$ by `true`.
- Otherwise, during the creation of symbolic transitions, replace every occurrence of $p(c_1, \ldots, c_n)$ by `false`.

5.4.2 Optimization B: Binary Compaction

Our second technique is to perform binary encoding on a set of Boolean variables, if we can be certain that at most one of them can be set to `true` in any time. E.g., consider a sample predicate `car-in` having one parameter `?location`, having an intuitive

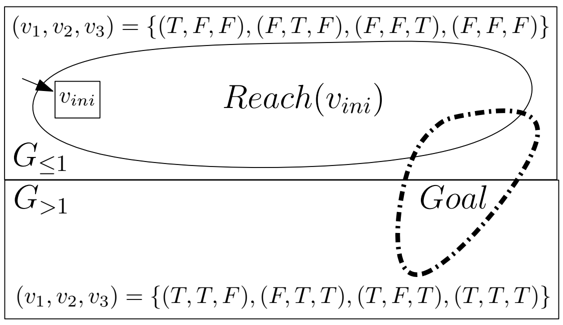

$(v_1, v_2, v_3) = \{(T,F,F),(F,T,F),(F,F,T),(F,F,F)\}$
v_{ini}
$Reach(v_{ini})$
$G_{\leq 1}$
$G_{>1}$
$Goal$
$(v_1, v_2, v_3) = \{(T,T,F),(F,T,T),(T,F,T),(T,T,T)\}$

Figure 5.13: The effect of binary compaction in game solving. $G_{\leq 1}$ ($G_{>1}$) represents the subarena from the original arena where at most (least) one variable in $\{v_1, v_2, v_3\}$ is `true`.

meaning that it specifies whether a car is in certain city. Assume that for parameter `?location`, its domain contains seven cities `A`, `B`, ..., `G`. For this predicate, the set of three locations and a unique empty symbol (representing the car is not in all locations) can be encoded as a number in binary format. Such a binary format uses only three Boolean variables.

Figure 5.13 illustrates how binary compaction helps to solve a game locally. Let v_{ini} be the initial location of the arena $G_{\leq 1} \cup G_{>1}$. As for the set of all reachable locations $Reach(v_{ini})$, at most one variable in $\{v_1, v_2, v_3\}$ is evaluated to `true`, then the set of all reachable locations are within the subarena $G_{\leq 1}$. Therefore, as we are not interested in $G_{>1}$ (as it is never reachable), we only need to encode arena $G_{\leq 1}$. In other words, finding a strategy starting from v_{ini} in the whole arena amounts to finding a strategy in $G_{\leq 1}$.

To process, we further assume that the precondition and postcondition are represented as a single conjunction of literals where negations are pushed close to atomic formula (this occurs in all testcases of our benchmark suite as well as examples from the PDDL4J library).

[Algorithm Sketch] Our implemented algorithm is as follows.

- (*Predicate with one parameter*) First perform applicability checking. Given predicate $p(x)$ where x is of domain C. we use $\lceil \log(|C| + 1) \rceil$ binary variables $p[1], \ldots, p[\lceil \log(|C| + 1) \rceil]$ to represent the set of binary variables $\{p(c) | c \in C\}$ if the following conditions hold.
  - In the initial configuration, at most one element $\{p(c) | c \in C\}$ can appear positively.
  - For every parameterized action which uses parameters of concrete value $x_1, \ldots, x_n$, either (i) p never occurs in both precondition and postcondition, or (ii)
    1. the precondition contains one positive occurrence of p.

2. in 1, if p uses parameter x_i, then in the postcondition, we allow the following three cases.
   a) p never occurs.
   b) $p(x_i)$ appears negatively.
   c) There exists exactly one parameter x_j, such that (i) $p(x_i)$ appears negatively and (ii) $p(x_j)$ appears positively.

When the above check holds, order elements in C by a list $list(C)$. Let $|C|$ be the size of C. Denote $BDD_{\cup_i var_i = v_i}$ to be a set of transition relations where variable var_i is evaluated to v_i while other variables are unconstrained, BDD_{false} to be the empty set and BDD_{true} to be the universe. During the creation of symbolic transitions, in a precondition or a postcondition,

– When encountering $p(c_i)$, replace it by $BDD_{\{p[1]=t_1,\ldots,p[\lceil\log(|C|+1)\rceil]=t_{\lceil\log(|C|+1)\rceil}\}}$, where $t_{\lceil\log(|C|+1)\rceil}\ldots t_1$ represents the binary encoding of $index(c_i, list(C))$.
  ∗ Denote $index(a, l)$ to be the index of a in list l with starting index value 0.
– When encountering variable $\neg p(c_1)$, the algorithm further splits into three cases.
  ∗ (i) When there exists $p(c_1)$, then replace it by BDD_{false} in the construction.
  ∗ (ii) When no $p(c_i)$ occurs, replace it by $BDD_{\{p[1]=t_1,\ldots,p[\lceil\log(|C|+1)\rceil]=t_{\lceil\log(|C|+1)\rceil}\}}$, where $t_{\lceil\log(|C|+1)\rceil}\ldots t_1$ represents the binary encoding of $|C|$.
  ∗ (iii) When there exists $p(c_2)$ where $c_1 \neq c_2$, then omit the construction by BDD_{true}.

- (*Predicate with two or more parameters*) Currently for predicates with more parameters, we check if binary encoding works for the last parameter. E.g., for predicate $p(x_1, x_2)$ where x_1, x_2 are having domain C, we check if the set of variables $S = \{p(c, c_1), \ldots, p(c, c_n)\}$, where $c, c_1, \ldots, c_n \in C$, we can apply binary encoding on S.

We remark on the correctness of such transformation for predicates with one parameter.

- **(Applicability checking)** Using the concept of invariants ensures that within $\{p(c)|c \in C\}$, at most one of it will be `true` starting from the initial configuration. We also need the to represent the case where all of them are `false`. Therefore, the set $\{0\ldots|C|\}$ of numbers suffices, and $\lceil\log(|C|+1)\rceil$ bits are required when using binary encoding.
- **(Creating the set of translation relations)** When encountering $p(c_i)$, the algorithm only performs a direct translation. However, $\neg p(c_1)$ needs to be considered in detail. The split of three cases matches the case of (b) and (c) in the applicability check.
  1. For case (i), in the original system, conjunction of sets $BDD_{p(c_1)=T}$ and $BDD_{p(c_1)=F}$ should return an empty set.

2. For case (ii), we know that p uses parameter c_1 and
   - $p(c_1)$ appears positively in the precondition from item 1 of the applicability check.
   - $p(c_1)$ appears negatively in the postcondition from item 2(b) of the applicability check.

   Therefore, for variables $\{p(c)|c \in C\}$, none is set to `true` after the postcondition. Thus we can safely represent it by the unique identifier $|C|$.
3. For case (iii), if we follow 2, then $BDD_{p(c_2)=T} \wedge BDD_{p(c_1)=F}$ in the original system will turn into BDD_{false} under our encoding, which is an incorrect result. Therefore, we shall directly return $BDD_{p(c_2)=T}$. This is achieved by omitting the construction of $\neg p(c_1)$ (or conjunct with BDD_{true}).

5.4.3 Optimization C: Goal-indifferent Variable Elimination

Our last optimization aims to identify variables that are unnecessary to satisfy the goal specification, i.e., if there exists a winning strategy that modifies these variables, then there exists another winning strategy that leaves them unchanged.

Recall the monkey example, if the goal of the monkey is to fetch the banana, then whether the monkey holds the glass can be viewed as irrelevant: whenever there exists a strategy which grasps the glass (and thus changes the `hasglass` variable), there exists another strategy that does not include grasping the glass. This knowledge corresponds to *inferring from the specification a set of actuators that are not required in the concrete setup to reach the goal*. This computation corresponds to computing the cone-of-influence computation used in verification.

[Algorithm Sketch] The analysis step for reachability games, which can also be applied to GR(1) games, proceeds as follows: first, add all variables that appears in the goal condition, i.e., the variables that do not have a “don't care” value in the goal condition, in a set S. We use the set S to store all the variables that have a potential influence on the goal. Then select a set of actions Δ whose postcondition (i.e., the `:effect` field in an action) changes variables in S. Add the set all variables S' which appear in the precondition (i.e., the `:precondition` field in an action) of Δ to S. Repeat until S saturates, then treat variables not contained in S as $\widehat{S}$. Denote the set of variables used either in the precondition or in the postcondition by player-1 transitions as S_{P_1}. Then variables within $\widehat{S} \setminus S_{P_1}$ can be omitted for construction.

The correctness relies on the following observations:

- For player-1, whether it can perform a move is insensitive to variables in $\widehat{S} \setminus S_{P_1}$ (i.e., outside S_{P_1}). Therefore, these variables can be omitted for player-1, and we do not restrict the ability of player-1.
- For player-0, to reach the goal, variables within $\widehat{S} \setminus S_{P_1}$ (i.e., within $\widehat{S}$) are not useful at all, as they do not appear in all of the possible control decisions leading to the goal.

E.g., consider the monkey example, where the goal is to fetch the banana. Initially, S

contains one element `hasbanana`. Action `GET-BANANA(p3)` contains `hasbananas` in the postcondition, so we add `(location p3)`, `hasknife`, `(at bananas p3)`, and `(onbox p3)` to S, as their values can be sensitive to `hasbananas`. Continue the process, it can be concluded that `hasglass` and `haswater` are not included in S. Also, for the experimenter, the set of variables used in `MOVE-BANANAS` and `DO-NOTHING` do not include `hasglass` and `haswater`. Therefore, in our symbolic encoding we safely ignore `haswater` and `hasglass`. Subsequently, we ignore constructing actions `PICK-GLASS` and `GET-WATER`, as they only update `hasglass` and `haswater`.

5.5 Implementation and Evaluation

5.5.1 Implementation

We have implemented the above features as an extension of GAVS+. To parse input files with the PDDL format, we use the open source library PDDL4J [PDD]. Concerning the specification,

- To specify reachability, safety, and Büchi conditions, we reuse the `:goal` field pre-existed in a PDDL problem.
- To specify LTL (GR(1)) conditions, we have explicitly implemented a GR(1) specification parser using the compiler construction tool JavaCC [Jav], such that users can specify an LTL specification with an extended format similar to PDDL. Prior to game solving, the specification is also translated such that $p_1, \ldots, p_m, q_1, \ldots, q_n$ are stored separately.
  – For example, the following PDDL-like specification is used to specify the following generalized Büchi condition $\Box\Diamond position(dep) \ \wedge \ \Box\Diamond position(store)$.
  ```
  ([]<> ( and (true))) -> ([]<> (and (position dep)) &&
  []<> (and (position store)))
  ```

5.5.2 Evaluation

In this section, we give a discussion between our work with others together with a preliminary evaluation over our implementation. Results are collected from an Intel 3.0 Ghz Machine (with 4GB RAM). For GAVS+, we allocate 3000MB memory for the Java Virtual Machine.

5.5.2.1 Effect of variable reduction (subarena creation)

We first give a general evaluation concerning the improvement by our preprocessing. Table 5.2 summarizes the performance measure on the classic example of Hanoi tower. We have also listed the result of the built-in GraphPlan algorithm [BF97] (a classic algorithm in AI planning) implemented in PDDL4J [PDD]. Under the preprocessing, we are able to solve more complex problems within reasonable amount of time. GraphPlan algorithm scales badly.

Table 5.2: Experimental results (seconds)

Problem	GraphPlan	Symbolic[n]	Symbolic[o]	Required steps to goal
Hanoi 4	0.69	1.127	0.554	15 steps
Hanoi 5	9.11	1.83	0.602	31 steps
Hanoi 6	t/o	4.34	0.934	63 steps
Hanoi 7	t/o	23.975	1.612	127 steps
Hanoi 8	t/o	t/o	1.918	255 steps
Hanoi 9	t/o	t/o	2.831	511 steps
Hanoi 10	t/o	t/o	5.394	1023 steps
Hanoi 11	t/o	t/o	13.515	2047 steps

[t/o] Timeout ($>$ 400 seconds)
[n] GAVS+: No optimization
[o] GAVS+: Optimization based on variable reduction

5.5.2.2 Plan Generation

We continue our evaluation by comparing our implementation with other tools. For this we have taken PDDL planners from the International Planning Competition (IPC'11). We have downloaded planners with executables available for download, namely *seq-opt-gamer*[8] [KE11] and *FastDownward* [Hel06][9].

Examples in our benchmark suite are either from the PDDL4J library or from the planning competition[10].

- (LOGISTICS from PDDL4J) In this domain a set of trucks, airplanes, airports are located at different cities. The goal is to find a plan of transportation to send pessengers/goods to desired destinations.
- (BARMAN from IPC'11) The following description is from the IPC'11 website: *In this domain there is a robot barman that manipulates drink dispensers, glasses and a shaker. The goal is to find a plan of the robot's actions that serves a desired set of drinks. In this domain deletes of actions encode relevant knowledge given that robot hands can only grasp one object at a time and given that glasses need to be empty and clean to be filled.*
- (VISIT-ALL from IPC'11) In this domain, a robot tries to traverse through all terrains, where connections between terrains are set as constraints.
- (GRIPPER from PDDL4J) In this domain, a robot with a specified number of arms is located in a room. The robot contains several predefined actions such as `pick-object`, `place-object`, or `move`. The goal is distribute objects to the their destinations.

Our implementation (Java-based, using JDD as our BDD package) is comparable with other implementations despite the implemented language. On larger examples, GAVS+

[8]It won the *first* place in the 2008 international planning competition (sequential optimization track). Also, it uses CUDD [CUD] for symbolic manipulation.
[9]FastDownward is now used by many teams as the initiative for attending the planning competition. It includes many algorithms prebuilt.
[10]Examples are directly taken from the repository. We remove the cost function in the Barman example, so all planners work on actions with unit cost.

Table 5.3: Experimental results (seconds)

Problem	Symbolic^{o_1}	Symbolic^{o_2}	seq-opt-gamer	FastDownward^{H_1}
Logistics pb0	1.09	1.018	0.875	3.29
Logistics pb1	6.852	7.124	9.474	m, t/o
Logistics pb2	10.616	16.924	p.e.	m, t/o
Logistics pb3	3.015	2.18	p.e.	p.e.
Logistics pb4	16.63	16.273	14.477	m, t/o
Barman pb1	72.52	23.742	45.631	39.26
Barman pb2	66.088	22.241	70.704	39.97
Barman pb3	66.672	20.727	99.147	39.47
Barman pb4	55.572	21.503	117.996	39.67
Visitall pb4full	0.899	0.873	0.677	0.09
Visitall pb5full	1.762	1.481	3.368	46.52
Visitall pb6full	15.646	23.63	t/o	m, t/o
Gripper pb4	1.913	3.328[1]	0.702	0.06
Gripper pb5	1.799	4.221[1]	1.149	0.80
Gripper pb6	31.219	57.183[1]	19.806	m, t/o
Gripper pb7	3.763	8.079[1]	6.240	m, t/o

t/o Timeout (> 300 seconds)
p.e. Parser error
m Memory error
^{o_1} GAVS+: Optimization based on variable reduction
^{o_2} GAVS+: Optimization based on variable reduction and FORCE ordering [AMS03]
[1] GAVS+: The FORCE heuristic is applied only once
^{H_1} FastDownward: Use option `-search "astar(blind)"` specified in the manual

outperforms on BARMAN and VISIT-ALL domains, while results vary on the LOGISTICS and GRIPPER domain[11]. Notice that using SAT or SMT solvers rather than BDD may achieve better performance. Nevertheless, our purpose is to have an idea concerning the compactness of our encoding for the use of game solving, and we view this result only as an additional benefit.

5.5.2.3 Game Solving (A): FESTO MPS System

In this section, we study how PDDL-based game solving can be applied to assist the control of FESTO MPS[12], a modular production system. MPS is mostly used for teaching purposes, but is a very good approximation for a real industrial automation process. Each unit of the MPS processes small colored work pieces that are made out of plastic or metal. Our demonstrator setup (see Figure 5.14) is composed of two modules, a processing unit and a storage unit.

- The processing platform is built up from a rotating plate (green element in Figure 5.14), together a probe sensor (which tests the form of the workpiece) and a drilling module (which processes on the workpiece).

[11]Here we list two results: one with FORCE heuristic and the other without; in our implementation we tune our heuristic scheme to invoke FORCE when appropriate.
[12]`http://www.festo-didactic.com/int-en/learning-systems/mps-the-modular-production-system/`

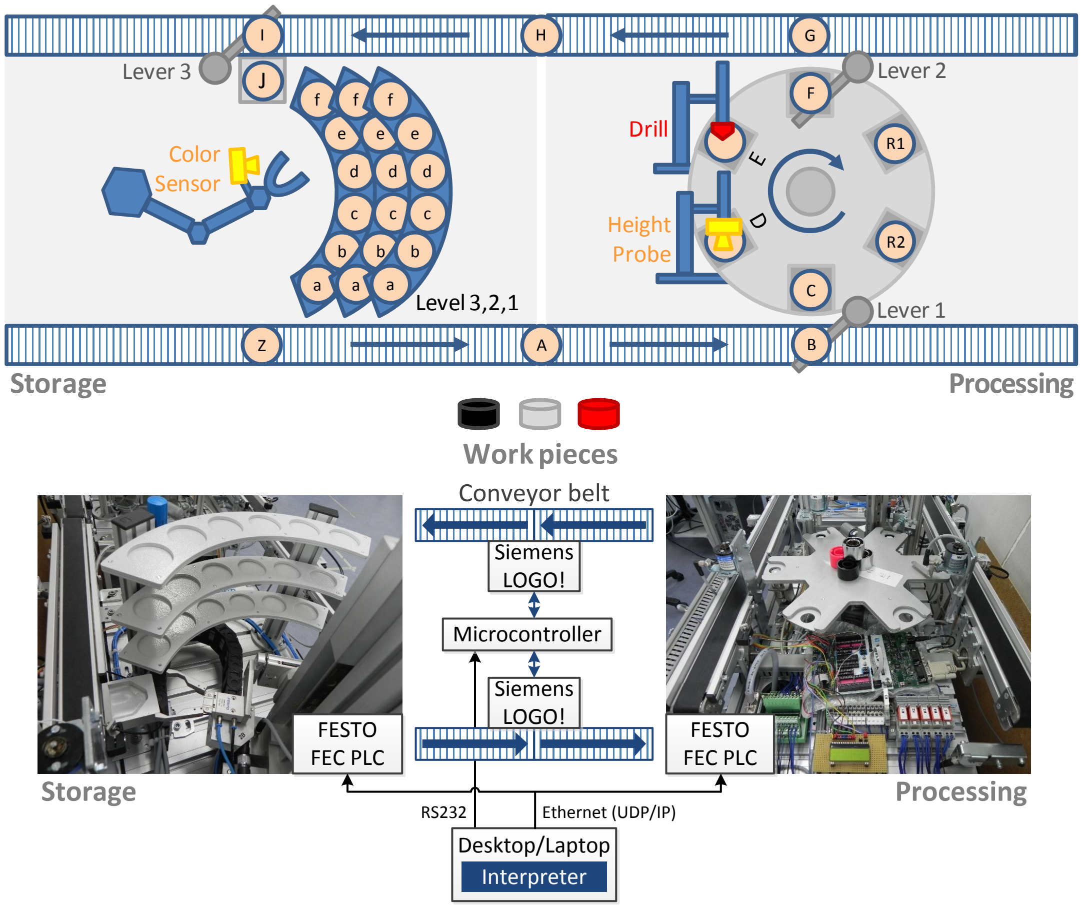


Figure 5.14: The FESTO MPS demonstrator setup (up) and screenshot of its actual execution (down).

- The storing station contains a robot arm with a three-layer rack, used to store and retrieve the object.
- Several rods are allocated on certain positions to move the object between the belt and the module.

We have tried to model components of each module (together with their predefined actions) using PDDL. Table 5.4 and 5.5 summarize some predicates and parameterized actions used to specify the system. We then try to specify various specifications and synthesize controllers (under sequentialized-reactive form)[13].

We apply our techniques to synthesize controllers for the FESTO MPS. Table 5.6 reports the time for the game engine to synthesize the strategy for different test cases. E.g., for

[13]The FORCE heuristics is not used in the synthesis process, as the number of interactions grows drastically and makes computing impossible. Instead, we only perform simple adjustments to make variables using the same constant close. We leave light-weight methods of finding good ordering heuristics as future work.

Table 5.4: Predicates defined in the PDDL domain of a FESTO MPS system

Predicate	Parameter - Type	Intuitive meaning
in-robot	?pos - robotposition	robot arm is in ?pos
at	?obj - object ?pos - (robotposition, beltposition, plateposition)	object ?obj is at position ?pos
free-hand	?gri - gripper	gripper ?gri has no object
carry	?obj - object, ?gri - gripper	gripper ?gri carries object ?obj
belt-connected	?pos1, ?pos2 - beltposition	transmission belt connected between ?pos1 and ?pos2
rod-located	?pos1, ?pos2 - (robotposition, beltposition, plateposition)	rod can push from ?pos1 to ?pos2
next	?pos1, ?pos2 - plateposition	an object on the plate will move from ?pos1 to the next position ?pos2 when rotating clockwise
drill-position	?pos - plateposition	drill is at position ?pos
have-color-sensor	?pos - (robotposition, beltposition, plateposition)	color sensor is at ?pos
color-sensor-on	?pos - (robotposition, beltposition, plateposition)	color sensor at ?pos is on
color	?obj - object, ?col - colortype	object ?obj has color ?col
have-form-sensor	?pos - (robotposition, beltposition, plateposition)	form sensor is at ?pos
form-sensor-on	?pos - (robotposition, beltposition, plateposition)	form sensor at ?pos is on
form	?obj - object, ?form - formtype	object ?obj has shape ?form
P0TRAN	none	system or environment's move

test case 3b in Table 5.6, the standard encoding needs a total number of 593×2 variables, while with optimizations we only use 60×2 variables. Here we offer two levels of optimization. Without optimization even case 2a cannot be synthesized. Deciding how optimization for synthesis is applied is done automatically and the total time is within than a second.

1. Our base setting is to move an object from position `A` to the rack. In this example we modify the domain to disable all sensor actions, so it amounts to forward reachability analysis. As the belt moving is only unidirectional (e.g., from `A` to `B`), the generated action sequence has 14 steps consisting (a) belt-moving, (b) plate-rotating, and (c) robot-arm processing. The processing time is 1.216 seconds.
2. Our first setting is to drill an object and store it on the rack based on the color. Initially the color value is unknown, and shall only be known when triggering the color sensor. Thus in our goal specification, we add an additional constraint specifying that an object shall be of color white, red, or black. Our engine creates sequentialzed reactive code-blocks in 1.49 seconds.
3. Our second setting is to modify from 1, and it is required that an object shall only be drilled when it is facing up (formtype: up). When it is placed down, then it shall be returned to position `A` back to the operator.
   a) When removing one color and one layer of storage, the engine synthesizes the code in 1.26 seconds.
   b) When restricting each layer to only two storage positions, the engine synthesizes the code in 1.36 seconds.
   c) When restricting each layer to only three storage positions, the engine synthesizes the code in 1.37 seconds.
4. The last setting is to modify from 2, and use the object detection sensor to detect

Table 5.5: Actions defined in the PDDL domain of a FESTO MPS system

Action	Parameter - Type	Intuitive meaning	Change
robot-move	?pos1 ?pos2 - robotposition	robot arm moves from ?pos1 to ?pos2	Sys -> Sys
robot-pick	?gri - gripper ?pos - robotposition, ?obj - object	robot arm ?gri picks object ?obj (both at position ?pos)	Sys -> Sys
robot-check	?pos - robotposition	trigger sensor to check if ?pos is occupied	Sys -> Env
return-check	?pos - robotposition	return whether position ?pos is occupied	Env -> Sys
robot-drop	?gri - gripper ?pos - robotposition, ?obj - object	robot arm ?gri drops object ?obj (both at position ?pos)	Sys -> Sys
belt-move	?obj - object, ?pos1 ?pos2 - beltposition	belt transfers object ?obj from ?pos1 to ?pos2	Sys -> Sys
drill-in	?obj - object, ?pos - plateposition	drill located at position ?pos drills the object	Sys -> Sys
plate-rotate	?obj1 ... ?obj6 - object ?pos1 ... ?pos6 - ?plateposition	rotate the plate to move the object	Sys -> Sys
rod-push	?obj - object ?pos1 ?pos2 - (robotposition, beltposition, plateposition)	push the rod located between ?pos1 and ?pos2 to change the position of object ?obj	Sys -> Sys
trigger-color-sensor	?obj - object ?pos - (robotposition, beltposition, plateposition)	trigger the color sensor located at ?pos	Sys -> Env
return-color-value	?color - colortype ?obj - object ?pos - (robotposition, beltposition, plateposition)	return the color value ?color when the color sensor located at ?pos is on	Env -> Sys
trigger-form-sensor	?obj - object ?pos - (robotposition, beltposition, plateposition)	trigger the shape-detector located at ?pos	Sys -> Env
return-form-value	?form - formtype ?obj - object ?pos - (robotposition, beltposition, plateposition)	return the shape ?form when the shape-detector located at ?pos is on	Env -> Sys

whether the rack is full. If it is full, then return object to position `A` back to the operator.

a) When removing one color and one layer while restricting each remaining layer to only two storage positions, the engine synthesizes the code in 1.65 seconds.

b) When removing one color and one layer while restricting each remaining layer to only three storage positions, the engine synthesizes the code in 2.38 seconds.

c) When restricting each layer to only three storage positions while maintaining three colors, the engine synthesizes the code in 1.97 seconds.

5.5.2.4 Game Solving (B): Other Examples

Table 5.7 summarizes the performance of our synthesis engine on other selected benchmark suites.

- (RobotPlanning modified from MBP [BCP$^+$01]) In this domain, a robot is placed in a house with many rooms. The goal varies from simple reachability to repeatedly visit several rooms (generalized Büchi). We also experiment coordination within two robots: the goal is that two robots shall never be in the same room, and in our synthesis framework, we let one robot perform its move freely while the other shall win the safety game.

Table 5.6: Game solving time on FESTO MPS (seconds)

Case	Partial Opt.[H1]	Optimization[H2]	Remark
1	30.42	1.49	Storing by color
2a	29.93	1.26	1 + object detection
2b	86.34	1.36	1 + object detection
2c	97.47	1.37	1 + object detection
3a	231.21	1.65	2 + capacity detection
3b	420.06	2.38	2 + capacity detection
3c	t.o.	1.97	2 + capacity detection

[H1] Evaluated under an earlier (and slower) version of GAVS+ which is partially optimized. We also raise the default memory setting to 6000 MB for JVM.
[H2] Evaluated under current version of GAVS+. We reduce the default memory setting to 1000 MB for JVM.
[t.o.] Time out (> 500 seconds)

- (GRIPPER from PDDL4J) In this domain, we model the error of the robot arm, and the purpose is to achieve goal-oriented behavior.
- (ELEVATOR from [PPS06]) In this domain, the request of the user is modeled, and the goal is a GR(1) specification indicating that request shall be responded. See the original paper [PPS06] for details.
- (MODELTRAIN) Lastly, we try to extract from the example shown in Figure 5.15 a PDDL model for a train system. In this setting, we model the controller with the ability to raise the red light (enforce the train to stop) and to perform change over switches.
  - To create precise modeling over interactions between the train and the track, we have performed case split to partition the set of all tracks into three categories, i.e., in Figure 5.15 the yellow rectangle (`seg`), the red circle (`splitsegP1`), and the mesh rectangle (`splitsegP2`). For example, when a train is on `splitsegP2` moving towards `splitsegP1`, it suffers from derailing when two segments are not connected by switch and when no light signal is on to stop the train.
  - The goal is either to perform repeated traversal, to avoid derailing, or to avoid collision (i.e., two trains are never within the same track during their service to destinations).
  - We have also experimented our variable reduction techniques under the extension of conditional effects. In this domain, this enables to further reduce the number of declared variables by 30%.

During our evaluation, we found that it is slightly awkward to model simultaneous movement of multiple trains using PDDL, as PDDL lacks the ability to describe parallel actions (e.g., we must model the behavior of two trains as the set of product moves explicitly). We plan to support modularity under the existing framework in near future (similar to the BIP framework for component-based design in Chapter 4).

Table 5.7: Experimental results (seconds)

Problem	Execution Time[R]	Remark
MBP pb1	0.401	reachability
MBP pb1	1.4	Büchi
MBP pb1	0.399	GR(1)
MBP pb2	6.45	Safety
MBP pb3	7.435	Safety
Gripper pb1	1.48	reachability
Gripper pb2	2.785	reachability
Gripper pb3	6.529	reachability
Gripper pb4	17.963	reachability
Elevator pb1	6.421	GR(1), 8 floors
Elevator pb2	15.557	GR(1), 16 floors
Elevator pb3	31.67	GR(1), 24 floors
Train pb1	1.163	Simple derail prevention (safety)
Train pb2	15.538	Repeated visit (generalized Büchi)
Train pb3	16.616	Collision avoidance (reachability+safety)

[R] Total time includes game creation, game solving, and strategy print-out. No optimization (e.g., variable ordering, sequentialize strategy) is used.

5.6 Summary

5.6.1 Relation to other work

We compare our results with existing work in (a) local game solving, (b) planning in artificial intelligence and (c) LTL synthesis.

Our key insight is to perform faster synthesis by solving games locally. The concept of local game solving was only used in parity or winning conditions with known complexity at least $\mathtt{NP} \cap \mathtt{co\text{-}NP}$ [FL10, NRZ11]. Our concept to view program optimization as techniques for local game solving is new, and our focus is only to solve games with strategy finding in polynomial time, as when encountering large examples, we need to solve games symbolically. Also, we found work by Marinov et al. [MKB$^+$05] related, as they also use techniques in program optimization for Alloy Analyzer to create optimized SAT formulas. However, our optimization techniques are different from theirs, and our transformation amounts to subgame solving.

Concrete theoretical results from synthesis and games encourage us to seek for applications in various domains. Under this initiative, we find it natural to relate synthesis with planning problems in AI and robotics. The purpose of planning is to create a sequence of actions under a given domain and problem setting such that goal configuration can be reached from the initial configuration via configuration updates from the generated action sequence. In a Dagstuhl seminar held previously in 2006 [KTV06], the participants drew a conclusion that two fields (planning, synthesis) are intimately related and encouraged researchers to increase the interaction and collaboration between two research communities. Nevertheless, for such collaboration to succeed, a platform sharing mutual knowledge is required (yet lacking) for dialogue, which strengthens our motivation.

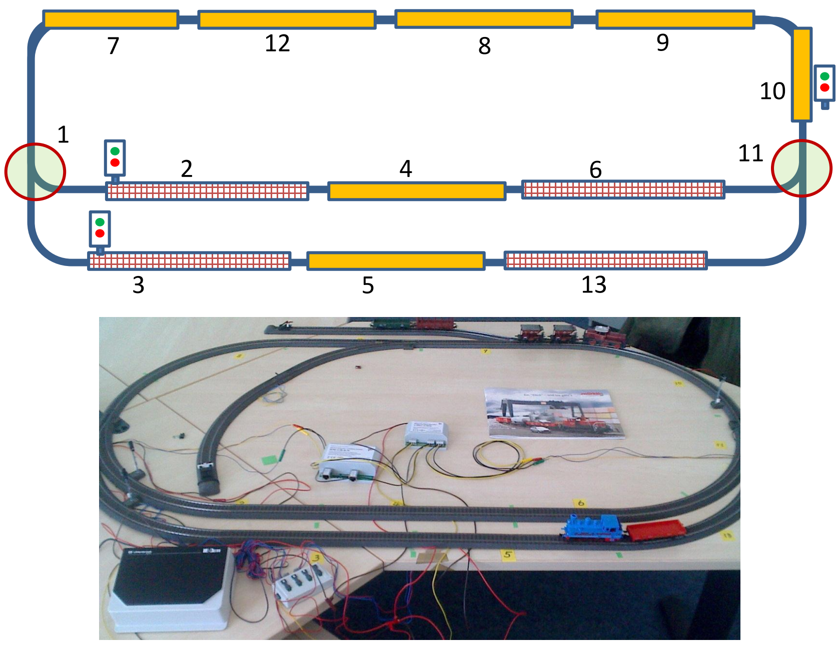


Figure 5.15: The model train system setup (up) and screenshot of its actual execution (down).

We briefly remark on other PDDL planners. Although a large collection of PDDL specific solvers is available, only [EK07] (i.e., the implementation *seq-opt-gamer*) is based on a similar approach combining planning and games. The goal of [EK07] is to "*study the application and extension of planning technology for general game playing*". Therefore, we find that games described in [KE11] are mostly AI games (e.g., tic-tac-toe, hanoi tower specified in [KE11] as benchmark), where most of them are having reachability objectives. Contrarily, our goal is rather to allow researchers in the verification community to profit from the rich collection of models coming from the AI community, while introducing AI community prebuilt algorithms to plan for complex behaviors. Therefore, our algorithms for safety, Büchi, and GR(1) brings difference with the tool above. Besides, we have shown in our evaluation that our encoding scheme enables faster processing (and thus faster game solving), despite of our Java implementation.

The concept of the second player is similar to non-deterministic planning, where uncertainty can be modeled as the environment move. Thus we also give a brief remark on the tool MBP [BCP^{+}01], where they seek to perform planning on non-deterministic domains. They propose three conditions, namely weak, strong, and strong cyclic planning. The weak condition means to make an optimistic assumption on the environment's behavior, thus is equal to normal planning. The strong planning can be viewed as an instance of reachability game. Lastly, the strong acyclic plan assumes that the environment player can not indefinitely perform the bad behavior (something close to almost-sure winning). Our argument over such tools follows from above. ALisp [MRL05] also performs synthesis by letting a user specify non-determinism over choices, but its strategy finding is based on machine learning.

Concerning LTL synthesis, our main argument is emphasized in the introduction, where our focus is on building a language with ease of refinement. In addition, our optimization technique is general for preprocessing. E.g., during the evaluation with the AMBA bus case study with the Anzu tool, Jobstmann et al. [JGWB07] manually extract from the specification new mutual exclusion conditions specifying that at most one variable is active. Then these conditions are added to constrain the system. The above work can be summarized by our program optimization technique (B) - extract conditions for binary encoding, but our approach is fully automatic. In our extended technical report, we have experienced a speed up of an order of magnitude by applying our optimization techniques on an arbiter example in the tool Anzu. Also, with such knowledge, we directly change the symbolic encoding to use less BDD variables, thus increasing the speed of synthesis.

5.6.2 Conclusion

In this chapter, we illustrate how games can be combined with PDDL for synthesizing component-based systems. Our purpose is to create an easy-to-use modeling language for behavioral-level synthesis. To create games, we need to adapt program optimization techniques to create compact representation such that game engines are able to synthesize strategies successfully (within reasonable computation resource). Furthermore, the generated strategies shall be further pruned for the ease of understanding.

Currently, we try to apply our results and implementation within two applications, namely (1) the AutoPnP project, where the purpose is to model the ability of components (for FESTO MPS) and use GAVS+ for controller synthesis, and (2) the JAMES project for human-robot interaction, where the purpose is to create planners subject to change with guaranteed safety (e.g., Büchi/reachability + safety). We have also created a tool called MGSyn [CGR$^+$12b], which uses the underlying engine to automate the synthesis process for automation systems. To bring practical impact, an ongoing work is to synthesize controllers with optimality criterion. E.g., it can be interesting to use sketching [SLRBE05] to parallelize or distribute the generated program for performance considerations. Besides, another direct extension which includes integer cost (which can also be encoded symbolically using BDDs) for actions is under implementation. Lastly, to increase the speed of synthesis, we plan to study how other program optimization techniques can be used while replacing our Java-based BDD engine with native C implementation.

CHAPTER 6

Application C. A Game-Theoretic Approach for Synthesizing Fault-Tolerant Embedded Systems

Abstract

We present an approach for fault-tolerant synthesis by combining predefined patterns for fault-tolerance with algorithmic game solving. A non-fault-tolerant system, together with the relevant fault hypothesis and fault-tolerant mechanism templates in a pool are translated into a distributed game, and we perform an incomplete search of strategies to cope with undecidability. One result is that finding a distributed positional strategy for reachability winning is NP-complete. The result of the game is translated back to executable code concretizing fault-tolerant mechanisms using constraint solving. The overall approach is implemented to a prototype tool chain and is illustrated using examples.

Contents

6.1 Introduction

In this chapter, we investigate methods to perform automatic fault-tolerant (FT for short) synthesis under the context of embedded systems, where our goal is to generate executable code which can be deployed on dedicated hardware platforms.

Creating such a tool supporting the fully-automated process is very challenging as the inherent complexity is high: bringing FT synthesis from theory to practice means solving a problem consisting of (a) interleaving semantics, (b) timing, (c) fault-tolerance, (d) dedicated features of concrete hardware, and optionally, (e) the code generation framework. To generate tamable results, we first constrain our problem space to some simple yet reasonable scenarios (sec. 6.2). Based on these scenarios we can start system modeling (sec. 6.3) taking into account all above mentioned aspects.

To proceed further, we find it important to observe the approach nowadays to understand the need: for engineers working on ensuring fault-tolerance of a system, once the corresponding fault model is decided, a common approach is to select some fault-tolerant patterns [Han07] (e.g., fragments of executable code) from a pattern pool. Then engineers must fine-tune these mechanisms, or fill in unspecified information in the patterns to make them work as expected. With the above scenario in mind, apart from generating complete FT mechanisms from specification, our synthesis technique emphasizes automatic selection of predefined FT patterns and automatic tuning such that details (e.g., timing) can be filled without human intervention. This also reduces a potential problem where unwanted FT mechanisms are synthesized due to under-specification. Following the statement, we translate the system model, the fault hypothesis, and the set of available FT patterns into a distributed game [MW03] (sec. 6.4), and a strategy generated by the game solver can be interpreted as a selection of FT patterns together with guidelines of tuning.

For games, it is known that solving distributed games is, in most cases, undecidable [MW03]. To cope with undecidability, we restrict ourselves to the effort of finding positional strategies (mainly for reachability games). We argue that finding positional strategies is still practical, as the selected FT patterns may introduce additional memory during game creation. Hence, a positional strategy (by pattern selection) combined with selected FT patterns generates mechanisms using memory. By posing this restriction, the problem of finding a strategy of the game (for control) is NP-Complete (sec. 6.5), and searching techniques (e.g., SAT translation or combining forward search with BDD) are thus applied to assist the finding of solutions.

The final step of the automated process is to translate the result of synthesis back to concrete implementation: the main focus is to ensure that the newly synthesized mechanisms do not change the implementability of the original system (i.e., the new system is schedulable). Based on our modeling framework, this problem can be translated to a linear constraint system, which can be solved efficiently by existing tools.

To evaluate our methods, we have created our prototype software, which utilizes the model-based approach to facilitate the design, synthesis, and code generation for fault-tolerant embedded systems. We demonstrate two small yet representative examples with our tool for a proof-of-concept (sec. 6.7); these examples indicate the applicability

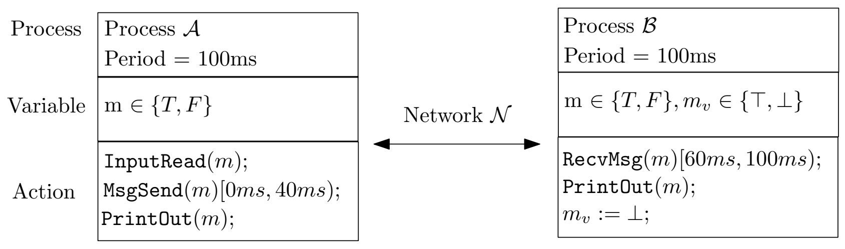


Figure 6.1: An example for two processes communicating over an unreliable network.

of the approach. Lastly, we conclude this chapter with an overview of related work (sec. 7.5) and a brief summary including the flow of our approach (sec. 6.9).

6.2 Motivating Scenario

6.2.1 Adding FT Mechanisms to Resist Message Loss

We give a motivating scenario in embedded systems to facilitate our mathematical definitions. The simple system described in Figure 6.1 contains two *processes* $\mathcal{A}$, $\mathcal{B}$ and one bidirectional *network* $\mathcal{N}$. Processes $\mathcal{A}$ and $\mathcal{B}$ start executing sequential actions together with a looping period of $100ms$. In each period, $\mathcal{A}$ first reads an input using a sensor to variable m, followed by sending the result to the network $\mathcal{N}$ using the action `MsgSend(m)`, and outputing the value (e.g., to a log).

In process $\mathcal{A}$, for the action `MsgSend(m)`, a message containing value of m is forwarded to $\mathcal{N}$, and $\mathcal{N}$ broadcasts the value to all other processes which contain a variable named m, and set the variable m_v in $\mathcal{B}$ as $\top$ (indicating that the content is valid). However, $\mathcal{A}$ is unaware whether the message has been sent successfully: the network component $\mathcal{N}$ is unreliable, which has a faulty behavior of *message loss*. The fault type and the frequency of the faulty behavior are specified in the *fault model*: in this example for every complete period ($100ms$), at most one message loss can occur.

In $\mathcal{B}$, its first action `RecvMsg(m)` has a property describing an interval $[60, 100)$, which specifies the *release time* and *deadline* of this action to be $60ms$ and $100ms$, respectively. By posing the release time and the deadline, in this example, $\mathcal{B}$ can finalize its decision whether it has received the message m successfully using the equality constraint ($m_v = \perp$), provided that the time interval $[40, 60)$ between (a) deadline of `MsgSend(m)` and (b) release time of `RecvMsg(m)` overestimates the *worst case transmission time* for a message to travel from $\mathcal{A}$ to $\mathcal{B}$. After `RecvMsg(m)`, it outputs the received value (e.g., to an actuator).

Due to the unreliable network, it is easy to observe that two output values may not be the same. Thus the *fault-tolerant synthesis* problem in this example is to perform suitable modification on $\mathcal{A}$ and $\mathcal{B}$, such that two output values from $\mathcal{A}$ and $\mathcal{B}$ are the same at the end of the period, regardless of the disturbance from the network.

6.2.2 Solving Fault-Tolerant Synthesis by Instrumenting Primitives

To perform FT synthesis in the example above, our method is to introduce several slots (the size of slots are fixed by the designer) between actions originally specified in the system. For each slot, an atomic operation can be instrumented, and these actions are among the pool of predefined *fault-tolerant primitives*, consisting of message sending, message receiving, local variable modifications, or `null-ops`. Under this setting we have created a game, as the original transitions in the fault-intolerant system combined with all FT primitives available constitute the controller (player-0) moves, and the triggering of faults and the networking can be modeled as environment (player-1) moves.

6.3 System Modeling

6.3.1 Platform Independent System Execution Model

We first define the execution model where timing information is included; it is used for specifying embedded systems and is linked to our code-generation framework. In the definition, for ease of understanding we also give each term intuitive explanations.

Definition 24. *Define the syntax of the* ***Platform-Independent System Execution Model (PISEM)*** *be* $\mathcal{S} = (\mathcal{A}, \mathcal{N}, \mathcal{T})$.

- $\mathcal{T} \in \mathbf{Q}$ *is the replication period of the system.*
- $\mathcal{A} = \bigcup_{i=1\ldots n_A} \mathcal{A}_i$ *is the set of processes, where in* $\mathcal{A}_i = (V_i \cup V_{env_i}, \overline{\sigma_i})$,
  - V_i *is the set of variables, and* V_{env_i} *is the set of environment variables. For simplicity assume that* V_i *and* V_{env_i} *are of integer domain.*
  - $\overline{\sigma_i} := \sigma_1[\alpha_1, \beta_1); \ldots; \sigma_j[\alpha_j, \beta_j); \ldots; \sigma_{k_i}[\alpha_{k_i}, \beta_{k_i})$ *is a sequence of actions.*
    - $\sigma_j :=$ `send`$(pre, index, n, s, d, v, c) \mid a \leftarrow$ `e` $\mid$ `receive`(pre, c) *is an atomic action (action pattern), where*
      - · $a, c \in V_i$,
      - · `e` *is function from* $V_{env_x} \cup V_i$ *to* V_i *(this includes* `null-op`*),*
      - · pre *is a conjunction of over equalities/inequalities of variables,*
      - · $s, d \in \{1, \ldots, n_A\}$ *represents the source and destination,*
      - · $v \in V_d$ *is the variable which is expected to be updated in process* d,
      - · $n \in \{1, \ldots, n_N\}$ *is the network used for sending, and*
      - · $index \in \{1, \ldots, size_n\}$ *is the index of the message used in the network.*
    - $[\alpha_j, \beta_j)$ *is the execution interval, where* $\alpha_j \in \mathbb{Q}$ *is the release time and* $\beta_j \in \mathbb{Q}$ *is the deadline.*
- $\mathcal{N} = \bigcup_{i=1\ldots n_N} \mathcal{N}_i$, $\mathcal{N}_i = (\mathcal{T}_i, size_i)$ *is the set of network.*
  - $\mathcal{T}_i : \mathbb{N} \rightarrow \mathbb{Q}$ *is a function which maps the index (or priority) of a message to the worst case message transmission time (WCMTT).*
  - $size_i$ *is the number of messages used in* $\mathcal{N}_i$.

[Example] Based on the above definitions, the system under execution in section 6.2.1 can be easily modeled by PISEM: let $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{N}$ in section 6.2.1 be renamed in a PISEM as $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{N}_1$. For simplicity, we use $\mathcal{A}.j$ to represent the variable j in process $\mathcal{A}$, assume that the network transmission time is 0, and let v_{env} contain only one variable v in $\mathcal{A}_1$. Then in the modeled PISEM, we have $\mathcal{N}_1 = (f : \mathbb{N} \rightarrow 0, 1)$, $\mathcal{T} = 100$, and the action sequence of process $\mathcal{A}_1$ is

$$m \leftarrow \texttt{InputRead}(v)[0, 40); \texttt{send}(true, 1, 1, 1, 2, m, \mathcal{A}_1.m)[0, 40); v \leftarrow \texttt{PrintOut}(m)[40, 100);$$

For convenience, we use $|\overline{\sigma_i}|$ to represent the length of the action sequence $\overline{\sigma_i}$, $\sigma_j.deadline$ to represent the deadline of σ_j, and $iSet(\overline{\sigma_i})$ to represent a set containing (a) the set of subscript numbers in $\overline{\sigma_i}$ and (b) $|\overline{\sigma_i}| + 1$, i.e., $\{1, \ldots, k_i, k_i + 1\}$.

Definition 25. *The configuration of $\mathcal{S}$ is $(\bigwedge_{i=1 \ldots n_A}(v_i, v_{env_i}, \Delta_{next_i})$, $\bigwedge_{j=1 \ldots n_N}(occu_j, s_j, d_j, var_j, c_j, t_j, ind_j), t)$, where*

- *v_i is the set of the current values for the variable set V_i,*
- *v_{env_i} is the set of the current values for the variable set V_{env_i},*
- *$\Delta_{next_i} \in [1, |\overline{\sigma_i}| + 1]$ is the next atomic action index taken in $\overline{\sigma_i}$*[1]*,*
- *$occu_j \in \{\texttt{false}, \texttt{true}\}$ is for indicating whether the network is busy,*
- *$s_j, d_j \in \{1, \ldots, n_A\}$,*
- *$var_j \in \bigcup_{i=1,\ldots,n_A}(V_i \cup V_{env_i})$,*
- *$c_j \in \mathbb{Z}$ is the content of the message,*
- *$ind_j \in \{1, \ldots, size_j\}$ is the index of the message occupied in the network,*
- *t_j is the reading of the clock used to estimate the time required for transmission,*
- *t is the current reading of the global clock.*

The change of configuration is caused by the following operations.

1. (*Execute local action*) For machine i, let s and j be the current configuration for var and Δ_{next_i}, and v_i, v_{env_i} are current values of V_i and V_{env_i}. If $j = |\overline{\sigma_i}| + 1$ then do nothing (all actions in $\overline{\sigma_i}$ have been executed in this cycle); else the action $\sigma_j := var \leftarrow \texttt{e}[\alpha_j, \beta_j)$ updates var from s to $\texttt{e}(v_i, v_{env_i})$, and changes Δ_{next_i} to $min\{x | x \in iSet(\overline{\sigma_i}), x > j\}$. This action should be executed between the time interval $t \in [\alpha_j, \beta_j)$.
2. (*Send to network*) For machine i, let s and j be the current configuration for var and Δ_{next_i}. If $j = |\overline{\sigma_i}| + 1$ then do nothing; else the action $\sigma_j := \texttt{send}(pre, index, n, s, d, v, c)[\alpha_j, \beta_j)$ should be processed between the time interval $t \in [\alpha_j, \beta_j)$, and changes Δ_{next_i} to $min\{x | x \in iSet(\overline{\sigma_i}), x > j\}$.
   - When pre is evaluated to true (it can be viewed as an `if` statement), it then checks the condition $occu_n = false$: if the condition holds, it updates network n with value $(occu_n, s_n, d_n, var_n, c_n, t_n, ind_n) := (true, i, d, v, c, 0, index)$. Otherwise it blocks until the condition holds.
   - When pre is evaluated to false, it skips the sending.

[1]Here an interval $[1, |\overline{\sigma_i}| + 1]$ is used for the introduction of FT mechanisms described later.

3. (*Process message*) For network j, for configuration $(occu_j, s_j, d_j, var, c_j, t_j, ind_j)$ if $occu_j = true$, then during $t_j < \mathcal{T}_j(ind_j)$, a transmission occurs, which updates $occu_j$ to `false`, $A_{d_j}.var$ to c_j, and $A_{d_j}.var_v$ to `true`.
4. (*Receive*) For machine i, let s and j be the current configuration for c and Δ_{next_i}. If $j = |\overline{\sigma_i}| + 1$ then do nothing; else for $\texttt{receive}(pre, c)[\alpha_j, \beta_j)$ in machine i, it is processed between the time interval $t \in [\alpha_j, \beta_j)$ and changes Δ_{next_i} to $min\{x | x \in iSet(\overline{\sigma_i}), x > j\}$[2].
5. (*Repeat Cycle*) When $t = \mathcal{T}$, t is reset to 0, and for all $x \in \{1, \ldots, n_A\}$, Δ_{next_x} are reset to 1.

Notice that by using this model to represent the embedded system under analysis, we make the following assumptions:

- *All processes and networks in $\mathcal{S}$ share a globally synchronized clock.* Note that this assumption can be fulfilled in many hardware platforms, e.g., components implementing the IEEE 1588 [EFW02] protocol.
- For all actions σ, $\sigma.deadline < \mathcal{T}$; for all send actions $\sigma := \texttt{send}(pre, index, n, s, d, v, c)$, $\sigma.deadline + \mathcal{T}_n(index) < \mathcal{T}$, i.e., all processes and networks should finish its work within one complete cycle.

6.3.2 Interleaving Model (IM)

Next, we establish the idea of interleaving model (IM) which is used to offer an intermediate representation to bridge PISEM and game solving, such that (a) it captures the execution semantics of PISEM without explicit statements of timing, and (b) by using this model it is easier to connect to the standard representation of games.

Definition 26. *Define the syntax of the **Interleaving Model (IM)** be $S_{IM} = (A, N)$.*

- $A = \bigcup_{i=1\ldots n_A} A_i$ *is the set of processes, where in* $A_i = (V_i \cup V_{env_i}, \overline{\sigma_i})$,
  - V_i *is the set of variables, and* V_{env_i} *is the set of environment variables.*
  - $\overline{\sigma_i} := \sigma_1[\wedge_{m=1\ldots n_A}[pc_{1,m_{low}}, pc_{1,m_{up}})]; \ldots; \sigma_j[\wedge_{m=1\ldots n_A}[pc_{j,m_{low}}, pc_{j,m_{up}})]; \ldots; \sigma_{k_i}[\wedge_{m=1\ldots n_A}[pc_{k_i,m_{low}}, pc_{k_i,m_{up}})]$ *is a fixed sequence of actions.*
    - * $\sigma_j := \texttt{send}(pre, index, n, s, d, v, c) \mid \texttt{receive}(pre, c) \mid a \leftarrow \texttt{e}$ *is an atomic action, where* $a, c, e, pre, v, n,$ s, d *are defined similarly as in PISEM.*
    - * *For* σ_j, $\forall m \in \{1, \ldots, n_A\}$, $pc_{j,m_{low}}, pc_{j,m_{up}} \in \{1, \ldots, |\overline{\sigma_m}| + 2\}$ *is the lower and the upper bound (PC-precondition interval) concerning*
      1. *precondition of program counter in machine* k, *when* $m \neq i$.
      2. *precondition of program counter for itself, when* $m = i$.
- $N = \bigcup_{i=1\ldots n_N} N_i$, $N_i = (T_i, size_i)$ *is the set of network.*
  - $T_i : \mathbb{N} \rightarrow \bigwedge_{m=1\ldots n_A}(\{1, \ldots, |\overline{\sigma_m}| + 2\}, \{1, \ldots, |\overline{\sigma_m}| + 2\})$ *is a function which*

[2] In our formulation, the $\texttt{receive}(pre, c)$ action can be viewed as a syntactic sugar of `null-op`; its purpose is to facilitate the matching of send-receive pair with variable c.

maps the index (or priority) of a message to the PC-precondition interval of other processes.

– $size_i$ *is the number of messages used in* $\mathcal{N}_i$.

Definition 27. *The configuration of* S_{IM} *is* $(\bigwedge_i(v_i, v_{env_i}, \Delta_{next_i}), \bigwedge_j(occu_j, s_j, d_j, c_j))$, *where* $v_i, v_{env_i}, \Delta_{next_i}, occu_j, s_j, d_j, c_j$ *are defined similarly as in PISEM.*

The change of configurations in IM can be interpreted analogously to PISEM; we omit details here but mention three differences:

1. For an action σ_j having the precondition $[\wedge_{m=1\dots n_A}[pc_{j,m_{low}}, pc_{j,m_{up}})]$, it should be executed between $pc_{j,m_{low}} \leq \Delta_{next_m} < pc_{j,m_{up}}$, for all m.
2. For processing a message, constraints concerning the timing of transmission in PISEM are replaced by referencing the PC-precondition interval of other processes in IM, similar to 1.
3. The system repeats the cycle when $\forall x \in \{1, \dots, n_A\}$, $\Delta_{next_x} = |\overline{\sigma_x}| + 1$ and $\forall x \in \{1, \dots, n_N\}$, $occu_x = \texttt{false}$.

6.4 Step A: Front-end Translation from Models to Games

6.4.1 Step A.1: From PISEM to IM

To translate from PISEM to IM, the key is to generate abstractions from the release time and the deadline information specified in PISEM. As in our formulation, the system is equipped with a globally synchronized clock, the execution of actions respecting the release time and the deadline can be translated into a partial order. Algorithm 5 concretizes this idea by generating PC-intervals in all machines as

- temporal preconditions for an action to execute, or
- temporal preconditions for a network to finish its message processing, i.e., to update a variable in the destination process with the value in the message[3].

Starting from the initialization where no PC is constrained, the algorithm performs a restriction process using four if-statements {**(1)**, **(2)**, **(3)**, **(4)**} listed.

- In (1), if $\sigma_m.releaseTime > \sigma_n.deadline$, then before σ_m is executed, σ_n should have been executed.
- In (2), if $\sigma_m.deadline < \sigma_n.releaseTime$, then σ_n should not be executed before executing σ_m.
- Similar analysis is done with (3) and (4). However, we need to consider the combined effect together with the network transmission time: we use 0 to represent the best case, and $\mathcal{T}_n(ind)$ for the worst case.

[Example] For the example in sec. 6.2, consider the action $\sigma_1 := m \leftarrow \texttt{InputRead}(v)[0, 40)$ in $\mathcal{A}_1$ of a PISEM. Algorithm 5 returns $mapLB(\sigma)$ and $mapUB(\sigma)$ with two arrays $[1, 1]$ and $[2, 2]$, indicated in Figure 6.2a. Based on the definition of IM, σ_1 should be

[3]Here we assume that in each period, for all $\mathcal{N}_j$, each message of type $ind \in \{1, \dots, size_j\}$ is sent at most once. In this way, the algorithm can assign an unique PC-precondition interval for every message type.

Algorithm 5: GeneratePreconditionPC

```
Data: PISEM model $\mathcal{S} = (\mathcal{A}, \mathcal{N}, \mathcal{T})$
Result: Two maps $mapLB$, $mapUB$ which map from an action $\sigma$ (or a msg processing
        by network) to two integer arrays $lower[1 \dots n_A]$, $upper[1 \dots n_A]$
begin
    /* Initial the map for recording the lower and upper bound for action */
    for action $\sigma_k$ in $\mathcal{A}_i$ of $\mathcal{A}$ do
        $mapLB.put(\sigma_k$, new int[1...$n_A$](1)) /* Initialize to 1 */
        $mapUB.put(\sigma_k$, new int[1...$n_A$])
        for $\mathcal{A}_j \in \mathcal{A}$ do $mapUB.get(\sigma_k)$[j] := $|\overline{\sigma_j}| + 2$ /* Initialize to upperbound */
        $mapLB.get(\sigma_k)$[i] = k; $mapUB.get(\sigma)$[i] = k+1; /* self PC */
    for action $\sigma_m$ in $\mathcal{A}_i$ of $\mathcal{A}$, $m = 1, \dots, |\overline{\sigma_i}|$ do
        for action $\sigma_n$ in $\mathcal{A}_j$ of $\mathcal{A}$, $n = 1, \dots, |\overline{\sigma_j}|$ , $j \neq i$ do
1           if $\sigma_m.releaseTime > \sigma_n.deadline$ then
                $mapLB.get(\sigma_m)$[j] := max{$mapLB.get(\sigma_m)$[j], $n + 1$}
2           if $\sigma_m.deadline < \sigma_n.releaseTime$ then
                $mapUB.get(\sigma_m)$[j] := min{$mapUB.get(\sigma_m)$[j], $n + 1$};
    /* Initialize the map for recording the lower and upper bound for msg
    transmission */
    for action $\sigma_k = send(pre, ind, n, s, d, v, c)$ in $\mathcal{A}_i$ of $\mathcal{A}$ do
        $mapLB.put(n.ind$, new int[1...$n_A$](1)) /* Initialize to 1 */
        $mapLB.get(n.ind)$[i] := k+1 /* Strictly later than executing send() */
        $mapUB.put(n.ind$, new int[1...$n_A$])
        for $\mathcal{A}_j \in \mathcal{A}$ do $mapUB.get(n.ind)$[j] := $|\overline{\sigma_j}| + 2$ /* Initialize to upperbound */
    for action $\sigma_k = send(pre, ind, n, s, d, v, c)$ in $\mathcal{A}_i$ of $\mathcal{A}$ do
        for action $\sigma_m$ in $\mathcal{A}_j$ of $\mathcal{A}$, $n = 1, \dots, |\overline{\sigma_j}|$ do
3           if $\sigma_k.releaseTime + 0 > \sigma_m.deadline$ then
                $mapLB.get(n.ind)$[j] := max{$mapLB.get(n.ind)$[j], $m + 1$}
4           if $\sigma_k.deadline + \mathcal{T}_n(ind) < \sigma_m.releaseTime$ then
                $mapUB.get(n.ind)$[j] := min{$mapUB.get(n.ind)$[j], $m + 1$};
```

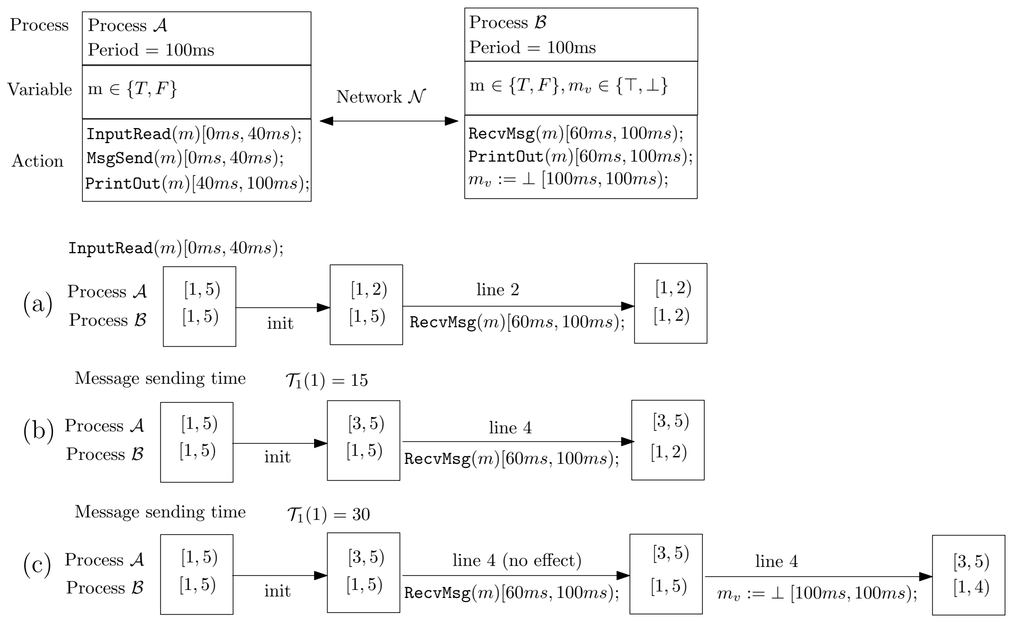


Figure 6.2: An illustration for Algorithm 5.

executed with the temporal precondition that no action in $\mathcal{A}_2$ is executed, satisfying the semantics originally specified in PISEM. For the analysis of message sending time, two cases are listed in Figure 6.2b and Figure 6.2c, where the WCMTT is estimated as 15ms and 30ms, respectively.

6.4.2 Step A.2: From IM to Distributed Game

Here we give main concepts how a game is created after step A.1 is executed. To create a distributed game from a given interleaving model $S_{IM} = (A, N)$, we need to proceed with the following three steps:

6.4.2.1 Step A.2.1: Creating non-deterministic timing choices for existing actions

During the translation from a PISEM $\mathcal{S} = (\mathcal{A}, \mathcal{N}, \mathcal{T})$ to its corresponding IM $S_{IM} = (A, N)$, for all process $\mathcal{A}_i$ in $\mathcal{A}$, for every action $\sigma[\alpha, \beta)$ where $\sigma[\alpha, \beta) \in \overline{\sigma_i}$, algorithm 1 creates the PC-precondition interval $[\wedge_{m=1\ldots n_A}[pc_{m_{low}}, pc_{m_{up}})]$ of other processes. Thus in the corresponding game, for $\sigma[\wedge_{m=1\ldots n_A}[pc_{m_{low}}, pc_{m_{up}})]$, each element $\sigma[\wedge_{m=1\ldots n_A}(pc_m)]$, where $pc_{m_{low}} \leq pc_m < pc_{m_{up}}$, is a nondeterministic transition choice which can be selected separately by the game engine.

6.4.2.2 Step A.2.2: Introducing fault-tolerant choices as $\sigma_{\frac{a}{b}}$

In our framework, fault-tolerant mechanisms are similar to actions, which consist of two parts: *action pattern* σ and *timing precondition* $[\wedge_{m=1\ldots n_A}[pc_{m_{low}}, pc_{m_{up}})]$. Compared to existing actions where nondeterminism comes from timing choices, for fault-tolerance transition choices include all combinations from (1) timing precondition and (2) action patterns available from a predefined pool.

We use the notation $\sigma_{\frac{a}{b}}$, where $\frac{a}{b} \in \mathbb{Q} \backslash \mathbb{N}$, to represent an inserted action pattern between $\sigma_{\lfloor \frac{a}{b} \rfloor}$ and $\sigma_{\lceil \frac{a}{b} \rceil}$. With this formulation, multiple FT mechanisms can be inserted within two consecutive actions σ_i, σ_{i+1} originally in the system, and the execution semantic follows what has been defined previously: as executing an action updates Δ_{next_i} to $min\{x | x \in iSet(\overline{\sigma_i}), x > j\}$, updating to a rational value is possible. Note that as $\sigma_{\frac{a}{b}}$ is only a fragment without temporal preconditions, we use algorithm 6 to generate all possible temporal preconditions satisfying the semantics of the original interleaving model: after the synthesis only temporal conditions satisfying the acceptance condition will be chosen.

Algorithm 6: DecideInsertedFTTemplateTiming

```
Data: $\sigma_c[\wedge_{m=1 \ldots n_A}[pc_{c,m_{low}}, pc_{c,m_{up}})]$, $\sigma_d[\wedge_{m=1 \ldots n_A}[pc_{d,m_{low}}, pc_{d,m_{up}})]$, which are
      consecutive actions in $\overline{\sigma_i}$ of $A_i$ of $S_{IM} = (A, N)$, and one newly added action
      pattern $\sigma_{\frac{a}{b}}$ to be inserted between
Result: Temporal preconditions for action pattern $\sigma_{\frac{a}{b}}$: $[\wedge_{m=1 \ldots n_A}[pc_{\frac{a}{b},m_{low}}, pc_{\frac{a}{b},m_{up}})]$
begin
    for $m = 1, \ldots, n_A$ do
        if $m \neq i$ then
            $pc_{\frac{a}{b},m_{low}} := pc_{c,m_{low}}$ /* Use the lower bound of $c$ for its lower bound */
            $pc_{\frac{a}{b},m_{up}} := pc_{d,m_{up}}$ /* Use the upper bound of $d$ for its upper bound */
        else
            $pc_{\frac{a}{b},m_{low}} := \frac{a}{b}$; $pc_{\frac{a}{b},m_{up}} := d$
```

We conclude this step with two remarks:

- For all existing actions, the non-deterministic choice generation in step A.2.1 must be modified to contain these rational points introduced by FT mechanisms.
- A problem induced by FT synthesis is whether the system behavior changes due to the introduction of FT mechanisms. We answer the problem by splitting into two subproblems:
  - **[Problem 1]** Whether the system is still schedulable due to the introduction of FT actions, as these FT actions also consume time. This can only be answered when the result of synthesis is generated, and we leave this to section 6.6.
  - **[Problem 2]** Whether the networking behavior remains the same. This problem *must* be handled before game creation, as introducing a FT message may significantly influence the worst case message transmission time (WCMTT) of all existing messages, leading a completely different networking behavior. The answer of this problem depends on many factors, including the hardware in use, the configuration setting, and the analysis technique used for the estimation of WCMTT. In Appendix A we give a simple analysis for

ideal CAN buses [DBBL07], which are used most extensively in industrial and automotive embedded systems: in the analysis, we propose conditions where newly added messages do ***not*** change the existing networking behavior. Similar analysis can be done with other timing-predictable networks, e.g., FlexRay [PPE$^+$08].

6.4.2.3 Step A.2.3: Game Creation by Introducing Faults

In our implementation, we do not generate the primitive form of distributed games (DG), as the definition of DG is too primitive to manipulate. Instead, algorithms in our implementations are based on our created variant called ***symbolic distributed games*** *(SDG)*:

Definition 28. *Define a symbolic distributed game $\mathcal{G}_{ABS} = (V_f \uplus V_{CTR} \uplus V_{ENV}, A, N, \sigma_f, pred)$.*

- *V_f, V_{CTR}, V_{ENV} are disjoint sets of (fault, control, environment) variables.*
- *$pred : V_f \times V_{CTR} \times V_{ENV} \rightarrow \{\texttt{true}, \texttt{false}\}$ is the partition condition.*
- *$A = \bigcup_{i=1\ldots n_A} A_i$ is the set of* ***symbolic local games (processes)*** *, where in $A_i = (V_i \cup V_{env_i}, \overline{\sigma_i})$,*
  - *V_i is the set of variables, and $V_{env_i} \subseteq V_{ENV}$.*
  - *$\overline{\sigma_i} := \bigcup \sigma_{i_1} \langle \wedge_{m=1,\ldots,n_A} pc_{i_{1m}} \rangle; \ldots ; \bigcup \sigma_{i_k} \langle \wedge_{m=1,\ldots,n_A} pc_{i_{km}} \rangle$ is a sequence, where $\forall j = 1, \ldots, k$, $\bigcup \sigma_{i_j} \langle \wedge_{m=1,\ldots,n_A} pc_{i_{jm}} \rangle$ is a set of choice actions for player-0 in A_i.*
    - *σ_{i_j} is defined similarly as in IM.*
    - *$\forall m = \{1, \ldots, n_A\}$, $pc_{i_{jm}} \in [pc_{i_j, m_{low}}, pc_{i_j, m_{up}})$, $pc_{i_j, m_{low}}, pc_{i_j, m_{up}} \in iSet(\overline{\sigma_m})$.*
  - *$V_{CTR} = \bigcup_{i=1\ldots n_A} V_i$.*
- *$N = \bigcup_{i=1\ldots n_N} N_i$, $N_i = (T_i, size_i, tran_i)$ is the set of network processes.*
  - *T_i and $size_i$ are defined similarly as in IM.*
  - *$tran_i : V_f \times (\{\texttt{true}, \texttt{false}\} \times \{1, \ldots, n_A\}^2 \times \bigcup_{i=1,\ldots,n_A} (V_i \cup V_{env_i}) \times \mathbb{Z} \times \{1, \ldots, size_i\}) \rightarrow V_f \times (\{\texttt{true}, \texttt{false}\} \times \{1, \ldots, n_A\}^2 \times \bigcup_{i=1,\ldots,n_A} (V_i \cup V_{env_i}) \times \mathbb{Z} \times \{1, \ldots, size_i\})$ is the network transition relation for processing messages (see sec. 6.3.1 for meaning), but can be influenced by additional variables in V_f.*
- *$\sigma_f : V_f \times V_{CTR} \times V_{ENV} \times \bigwedge_{i=1\ldots n_A} iSet(\overline{\sigma_i}) \rightarrow V_{ENV} \times V_f \times \bigwedge_{i=1\ldots n_A} iSet(\overline{\sigma_i})$ is the environment update relation.*

We establish an analogy between SDG and DG using Figure 6.3.

1. The configuration v of a SDG is defined as the product of all variables used.
2. A play for a SDG starting from state v_0 is a maximal path $\pi = v_0 v_1 \ldots$, where
   - In v_k, player-1 determines the move $(v_k, v_{k+1}) \in E$ when $pred(v_k)$ is evaluated to `true` (`false` for player-0); the partition of vertices V_0 and V_1 in a SDG is implicitly defined based on this, rather than specified explicitly as in a distributed game.
   - A move (v_k, v_{k+1}) is a selection of executable transitions defined in N, σ_f, or

	DG	SDG
State space	product of all vertices in local games	product of all variables (including variables used in local games)
Vertex partition (V_0 and V_1)	explicit partition	use $pred$ to perform partition
Player-0 transitions	defined in local games	defined in $\overline{\sigma_i}$ of A_i, for all $i \in \{1, \ldots, n_A\}$
Player-1 transitions	explicitly specified in the global game	defined in N and σ_f

Figure 6.3: Comparison between DG and SDG

A; in our formulation, transitions in N and σ_f are all environment moves[4], while transitions in A are control moves[5].

3. Lastly, a distributed positional strategy for player-0 in a SDG can be defined analogously as to uniquely select an action from the set $\bigcup \sigma_{\alpha_j} \langle \wedge_{m=1,\ldots,n_A}, pc_{\alpha_{jm}} \rangle$, for all A_i and for all program counter j defined in $\overline{\sigma_i}$. Each strategy should be insensitive of contents in other symbolic local games.

We now summarize the logical flow of game creation using Figure 6.4.

- (a) Based on the fixed number of slots (for FT mechanisms) specified by the user, extend IM to IM_{frac} to contain fractional PC-values induced by the slot.
- (b) Create $IM_{frac+FT}$, including the sequence of choice actions (as specified in the SDG) by
  - Extracting action sequences defined in IM_{frac} to choices (step A.2.1).
  - Inserting FT choices (step A.2.2).
- (c) Introduce faults and partition player-0 and player-1 vertices: In engineering, a *fault model* specifies potential undesired behavior of a piece of equipment, such that engineers can predict the consequences of system behavior. Thus, a *fault* can be formulated with three tuples[6]:
  1. The fault type (an unique identifier, e.g., `MsgLoss`, `SensorError`).
  2. The maximum number of occurrences in each period.
  3. Additional transitions not included in the original specification of the system (*fault effects*).

  We perform the translation into a game using the following steps.
  - For (1), introduce variables to control the triggering of faults.

[4]As the definition of distributed games features multiple processes having no interactions among themselves but only with the environment, a SDG is also a distributed game. In the following section, our proof of results and algorithms are all based on DG.

[5]This constraint can be released such that transitions in A can either be control (normal) or environment (induced by faults) moves; here we leave the formulation as future work.

[6]For complete formulation of fault models, we refer readers to our earlier work [CBEK09].

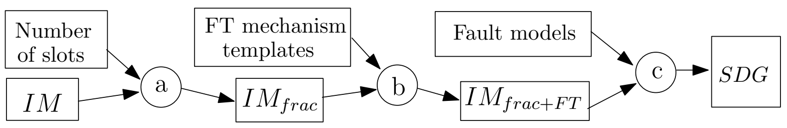


Figure 6.4: Creating the SDG from IM, FT mechanisms, and faults.

- For (2), introduce counters to constrain the maximum number of fault occurrences in each period.
- For (3), for each transition used in the component influenced by the fault, create a corresponding fault transition which is triggered by the variable and the counter; similarly create a transition with normal behavior (also triggered by the variable and the counter). Notice that our framework is able to model faults actuating on the FT mechanisms, for instance, the behavior of network loss on the newly introduced FT messages.

[Example] We outline how a game (focusing on fault modeling) is created with the example in sec. 6.2; similar approaches can be applied for input errors or message corruption; here the modeling of input (for `InputRead(m)`) is skipped.

- Create the predicate $pred$: $pred$ is evaluated to `false` in all cases except (a) when the boolean variable $occu$ (representing the network occupance) is evaluated to `true` and (b) when for all $i \in \{1, \ldots, n_A\}$, $\Delta_{next_i} = |\overline{\sigma_i}| + 1$ (end of period); the predicate partitions player-0 and player-1 vertices.
- For all process i and program counter j, the set of choice actions $\bigcup \sigma_{\alpha_j} \langle \wedge_{m=1,\ldots,n_A}, pc_{\alpha_{jm}} \rangle$ are generated based on the approach described previously.
- Create variable $v_f \in V_f$, which is used to indicate whether the fault (`MsgLoss`) has been activated in this period.
- In this example, as the maximum number of fault occurrences in each period is 1, we do not need to create additional counters.
- For each message sending transition t in the network, create two normal transitions $(v_f = \texttt{true} \wedge v'_f = \texttt{true}) \wedge t$ and $(v_f = \texttt{false} \wedge v'_f = \texttt{false}) \wedge t$ in the game.
- For each message sending transition t in the network, generate a transition t' where the message is sent, but the value is not updated in the destination. Create a fault transition $(v_f = \texttt{false} \wedge v'_f = \texttt{true}) \wedge t'$ in the game.
- Define σ_f to control v_f: if for all $i \in \{1, \ldots, n_A\}$, $\Delta_{next_i} = |\overline{\sigma_i}| + 1$, then update v_f to `false` as Δ_{next_i} updates to 1 (reset the fault counter at the end of the period).

6.5 Step B: Solving Distributed Games

We summarize the result from [MW03] as a general property of distributed games.

Theorem 6. *There exists distributed games with global winning strategy but (a) without dis-*

tributed memoryless strategies, or (b) all distributed strategies require memory. In general, for a finite distributed game, it is undecidable to check whether a distributed strategy exists from a given position [MW03].

As the problem is undecidable in general, we restrict our interest in finding a distributed positional strategy for player 0, if there exists one. We also focus on games with reachability winning conditions. By posing the restriction, the problem is NP-Complete.

Theorem 7. *[$PositionalDG_0$] Given a distributed game $\mathcal{G} = (\mathcal{V}_0 \uplus \mathcal{V}_1, \mathcal{E})$, an initial state $x = (x_1, \dots, x_n)$ and a target state $t = (t_1, \dots, t_n)$, deciding whether there exists a positional (memoryless) distributed strategy for player-0 from x to t is NP-Complete.*

Proof. We first start by recalling the definition of attractor, a term which is commonly used in the game and later applied in the proof. Given a game graph $G = (V_0 \uplus V_1, E)$, for $i \in \{0, 1\}$ and $X \subseteq V$, the map $\text{attr}_i(X)$ is defined by

$$\text{attr}_i(X) := X \cup \{v \in V_i \mid vE \cap X \neq \emptyset\} \cup \{v \in V_{1-i} \mid \emptyset \neq vE \subseteq X\},$$

i.e., $\text{attr}_i(X)$ extends X by all those nodes from which either player i can move to X within one step or player $1-i$ cannot prevent to move within the next step. (vE denotes the set of successors of v.) Then $\text{Attr}_i(X) := \bigcup_{k \in \mathbb{N}} \text{attr}_i^k(X)$ contains all nodes from which player i can force any play to visit the set X.

We continue our argument as follows.

[NP] The reachability problem for a distributed game can be solved in NP: a solution instance $\xi = \langle f_1, \dots, f_n \rangle$ is a strategy which selects exactly one edge for every control vertex in the local game. As the distributed game graph is known, after the selection we calculate the reachability attractor $\text{Attr}_0(\{t\})$ of the distributed game: during the calculation we overlook transitions which is not selected (in the strategy) in the local game. This means that in the distributed game, to add a control vertex $v \in \mathcal{V}_0$ to the attractor using the edge (v, u), we must ensure that $\forall j \in \{1, \dots, n\}.\ (proj(v_i, j) \in V_{0_j} \rightarrow proj(u, j) = proj(f_j(v), j))$. Lastly, we check if the initial state is contained; the whole calculation and checking process can be done in deterministic P-time.

[NP-C] For completeness proof, we perform a reduction from 3SAT to the finding of positional strategies in a distributed game. Given a set of 3CNF clauses $\{C_1, \dots, C_m\}$ under the set of literals $\{var_1, \overline{var_1}, \dots, var_n, \overline{var_n}\}$ and variables $\{var_1, \dots, var_n\}$, the distributed game $\mathcal{G}$ is created as follows (see Figure 6.5 for illustration):

- Create 3 local games G_1, G_2, and G_3, where for $G_i = (V_{0_i} \uplus V_{1_i}, E_i)$:
  - $V_{0_i} = \{var_1, \dots, var_n\}$, $V_{1_i} = \{S, T_{var_1}, F_{var_1}, \dots, T_{var_n}, F_{var_n}\}$.
  - $E_i = \bigcup_{j=1,\dots,n} \{(var_j, T_{var_j}), (var_j, F_{var_j})\}$.
- Create local game $G_4 = (V_{0_4} \uplus V_{1_4}, E_4)$:
  - $V_{0_4} = \{OK_0, NO_0\} \cup \bigcup_{j=1,\dots,m+n} \{v_{j0}\}$.
  - $V_{1_4} = \{S, OK_1, NO_1\} \cup \bigcup_{j=1,\dots,m+n} \{v_{j1}\}$.
  - $E_4 = \bigcup_{j=1,\dots,m+n} \{(v_{j0}, v_{j1})\} \cup \{(OK_0, OK_1), (NO_0, NO_1)\}$.

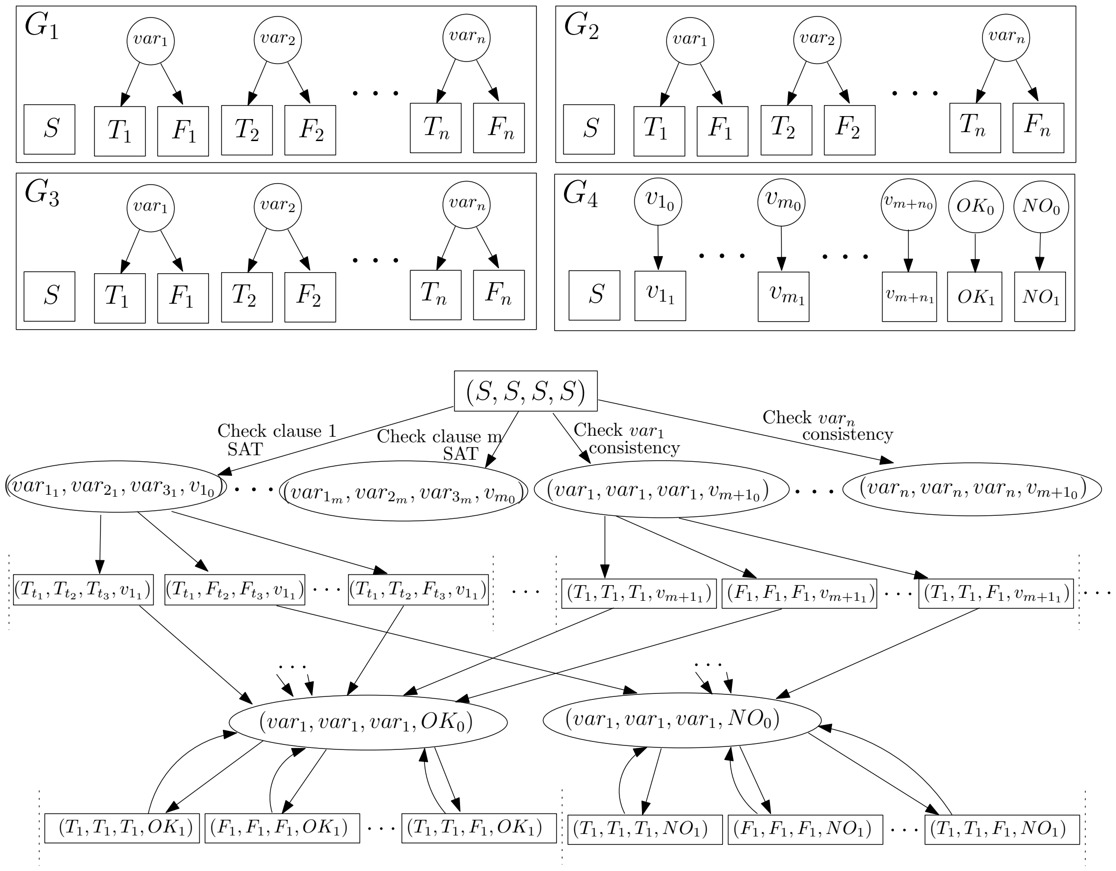

G_1
G_2
G_3
G_4
var_1
var_2
var_n
S
T_1
F_1
T_2
F_2
T_n
F_n
v_{1_0}
v_{m_0}
v_{m+n_0}
OK_0
NO_0
v_{1_1}
v_{m_1}
v_{m+n_1}
OK_1
NO_1
(S, S, S, S)
Check clause 1 SAT
Check clause m SAT
Check var_1 consistency
Check var_n consistency
$(var_{1_1}, var_{2_1}, var_{3_1}, v_{1_0})$
$(var_{1_m}, var_{2_m}, var_{3_m}, v_{m_0})$
$(var_1, var_1, var_1, v_{m+1_0})$
$(var_n, var_n, var_n, v_{m+1_0})$
$(T_{t_1}, T_{t_2}, T_{t_3}, v_{1_1})$
$(T_{t_1}, F_{t_2}, F_{t_3}, v_{1_1})$
$(T_{t_1}, T_{t_2}, F_{t_3}, v_{1_1})$
$(T_1, T_1, T_1, v_{m+1_1})$
$(F_1, F_1, F_1, v_{m+1_1})$
$(T_1, T_1, F_1, v_{m+1_1})$
$(var_1, var_1, var_1, OK_0)$
$(var_1, var_1, var_1, NO_0)$
(T_1, T_1, T_1, OK_1)
(F_1, F_1, F_1, OK_1)
(T_1, T_1, F_1, OK_1)
(T_1, T_1, T_1, NO_1)
(F_1, F_1, F_1, NO_1)
(T_1, T_1, F_1, NO_1)

Figure 6.5: Illustrations for the reduction from 3SAT to $PositionalDG_0$.

- Second, create the distributed game $\mathcal{G}$ from local games above, and define the set of environment transition to include the following types using the 3SAT problem:
  1. (Intention to check SAT) In the 3SAT problem, for clause $C_i = (l_{1_i} \vee l_{2_i} \vee l_{3_i})$, let the variable for literals $l_{1_i}, l_{2_i}, l_{3_i}$ be $var_{1_i}, var_{2_i}, var_{3_i}$. Create a transition in the distributed game from (S, S, S, S) to $(var_{1_i}, var_{2_i}, var_{3_i}, v_{i_0})$.
  2. (Intention to check consistency) In the 3SAT problem, for variable var_i, Create a transition in the distributed game from (S, S, S, S) to $(var_i, var_i, var_i, v_{m+i_0})$.
  3. (Result of clause) In the 3SAT problem, for clause $C_i = (l_{1_i} \vee l_{2_i} \vee l_{3_i})$, let the variable for the clause be $var_{1_i}, var_{2_i}, var_{3_i}$. We refer the vertex evaluating var_{j_i} as `true` to T_i in the local game G_j; similarly, we use F_i for a variable being evaluated `false`. For each clause C_i, enumerate over 8 cases for the assignments of $var_{1_i}, var_{2_i}, var_{3_i}$ which make C_i true.
     a) For cases which makes the assignment true, create an edge from the assignment to $(var_1, var_1, var_1, OK_0)$; for example, if $var_{1_i} = $ `true`, $var_{2_i} =$ `false`, $var_{3_i} =$ `true` makes a satisfying assignment to C_i, create an edge $((T_{1_i}, F_{2_i}, T_{3_i}, v_{i_1}), (var_1, var_1, var_1, OK_0))$.

   b) For cases which makes the assignment false, create an edge from the assignment to $(var_1, var_1, var_1, NO_0)$.

4. (Result of variable consistency) For all $i \in \{1, \ldots, n\}$:
   a) Create two edges $((T_i, T_i, T_i, v_{m+i_1}), (var_1, var_1, var_1, OK_0))$ and $((F_i, F_i, F_i, v_{m+i_1}), (var_1, var_1, var_1, OK_0))$.
   b) For other 6 combinations $(T_i, F_i, F_i, v_{m+i_1}), (F_i, F_i, T_i, v_{m+i_1})$, $(T_i, F_i, F_i, v_{m+i_1}), (T_i, T_i, F_i, v_{m+i_1}), (F_i, T_i, T_i, v_{m+i_1})$, $(T_i, F_i, T_i, v_{m+i_1})$, create edges to $(var_1, var_1, var_1, NO_0)$.

5. (Continuous execution) For all $i \in \{1, \ldots, n\}$:
   a) For all combinations $(T_i, F_i, F_i, OK_1), (F_i, F_i, T_i, OK_1), (T_i, F_i, F_i, OK_1)$, $(F_i, F_i, T_i, OK_1), (T_i, F_i, F_i, OK_1), (T_i, T_i, F_i, OK_1), (F_i, T_i, T_i, OK_1)$, (T_i, F_i, T_i, OK_1), create edges to $(var_1, var_1, var_1, OK_0)$.
   b) For all combinations $(T_i, F_i, F_i, NO_1), (F_i, F_i, T_i, NO_1), (T_i, F_i, F_i, NO_1)$, $(F_i, F_i, T_i, NO_1), (T_i, F_i, F_i, NO_1), (T_i, T_i, F_i, NO_1), (F_i, T_i, T_i, NO_1)$, (T_i, F_i, T_i, NO_1), create edges to $(var_1, var_1, var_1, NO_0)$.

We claim that $\{C_1, \ldots, C_m\}$ is satisfiable iff $\mathcal{G}$ has a positional distributed strategy to reach $(var_1, var_1, var_1, OK_0)$ from (S, S, S, S).

1. If $\{C_1, \ldots, C_m\}$ is satisfiable, let the set of satisfying literals be L', and assume that for all literals, in each pair $(var_i, \overline{var_i})$ exactly one of them is in L' (this is always possible). For the distributed game $\mathcal{G}$, in local games G_1, G_2 and G_3, let the positional strategy for control vertex var_i move to T_i if $var_i \in L'$, and move to F_i if $\overline{var_i} \in L'$ (for G_4, simply use the local edge). In a play, as player-1 starts the move, any of his selection leads to a player-0 vertex:
   - If player-1 choose edges of type 1 (intension to check the clause of SAT), for G_1, G_2 and G_3, the vertex uses its positional strategy, which corresponds to the assignment in the clause. The combined move then forces player-1 to choose an edge of type 3(a), leading to the target state.
   - If player-1 choose edges of type 2 (intension to check the consistency), as the positional strategies for G_1, G_2 and G_3 are all derived from the same satisfying instance of the 3SAT problem, for each strategy, it performs the same move from var_i to T_i or to F_i; the combined move of player-0 forces player-1 to choose an edge of type 4(a), leading to the target state.
2. Consider a distributed positional strategy $\langle f_1, f_2, f_3, f_4 \rangle$ which reaches $(var_1, var_1, var_1, OK_0)$ from (S, S, S, S). In G_1, for each control vertex var_i, it points to T_i or F_i. The positional strategy of G_1 generates a satisfying instance of the 3SAT problem:
   - Assign var_i in the 3SAT problem to `true` if the strategy points vertex var_i in G_1 to T_i.
   - Assign var_i in the 3SAT problem to `false` if the strategy points vertex var_i in G_1 to F_i.

We analyze the size of the game and the time required to perform the reduction.

- For $i = 1, 2, 3$, G_i contains $3n + 1$ vertices, and G_4 has $2(m + n + 2) + 1$ vertices.

As the total vertices of the distributed game is the product, it is polynomial to the original 3SAT problem instance.

- Consider the time required to perform reduction from 3SAT to $PositionalDG_0$:
  - For $i = 1, 2, 3$, G_i, they are constructed in $\mathcal{O}(n)$.
  - G_4 is constructed in $\mathcal{O}(m + n)$.
  - For the distributed game, vertices are constructed polynomial to m and n, more precisely $\mathcal{O}(n^3(m + n))$.
  - For edges in the distributed game, we consider the most complicated case, i.e. creating an edge of type 3. Yet it takes constant time to check and establish the connection, and for each player-1 vertex except (S, S, S, S) which has $m + n$ edges, at most 8 edges are created. Therefore, the total required time for edge construction is also polynomial to m and n.

Therefore, 3SAT $\leq_{poly}$ $PositioalDG_0$, which concludes the proof. □

With the NP-completeness proof, finding a distributed reachability strategy for distributed games amounts to the process of searching. For example, it is possible to perform a bounded-depth forward search over choices of local transitions: during the search, the selection of edges is constructed as a tree node in the search tree, and the set of reachable vertices (represented as BDD) based on the selection is also stored in the tree node. This method is currently implemented in our framework.

6.5.1 Solving Distributed Games using SAT Methods

Apart from the search method above, in this section we give an alternative approach based on a reduction to SAT. Madhusudan, Nam, and Alur [AMN05] designed the *bounded witness algorithm* (based on unrolling) for solving reachability (local) games. Although based on their experiment, the witness algorihm is not as efficient as the BDD based approach in centralized games, we find this concept potentially useful for solving distributed games. For this, we have created a variation (Algorithm 7) for this purpose.

To provide an intuition, first we paraphrase the concept of witness defined in [AMN05], a set of states which witnesses the fact that player 0 wins. In [AMN05], consider the generated SAT problem from a local game $G = (V_0 \uplus V_1, E)$ trying to reach from V_{init} to V_{goal}: for $i = 1, \ldots, d$ and vertex $v \in V_0 \uplus V_1$, variable $\langle v \rangle_i = \texttt{true}$ when one of the following holds:

1. $v \in V_{init}$ and $i = 1$ (if $v \notin V_{init} \wedge i = 1$ then $\langle v \rangle_i = \texttt{false}$).
2. $v \in V_{goal}$ (if $v \notin V_{goal} \wedge i = d$ then $\langle v \rangle_i = \texttt{false}$).
3. $v \in V_0 \setminus V_{goal}$ and $\exists v' \in V_0 \uplus V_1.\ \exists e \in E.\ \exists j > i.\ (e = (v, v') \wedge \langle v' \rangle_j = \texttt{true})$
4. $v \in V_1 \setminus V_{goal}$ and $\forall e = (v, v') \in E.\ \exists j > i.\ \langle v' \rangle_j = \texttt{true}$

This recursive definition implies that if v in V_0 (resp. in V_1) is not the goal but in the witness set, then exists one (resp. for all) successor v' which should either be (i) in a goal state or (ii) also in the witness: note that for (ii), the number of allowable steps to reach the goal is decreased by one. This definition ensures that all plays defined in the witness

Algorithm 7: PositionalDistributedStrategy_BoundedSAT_0

```
Data: Distributed game graph $\mathcal{G} = (\mathcal{V}_0 \uplus \mathcal{V}_1, \mathcal{E})$, set of initial states $V_{init}$, set of target
      states $V_{goal}$, the unrolling depth $d$
Result: Output: whether a distributed positional strategy exists to reach $V_{goal}$ from
      $v_{init}$
begin
    let clauseList := getEmptyList() /* Store all clauses for SAT solvers */
    /* STEP 1: Variable creation */
    for $v = (v_1, \ldots, v_m) \in \mathcal{V}_0 \uplus \mathcal{V}_1$ do
        create $d$ boolean variables $\langle v_1, \ldots, v_m \rangle_1, \ldots, \langle v_1, \ldots, v_m \rangle_d$;
    for local control transition $e = (x_i, x'_i) \in E_i, x_i \in V_{0_i}$ do
        create boolean variable $\langle e \rangle$;
    /* STEP 2: Initial state constraints */
    for $v = (v_1, \ldots, v_m) \in \mathcal{V}_0 \uplus \mathcal{V}_1$ do
        if $(v_1, \ldots, v_m) \in V_{init}$ then
            clauseList.add([$\langle v_1, \ldots, v_m \rangle_1$])
        else
            clauseList.add([$\neg \langle v_1, \ldots, v_m \rangle_1$])
    /* STEP 3: Target state constraints */
    for $v = (v_1, \ldots, v_m) \in \mathcal{V}_0 \uplus \mathcal{V}_1$ do
        if $(v_1, \ldots, v_m) \in V_{goal}$ then
            clauseList.add([$\langle v_1, \ldots, v_m \rangle_1 \wedge \ldots \wedge \langle v_1, \ldots, v_m \rangle_d$])
        else
            clauseList.add([$\neg \langle v_1, \ldots, v_m \rangle_d$])
    /* STEP 4: Unique selection of local transitions (for distributed positional strategy)
    */
    for local control transition $e = (x_i, x'_i) \in E_i, x_i \in V_{0_i}$ do
        for local transition $e_1 = (x_i, x'_{i_1}), \ldots, e_k = (x_i, x'_{i_k}) \in E_i, e_1 \ldots e_k \neq e$ do
            clauseList.add([$\langle e \rangle \Rightarrow (\neg \langle e_1 \rangle \wedge \ldots \wedge \neg \langle e_k \rangle)$])
    /* STEP 5: If a control vertex is in the attractor (winning region) but not a goal,
    an edge should be selected to reach the goal state */
    for $v = (v_1, \ldots, v_m) \in \mathcal{V}_0$ do
        for $v_i, i = 1, \ldots, m$ do
            if $v_i \in V_{0_i} \setminus V_{goal}$ then
                let $\bigcup_j e_j$ be the set of local transitions starting from $v_i$ in $G_i$
                if $\bigcup_j e_j \neq \phi$ then
                    clauseList.add([$(\bigvee_{i=1 \ldots d} \langle v \rangle_i) \Rightarrow (\bigvee_j \langle e \rangle_j)$])
    continue with Algorithm 8.
```

Algorithm 8: PositionalDistributedStrategy_BoundedSAT_0 (continuing Algorithm 7)

```
begin
    /* STEP 6: Impact of control edge selection (simultaneous progress) */
    for $v = (v_1, \ldots, v_m) \in \mathcal{V}_0$ do
        forall the edge combination $(e_1, \ldots, e_m)$: $e_i = (v_i, v'_i) \in E_i$ when $v_i \in V_{0_i}$ or
        $e_i = (v_i, v_i)$ when $x_i \in V_{1_i}$ do
            /* $e_i = (v_i, v_i)$ when $x_i \in V_{1_i}$ are simply dummy edges for ease of
            formulation */
            for $j = 1, \ldots, d-1$ do
                clauseList.add([$\langle v_1, \ldots, v_m \rangle_j \Rightarrow ((\bigwedge_{\{i | v_i \in V_{0_i}\}} \langle e_i \rangle) \Rightarrow (\langle v'_1, \ldots, v'_m \rangle_{j+1}))$])
    /* STEP 7: Impact of environment vertex */
    for environment vertex $v = (v_1, \ldots, v_m) \in \mathcal{V}_1$ do
        let the set of successors be $\bigcup_i v_i$; for $j = 1, \ldots, d-1$ do
            clauseList.add([$\langle v \rangle_j \Rightarrow (\bigwedge_i (\langle v_i \rangle_{j+1} \vee \ldots \vee, \langle v_i \rangle_d))$]);
    /* STEP 8: Invoke the SAT solver: return true when satisfiable */
    return invokeSATsolver(clauseList)
```

reaches the goal from the initial state within $d-1$ steps: If a play (starting from initial state) has proceeded $d-1$ steps and reached $u \notin V_{goal}$, then based on (2), $\langle u \rangle_d$ should be `false`. However, based on (1), (3), (4) the $\langle u \rangle_d$ should be set to `true` (reachable from initial states using $d-1$ steps). Thus the SAT problem should be unsatisfiable.

In general, Algorithm 7 creates constraints based on the above concept, but compared to the bounded local game reachability algorithm in [AMN05], it contains slight modifications:

1. When a variable $\langle v \rangle_i$ is evaluated to `true`, it means that vertex v can reach the target state within $d-i$ steps, which is the same as what is defined in [AMN05]. However, we introduce more variables for edges in local games, which is shown in STEP 1: when a variable $\langle e \rangle$ is evaluated to `true`, the distributed strategy uses the local transition e.
2. To achieve locality, we must include constraints specified in STEP 4: the positional (memoryless) strategy disallows to change the use of local edges from a given vertex.
3. We modify the impact of control edge selection in STEP 6 by adding an additional implication "$\langle e \rangle \Rightarrow$" over the original constraint in the witness algorithm [AMN05]. Here as in Mohalik and Walukiwitz's formulation, all subgames in a control position should proceed a move (the progress of a global move is a combination of local moves), we need to create constraints considering all possible local edge combinations.

We explain the concept of the witness algorithm (Algorithm 7 and 8) using Figure 6.6, where a sample witness of an imaginary distributed game is represented as a tree structure. We may interpret a witness as follows: a witness describes how the strategy un-

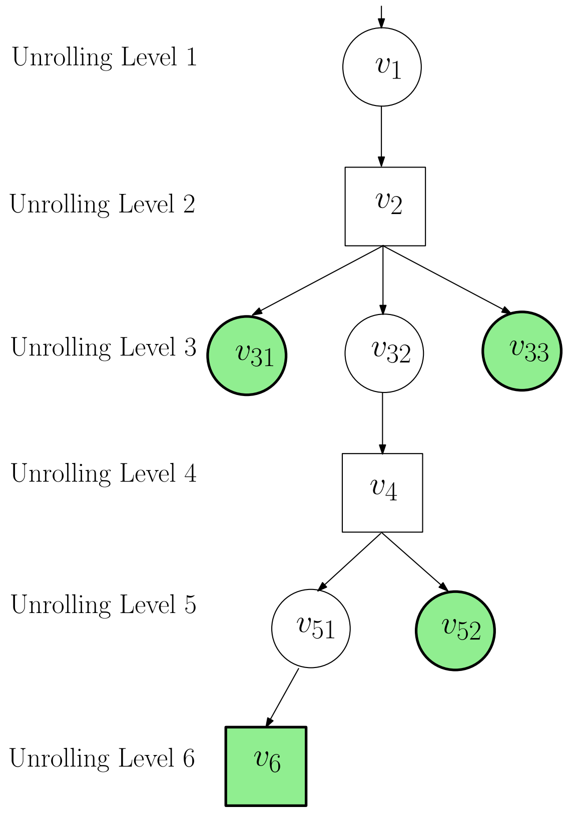


Figure 6.6: The concept of witness.

avoidably leads each possible play starting from the initial location to a goal location. In Figure 6.6, a goal state is labeled with the green color (e.g., v_{31}). Starting from the initial state v_1 (instance of STEP 2 constraints), first the unique distributed positional strategy decides the unique successor v_2 (instance of STEP 4 and 6 constraints). From v_1, it has three possible successors v_{31}, v_{32}, v_{33}. The condition where three successors should all lead to a goal state is captured by STEP 7 constraints. For v_{31} and v_{33}, they are goal states (instance of STEP 3 constraints). For v_{32}, as it is not a goal state, again the unique distributed positional strategy decides the next move; however, as it inevitably reaches goal states (with a recursive probing), the corresponding Boolean variable in the formula is set to `true`. With back propagation the corresponding variable of v_2 is also set to `true`. Following the argument above, it is not difficult to observe that if the unrolling depth equals 6, the satisfiability of the generated SAT formula indeed creates such a witness of the distributed game.

In appendix B, we give an alternate algorithm working with different formulation of distributed games where in each control location, only one local game can move: a run of the game may execute multiple local moves until it reaches a state where all local games are in an environment position. We find this alternative formulation closer to the interleaving semantics of distributed systems.

6.6 Step C: Conversion from Strategies to Concrete Implementations

Once when the distributed game has returned a positive result, and assume that the result is represented as an IM, the remaining problem is to check whether the synthesized result can be translated to PISEM and thus further to concrete implementation. If for each existing action or newly generated FT mechanism, the worst case execution time is known (with available WCET tools, e.g., AbsInt[7]), then we can always answer whether the system is implementable by a full system rescheduling, which can be complicated. Nevertheless, based on our system modeling (assumption with a globally synchronized clock), perform modification on the release time or the deadline on existing actions from the synthesized IM can be translated to a linear constraint system, as in the synthesized IM each action contains a timing precondition based on program counters. Here we give a simplified algorithm which performs *local timing modification (LTM)*. Intuitively, LTM means to perform partitions on either

1. the interval d between the deadline of action $\sigma_{\lfloor \frac{a}{b} \rfloor}$ and release time of $\sigma_{\lceil \frac{a}{b} \rceil}$, if (a) $\sigma_{\frac{a}{b}}$ exists and (b) $d \neq 0$, or
2. the execution interval of action $\sigma_{\lfloor \frac{a}{b} \rfloor}$, if $\sigma_{\frac{a}{b}}$ exists.

In the algorithm, we assume that for every action σ_d, $d \in \mathbb{N}$ where FT mechanisms are not introduced between σ_d and σ_{d+1} during synthesis, its release-time and deadline should not change; this assumption can be checked later or added explicitly to the constraint system under solving (but it is not listed here for simplicity reasons). Then we solve a constraint system to derive the release time and deadline of all FT actions introduced. Algorithm 9 performs such execution[8]: for simplicity assume at most one FT action exists between two actions σ_i, σ_{i+1}; in our implementation this assumption is released:

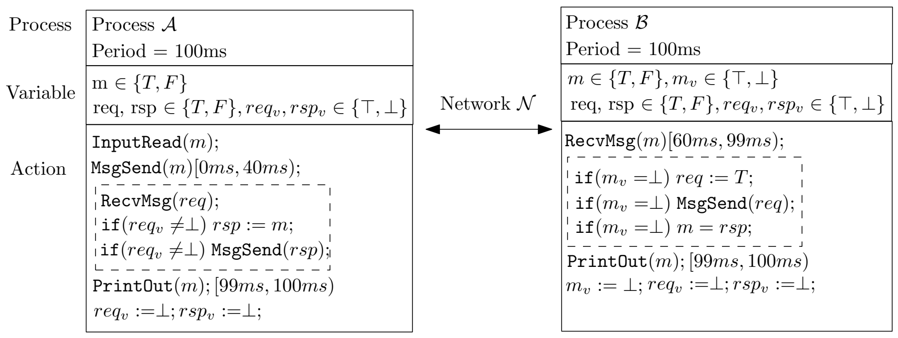
- Item (1) performs a interval split between $\sigma_{\lfloor \frac{a}{b} \rfloor}$ and $\sigma_{\frac{a}{b}}$.
- Item (3) assigns the deadline of $\sigma_{\lfloor \frac{a}{b} \rfloor}$ to be the original deadline of $\sigma_{\frac{a}{b}}$.
- Item (4), (5) ensure that the reserved time interval is greater than the WCET.
- Item (6) to (11) introduce constraints from other processes:
  - Item (6) (7) (8) consider existing actions which do not change the deadline and release time; for these fetch the timing information from PISEM.
  - Item (9) (10) (11) consider newly introduced actions or existing actions which change their deadline and release time; for these actions use variables to construct the constraint.
- Item (12) is a conservative dependency constraint between $\sigma_{\frac{a}{b}}$ and a send σ_d.

[7]`http://www.absint.com/`

[8]Here we list case 2 only; for case 1 similar analysis can be applied.

Algorithm 9: LocalTimingModification

```
Data: Original PISEM $\mathcal{S} = (\mathcal{A}, \mathcal{N}, \mathcal{T})$, synthesized IM $S = (A, N)$
Result: For each $\sigma_{\frac{a}{b}}$ and $\sigma_{\lfloor\frac{a}{b}\rfloor}$, their execution interval $[\alpha_{\frac{a}{b}}, \beta_{\frac{a}{b}})$, $[\alpha_{\lfloor\frac{a}{b}\rfloor}, \beta_{\lfloor\frac{a}{b}\rfloor})$
For convenience, use ($X\ in\ \mathcal{S}$) to represent the retrieved value $X$ from PISEM $\mathcal{S}$.
begin
    for $\sigma_{\frac{a}{b}}[\wedge_{m=1\dots n_A}[pc_{\frac{a}{b},m_{low}}, pc_{\frac{a}{b},m_{up}})]$ in $\overline{\sigma_i}$ of $A_i$ do
        let $\alpha_{\frac{a}{b}}, \beta_{\frac{a}{b}}, \alpha_{\lfloor\frac{a}{b}\rfloor}, \beta_{\lfloor\frac{a}{b}\rfloor}$ // Create a new variable for the constraint system
        /* Type A constraint: causalities within the process */
1       $constraints$.add($\alpha_{\frac{a}{b}} = \beta_{\lfloor\frac{a}{b}\rfloor}$)
2       $constraints$.add($\alpha_{\lfloor\frac{a}{b}\rfloor} = (\alpha_{\lfloor\frac{a}{b}\rfloor} in\ \mathcal{S})$)
3       $constraints$.add($\beta_{\frac{a}{b}} = (\beta_{\lfloor\frac{a}{b}\rfloor} in\ \mathcal{S})$)
4       $constraints$.add($\beta_{\frac{a}{b}} - \alpha_{\frac{a}{b}} > WCET(\sigma_{\frac{a}{b}})$)
5       $constraints$.add($\beta_{\lfloor\frac{a}{b}\rfloor} - \alpha_{\lfloor\frac{a}{b}\rfloor} > WCET(\sigma_{\lfloor\frac{a}{b}\rfloor})$)
    /* Type B constraint: causalities crossing different processes */
    for $\sigma_{\frac{a}{b}}[\wedge_{m=1\dots n_A}[pc_{\frac{a}{b},m_{low}}, pc_{\frac{a}{b},m_{up}})]$ in $\overline{\sigma_i}$ of $A_i$ do
        for $\sigma_d[\wedge_{m=1\dots n_A}[pc_{d,m_{low}}, pc_{d,m_{up}})]$ in $\overline{\sigma_j}$ of $A_j$ do
            if $d \in \mathbb{N}$ and not exists $\sigma_{\frac{x}{y}} \in \overline{\sigma_j}$ where $\lfloor\frac{x}{y}\rfloor = d$ then
6               if $pc_{d,j_{up}} < pc_{\frac{a}{b},j_{low}}$ then $constraints$.add(($\beta_d\ in\ \mathcal{S}$) $< \alpha_{\frac{a}{b}}$)
7               if $pc_{d,j_{low}} > pc_{\frac{a}{b},j_{up}}$ then $constraints$.add(($\alpha_d\ in\ \mathcal{S}$) $> \beta_{\frac{a}{b}}$)
                if $\sigma_{\frac{a}{b}} := \mathtt{send}(pre, ind, n, dest, v, c) \wedge pc_{d,j_{low}} > pc_{\frac{a}{b},j_{up}}$ then
8                   $constraints$.add(($\alpha_d\ in\ \mathcal{S}$) $> \beta_{\frac{a}{b}} + WCMTT(n, ind)$)
            else
9               if $pc_{d,j_{up}} < pc_{\frac{a}{b},j_{low}}$ then $constraints$.add($\beta_d < \alpha_{\frac{a}{b}}$)
10              if $pc_{d,j_{low}} > pc_{\frac{a}{b},j_{up}}$ then $constraints$.add($\alpha_d > \beta_{\frac{a}{b}}$)
                if $\sigma_{\frac{a}{b}} := \mathtt{send}(pre, ind, n, dest, v, c) \wedge pc_{d,j_{low}} > pc_{\frac{a}{b},j_{up}}$ then
11                  $constraints$.add($\alpha_d > \beta_{\frac{a}{b}} + WCMTT(n, ind)$)
    /* Type C constraint: conservative data dependency constraints */
    for $\sigma_{\frac{a}{b}}[\wedge_{m=1\dots n_A}[pc_{\frac{a}{b},m_{low}}, pc_{\frac{a}{b},m_{up}})]$ in $\overline{\sigma_i}$ of $A_i$ do
        for $\sigma_d[\wedge_{m=1\dots n_A}(pc_{d,m_{low}}, pc_{d,m_{up}})]$ in $\overline{\sigma_j}$ of $A_j$ do
            if $\sigma_d := \mathtt{send}(pre, ind, n, dest, v, c) \wedge \sigma_{\frac{a}{b}}$ reads variable $c \wedge pc_{d,j_{up}} < pc_{\frac{a}{b},j_{low}}$
            then
12              $constraints$.add(($\beta_d\ in\ \mathcal{S}$) $+ WCMTT(n, ind) < \alpha_{\frac{a}{b}}$)
    solve $constraints$ using (linear) constraint solvers.
```



```
Process A
Period = 100ms
m ∈ {T, F}
req, rsp ∈ {T, F}, req_v, rsp_v ∈ {⊤, ⊥}
InputRead(m);
MsgSend(m)[0ms, 40ms);
  RecvMsg(req);
  if(req_v ≠⊥) rsp := m;
  if(req_v ≠⊥) MsgSend(rsp);
PrintOut(m); [99ms, 100ms)
req_v :=⊥; rsp_v :=⊥;
```

```
Process B
Period = 100ms
m ∈ {T, F}, m_v ∈ {⊤, ⊥}
req, rsp ∈ {T, F}, req_v, rsp_v ∈ {⊤, ⊥}
RecvMsg(m)[60ms, 99ms);
  if(m_v =⊥) req := T;
  if(m_v =⊥) MsgSend(req);
  if(m_v =⊥) m = rsp;
PrintOut(m); [99ms, 100ms)
m_v := ⊥; req_v :=⊥; rsp_v :=⊥;
```

Figure 6.7: An example where FT primitives are introduced for synthesis.

6.7 Implementation and Case Studies

For implementation, we have created our prototype software as an Eclipse-plugin, called GECKO (we refer readers to a full report in the appendix on the design methodology of Gecko), which offers an open-platform based on the model-based approach to facilitate the design, synthesis, and code generation for fault-tolerant embedded systems. Currently the engine implements the search-based algorithms, and the SAT-based algorithm is experimented independently under GAVS+.To evaluate our approach, here we reuse the example in sec. 6.2 and perform automatic tuning synthesis for the selected FT mechanisms.

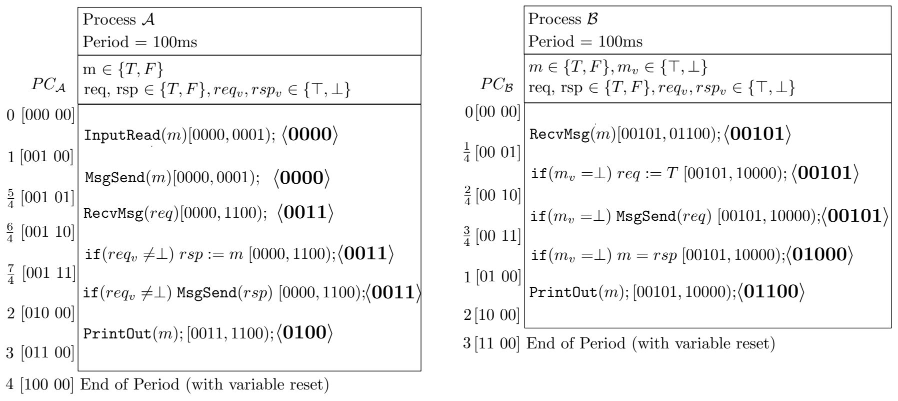
6.7.1 Example from Section 2

In this example, the user selects a set of FT mechanism templates with the intention to implement a *fail-then-resend* operation, which is shown in Figure 6.7. The selected patterns introduce two additional messages in the system, and the goal is to orchestrate multiple synchronization points introduced by the FT mechanisms between $\mathcal{A}$ and $\mathcal{B}$ (the timing in FT mechanisms is unknown). The fault model, similar to sec. 6.2, assumes that in each period at most one message loss occurs.

Once when GECKO receives the system description (including the fault model) and the reachability specification, it translates the system into a distributed game. In Figure 6.8, the set of possible control transitions are listed[9]; the solver generates an appropriate PC-precondition for each action to satisfy the specification. In Figure 6.8, bold numbers (e.g., $\langle \mathbf{0000} \rangle$) indicate the synthesized result. The time line of the execution (the synthesized result) is explained as follows:

1. Process $\mathcal{A}$ reads the input, sends `MsgSend`(m), and waits.
2. Process $\mathcal{B}$ first waits until it is allowed to execute (`RecvMsg`(m)). Then it performs a conditional send `MsgSend`(req) and waits.
3. Process $\mathcal{A}$ performs `RecvMsg`(req), following a conditional send `MsgSend`(rsp).

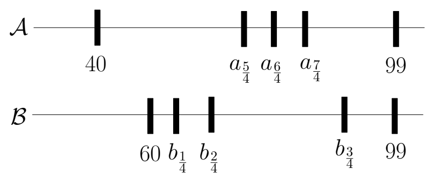
[9]In our implementation, the PC starts from 0 rather than 1; which is different from the formulation in IM and PISEM.



Process $\mathcal{A}$
Period $= 100$ms
m $\in \{T, F\}$
$PC_{\mathcal{A}}$ req, rsp $\in \{T, F\}, req_v, rsp_v \in \{\top, \bot\}$
0 [000 00]
`InputRead`$(m)[0000, 0001)$; $\langle \mathbf{0000} \rangle$
1 [001 00]
`MsgSend`$(m)[0000, 0001)$; $\langle \mathbf{0000} \rangle$
$\frac{5}{4}$ [001 01]
`RecvMsg`$(req)[0000, 1100)$; $\langle \mathbf{0011} \rangle$
$\frac{6}{4}$ [001 10]
`if`$(req_v \neq \bot)$ $rsp := m$ $[0000, 1100)$;$\langle \mathbf{0011} \rangle$
$\frac{7}{4}$ [001 11]
`if`$(req_v \neq \bot)$ `MsgSend`(rsp) $[0000, 1100)$;$\langle \mathbf{0011} \rangle$
2 [010 00]
`PrintOut`(m); $[0011, 1100)$;$\langle \mathbf{0100} \rangle$
3 [011 00]
4 [100 00] End of Period (with variable reset)

Process $\mathcal{B}$
Period $= 100$ms
$m \in \{T, F\}, m_v \in \{\top, \bot\}$
$PC_{\mathcal{B}}$ req, rsp $\in \{T, F\}, req_v, rsp_v \in \{\top, \bot\}$
0 [00 00]
`RecvMsg`$(m)[00101, 01100)$;$\langle \mathbf{00101} \rangle$
$\frac{1}{4}$ [00 01]
`if`$(m_v = \bot)$ $req := T$ $[00101, 10000)$; $\langle \mathbf{00101} \rangle$
$\frac{2}{4}$ [00 10]
`if`$(m_v = \bot)$ `MsgSend`(req) $[00101, 10000)$;$\langle \mathbf{00101} \rangle$
$\frac{3}{4}$ [00 11]
`if`$(m_v = \bot)$ $m = rsp$ $[00101, 10000)$;$\langle \mathbf{01000} \rangle$
1 [01 00]
`PrintOut`(m); $[00101, 10000)$;$\langle \mathbf{01100} \rangle$
2 [10 00]
3 [11 00] End of Period (with variable reset)

Figure 6.8: A concept illustration for the control choices in the generated game.



Type A		Type B		Type C
$a_{\frac{5}{4}} > 40$	$b_{\frac{1}{4}} - 60 > 1$	$a_{\frac{5}{4}} > b_{\frac{2}{4}} + 1 + 3$	$b_{\frac{1}{4}} > 40 + 3$	$b_{\frac{3}{4}} - 40 > 3$
$a_{\frac{6}{4}} - a_{\frac{5}{4}} > 1$	$b_{\frac{2}{4}} - b_{\frac{1}{4}} > 1$	$a_{\frac{6}{4}} > b_{\frac{2}{4}} + 1 + 3$	$b_{\frac{2}{4}} > 40 + 3$	
$a_{\frac{7}{4}} - a_{\frac{6}{4}} > 1$	$b_{\frac{3}{4}} - b_{\frac{2}{4}} > 1$	$a_{\frac{7}{4}} > b_{\frac{2}{4}} + 1 + 3$	$b_{\frac{3}{4}} > a_{\frac{7}{4}} + 1 + 3$	
$99 - a_{\frac{7}{4}} > 1$	$99 - b_{\frac{3}{4}} > 1$	$a_{\frac{5}{4}} + 1 < b_{\frac{3}{4}}$	$b_{\frac{1}{4}} + 1 < a_{\frac{5}{4}}$	
		$a_{\frac{6}{4}} + 1 < b_{\frac{3}{4}}$	$b_{\frac{2}{4}} + 1 < a_{\frac{5}{4}}$	
		$a_{\frac{7}{4}} + 1 < b_{\frac{3}{4}}$	$b_{\frac{1}{4}} + 1 < 99$	

Figure 6.9: An illustration for applying LTM for the example in sec. 6.7.1, and the corresponding linear constraints.

4. Process $\mathcal{B}$ performs conditional assignment, which assigns the value of rsp to m, if m_v is empty.

We continue the case study by stating assumptions over hardware and timing; these can be specified in GECKO as properties of the model.

1. Process $\mathcal{A}$ and $\mathcal{B}$ are running on two Texas Instrument LM3S8962 development boards[10] under FreeRTOS[11] (a real-time operating system), and messages are communicating over a CAN bus.
2. For each existing or FT action, its WCET on the hardware is 1ms.
3. For all messages communicating using the network, the WCMTT is 3ms.

We apply the LTM algorithm, such that we can generate timing constraints on dedicated hardware; these timing constraints will be translated to executable C code (based

[10] `http://www.luminarymicro.com/products/LM3S8962.html`
[11] `http://www.freertos.org/`

on FreeRTOS). Figure 6.9 is used to assist the explanation of LTM, where variables used in the linear constraint solver are specified as follows:

- $a_{\frac{5}{4}}$: release time for action "`RecvMsg`(req)" in process $\mathcal{A}$.
- $a_{\frac{6}{4}}$: release time for action "`if`($req_v \neq\perp$) $rsp := m$" (and similarly, the deadline for "`RecvMsg`(req)") in process $\mathcal{A}$.
- $a_{\frac{7}{4}}$: release time for action "`if`($req_v \neq\perp$) `MsgSend`(rsp)" in process $\mathcal{A}$.
- $b_{\frac{1}{4}}$: release time for action "`if`($m_v =\perp$) $req := T$" in process $\mathcal{B}$.
- $b_{\frac{2}{4}}$: release time for action "`if`($m_v =\perp$) `MsgSend`(req)" in process $\mathcal{B}$.
- $b_{\frac{3}{4}}$: release time for action "`if`($m_v =\perp$) $m = rsp$" in process $\mathcal{B}$.

As in process $\mathcal{A}$, there exists a time interval $[40, 99)$ between two existing actions `MsgSend`(m) and `PrintOut`(m), the LTM algorithm will prefer to utilize this interval than splitting [0, 40), as using $[40, 99)$ generates the least modification on the scheduling. The generated linear constraint system is also shown in Figure 6.9. An satisfying instance for $(a_{\frac{5}{4}}, a_{\frac{6}{4}}, a_{\frac{7}{4}}, b_{\frac{1}{4}}, b_{\frac{2}{4}}, b_{\frac{3}{4}})$ could be $(72, 77, 82, 62, 67, 87)$; instructions concerning the release time and the deadline for the generated fault-tolerant model can be annotated based on this.

6.7.2 Another Example

For the second example, the user selects an inappropriate set of FT mechanisms[12]. Compared to Figure 6.7, in process $\mathcal{A}$ an equality constraint "`if`($req_v =\perp$)" is used, instead of "`if`($req_v \neq\perp$)". In this way, the combined effect of FT mechanisms in Example B changes dramatically from that of Example A:

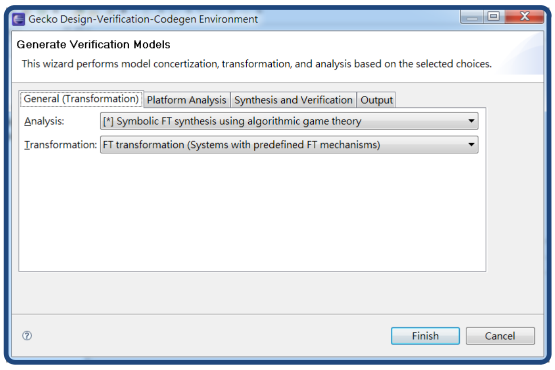
- When $\mathcal{B}$ does not receive m from $\mathcal{A}$, it sends a request command.
- When $\mathcal{A}$ receives a request message, it does not send the response; this violates the original intention of the designer.

Surprisingly, GECKO reports a positive result with an interesting sequence! For all FT actions in process $\mathcal{A}$, they should be executed with the procondition of $PC_{\mathcal{B}}$ equal to 0000, meaning that FT mechanisms in $\mathcal{A}$ are executed before `RecvMsg`(m) in $\mathcal{B}$ starts. In this way, $\mathcal{A}$ always sends the message `MsgSend`(rsp) containing the value of m, and as at most one message loss exists in one period, the specification is satisfied.

6.7.3 Discussion

Concerning the running time of the above two examples, the searching engine (based on forward searching + BDD for intermediate image storing) is able to report the result in 3 seconds, while constraint solving is also relatively fast (within 1 second). Our engine offers a translation scheme to dump the BDD to mechanisms in textual form; this process occupies most of the execution time. Note that the NP-completeness result

[12]This is originally a design mistake when we specify our FT mechanism patterns; however interesting results are generated.



Gecko Design-Verification-Codegen Environment

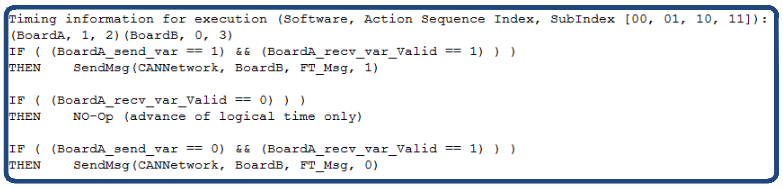
Generate Verification Models

This wizard performs model concertization, transformation, and analysis based on the selected choices.

Analysis: [*] Symbolic FT synthesis using algorithmic game theory

Transformation: FT transformation (Systems with predefined FT mechanisms)

(a) Pop-up window of Gecko

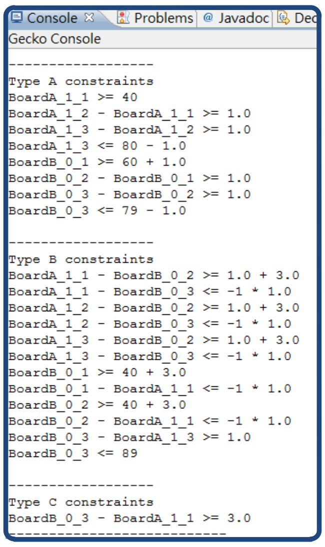


```
Timing information for execution (Software, Action Sequence Index, SubIndex [00, 01, 10, 11]):
(BoardA, 1, 2)(BoardB, 0, 3)
IF ( (BoardA_send_var == 1) && (BoardA_recv_var_Valid == 1) ) )
THEN    SendMsg(CANNetwork, BoardB, FT_Msg, 1)

IF ( (BoardA_recv_var_Valid == 0) ) )
THEN    NO-Op (advance of logical time only)

IF ( (BoardA_send_var == 0) && (BoardA_recv_var_Valid == 1) ) )
THEN    SendMsg(CANNetwork, BoardB, FT_Msg, 0)
```

(b) Synthesized mechanism (interleaving model; textual form)



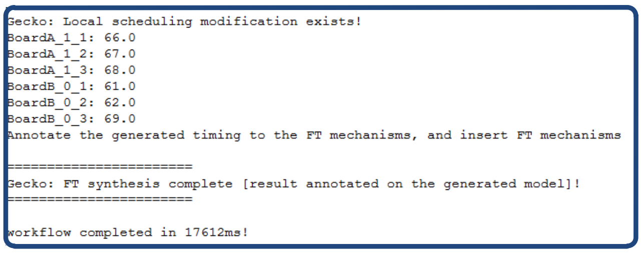
Gecko Console

```
------------------
Type A constraints
BoardA_1_1 >= 40
BoardA_1_2 - BoardA_1_1 >= 1.0
BoardA_1_3 - BoardA_1_2 >= 1.0
BoardA_1_3 <= 80 - 1.0
BoardB_0_1 >= 60 + 1.0
BoardB_0_2 - BoardB_0_1 >= 1.0
BoardB_0_3 - BoardB_0_2 >= 1.0
BoardB_0_3 <= 79 - 1.0

------------------
Type B constraints
BoardA_1_1 - BoardB_0_2 >= 1.0 + 3.0
BoardA_1_1 - BoardB_0_3 <= -1 * 1.0
BoardA_1_2 - BoardB_0_2 >= 1.0 + 3.0
BoardA_1_2 - BoardB_0_3 <= -1 * 1.0
BoardA_1_3 - BoardB_0_2 >= 1.0 + 3.0
BoardA_1_3 - BoardB_0_3 <= -1 * 1.0
BoardB_0_1 >= 40 + 3.0
BoardB_0_1 - BoardA_1_1 <= -1 * 1.0
BoardB_0_2 >= 40 + 3.0
BoardB_0_2 - BoardA_1_1 <= -1 * 1.0
BoardB_0_3 - BoardA_1_3 >= 1.0
BoardB_0_3 <= 89

------------------
Type C constraints
BoardB_0_3 - BoardA_1_1 >= 3.0
---------------------------
```

(c) Constraint system by LTM



```
Gecko: Local scheduling modification exists!
BoardA_1_1: 66.0
BoardA_1_2: 67.0
BoardA_1_3: 68.0
BoardB_0_1: 61.0
BoardB_0_2: 62.0
BoardB_0_3: 69.0
Annotate the generated timing to the FT mechanisms, and insert FT mechanisms

=======================
Gecko: FT synthesis complete [result annotated on the generated model]!
=======================

workflow completed in 17612ms!
```

(d) Results of synthesized timing constraints

Figure 6.10: Screenshots of GECKO when executing the example in sec. 6.7.1.

does not bring huge benefits, as another exponential blow-up caused by the translation from variables to states is also unavoidable: this is the reason why currently we use a forward search algorithm combining with BDDs in the implementation.

Nevertheless, this does not means that FT synthesis in practice is not possible; our argument is as follows:

1. We have indicated that this method is applicable for small examples (similar to the test case in the paper).
2. To fight with complexity we consider it important to respect the compositional (layered) approach used in the design of embedded systems: once when a system have been refined to several subsystems, it is more likely for our approach to be applicable.

6.8 Related Work

Verification and synthesis of fault tolerance is an active field [FKL08, LM94, KA00, GR09, BFS00, ORSvH95, AAE98, DF09]. Among all existing works, we find that the work closest to ours is by Kulkarni et.al. [KA00]. Here we summarize the differences in three aspects.

1. (Problem) As we are interested in real-time embedded systems, our starting model resembles existing formulations used in the real-time community, where time is explicitly stated in the model. Their work is more closely to protocol synthesis and the starting model is based on (a composition of) FSMs.
2. (Approach) As our original intention is to facilitate the pattern selection and tuning process, our approach does not seek for the synthesis of complete FT mechanisms and can be naturally connected to games (having a set of predefined moves). Contrarily, their results focus on synthesizing complete FT mechanisms, for example voting machines or mechanisms for Byzantine generals' problem.
3. (Algorithm) To apply game-based approach for embedded systems, our algorithms includes the game translation (timing abstraction) and constraint solving (for implementability). In addition, our game formulation enables us to connect and modify existing and rich results in algorithmic game solving: for instance, we reuse the idea of witness in [AMN05] for distributed games, and it is likely to establish connections between incomplete methods for distributed games and algorithms for games of imperfect information [DWDR06].

A recent work by Girault et.al. [GR09] follows similar methodologies (i.e., on protocol level FSMs) to [KA00] and performs discrete controller synthesis for fault-tolerance; the difference between our work and theirs follows the argument above.

Lastly, we would like to comment on the application of algorithmic games. Several important work for game analysis or LTL synthesis can be found from Bloem and Jobstmann et.al. (the program repair framework [JGB05]), Henzinger and Chatterjee et.al. (Alpaga and the interface synthesis [BCDW^{+}09, DHJP08]), or David and Larson et.al. (Uppaal TIGA [BCD^{+}07b]). One important distinction is that due to our system modeling, we naturally start from a problem of solving distributed games and need to

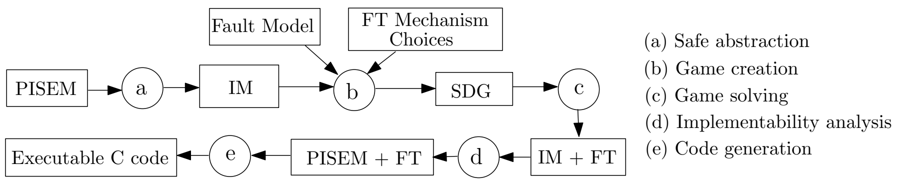


Figure 6.11: Concept illustration of the overall approach for fault-tolerant synthesis; IM+FT means that an IM model is equipped with FT mechanisms.

fight with undecidability immediately, while the above works are all based on a non-distributed setting.

6.9 Concluding Remarks

This chapter presents a comprehensive approach (see Figure 6.11 for concept illustration) for the augmenting of fault-tolerance for real-time distributed systems under a game-theoretic framework. We use simple yet close-to-reality models (PISEM) as a starting point of FT synthesis, translate PISEM to distributed games with safe abstractions, perform game solving and later implementability analysis. The above flow is experimented in a prototype, enabling us to utilize model-based development framework to perform FT synthesis. These mechanisms may have interesting applications in distributed process control and robotics. To validate our approach, we plan to increase the maturity of our prototype system and study new algorithms for performance gains.

6.10 Appendix

6.10.1 A. The Need of Reestimating the WCMTT in CAN Buses when FT messages are Introduced

To have an understanding whether newly introduced FT messages can change the existing networking behavior is both hardware and configuration dependent. In this section, we only describe the behavior when FT messages are introduced in a Control Area Network (CAN bus), which is widely used in automotive and automation domains. Here we give configuration settings (conditions) such that newly introduced messages do ***not*** influence the existing networking behavior. For details concerning the timing analysis of CAN, we refer readers to [TBW95, DBBL07].

Proposition 1. *Given an ideal CAN bus with message priority from 1 to k, when the three conditions are satisfied:*

1. *No message with priority k is not used in the existing network.*
2. *The predefined size of the message for priority k is larger than all messages with priority 1 to $k-1$,*
3. *All FT messages are having priority smaller or equal to k, and the size is less than the message size stated in (2).*

When the WCMTT is derived using the analysis in [TBW95], concerning the WCMTT of all messages with priority 1 *to* k*, it is indifferent to the newly introduced messages.*

Proof. (Outline) Based on the algorithm in [TBW95], for a message with priority $i \in \{1, \dots, k\}$, its timing behavior only changes with two factors:

- (a) The blocking time caused by a message with lower priority $j > i$ changes: when a message with lower priority changes to a bigger message size, the blocking time increases.
- (b) The interference from messages with higher priority $j < i$.

We proceed the argument as follows.

- For timing changes due to (b), as FT messages are all with lower priorities (based on condition 3), they do not create or increase interferences with this type.
- For timing changes due to (a), we separate two two cases:
  - As the size of all FT messages are smaller or equal than the message size specified in (2), then the timing behavior for messages with priority 1 to $k - 1$ do not change.
  - Lastly, although the message with priority k can change as it can now be blocked by a lower priority message, such message does not exist based on condition (1).

□

By the above information, in our framework we may assume that all messages transmitted in a CAN bus are with lowest priority $k + 1$, and then perform a simple timing analysis at Step A.2.2 before creating the game; in this way, the problem is Step A.2.2 ([**Problem 2**]) is safe to neglect.

6.10.2 B. Algorithm Modification for Interleaving of Local Games

(Remark) Compared to Mohalik and Walukiwitz's formulation, as in this formulation, only one subgame in a control position can proceed a move, we do not need to create constraints considering all possible combinations in STEP 6, which is required in the algorithm PositionalDistributedStrategy_BoundedSAT_0 (sec. 6.5).

6.10.3 C. GECKO: Tool-based Development of Light-weight Fault-tolerant Embedded Systems

6.10.3.1 Introduction

In this section, we present Gecko, a prototype framework for model-driven development of embedded (distributed) systems, focusing on fault-tolerance and dependability aspects. The goal of Gecko is to introduce fault-tolerance aspects into existing design flows of embedded systems with minimal overhead.

Algorithm 10: PositionalDistributedStrategy_ControlInLocalGameInterleaving_-BoundedSAT_0

```
Data: Distributed game graph $\mathcal{G} = (\mathcal{V}_0 \uplus \mathcal{V}_1, \mathcal{E})$, set of initial states $V_{init}$, set of target
      states $V_{goal}$, the unrolling depth $d$
Result: Output: whether a distributed positional strategy exists to reach $V_{goal}$ from
      $v_{init}$
begin
    let clauseList := getEmptyList() /* Store all clauses for SAT solvers */
    execute STEP 1 to STEP 5 mentioned in
    PositionalDistributedStrategy_BoundedSAT_0
    /* STEP 6: Impact of control edge selection */
    for local control transition $e = (x_i, x'_i) \in E_i, x_i \in V_{0_i}$ do
        for $v = (v_1, \ldots, v_m) \in \mathcal{V}_0 \uplus \mathcal{V}_1$ where $x_i = v_i$ do
            for $j = 1, \ldots, d-1$ do
                clauseList.add([$\langle e \rangle \Rightarrow (\langle v_1, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots, v_m \rangle_j \Rightarrow$
                $(\langle v_1, \ldots, v_{i-1}, x'_i, v_{i+1}, \ldots, v_m \rangle_{j+1} \vee \ldots \vee \langle v_1, \ldots, v_{i-1}, x'_i, v_{i+1}, \ldots, v_m \rangle_d))$])
    /* STEP 7: Impact of environment vertex */
    for environment vertex $v = (v_1, \ldots, v_m) \in \mathcal{V}_1$ do
        let the set of successors be $\bigcup_i v_i$; for $j = 1, \ldots, d-1$ do
            clauseList.add([$\langle v \rangle_j \Rightarrow (\bigwedge_i (\langle v_i \rangle_{j+1} \vee \ldots \vee, \langle v_i \rangle_d))$])
    /* STEP 8: Invoke the SAT solver: return true when satisfiable */
    return invokeSATsolver(clauseList)
```

The predecessor of Gecko is called FTOS, which was developed as a text-based modeling tool for the support of designing fault-tolerant systems. Technical details about FTOS can be found in [Buc08]. Since then, we have extended and reimplemented the tool with the following goals:

- Closing the visibility gap between models and implementations. As FTOS works over dedicated models of computation, a single-step transformation from model sketches to executable code is non-intuitive. By introducing intermediate models (functional and architectural) and several analysis techniques, we acquire stronger guarantees over the resulting artifact[13]. Furthermore, it enables a generalized usage for any application specific models to be introduced in the Gecko development flow.
- Giving more control to the user with ease of use. The tool should allow users to inspect intermediate results and to perform modifications on-the-fly. The latter case can be useful, in case an automatic global optimization of system aspects would require a search in a huge state space, while developers can find a nearly optimal solution based on their experience.

The resulting tool has been developed as a plug-in under the Eclipse Modeling Frame-

[13]We are aware of theoretical frameworks of composition and refinement process of systems, whilst it is not our current focus.

work (EMF) [emf, oaw], which is now called Gecko. It automates the implementation of extra-functional aspects with a focus on fault-tolerance, integrates formal verification, but also enables developers to manually optimize the system.

In the following, we introduce the design flow and components currently included in Gecko using figure 6.12.

6.10.3.2 Functional Concretization

In the first step, a given *application-specific model* in high-level descriptions is concretized into an *abstract actor model* that is based on concepts similar to actor-oriented languages, and contains explicit timing information based on specifications included in the original models or information added by the user within this step. In fig. 6.12, the application-specific model consists of four partial models, namely HW, SW, Fault, FT (for fault tolerance). During the concretization, only SW and FT are involved, where we mainly generate required actors, ports, and token transfers (similar to connectors in Ptolemy [EJL$^+$03], differences mentioned later) concretizing user specified fault-tolerance mechanisms. Fig. 6.13 illustrates an example from model sketches (upper part) to functional models (lower part). For the timing specification, we use the information contained in the SW model, which is based on the concept of logical execution time [HHK03, KS08].

The *actor model concretizer* in Gecko translates fault-tolerance mechanisms into according actors represented by a tree structure, whose elements can reference other elements of the model. To generate executable code for actors in later stages, for each platform only a generalized tree interpreter is required. This greatly reduces the possibility of programming errors when introducing new platforms, as no parameterized skeletons for each actor and each platform are required[14].

6.10.3.3 Function-Architectural Mapping under Local Scale

After the first transformation, different application-specific models are mapped to unified abstract actor models, allowing us to follow the same transformation scheme to generate *computational (software) models*. These models concretize the implementation of the communication between software components and the timing without concrete parameters. These parameters are derived in the next phases, when the concrete hardware is considered. These steps are performed by the *simple platform analyzer* as depicted in fig. 6.12.

The above process can be challenging concerning design space exploration with multi-objective function optimization. Therefore, we constrain ourselves, and assume that a major part of the functionalities are mapped to architectures by user-guided information, as it is currently specified in the FTOS software and hardware model. Our goal is to perform a ***local-scale*** design space exploration concerning extra-functional properties, i.e., we consider how to integrate newly generated actors (for dependabil-

[14]For instance, when concretizing actors in UPPAAL verification models, all &, <, > tokens are replaced by $\&amp;$, $\&lt;$, $\&gt;$. This can be easily achieved by a partial modification of the tree interpreter.

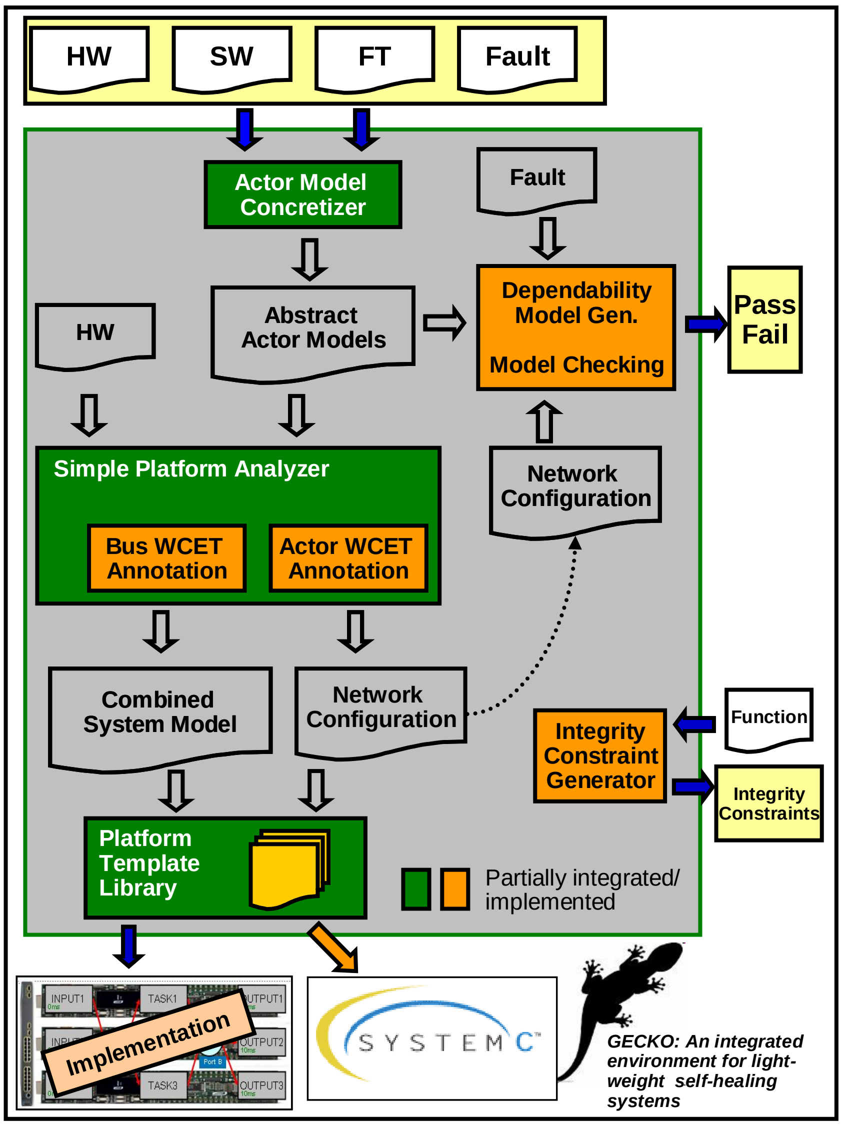


Figure 6.12: An overview for the flow of model transformations, code generation, and analysis scheme in Gecko.

ity, fault-tolerance) and newly introduced token transfers, such that the specified design constraints are still satisfied. Currently Gecko generates the mapping based on a small set of user-predefined parameters, and the analysis is done separately after the concretized model is generated. The resulting artifact is a combined model including network configurations (e.g., message groupings and priorities in the CAN bus).

The timing behavior in Gecko is based on (1) the required time for executing actor blocks (WCET) plus (2) the required time for token transfers (WCTT), which is grouped into messages in the architecture model. We follow two different approaches: a hard real-time approach based on temporarily disabling the system interrupts (not applicable for all applications) to assure the claimed WCET of actor blocks or a less stricter approach based on simulation to acquire confidence about the system behavior.

1. With respect to timing behavior of the network, currently only CAN buses are supported with analysis techniques. We implemented the *CAN bus WCTT analyzer* using the classical real-time scheduling theory [DBBL07, TBW95] based on resource contention. Nevertheless, the analysis technique does not scale to industrial examples, as in this idealized case, the CAN controller must have enough TxObjects to accommodate all the outgoing message streams. Our implementation template deploys the system based on this assumption, and raises alarms if it is violated. The mentioned problem is mainly an issue for models-of-computation with aperiodic behavior.
2. With respect to execution time of our automatically generated actors, to have stronger confidence for predictable timing, files containing the WCET of these actors (with parameterized form) can be read into Gecko as supplementary files.

6.10.3.4 Generation of Dependability Models

Regarding dependability, currently we are interested in whether a system with equipped fault-tolerant mechanisms is sufficient to resist faults defined in the fault-hypothesis. To achieve this goal, Gecko generates the dependability model from a combination of functional models, fault models, and network/hardware configuration files; the result is created in formats acceptable by the verification engine UPPAAL [BDL04]. Fig. 6.12 indicates required components for the generation of the dependability model:

1. The *network and hardware configurations* generated on the architectural level provide a sufficient abstraction on how faults are actuated. For instance, when a fault with type `MessageLoss` is considered, token transfers grouped in one single message can be lost simultaneously, which should be faithfully modeled.
2. The *abstract actor model* can be naturally translated into automata as input for formal verification engines. Gecko supports this translation and uses logical time / action causalities to avoid excessive use of variables in the verification model. Both actor actions and token transfers are represented by edges[15].

[15] An edge update takes zero time; time consumption is explicitly added, which is similar to discrete-event simulation in Ptolemy or SystemC [Sys].

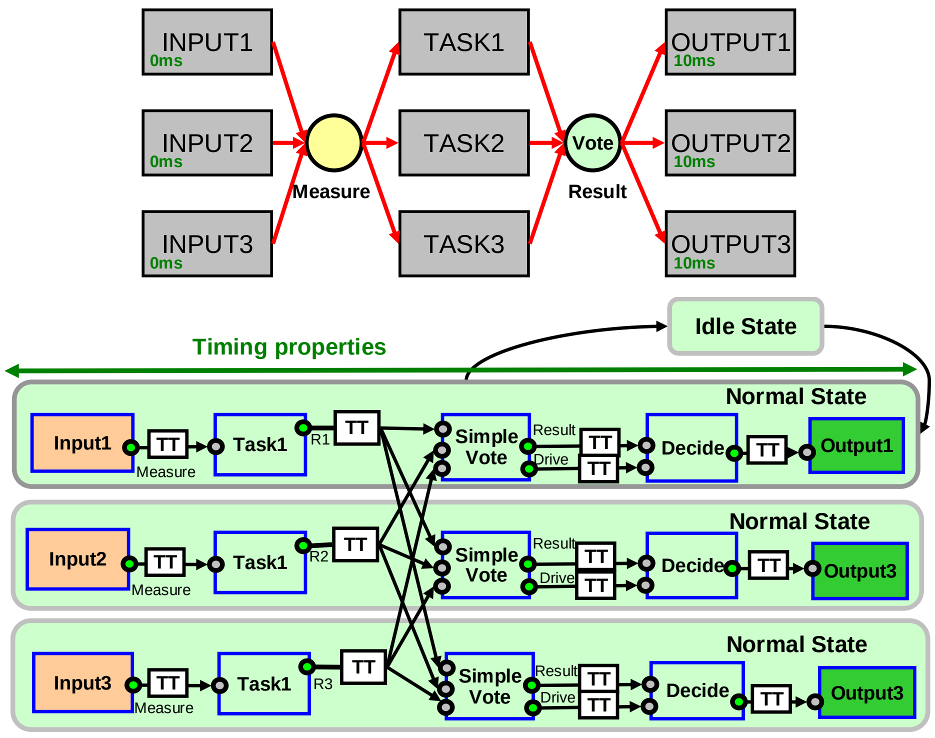


Figure 6.13: An example for concretizing FTOS models to functional models with timing annotations (some details omitted). TT stands for token transfer, which will later be refined during function-architecture mappings.

3. The *fault model* is used to annotate non-deterministic edges on the original model. As we try to eliminate the extensive use of clock variables, the sporadic behavior for the occurrence of faults requires appropriate abstractions to be tailored into the functional model.

Limitation: Currently, Gecko only generates partial stubs of the whole model. Specifications, as well as user defined functions must be annotated manually to the UPPAAL model, such that meaningful analysis results can be derived.

6.10.3.5 Deriving Integrity Constraints

With the above framework, we plan to investigate new techniques and algorithms and construct them on top of Gecko. In this section, we give an example how the tool can benefit from integration of these techniques.

In the fault-tolerance community, *integrity constraints* are conditions that hold in normal operation, but may fail to hold in the event of a fault [Hay09]. As sensors may have imprecisions, deriving integrity constraints to distinguish between effects caused by imprecisions or errors is crucial to prevent false alarms. Here a simplified scenario in fig. 6.14 is given to motivate the discussion and indicate our current progress.

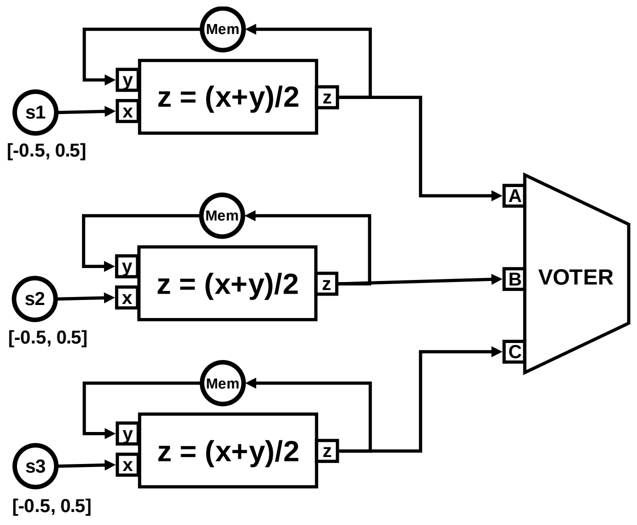
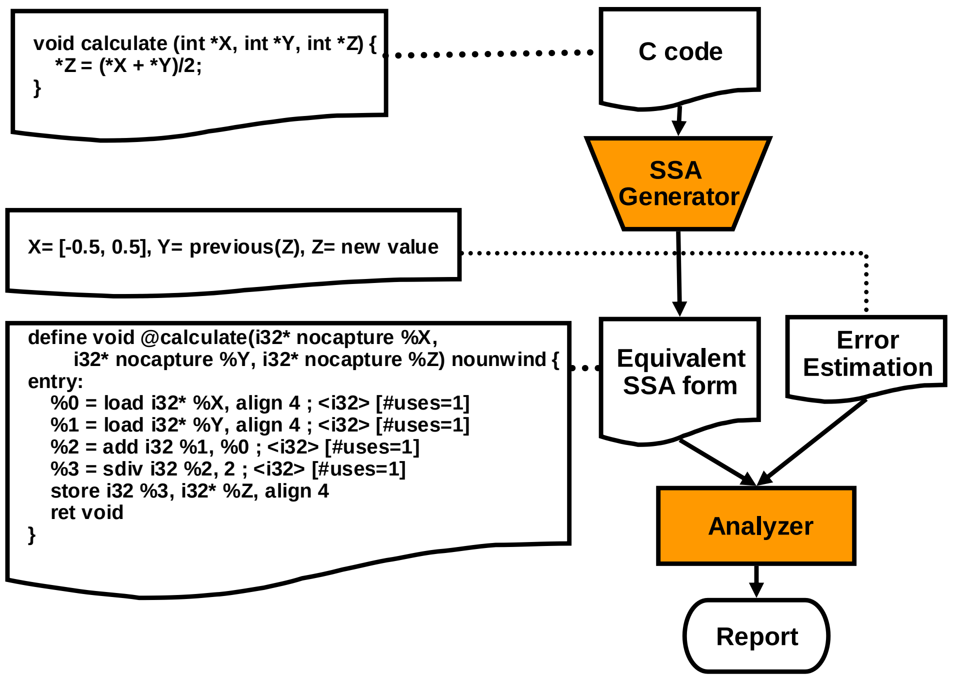


Figure 6.14: An example where integrity constraints are considered over the voter component.

In fig. 6.14, three identical sensor units `s1`, `s2`, and `s3` with imprecision ranging between $[-0.5, 0.5]$ perform readings from the environment, and pass their readings to the corresponding functional unit $z = (x + y)/2$. Results from each functional unit are transformed to the voter to detect erroneous results, and a copy of the previous result directly from port z is stored in the memory for next stage processing (similar to the integral part of the PID control). It can be observed that due to accumulative sensor imprecision, values on ports A (similarly B and C) can deviate from the actual value with a maximum ranging over $[(\frac{(-0.5+0)}{2} + \frac{(-0.5+\frac{(-0.5+0)}{2})}{2} + \ldots), (\frac{(0.5+0)}{2} + \frac{(0.5+\frac{(0.5+0)}{2})}{2} + \ldots)] = [-\Sigma_{i=2}^{\infty}\frac{1}{2^i}, \Sigma_{i=2}^{\infty}\frac{1}{2^i}] = [-0.5, 0.5]$. Therefore, when $|\alpha - \beta| > 1$ occurs, where $\alpha, \beta \in \{A, B, C\}$, we are certain that either α or β contains a erroneous sensor reading which leads to the result (or faulty in our definition)[16]; the integrity constraint can be designed based on the above information. The absence of such a criteria (no bounded interval available), indicates design errors. Following this example, in Gecko the process flow of generating integrity constraints (algorithm omitted) is sketched in fig. 6.15:

1. *(Preprocessing)* By adapting techniques in compiler technologies, the equivalent static single assignment (SSA) form using LLVM [LLV] can be generated, which enables efficient manipulations in the next stage.
2. *(Analysis)* The analyzer module gives each register variable (in fig. 6.15, `%0`, `%1`, `%2`, `%3`, `%X`, `%Y`, `%Z`) two values $(value, imprecision)$, and update these values based on the following two types:
    - *Atomic machine instructions* (in fig. 6.15, `load`, `store`, `add`, `sdiv`). Consider

[16]Here a single fault-occurrence assumption is applied, meaning that at most one sensor can generate erroneous readings at any instance.



```
void calculate (int *X, int *Y, int *Z) {
   *Z = (*X + *Y)/2;
}
```

```
define void @calculate(i32* nocapture %X,
      i32* nocapture %Y, i32* nocapture %Z) nounwind {
entry:
   %0 = load i32* %X, align 4 ; <i32> [#uses=1]
   %1 = load i32* %Y, align 4 ; <i32> [#uses=1]
   %2 = add i32 %1, %0 ; <i32> [#uses=1]
   %3 = sdiv i32 %2, 2 ; <i32> [#uses=1]
   store i32 %3, i32* %Z, align 4
   ret void
}
```

Figure 6.15: Process flow for the generation of integrity constraints.

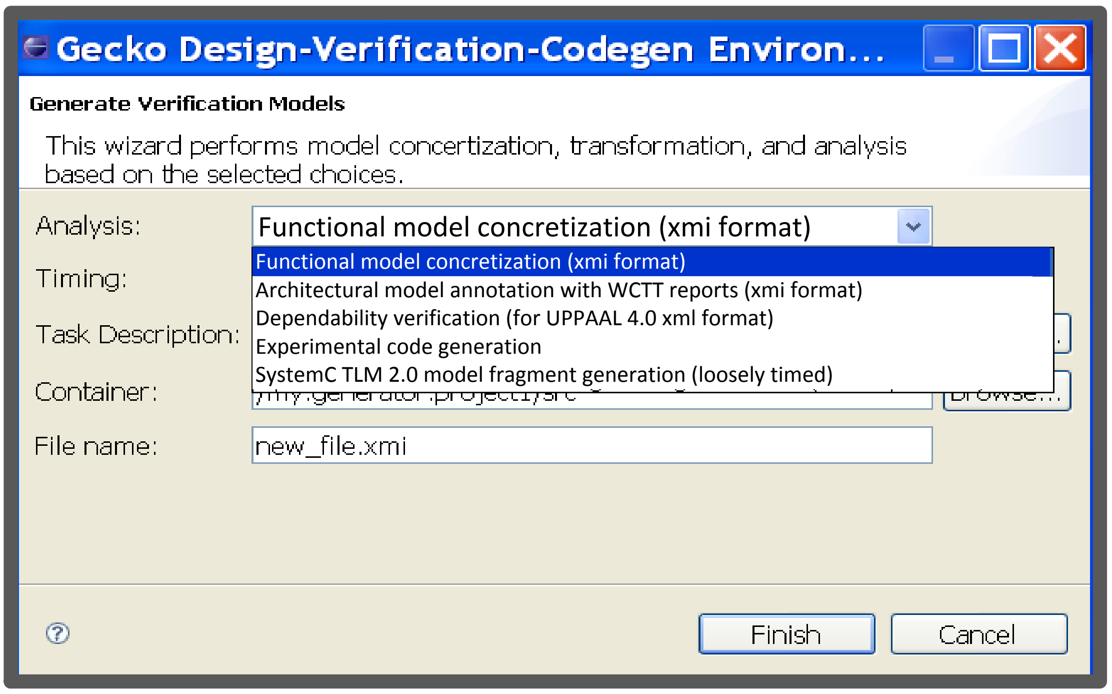
in fig. 6.15 with the instruction `%2 = add i32 %1, %0 ;`. Given `%0` with imprecision $[-0.5, 0.5]$, `%1` with imprecision $[-0.25, 0.25]$, `%2` would have imprecision ranging over $[-0.75, 0.75]$.

- *Global iterations* (e.g., a sensor reads one new value from the environment, load memory, which is recorded separately in the information "Error Estimation" in fig. 6.15). Note that as fixed points may never be reached, the number of iterations can be set explicitly.

Limitation: The integrity constraint generator module currently operates over functions under *affine transformations*; for conditional operations, a decision by taking both edges is approximated. The analysis works with sensor imprecision modeled using both intervals and Gaussian distribution, which is also closed under linear transformations.

6.10.3.6 Evaluation and Extension

The above functionalities of Gecko is implemented under the Eclipse platform, and Figure 6.16 shows some screenshots when designing with Gecko. Stepwise refinement operations can be performed to generate models or executable code. A complete demonstration using the tool over Luminary Micro LM3S8962 evaluation boards is also available, where a model containing distributed voting using the CAN bus is designed and deployed. To support detailed analysis, Gecko also enables designers to generate SystemC modules from Gecko models; currently only loosely-timed modeling methodology (using `b_transport()` primitives) is available.



Gecko Design-Verification-Codegen Environ...

Generate Verification Models

This wizard performs model concertization, transformation, and analysis based on the selected choices.

Analysis: Functional model concretization (xmi format)

- Functional model concretization (xmi format)
- Architectural model annotation with WCTT reports (xmi format)
- Dependability verification (for UPPAAL 4.0 xml format)
- Experimental code generation
- SystemC TLM 2.0 model fragment generation (loosely timed)

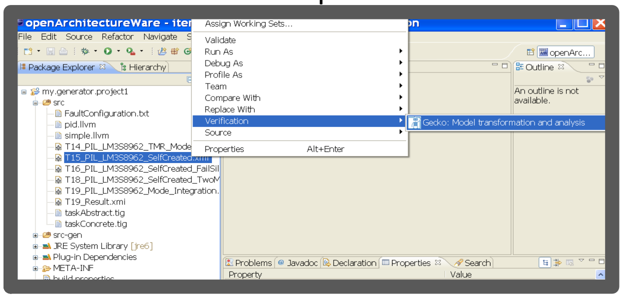
Timing:

Task Description:

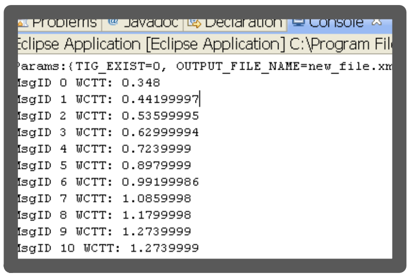
Container: /my.generator.project1/src

File name: new_file.xmi

Option selection windows in Gecko



Gecko pop-up options in Eclipse IDE



```
Eclipse Application [Eclipse Application] C:\Program Fil
Params:(TIG_EXIST=0, OUTPUT_FILE_NAME=new_file.xm
MsgID 0 WCTT: 0.348
MsgID 1 WCTT: 0.44199997
MsgID 2 WCTT: 0.53599995
MsgID 3 WCTT: 0.62999994
MsgID 4 WCTT: 0.7239999
MsgID 5 WCTT: 0.8979999
MsgID 6 WCTT: 0.99199986
MsgID 7 WCTT: 1.0859998
MsgID 8 WCTT: 1.1799998
MsgID 9 WCTT: 1.2739999
MsgID 10 WCTT: 1.2739999
```

Console for CAN bus WCET

Figure 6.16: Screenshots when designing systems using Gecko.

To provoke free extension and usage, the package will be released as an Eclipse add-on under GPL 2.0. For extensions, introducing new mechanisms requires users to design a parameterized skeleton using EMF, which can be easily adapted by referencing existing mechanisms; introducing new platforms requires users to implement links from abstract instructions (e.g., `Send()`, `Wait()`) to concretized mechanisms.

6.10.3.7 Brief Overview on Related Work

In general, different tools focus on different aspects in the overall design flow. For instance, Metropolis [BWH$^+$03] focuses on the joint modeling of applications and architectures, and Ptolemy focuses on the unified semantics for the execution of heterogeneous models of computation. Our focus, compared to others, is different: we expect to offer simplified (light-weight) means to achieve dependability and fault-tolerance under a unified design platform or to experiment novel techniques.

6.10.3.8 Concluding Remarks

Our contributions are summarized as follows:

- We presented a framework for embedded systems to equip with fault-tolerance with reduced design efforts. We bind actor-oriented methodologies and fault-tolerant patterns [Han07] using tree-structures, and perform local-scale design space exploration (for communication), and tree translation (for actors) in the refinement process.
- Analysis techniques are introduced (dependability model generation, simple timing analysis), or experimented (integrity constraint analysis) in Gecko.
- The tool enables interfacing and integration with other tooling by sharing a common platform (Eclipse IDE) and by translating Gecko models into SystemC models.

The introduction of model-driven development (MDD) is to facilitate the design process by providing an abstraction of the system behavior, where complexities of the system construction can be reduced. The Gecko tool, which is based on MDD and focuses on fault-tolerance and dependability aspects, will continue to be improved and extended.

6.10.4 D. Brief Instructions on Executing Examples for Synthesis in GECKO

Here we illustrate how FT synthesis is done in our prototype tool-chain using the example in sec. 6.7.1: first we perform model transformation and generate a new model which equips FT mechanisms. Then executable code can be generated based on performing code-generation over the specified model (optional). Once when the GECKO Eclipse add-on is installed (see our website for instructions), proceed with the following steps:

- The model (`F01_FT_Synthesis_Correct.xmi`) for sec. 6.7.1 contains the fault model, the hardware used in the system, and pre-inserted FT mechanism blocks, but their timing information is unknown.
- Right click on the selected model under synthesis, choose "`Verification`" `->` "`Gecko: Model Transformation and Analysis`". A pop-up window similar to fig. 6.10a is available.
- In the `General` tab, choose `Symbolic FT synthesis using algorithmic game theory`.
- In the `Platform Analysis` tab, set up the default actor WCET and network WCMTT to be 1 and 3.
- In the `Output` tab, select the newly generated output file.
- Press "`Finish`". Results of intermediate steps are shown in the console, including

FT mechanisms as interleaving models (fig. 6.10b), constraints derived from LTM (fig. 6.10c), and results of timing (fig. 6.10d) after executing the constraint solver.

  – In fig. 6.10b, the mechanism dumped from the engine specifies the action "`if`($req_v \neq \perp$) `MsgSend`(rsp)": note that this action implicitly implies that when $req_v = \perp$, a null-op which only updates the program counter should be executed; this is captured by our synthesis framework.
  – In fig. 6.10d, the total execution time is roughly 18s because the engine dumps the result back to mechanisms in textual form, which consumes huge amount of time: executing the game and performing constraint solving take only a small portion of the total time.

- When the model is generated, users can again right click on the newly generated model, and select `Code Generation` in the tab `General`: the code generator then combines the model description and software templates for dedicated hardware and OS to create executable C code.

CHAPTER 7

Resource-Bounded Strategies for Distributed Games

Abstract

Distributed games capture the interaction between a distributed system and its environment under operation. Applications, to name a few, include synthesis for fault-tolerance and distributed scheduling. As solving these games is undecidable even for simplest winning conditions, resource-bounded methods, i.e., strategies with (and computed from) limited resources, are fundamental gadgets to solve practical problems. In this chapter, we investigate such methods to synthesize controller strategies for distributed games, mainly on safety conditions. We first discuss the method based on projections (MP_c). This method can be further generalized using risk state partitions (MPP_c), where each local game is responsible of not entering certain risk states (in the distributed game), implying an implicit cooperation among local games. However, creating such partition for safety winning is NP-complete, therefore heuristics are required to guide the partition process. Lastly, we create local observations and combine with the lattice theory of antichains to give a method which generates the global observation strategy automaton for the distributed game; this automaton is further decomposed to several automata to generate local controllers.

Contents

7.1 Introduction

There is an ever-growing demand for scalable computing systems that supports goal-oriented composition. These computing assemblies are usually built from interacting components, where each component provides a specific, well-defined capability or service. It is a challenging task to orchestrate components in order to realize novel, added-value capabilities. In this chapter we focus on *distributed synthesis* [PR90], which is one approach that addresses this problem. The distributed synthesis problem [PR90] asks, given (i) a distributed architecture consisting of a set of processors communicating with each other and the environment over a network of synchronous input/output channels and (ii) a specification of the desired behavior, if there exists a distributed program that realize the specification. (A distributed program is a set of programs or implementations, one implementation for every component.)

The distributed synthesis problem is undecidable for most classes of architectures [PR90] and it stays undecidable when restricted to safety, reachability, or Büchi winning conditions [Jan07]. Only for simple architectures such as pipelines and one-way rings, the problem is decidable. For many applications, these architectures are typically too restrictive. Furthermore, the associated synthesis algorithms have often a very high complexity, e.g., non-elementary complexities [PR90].

Nevertheless, engineers perform distributed synthesis despite of this fact. They construct a distributed system from components, examine if it satisfies certain properties using verification or testing, and perform suitable modification until it works properly. Therefore in this chapter we take a practitioner's perspective. Our goal is to develop and present a set of efficient heuristics for distributed synthesis on arbitrary architectures. These methods reflect the way how users design systems, and should be inexpensive heuristics that are applicable in many interesting cases. The goal is, if these methods satisfy the design intention, distributed synthesis can be automated to a level of assistance - what is considered as a potential fix by a designer can be found automatically using our methods.

We use *distributed games* [MW03, GLZ04], an alternative formalism for distributed synthesis, to present our methods. In a distributed game, there are n players (e.g., n processes) playing a game against a single hostile environment (e.g., a scheduler). There is no explicit way for processes to interact, every such interaction has to go through the environment. Each player has only a local view of the global state of the system. A distributed strategy (program) is a collection of local strategies; one for each of the players. Distributed games have the same undecidability properties as the traditional distributed synthesis formalism but they provide (i) an easy way to integrate pre-defined behaviors of the components and (ii) an intuitive translation from concrete applications to the mathematical model for distributed synthesis. In Section 7.2.3, we give an example, in which a distributed system with priorities [BBS06] together with a specification is translated into a distributed game.

Our methods (see Table 7.1 for an overview and the underlying design intentions) syn-

Table 7.1: Heuristics and their corresponding design intentions

Index	Resource-Bounded Algorithm	Corresponding Design Intention
1	Projection	Minimal fix by modifying a single component
2	Projection + Risk Partition	Simple cooperation without mutual understanding among components
3	Local observation + Antichain	Enhanced cooperation based on the knowledge of global states inferred from local processes

thesize distributed strategies with bounded resources, e.g., positional strategies, which do not depend on the history of an execution. The synthesis of resource-bounded strategies is particularly important in the domain of distributed embedded systems, where only a limited amount of memory is available. We present our methods only for safety games but extensions to reachability or Büchi games are straightforward.

The first heuristic, presented in Section 7.3.1, reflects the intention of minimum repair: we may interpret the least modification over the system as modifying a single component. It uses projections to obtain local abstractions of the distributed game, and synthesizes local strategies with a finite history. The method runs in polynomial time but it is overly conservative, in the sense that it does not account for possible cooperations between local processes. An immediate extension, presented in Section 7.3.3, assigns each unsafe state to a process responsible for avoiding this state. This way of partitioning the set of unsafe states corresponds to a simple (context-unaware) form of cooperation between the processes. We show that the problem of finding a suitable partitioning is NP-complete. Finally, for enhanced cooperation, the method presented in Section 7.4 collects local observations derived from the distributed game graph. This algorithm is based on concepts of games with imperfect information; it relies heavily on antichains [DWDR06] and runs in EXPTIME. The complexity results are summarized in Table 7.2. Finally, in Section 7.5, we compare our heuristics with previous work on the distributed synthesis problem and provide an outlook in Section 7.6.

7.2 Preliminaries

We are describing distributed games as defined by Mohalik and Walukiewicz [MW03]. In this model there are no explicit means of interaction among processes as such interaction must take place through the environment. Moreover, each player has only a local view of the global system state, whereas the (hostile) environment has access to the global history. Distributed games are rich enough to model various variations of distributed synthesis problems proposed in the literature [MW03]. Partial contents for this section can also be found in Chapter 2.

7.2.1 (Local) Games

A *game graph* or *arena* is a directed graph $G = (V_s \uplus V_e, E)$ whose nodes are partitioned into two classes V_s and V_e. We only consider the case of two participants in the following and call them **player** (system) and **environment** for simplicity. A *play* starting from node v_0 is simply a maximal path $\pi = v_0 v_1 \dots$ in G where we assume that player determines the *move* $(v_k, v_{k+1}) \in E$ if $v_k \in V_s$; environment determines the move when $v_k \in V_e$. With $\mathrm{Occ}(\pi)$ ($\mathrm{Inf}(\pi)$) we denote the set of nodes visited (visited infinitely often) by a play π. A *winning condition* defines when a given play π is *won* by player; if π is not won by the player, it is won by environment. A node v is won by player if player can always choose his moves in such a way that he wins any resulting play starting from v, similarly for environment. We also use E_s (E_e) to represent the set of player (environment) edges in E.

7.2.2 Distributed games

Definition 29. *For all $i \in \{1, \dots, n\}$, let $G = (V_{si} \uplus V_{ei}, E_i)$ be a game graph with the restriction that it is bipartite. A* distributed game *$\mathcal{G}$ is of the form $(\mathcal{V}_s \uplus \mathcal{V}_e, \mathcal{E}, Acc)$*

- *$\mathcal{V}_e = V_{e1} \times \dots \times V_{en}$ is the set of environment vertices.*
- *$\mathcal{V}_s = (V_{s1} \uplus V_{e1}) \times \dots \times (V_{sn} \uplus V_{en}) \setminus \mathcal{V}_e$ is the set of player vertices.*
- *Let $(x_1, \dots, x_n), (x'_1, \dots, x'_n) \in \mathcal{V}_s \uplus \mathcal{V}_e$, then $\mathcal{E}$ satisfies:*
  - *If $(x_1, \dots, x_n) \in \mathcal{V}_s$, $((x_1, \dots, x_n), (x'_1, \dots, x'_n)) \in \mathcal{E}$ if and only if $\forall i.(x_i \in V_{si} \rightarrow (x_i, x'_i) \in E_i) \wedge \forall j.\ (x_j \in V_{1_j} \rightarrow x_j = x'_j)$.*
  - *For $(x_1, \dots, x_n) \in \mathcal{V}_e$, if $((x_1, \dots, x_n),$ $(x'_1, \dots, x'_n)) \in \mathcal{E}$, then for every x_i, either $x_i = x'_i$ or $x'_i \in V_{si}$, and moreover $(x_1, \dots, x_n) \neq (x'_1, \dots, x'_n)$*[1]
- *$Acc \subseteq (\mathcal{V}_s \uplus \mathcal{V}_e)^\omega$ is the winning condition*[2].

Notice that there is an asymmetry in the definition of environment's and player's moves. In a move from player's to environment's position, all components which are player's position must change. In the move from environment's to player's, all components are environment's position but only some of them need to change.

In a distributed game $\mathcal{G} = (\mathcal{V}_s \uplus \mathcal{V}_e, \mathcal{E}, Acc)$, a play is defined analogously as defined in local games: a *play* starting from node v_0 is a maximal path $\pi = v_0 v_1 \dots$ in $\mathcal{G}$ where player determines the *move* $(v_k, v_{k+1}) \in \mathcal{E}$ if $v_k \in \mathcal{V}_s$; the environment decides when $v_k \in \mathcal{V}_e$. For a vertex $x = (x_1, \dots, x_n)$, we use the function $proj(x, i)$ to retrieve the i-th component x_i, and use $proj(X, i)$ to retrieve the i-th component for a set of vertices X. For simplicity, denote $\pi_{\leq j}$ as $v_0 v_1 \dots v_j$ and use $proj(\pi_{\leq j}, i)$ for $proj(v_0, i) \dots proj(v_j, i)$, i.e., a sequence from $\pi_{\leq j}$ by projecting over i^{th} element.

[1]Another definition is to also add that all local moves of the environment should be explicitly listed in the local game; this is not required, as mentioned in the paper [MW03]. Thus for the ease of reading, we only construct environment moves in the distributed game and avoid drawing complicated environment edges in the local game.

[2]In this thesis we also use $\mathcal{G}$ as the identifier for the distributed game graph $(\mathcal{V}_s \uplus \mathcal{V}_e, \mathcal{E})$.

A *distributed strategy* of a distributed game for player is a tuple of functions $\xi = \langle f_1, \ldots, f_n \rangle$, where each function $f_i : (V_{si} \uplus V_{ei})^* \times V_{si} \to (V_{si} \uplus V_{ei})$ is a local strategy for G_i based on its observable history of local game i and current position of local game i. A distributed strategy is *positional* if
$f_i : V_{si} \to V_{si} \uplus V_{ei}$, i.e., the update of location depends only on the current position of local game. Contrarily, for environment a strategy is a function $f : (\mathcal{V}_s \uplus \mathcal{V}_e)^+ \to (\mathcal{V}_s \uplus \mathcal{V}_e)$ that assigns each play prefix $v_0 \ldots v_k$ a vertex v_{k+1} where $(v_k, v_{k+1}) \in \mathcal{E}$. The formulation of distributed games models the asymmetry between the environment (full observability) and the a set of local controllers (partial observability).

Definition 30. *A distributed game $\mathcal{G} = (\mathcal{V}_s \uplus \mathcal{V}_e, \mathcal{E}, Acc)$ is for player winning by a distributed strategy $\xi = \langle f_1, \ldots, f_n \rangle$ over initial states $\mathcal{V}_{ini} \in \mathcal{V}_e$, if for every play $\pi = v_0 v_1 v_2, \ldots$ where $v_0 \in \mathcal{V}_{ini}$, player wins π following his own strategy (regardless of strategies of environment), i.e.,*

- $\pi \in Acc.$
- $\forall i \in \mathbb{N}_0. \ (v_i \in \mathcal{V}_s \to (\forall j \in \{1, \ldots, n\}.(proj(v_i, j) \in V_{sj} \to proj(v_{i+1}, j) = f_j(proj(\pi_{\leq i}, j)))$.

Definition 31. *Given distributed game graph $(\mathcal{V}_s \uplus \mathcal{V}_e, \mathcal{E})$,*

- *the reachability winning condition is defined by $Acc = \{v_0 v_1 \ldots \in (\mathcal{V}_s \uplus \mathcal{V}_e)^\omega \mid Occ(v_0 v_1 \ldots) \cap \mathcal{V}_{goal} \neq \emptyset\}$, where $\mathcal{V}_{goal}$ is the set of goal states in $\mathcal{V}_s \uplus \mathcal{V}_e$.*
- *the safety (co-reachability) winning condition is defined by $Acc = \{v_0 v_1 \ldots \in (\mathcal{V}_s \uplus \mathcal{V}_e)^\omega \mid Occ(v_0 v_1 \ldots) \cap \mathcal{V}_{risk} = \emptyset\}$, where $\mathcal{V}_{risk}$ is the set of risk states in $\mathcal{V}_s \uplus \mathcal{V}_e$.*
- *the Büchi winning condition is defined by $Acc = \{v_0 v_1 \ldots \in (\mathcal{V}_s \uplus \mathcal{V}_e)^\omega \mid Inf(v_0 v_1 \ldots) \cap \mathcal{V}_{goal} \neq \emptyset\}$, where $\mathcal{V}_{goal}$ is the set of goal states in $\mathcal{V}_s \uplus \mathcal{V}_e$.*

7.2.3 Example: distributed scheduling [BBPS09]

For the scenario of distributed scheduling in BIP systems [BBPS09, BBS06], distributed games offer an intuitive translation for visualizing and solving such problems. Consider the distributed system with priorities [BBS06] shown in Figure 7.1. It consists of two processes, $\mathcal{P}_1$ and $\mathcal{P}_2$ and six actions (a, b, c, d, e, f). An action is executed by following the transitions with the matching label. If the label appears in several processes, then this action synchronized the processes. E.g., $\mathcal{P}_1$ and $\mathcal{P}_2$ are synchronized on Action a, i.e., they take the transitions labeled a simultaneously. At any step, a schedule decides which of available action to execute next. Priorities restrict the available actions. In our example, Action d has a lower priority than Action b, i.e., whenever d and b are enabled (e.g., in global state (v_2, v_3)), then Action d must not be taken. The same holds for Action a and e. Given a specification, e.g., avoid global state (v_2, v_4), we can construct a distributed game (Figure 7.1c; circles are player vertices and squares are environment vertices) that searches for local modification of the processes such that the specification is satisfied independent of the choices of the scheduler. Continuing the example of Figure 7.1c, starting with (v_1, v_3), when moving to (a, d), the environment can move to either (v_2, v_3) or (v_1, v_4), representing the granting of action a or d.

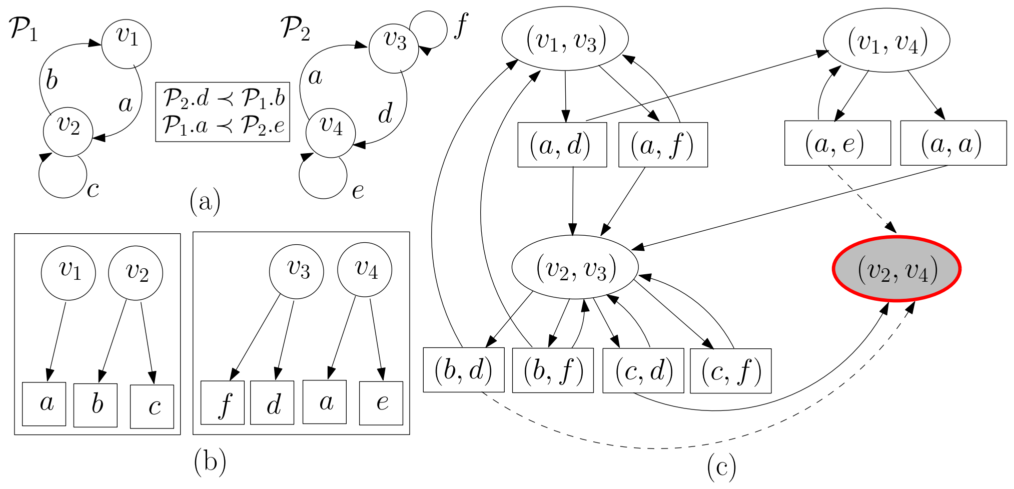


Figure 7.1: Two processes having a global priority $\mathcal{P}_2.d \prec \mathcal{P}_1.b$ and $\mathcal{P}_2.e \prec \mathcal{P}_1.a$ over actions (a), the generated local game (b), and the distributed game modeling the interaction and priority (the dashed line means an non-existing transition due to priority).

7.2.4 Solving Games of Imperfect Information using Antichains

Here we reuse the formulation in [DWDR06] to form the lattice of antichains of set of states, where a state corresponds to the location of a distributed game [DWDR06].

Let S be set of states. Let $q, q' \in 2^{2^S}$, and define $q \sqsubseteq q'$ iff $\forall s \in q : \exists s' \in q' : s \subseteq s'$. A set $s \subseteq S$ is *dominated* in q iff $\exists s' \in q : s \subset s'$, and define the set of dominated elements of q as $Dom(q)$. Lastly, donote $\lceil q \rceil$ to be $q \setminus Dom(q)$. $\langle L, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top \rangle$ forms a complete lattice [DWDR06], where

- L as the set $\{\lceil q \rceil \mid q \in 2^{2^S}\}$.
- For $Q \subseteq L$, $\bigsqcap Q = \lceil \{\bigcap_{q \in Q} s_q \mid s_q \in q\} \rceil$ is the greatest lower bound for Q.
- For $Q \subseteq L$, $\bigsqcup Q = \lceil \{s \mid \exists q \in Q : s \in q\} \rceil$ is the least upper bound for Q.
- $\bot = \emptyset$, $\top = \{S\}$.

A *game of imperfect information* [Rei84] works on a local game graph $G = (V_s \uplus V_e, E)$, where player is unaware of his position with absolute precision. This is defined by an observation set (Obs, γ) where $\gamma : \mathsf{Obs} \rightarrow 2^{V_s}$ such that $\forall v \in V_0 \cdot \exists \mathsf{obs} \in \mathsf{Obs} : v \in \gamma(\mathsf{obs})$ (Obs is a finite set of identifiers). During a play, when reaching a player vertex $v \in V_0$, an arbitrary observation accompanied with v will be assigned to player; he is only aware of the observation but not the location. As the location is not known with precision, the successors are imprecise as well. Thus edges for player are labeled with elements in a set Σ. A strategy for player is *observation-based* means that the it should based on the observation to make the decision, which is an element in Σ.

Without mentioning further details, we summarize how to use the operator CPre [DWDR06] over an antichain of set of states q to derive the set of Controllable Predecessors; by iterating backwards from the set of all locations until satura-

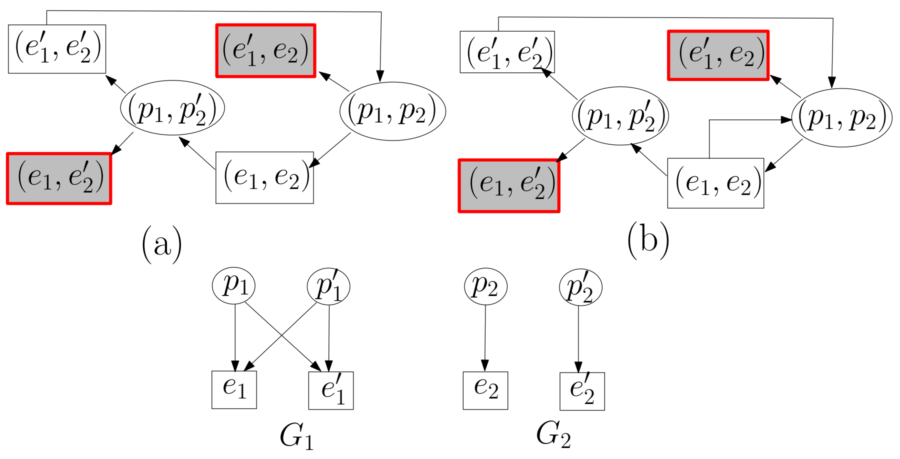


Figure 7.2: Two distributed games with safety-winning conditions.

tion, an observation-based strategy can be established. Define $Enabled(\sigma)$ to be the set of locations where an edge labeled σ is possible as an outgoing edge, and $Post_\sigma(S)$ (similarly $Post_e(S)$) be the set of locations by taking the edges labeled by σ (with transition e) from a set of locations S. Assume that the game graph is bipartite. Then, $\mathsf{CPre}(q) := \lceil\{s \subseteq V_s \mid \exists\sigma \in \Sigma \cdot \forall \mathsf{obs} \in \mathsf{Obs} \cdot \exists s' \in q : s \subseteq Enabled(\sigma) \wedge \bigcup_{e\in E_1} Post_e(Post_\sigma(s)) \cap \gamma(\mathsf{obs}) \subseteq s')\}\rceil$. Intuitively, CPre computes a set of player locations, such that by playing a common move σ, for all successor locations after the environment moves, player can decide the set of locations it belongs using the observation.

7.3 Distributed Safety Strategy based on Projections

7.3.1 Simple Projection

The first presented approach for safety games is based on projection. We call it **method using projections**, MP for short. The idea is to look at each subgame in isolation: consider a risk vertex $(v_1, \ldots, v_n)$ in the distributed game $\mathcal{G}$. Obviously, a strategy of local game G_i avoids v_i guarantees that the global state $(v_1, \ldots, v_n)$ is also avoided.

The method proceeds as follows: for each i, we abstract $\mathcal{G}$ to a game $\bar{\mathcal{G}}_i$ that refers only to the values of the i^{th} component, i.e., the other components are projected away. Every abstract state in $\bar{\mathcal{G}}_i$ that includes a risk state of $\mathcal{G}$ is marked as an error state. Then, we compute the environment attractor of the error states in $\bar{\mathcal{G}}_i$ to see if a winning local strategy exists.

We illustrate this idea using the distributed game in Figure 7.2a (an example from [MW03]). The resulting abstract games are in Figure 7.3a. Starting with node (e_1, e_2), the goal is to avoid $\{(e'_1, e_2), (e_1, e'_2)\}$.

- When abstracting component 2, $\bar{\mathcal{G}}_1$ is created. Any positional move of the local game G_1 leads to risk in $\bar{\mathcal{G}}_1$.

- When abstracting component 1, $\bar{\mathcal{G}}_2$ is created. Any positional move of the local game G_2 leads to risk in $\bar{\mathcal{G}}_2$.

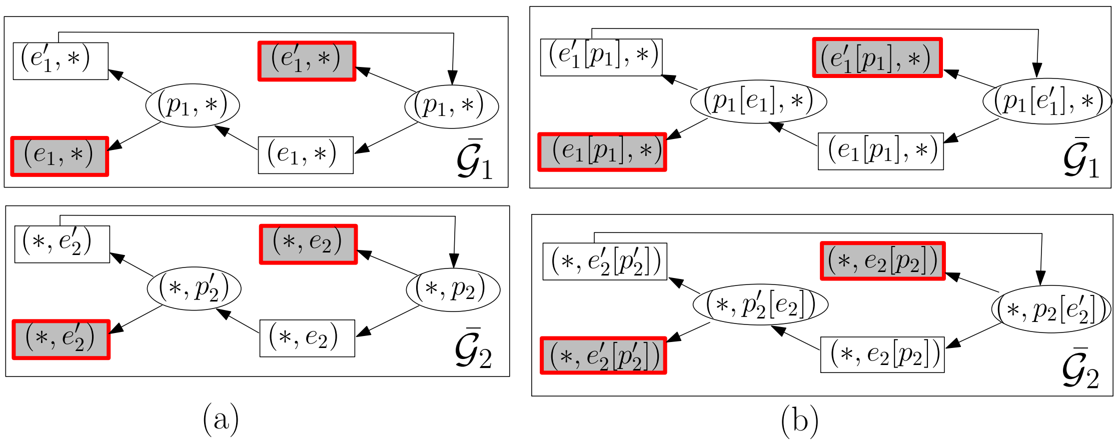


Figure 7.3: (a) Abstract games $\bar{\mathcal{G}}_1$ and $\bar{\mathcal{G}}_2$ (MP_0), and (b) fragment of the abstract games when one step memory is pushed (MP_1). Contents enclosed in square brackets are the history. The "$*$" symbol is used for the ease of explanation, meaning that the content is abstracted.

Therefore, this method cannot find a distributed winning strategy for this safety game, even though there exists a winning strategy: In $\mathcal{G}_1$, if its previous state is e_1, go to e_1', and vice versa; in $\mathcal{G}_2$ follow the only move. This strategy makes a local decision based on the finite observable history and the current position of a local game. We call a strategy *c-forgetful*, if a decision of a local game is made based on its history up to depth c.

Definition 32. *A distributed strategy $\xi = \langle f_1, \ldots, f_n \rangle$ is c-forgetful, if for all i, $f_i : (V_{si} \uplus V_{ei} \cup \{\Box\})^c \times V_{si} \rightarrow (V_{si} \uplus V_{ei})$ is a local strategy based on (a) its observable history of local game i up to depth c and (b) current position of local game i. The symbol $\Box$ represents an empty token in the history.*

We now formally define the concept of abstract games and methods generating such games to find c-forgetful distributed strategies. For ease of notation, we use $v \twoheadrightarrow u$ to indicate that there exists an edge from v to u between two vertices in the distributed game graph.

Definition 33. *Let $\mathcal{G} = (\mathcal{V}_s \uplus \mathcal{V}_e, \mathcal{E}, Acc)$ be a safety game defined using the set of risk states $\mathcal{V}_{risk}$, and $c \in \mathbb{N}_0$. Define the tuple of abstract games $(\bar{\mathcal{G}}_1, \ldots, \bar{\mathcal{G}}_n)$ of depth c, where $\bar{\mathcal{G}}_i = (\bar{\mathcal{V}}_{si} \uplus \bar{\mathcal{V}}_{ei}, \bar{\mathcal{E}}_i, Acc_i)$.*

- *$v = (v_1, \ldots, v_n) \in \mathcal{V}_s \uplus \mathcal{V}_e \Leftrightarrow v_i[v_{h_1} \ldots v_{h_c}]_v \in \bar{\mathcal{V}}_{si} \uplus \bar{\mathcal{V}}_{ei}$, where $v_{h_1}, \ldots, v_{h_c} \in V_{si} \uplus V_{ei} \cup \{\Box\}$, and one of the following holds.*
  - *In $\mathcal{G}$, there exist $v_{\alpha_1}, \ldots, v_{\alpha_c} \in \mathcal{V}_s \uplus \mathcal{V}_e$ such that $v_{\alpha_c} \twoheadrightarrow \ldots \twoheadrightarrow v_{\alpha_1} \twoheadrightarrow v$ and $\forall j \in \{1, \ldots, c\} : proj(v_{\alpha_j}, i) = v_{h_j}$.*
  - *In $\mathcal{G}$, there exist $k \leq c$, $v_{\alpha_1}, \ldots, v_{\alpha_k} \in \mathcal{V}_s \uplus \mathcal{V}_e$ such that v_{α_k} is an initial state, $v_{\alpha_k} \twoheadrightarrow \ldots \twoheadrightarrow v_{\alpha_1} \twoheadrightarrow v$ and $\forall j \in \{1, \ldots, k\} : proj(v_{\alpha_j}, i) = v_{h_j}$. For $k > c$, $v_{h_k} = \Box$.*

- $v_i[v_{h_1} \dots v_{h_c}]_v \in \bar{\mathcal{V}}_{si} \Leftrightarrow v_i \in V_{si}$.
- *For* $v = (v_1, \dots, v_n), u = (u_1, \dots, u_n) \in \mathcal{V}_s \uplus \mathcal{V}_e$, $(v, u) \in \mathcal{E} \Leftrightarrow (v_i[v_{h_1} \dots v_{h_c}]_v, u_i[v_i \ u_{h_1} \dots u_{h_{c-1}}]_u) \in \bar{\mathcal{E}}_i$, *where in* v *(similarly for* u*) one of the following holds.*
    - *In* $\mathcal{G}$, *there exists* $v_{\alpha_1}, \dots, v_{\alpha_c} \in \mathcal{V}_s \uplus \mathcal{V}_e$ *such that* $v_{\alpha_c} \twoheadrightarrow \dots \twoheadrightarrow v_{\alpha_1} \twoheadrightarrow v$ *and* $\forall j \in \{1, \dots, c\} : proj(v_{\alpha_j}, i) = v_{h_j}$.
    - *In* $\mathcal{G}$, *there exists* $k \leq c$, $v_{\alpha_1}, \dots, v_{\alpha_k} \in \mathcal{V}_s \uplus \mathcal{V}_e$ *such that* v_{α_k} *is an initial state,* $v_{\alpha_k} \twoheadrightarrow \dots \twoheadrightarrow v_{\alpha_1} \twoheadrightarrow v$ *and* $\forall j \in \{1, \dots, k\} : proj(v_{\alpha_j}, i) = v_{h_j}$. *For* $k > c$, $v_{h_k} = \Box$.
- *The safety condition* Acc_i *is defined using the set of risk states* $\{v_i[v_{h_1} \dots v_{h_c}]_v \mid v \in \mathcal{V}_{risk}\} \subseteq \bar{\mathcal{V}}_{si} \uplus \bar{\mathcal{V}}_{ei}$.
- *For the initial state,* $(v_1, \dots, v_n)$ *is initial in* $\mathcal{G}$ *iff* $v_i[\Box \dots \Box]_v$ *is initial in* $\bar{\mathcal{G}}_i$.

The method of MP_c performs sequentially in three steps[3].

1. First, construct $(\bar{\mathcal{G}}_1, \dots, \bar{\mathcal{G}}_n)$ out of game $\mathcal{G}$ and depth c. The definition of $\bar{\mathcal{G}}_i$ implies a direct algorithm for construction: first create all vertices, and connect vertices based on checking the relation between two vertices.
2. Second, solve $\bar{\mathcal{G}}_i$ in isolation and generate the set of risk edges. For vertex $v_i[v_{h_1} \dots v_{h_c}]_X$, $v_i[v_{h_1} \dots v_{h_c}]_Y$, the risk edge should be taken as the union from X and Y. The strategy for each local game is then to exclude the derived risk edges.
3. If one of the abstraction game returns a safety strategy, then report that a safety strategy exists.

Lemma 4. *Given a distributed game* $\mathcal{G}$ *(with* n *local games) and* $c \in \mathbb{N}_0$, MP_c *can be computed in time polynomial to* $n|\mathcal{G}|^c$.

Proof. Given a distributed game $\mathcal{G}$ and $c \in \mathbb{N}_0$, the size of the generated abstract game graph, i,e., $\bar{\mathcal{G}}_1, \dots, \bar{\mathcal{G}}_n$, has the size bounded by $|\mathcal{G}|^c$. Therefore the construction is time polynomial to the size of the distributed game. Considering the second step, for each abstraction game, it can be solved by computing the attractor set (time linear to the number of edges of the distributed game graph). $\Box$

Returning to the example in Figure 7.1, by creating abstraction games (Figure 7.4) and solve two games in isolation. We can generate two different safety strategies, namely

- apply $\{(v_1, a), (v_2, b)\}$ on $\mathcal{P}_1$ or
- apply $\{(v_3, f), (v_4, a)\}$ on $\mathcal{P}_2$.

7.3.2 Fixed-Size Strategy versus Fixed-Length History Strategy

For methods using projections, instead of considering strategies with the fixed length history c as in MP_c, an alternative approach is to consider the family of strategies using constant memory size c. Our argument is as follows.

[3]For details we refer readers to the Appendix.

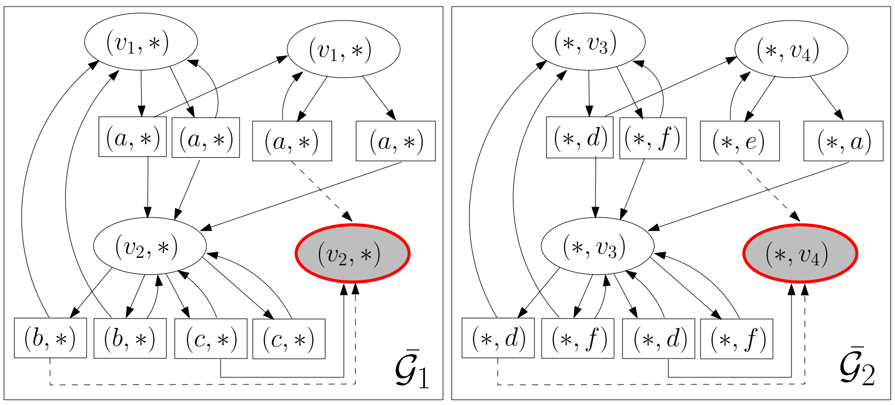


Figure 7.4: The resulting abstract game for the game in Figure 7.1.

- It is easier to implement MP_c: For using constant memory size, the criterion of memory update relies on the intelligence of designers. Contrarily, for MP_c, since the memory update in a transition can be described using double-implications between the preimage and the postimage, the method can be easily implemented symbolically using BDDs. We refer readers to the Appendix for details on the symbolic encoding.
- Our second argument is Lemma 5, specifying the incomparability between two approaches: there are some distributed games where using constant memory size for strategies is not enough.

Lemma 5. *There exists a class of distributed games where the size of the minimum memory used in the distributed strategy is proportional to the size of the game graph.*

Proof. We leave the proof in the Appendix. □

7.3.3 Projection + Risk Partition = Context-unaware Cooperation

The method described previously can find strategies for some problems, but it has limitations due to its *complete* projection of risk states: When projecting all risk states onto one abstraction game $\bar{\mathcal{G}}_i$, it is the sole responsibility of local game i to win the game, implying that no *cooperation* between local games is introduced. Based on this concept, we interpret the cooperation between local games, for safety conditions, as the *partition* of risk states.

Consider the distributed game in Figure 7.5. It can be observed that the MP algorithm with $c = 0$ ($c = 1$ as well) is unable to generate the strategy[4], meaning that a restriction over solely $\bar{\mathcal{G}}_1$ or solely $\bar{\mathcal{G}}_2$ is impossible. Now we partition the risk states based on the following criteria.

1. For G_1, it should ensure not entering (e_1, e'_2) in the distributed game.
2. For G_2, it should ensure not entering (e'_1, e_2) in the distributed game.

[4]In the abstract game $\bar{\mathcal{G}}_1$, both moves of p_1 lead to risk; in $\bar{\mathcal{G}}_2$, the only move of p'_2 leads to a risk state.

If G_1 has a strategy for 1 and G_2 has a strategy for 2, then the system is safe: in each safe location, for each successor state which is a risk state, there exists one local game to ensure of not entering. In fact, one distributed memoryless strategy exists for Figure 7.5:

- For $\mathcal{G}_1$, from p_1 move to e'_1.
- For G_2, from p_2 move to e''_2; from p'_2 move to e'_2.

Given (a) distributed game $\mathcal{G}$ with safety conditions defined by $\mathcal{V}_{risk}$, (b) a partition $(Risk_1, \ldots, Risk_n)$, where $Risk_i \subseteq \mathcal{V}_{risk}$ and $\biguplus_{i=1\ldots n} Risk_i = \mathcal{V}_{risk}$ and (c) an integer c, the method combining projection and partition (called MPP_c for short) finds the distributed strategy with the following steps.

1. For all $i = 1, \ldots, n$, create abstraction game $\bar{\mathcal{G}}_i$ with history depth c, where in the construction, the set of risk states $\mathcal{V}_{risk}$ is replaced by $Risk_i$.
2. Solve each abstraction game. If for each abstraction game, there exists a winning strategy, then we have found a distributed strategy.

Nevertheless, it can be observed that creating a "good" partition of risk states is combinational, implying that heuristics should be used in practice.

Definition 34. *Consider distributed game $\mathcal{G}$ with safety conditions defined by $\mathcal{V}_{risk}$, define a risk partition $(Risk_1, \ldots, Risk_n)$ over $\mathcal{V}_{risk}$ to be **good** when by applying it, MPP_c finds a distributed strategy for $\mathcal{G}$.*

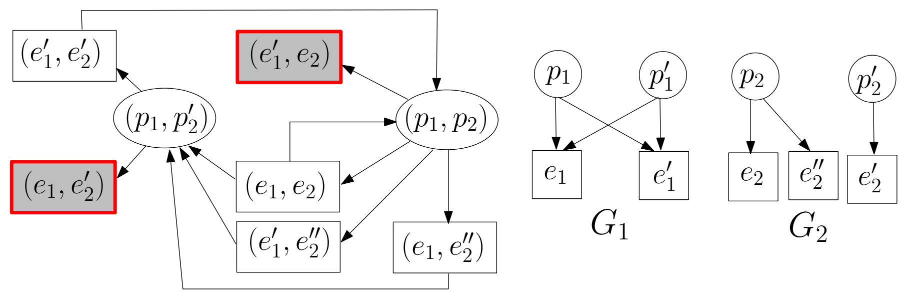


Figure 7.5: A distributed game where cooperation is possible for winning.

Lemma 6. *Given distributed game $\mathcal{G}$, finding a good risk partition for MPP_0 ($DGSafeMPP_0$) is NP-complete to the size of the distributed game graph.*

Proof. **(NP)** Finding a good risk partition for MPP_0 is in NP: once when a risk partition is selected, creating and solving all abstraction games to see if the distributed strategy exists (i.e., checking if the risk partition is good) can be done in polynomial time to the size of the distributed game graph.

(NP-C) We perform a reduction from 3SAT to $DGSafeMPP_0$ that operates in polynomial time. To give intuitive ideas how the reduction is performed, for an example 3SAT formula $\phi = (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$, the reduced distributed game is shown in Figure 7.6. Given a 3SAT Boolean formula ϕ with m variables and l clauses, we first create $m+1$ local game graphs $G_1, \ldots, G_m, G_{choose}$ as follows.

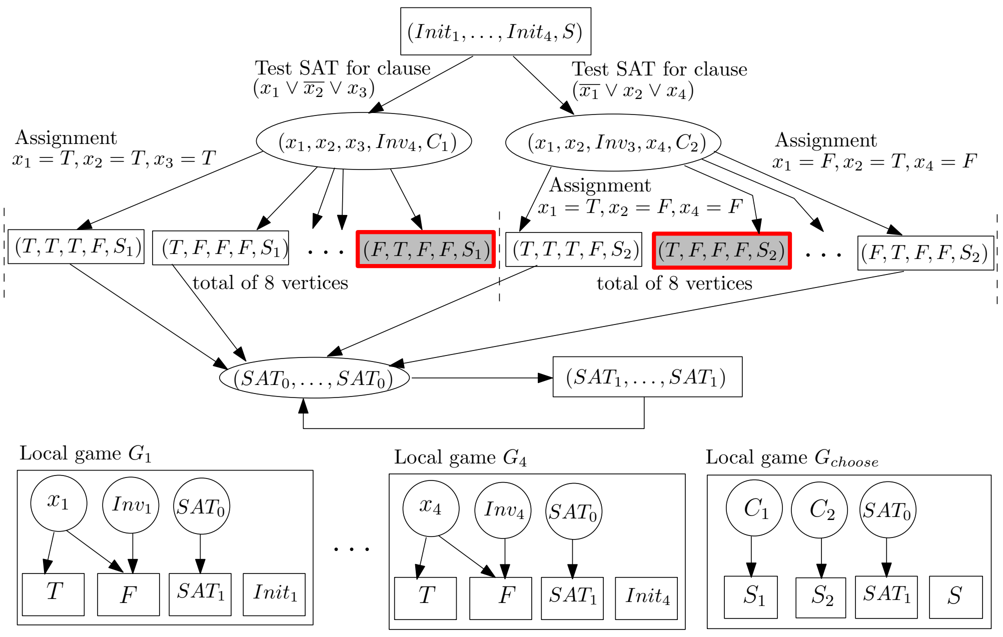

$(Init_1, \dots, Init_4, S)$
Test SAT for clause $(x_1 \vee \overline{x_2} \vee x_3)$
Test SAT for clause $(\overline{x_1} \vee x_2 \vee x_4)$
$(x_1, x_2, x_3, Inv_4, C_1)$
$(x_1, x_2, Inv_3, x_4, C_2)$
Assignment $x_1 = T, x_2 = T, x_3 = T$
Assignment $x_1 = T, x_2 = F, x_4 = F$
Assignment $x_1 = F, x_2 = T, x_4 = F$
(T, T, T, F, S_1)
(T, F, F, F, S_1)
(F, T, F, F, S_1)
(T, T, T, F, S_2)
(T, F, F, F, S_2)
(F, T, F, F, S_2)
total of 8 vertices
total of 8 vertices
$(SAT_0, \dots, SAT_0)$
$(SAT_1, \dots, SAT_1)$
Local game G_1
Local game G_4
Local game G_{choose}
x_1 Inv_1 SAT_0 T F SAT_1 $Init_1$
x_4 Inv_4 SAT_0 T F SAT_1 $Init_4$
C_1 C_2 SAT_0 S_1 S_2 SAT_1 S

Figure 7.6: The reduced distributed game for the SAT problem $\phi = (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$.

- For $i = 1, \dots, m$, $G_i = (V_{s_i} \uplus V_{e_i}, E_i)$, where
  - $V_{s_i} = \{x_i, Inv_i, SAT_0\}$.
  - $V_{e_i} = \{T, F, SAT_1, Init_i\}$.
  - $E_i = \{(x_i, T), (x_i, F), (Inv_i, F), (SAT_0, SAT_1)\}$.
- $G_{choose} = (V_{s_{choose}} \uplus V_{e_{choose}}, E_{choose})$, where
  - $V_{s_{choose}} = \bigcup_{j=1,\dots,l} C_j \cup \{SAT_0\}$.
  - $V_{e_{choose}} = \bigcup_{j=1,\dots,l} S_j \cup \{SAT_0, S\}$.
  - $E_{choose} = \bigcup_{j=1,\dots,l} (C_j, S_j) \cup \{(SAT_0, SAT_1)\}$.

From local game graphs, we create the distributed game $\mathcal{G} = (\mathcal{V}_s \uplus \mathcal{V}_e, \mathcal{E}, Acc)$. Notice that when performing a direct product over local game graphs, we have generated a distributed game graph of size exponential to the number of variables. This breaks the polynomial time reduction and should be avoided. Instead we construct a distributed game graph which is linear to the number of clauses, and algorithm MPP_0 is performed on this smaller game graph. Algorithm 11 describes such construction of the distributed game (see Figure 7.6 for concepts). The total number of vertices in the generated distributed game is of size $9l + 3$. The required time for construction is polynomial to m and l.

The final argument is to show that ϕ is satisfiable if and only if in the reduced game $\mathcal{G}$, there exists a good risk partition (for player to win from the initial state).

- ($\Rightarrow$) We first show how to create a partition of the risk states, followed by showing the the distributed strategy.

Algorithm 11: Distributed Game Construction Algorithm

```
Data: 3SAT Boolean formula $\phi$ with $m$ variables and $l$ clauses, local game graphs
      $G_1, \ldots, G_m, G_{choose}$
Result: Distributed game $\mathcal{G} = (\mathcal{V}_s \uplus \mathcal{V}_e, \mathcal{E}, Acc)$
begin
    Initialize $\mathcal{V}_s, \mathcal{V}_e, \mathcal{E}$ as empty sets.
    Initialize the set of risk states $\mathcal{V}_{risk}$ as empty sets.
    $\mathcal{V}_e := \mathcal{V}_e \cup \{(Init_1, \ldots, Init_m, S)\}$ // Initial vertex
    $\mathcal{V}_s := \mathcal{V}_s \cup \{(SAT_0, \ldots, SAT_0)\}$
    $\mathcal{V}_e := \mathcal{V}_e \cup \{(SAT_1, \ldots, SAT_1)\}$
    // We use $l_\alpha$ to represent the literal which can be either $x_\alpha$ or $\overline{x_\alpha}$.
    for clause $Clause_i = (l_\alpha \vee l_\beta \vee l_\gamma)$ using variable $x_\alpha, x_\beta, x_\gamma$, $i = 1, \ldots, l$ do
        $\mathcal{V}_s := \mathcal{V}_s \cup \{(v_1, \ldots, v_j, \ldots, v_n, C_i)\}$: if $j \in \{\alpha, \beta, \gamma\}$ then $v_j := x_j$ else $v_j := Inv_j$.
        for 8 possible assignments $(a_\alpha, a_\beta, a_\gamma) \in \{T, F\}^3$ of $Clause_i$ do
            $\mathcal{V}_e := \mathcal{V}_e \cup \{(v'_1, \ldots, v'_j, \ldots, v'_n, S_i)\}$: if $j \in \{\alpha, \beta, \gamma\}$ then $v'_j := a_j$ else
            $v'_j := F$.
            $\mathcal{E} := \mathcal{E} \cup \{((v_1, \ldots, v_j, \ldots, v_n, C_i),$
            $(v'_1, \ldots, v'_j, \ldots, v'_n, S_i))\}$
            if assignment $(a_\alpha, a_\beta, a_\gamma)$ makes $Clause_i$ UNSAT then
                $\mathcal{V}_{risk} := \mathcal{V}_{risk} \cup \{(v'_1, \ldots, v'_j, \ldots, v'_n, S_i)\}$
            else
                $\mathcal{E} := \mathcal{E} \cup \{((v'_1, \ldots, v'_j, \ldots, v'_n, S_i),$
                $(SAT_0, \ldots, SAT_0))\}$
    $\mathcal{E} := \mathcal{E} \cup \{((SAT_0, \ldots, SAT_0), (SAT_1, \ldots, SAT_1))\}$
    $\mathcal{E} := \mathcal{E} \cup \{((SAT_1, \ldots, SAT_1), (SAT_0, \ldots, SAT_0))\}$
    Let $Acc$ be defined by the set of risk states $\mathcal{V}_{risk}$
    return $(\mathcal{V}_s \uplus \mathcal{V}_e, \mathcal{E}, Acc)$
```

1. If ϕ is satisfiable, let the set of the satisfying literals be $(l_1, \ldots, l_m)$. Define the risk partition as follows: If for clause $Clause_i$, literal l_j makes it SAT, then add the risk vertex $v \in \mathcal{V}_e \cap \mathcal{V}_{risk}$ where $\exists v_1 \in \{x_1, Inv_1\} \ldots \exists v_m \in \{x_m, Inv_m\} \cdot ((v_1, \ldots, v_m, C_i), v) \in \mathcal{E}$ to $Risk_j$ (there is only one such vertex), if it is not added to any partition before. As each clause in ϕ is satisfiable, for every risk state v_i (representing UNSAT of clause i) in $\mathcal{G}$, it is contained by one $Risk_j$ of the abstraction game $\bar{\mathcal{G}}_j$, $j \in \{1, \ldots, m\}$.
   - For example, in Figure 7.6, one satisfying assignment is $(x_1, \overline{x_2}, x_3, x_4)$. Then define the risk partition by setting $Risk_1 = \{(F, T, F, F, S_1)\}$, $Risk_4 = \{(T, F, F, F, S_2)\}$, and $Risk_2 = Risk_3 = \emptyset$.
2. The set of satisfying literals also defines the positional strategy for each local game: For literal $l_j = x_j$, in local game G_j move from vertex x_j to T; for literal $l_j = \overline{x_j}$, in local game G_j move from vertex x_j to F.

3. Solving the game using MPP_0 based on the partition specified in (1) generates the distributed strategy in (2).

- ($\Leftarrow$) For the other direction it follows similar arguments.

$\square$

7.3.3.1 Limitation

Before ending this section, we consider the distributed game in Figure 7.2b, where we cannot find a solution with $c = 0$ or $c = 1$, for all possible partitions of risk states.

- As the MPP method simply unrolls the game graph and is unaware of on cases when $c' > c$, it cannot inform us the *impossibility* of strategies.
- The result of directly solving the distributed game graph using attractor is *uninformative*: it is known that if a global strategy (both local games can see contents of others) does not exist, then no distributed strategy exists. However, in this case there exists a global strategy by performing $((p_1, p_2), (e_1, e_2))$ and $((p_1, p'_2), (e'_1, e_2))$.

7.4 Observation + Antichain + Decomposition

We present a three-step algorithm to find safety strategies for distributed games. This algorithm creates enhanced cooperation among local controllers.

7.4.1 Local Observations

The structure of a distributed game graph is important for generating the distributed strategy using memory. From the view of control in a local game, during a play the position of other local games is not known with absolute precision. However, this does not mean that it knows nothing: as the game graph of the distributed game is known *prior to the play*, each local control can infer its **local observations** (based on the structure of the distributed game graph) and should use it when executing the strategy.

Consider the distributed game in Figure 7.2a where G_1 and G_2 are local game graphs. For position p_1, if its previous state is e_1, then the local controller of G_1 knows that it can only be in the global state (p_1, p'_2): the move (e_1, p_1) assigns a local observation $\{(p_1, p'_2)\}$ to p_1. It can be further observed that p_1 can clearly locate itself due to the emptiness of the intersection between two observations induced by e_1 and e'_1. But for the distributed game in Figure 7.2b, for p_1, if its previous state is e_1, then it can be either (p_1, p'_2) or (p_1, p_2): the move (e_1, p_1) assigns a local observation $\{(p_1, p'_2), (p_1, p_2)\}$ to p_1.

Definition 35. *For a distributed game $\mathcal{G}$, a local observation obs of local game G_i maps from an edge $(v_1, v_2) \in E_i$ to $2^{\mathcal{V}_s \uplus \mathcal{V}_e}$. We denote the set of local observations for G_i to be Obs_i.*

Consider Figure 7.2b, for G_1 we create two observations $obs_{11}, obs_{12} \in Obs_1$.

- For $obs_{11}((v_{pre}, v))$, it returns $\{(p_1, p'_2), (p_1, p_2)\}$ when $v_{pre} = e_1$ and $v = p_1$; else it returns $\emptyset$.
- For $obs_{12}((v_{pre}, v))$, it returns $\{(p_1, p_2)\}$ when $v_{pre} = e'_1$ and $v = p_1$; else it returns $\emptyset$.

(Generating local observations for distributed games) To create the set of local observations Obs_i for each local game G_i, proceed as follows. For the local observation over edge (a, b) in local game G_i, extract the set of edges $Set := \{e = ((e_1, \ldots, e_n), (s_1, \ldots, s_n)) \mid e \in \mathcal{E} \ \wedge \ e_i = a \ \wedge \ s_i = b\}$ to create $\{(v_1, \ldots, v_n) \mid \exists((e_1, \ldots, e_n), (s_1, \ldots, s_n)) \in Set : (v_1, \ldots, v_n) = (s_1, \ldots, s_n)\}$. The algorithm is straightforward and is omitted here.

Lemma 7. *For a distributed game $\mathcal{G}$, the size of Obs_i in a local game G_i is bounded by $\mathcal{O}(|V_{ei}||V_{si}|)$, and for each observation $obs_i \in Obs_i$, obs_i may return a state set of size $\mathcal{O}(|\mathcal{V}_s \uplus \mathcal{V}_e|)$.*

Proof. Straightforward based on the definition. □

Lemma 8. *It takes time $\mathcal{O}((\sum_{i=1}^{n} |V_{ei}||V_{si}|)|\mathcal{V}_s \uplus \mathcal{V}_e|)$ to create all local observations for a distributed game $\mathcal{G}$.*

Proof. Straightforward based on the definition. □

7.4.2 Distributed Controllable Predecessor

The second step of this algorithm computes the **distributed controllable predecessor**, which is adapted from techniques of solving games of incomplete (imperfect) information [DWDR06, Rei84, DWDHR06]. For a safety game, we can always modify the game to let all risk states having no outgoing edges. By doing so, asking whether a system is safe amounts to the query of continuous execution (no deadlock) for each local controller. For details, see [DWDR06].

For a distributed game $\mathcal{G}$, define $Enabled(\langle e_1, \ldots, e_n \rangle)$ be the set of states $S \in \mathcal{V}_s$ where $\forall s = (s_1, \ldots, s_n) \in S$, for local game G_i, if $s_i \in V_{si}$, then s_i can move with one edge to location e_i; else (the case when $s_i \in V_{ei}$) $s_i = e_i$. Also, let $Post_{\langle \sigma_1, \ldots, \sigma_n \rangle}(S)$ be the set S' of reachable states from S where $\forall e = (e_1, \ldots, e_n) \in S$, $\exists s' \in S'$ such that e_i moves via the i^{th} component of the environment edge $\langle \sigma_1, \ldots, \sigma_n \rangle$ to $proj(s', i)$.

Let q be an antichain of set of locations in the distributed game, define the distributed controllable predecessor DCPre as follows.

$$\begin{gathered}\mathsf{DCPre}(q) := \lceil \{ s \subseteq \mathcal{V}_s \mid \exists e_1 \in V_{e1} \ldots \exists e_n \in V_{en} \cdot \\ \forall obs_1 \in Obs_i \ldots \forall obs_n \in Obs_n \cdot \exists s' \in q : \\ s \subseteq Enabled(\langle e_1, \ldots, e_n \rangle) \wedge \\ \bigcup_{\langle \sigma_1, \ldots, \sigma_n \rangle \in \mathcal{E}} (Post_{\langle \sigma_1, \ldots, \sigma_n \rangle}(e_1, \ldots, e_n)) \\ \cap\, obs_1(\sigma_1) \cap \ldots \cap obs_n(\sigma_n) \subseteq s' \} \rceil\end{gathered}$$

Intuitively, a set s belongs to $\mathsf{DCPre}(q)$ iff

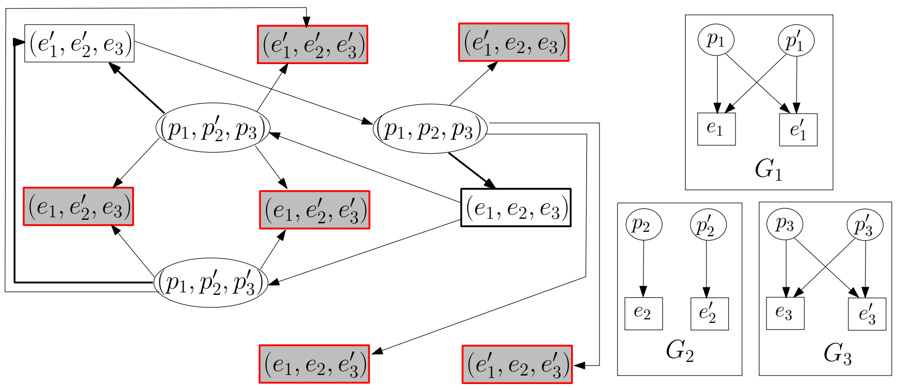


Figure 7.7: A distributed game with three local games.

- In s, a tuple of controllable actions moving to $(e_1, \ldots, e_n)$ can be enabled, where e_i is in G_i.
- When player moves to $(e_1, \ldots, e_n)$, after the environment has played, any local observation suffices to determine in which set s' of q the next state lies.
- s is maximal.

Analysis We establish an analogy between DCPre and the definition of the controllable predecessor CPre for games of imperfect information (Section 7.2.4):

- Relate the tuple $\langle e_1, \ldots, e_n \rangle$ in DCPre to an edge label $\sigma \in \Sigma$ for player in CPre.
- Relate the combined observation $(obs_1, \ldots, obs_n)$, where $\forall i : obs_i \in Obs_i$ in DCPre to a single observation in CPre.

Therefore, calculating DCPre employs the same complexity as CPre but the number of actions and the number of observations are larger (due to the distributed setting). For concrete algorithm of CPre (a worklist algorithm over set of locations) it can be found in [DWDR06] and is omitted here. By (a) calculating DCPre$^*(\{\mathcal{V}_s \uplus \mathcal{V}_e\})$, i.e., applying DCPre continuously until the fixed point is reached and (b) intersecting the result with the initial state, we can generate a **global observation strategy automaton**, which is the similar to the strategy automaton for games of imperfect information. Here we illustrate the process of automaton construction using an example. For algorithmic details, we refer readers to [DWDR06] for the analogy.

Definition 36. *For the distributed game $\mathcal{G}$ with local observation Obs_i for each local game i, define the global observation strategy automaton $\mathcal{A}_{\mathcal{G}} = (\mathcal{Q}, q_0, \mathcal{L}, \delta)$.*

- $\mathcal{Q} =$ *DCPre*$^*(\{\mathcal{V}_s \uplus \mathcal{V}_e\}) \cup \{q_0\}$ *is the set of states.*
- q_0 *is the initial state.*
- $\mathcal{L} : \mathcal{Q} \setminus \{q_0\} \rightarrow \{\langle \sigma_1, \ldots, \sigma_n \rangle \mid \sigma_i \in V_{ei}\}$ *is the state labeling.*
- $\delta : \mathcal{Q} \times \{\langle obs_1, \ldots, obs_n \rangle \mid obs_i \in Obs_i\} \rightarrow \mathcal{Q}$ *is the transition function.*

Lemma 9. *DCPre can be computed in* `EXPTIME` *to the size of the distributed game.*

Proof. It is computable using the antichain algorithm in [DWDR06] running in

EXPTIME. The number of local observations created and the number of actions in the distributed game do not change the EXPTIME complexity. □

7.4.2.1 Example: Calculating DCPre*

Here we give an example how a global observation strategy automaton is constructed from a distributed game. Consider the distributed game $\mathcal{G}$ in Figure 7.7. To solve the safety game, we first construct the set of local observations, and calculate the DCPre$^*(\{\mathcal{V}_s \uplus \mathcal{V}_e\})$. Let the set of initial states be $\{(e'_1, e'_2, e_3), (e_1, e_2, e_3)\}$.

- Local observations in G_1 (Obs_1):
  - (obs_{11}) The edge (e_1, p_1) creates an observation $\{(p_1, p'_2, p_3), (p_1, p'_2, p'_3)\}$.
  - (obs_{12}) The edge (e'_1, p_1) creates an observation $\{(p_1, p_2, p_3)\}$.
- Local observations in G_2 (Obs_2) is not required, as the strategy is fixed to positional only.
- Local observations in G_3 (Obs_3):
  - (obs_{31}) The edge (e_3, p_3) creates an observation $\{(p_1, p'_2, p_3), (p_1, p_2, p_3)\}$.
  - (obs_{32}) The edge (e_3, p'_3) creates an observation $\{(p_1, p'_2, p'_3)\}$.

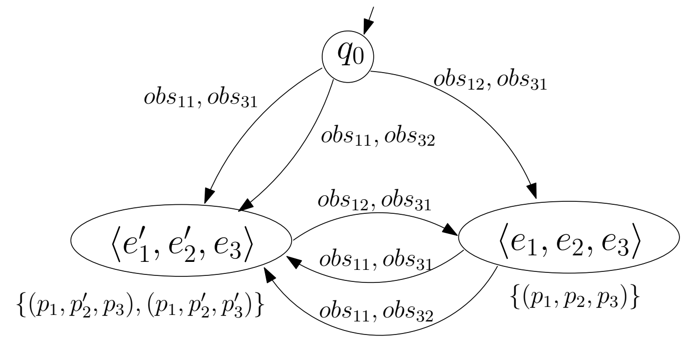


Figure 7.8: The global observation strategy automaton for the game in Figure 7.7.

The intermediate steps are as follows.

- $S_1 = \mathsf{DCPre}(\{\mathcal{V}_s \uplus \mathcal{V}_e\}) = \{(p_1, p_2, p_3)\}_{\langle e_1, e_2, e_3\rangle}, \{\{(p_1, p'_2, p_3), (p_1, p'_2, p'_3)\}_{\langle e'_1, e'_2, e_3\rangle}\}$. Here for explanation we check each step of $\{(p_1, p_2, p_3)\}$:
  - For $\{(p_1, p_2, p_3)\}$, the enabled control action is $\langle e_1, e_2, e_3\rangle$, meaning that $Post(\{(p_1, p_2, p_3)\}) = (e_1, e_2, e_3)$.
  - The environment then move to $S' = \{(p_1, p'_2, p_3), (p_1, p'_2, p'_3)\}$. Now intersect with all combinations of observations.
    - ∗ Intersect $obs_{11}(e_1) \cap obs_{31}(e_3)$ we get $\{(p_1, p'_2, p_3)\} \subseteq S$.
    - ∗ Intersect $obs_{11}(e_1) \cap obs_{32}(e_3)$ we get $\emptyset \subseteq S$.
    - ∗ Intersect $obs_{12}(e_1) \cap obs_{31}(e_3)$ we get $\emptyset \subseteq S$.
    - ∗ Intersect $obs_{12}(e_1) \cap obs_{32}(e_3)$ we get $\{(p_1, p'_2, p'_3)\} \subseteq S$.

```
Algorithm 12: Direct Decomposition
Data: Global observation strategy automaton $\mathcal{A}_\mathcal{G} = (\mathcal{Q}, q_0, \mathcal{L}, \delta)$ and local observations
      $(Obs_1, \ldots, Obs_n)$
Result: NFA $\{A_1, \ldots, A_n\}$, $A_i = (Q_i, q_{0_i}, L_i, \delta_i)$. $\delta_i \subseteq Q_i \times Obs_i \times Q_i$
begin
    Denote $proj(\langle a_1, \ldots, a_n \rangle, i)$ as the $i^{\text{th}}$ component of in tuple $\langle a_1, \ldots, a_n \rangle$
    for $i = 1 \ldots n$ do
        create $A_i$ from $\mathcal{A}_\mathcal{G}$ as follows:
        $Q_i = \mathcal{Q}$, $q_{0_i} = q_0$
        $\forall q \in \mathcal{Q} : \mathcal{L}(q) = \langle e_1, \ldots, e_n \rangle \rightarrow L_i(q) = e_i$
        $\forall q_1, q_2 \in \mathcal{Q} : q_2 \in \delta(q_1, \langle e_1, \ldots, e_n \rangle) \rightarrow \delta_i(q_1, e_i) = q_2$
```

  - Thus, we add $\{(p_1, p_2, p_3)\}$ to $\mathsf{DCPre}(\{S\})$
- $S_2 = \mathsf{DCPre}(\{S_1\}) = S_1$, and we have reached the fixed point. Figure 7.8 shows the generated global observation strategy automaton.

7.4.2.2 Example in Figure 7.2b

We now revisit the distributed game in Figure 7.2b by calculating $\mathsf{DCPre}^*(\{\mathcal{V}_s \uplus \mathcal{V}_e\})$.

- $S_1 = \mathsf{DCPre}(\{\mathcal{V}_s \uplus \mathcal{V}_e\}) = \{\{(p_1, p_2)\}_{\langle e_1, e_2 \rangle}, \{(p_1, p'_2)\}_{\langle e'_1, e'_2 \rangle}\}$.
- $S_2 = \mathsf{DCPre}(\{S_1\}) = \{\{(p_1, p'_2)\}_{\langle e'_1, e'_2 \rangle}\}$.
- $S_3 = \mathsf{DCPre}(\{S_2\}) = \{\emptyset\}$; $S_4 = \mathsf{DCPre}(\{S_3\})$ and the fixed point is reached.

Therefore, although the distributed game has a global strategy for safety, the calculation of DCPre suggests the impossibility of practical strategies.

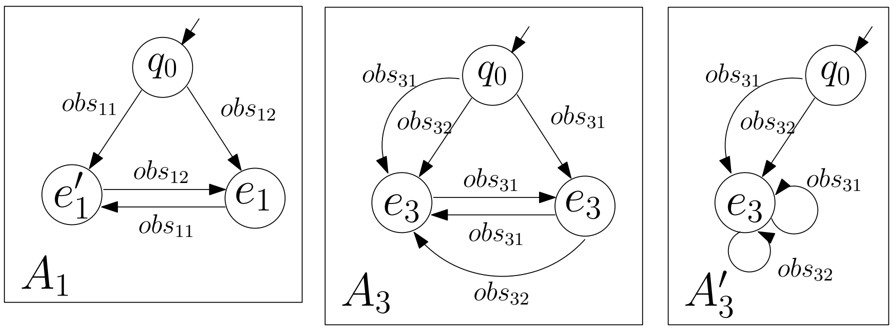
7.4.3 Direct Decomposition of Automaton for Strategies

The last step of our method is to decompose the global observation strategy automaton $\mathcal{A}_\mathcal{G}$ for the distributed game $\mathcal{G}$ to a tuple of **local strategy automata** $\langle A_1, \ldots, A_n \rangle$, where each strategy automaton A_i employs a distributed strategy for local game G_i. Formally, $A_i = (Q_i, q_{0_i}, L_i, \delta_i)$, where Q_i is the set of states, q_{0_i} is the initial state, $L_i \in V_{i_1}$ is the labeling function (to indicate which action to take), and $\delta_i : Q_i \times Obs_i \rightarrow Q_i$ is the state update function.

As in $\mathcal{A}_\mathcal{G}$, each edge is equipped with a tuple $\langle obs_1, \ldots, obs_n \rangle$ for the indication of the next move, it is at first natural to split $\mathcal{A}_\mathcal{G}$ to $A_i, \ldots, A_n$ as follows: for an edge in A_i, it is labeled by σ_i, which is the i^{th} component in the corresponding edge $\langle obs_1, \ldots, obs_n \rangle$ in $\mathcal{A}_\mathcal{G}$. Similar modification can be done on the labeling function. We refer this operation as a **direct decomposition** of a global observation strategy automaton, and Algorithm 12 formulates this idea.

Nevertheless, the direct decomposition may be problematic and needs further fixing, if the generated local strategy automata are *nondeterministic*. Consider the generated global observation strategy automaton in Figure 7.8. The direct decomposition creates three local strategy automata A_1, A_2, and A_3 (A_2 is omitted as it is positional), as shown in Figure 7.9. Consider the run of A_3:

- Starting from q_0, when receiving observation obs_{31} (i.e., from previous location e_3 to p_3), the controller moves can move to two vertices labeled e_3.
- If it moves to the left vertex, then it is not able to react to obs_{32}, implying a deadlock of moves.



$obs_{11} : (e_1, p_1) \quad obs_{12} : (e_1', p_1) \quad obs_{31} : (e_3, p_3) \quad obs_{32} : (e_3, p_3')$

Figure 7.9: Direct decomposition for the global observation strategy automaton in Figure 7.8; A_3' is a proposed fix of A_3. The box below indicates the corresponding (previous-location, current-location) pair for the observation.

Still, if the decomposed local automata are all deterministic, we have a distributed solution via direct decomposition.

Lemma 10. *For distributed game $\mathcal{G}$, if direct decomposition (using Algorithm 12) from the global observation strategy automaton $\mathcal{A}_{\mathcal{G}}$ generates automata $A_1, \ldots, A_n$ which are all deterministic, then $A_1, \ldots, A_n$ are the corresponding local strategy automata.*

Proof. If $A_1, \ldots, A_n$ are deterministic, then for the product of $A_1, \ldots, A_n$, starting from the initial state, $A_1 \times \ldots \times A_n$ directly defines $\mathcal{A}$. $\square$

7.4.4 Decomposing Automaton for Strategies via Fixing

For cases where direct decomposition generates non-deterministic automata, it is then required to modify the generated result $\langle A_1, \ldots, A_n \rangle$ to maintain determinacy and deadlock freedom while ensuring the product behavior to be constrained by $\mathcal{A}_{\mathcal{G}}$. For this purpose, we consider the *fixing of automata*. For simplicity, from now on we assume that for the safety game only one initial location exists.

Definition 37. *For distributed game $\mathcal{G}$, let the generated strategy automaton be $\mathcal{A}_{\mathcal{G}}$ and the direct decomposition of the automaton be $\langle A_1, \ldots, A_n \rangle$, where $A_i = (Q_i, q_{0_i}, L_i, \delta_i)$. Define an automata fixing to be $\langle (A_1', \gamma_1), \ldots, (A_n', \gamma_n) \rangle$, where $A_i' = (Q_i', q_{0_i}, L_i', \delta_i')$, $\delta_i' : Q_i' \times Obs_i \rightarrow Q_i'$. $\gamma_i : Q_i \rightarrow Q_i' \cup \{\texttt{NULL}\}$ is a function which maps vertices in Q_i to elements in Q_i' or to a special element NULL. A fix should follow the following properties.*

1. $\forall q \in Q_i : \gamma_i(q) \neq \texttt{NULL} \rightarrow L_i(q) = L'_i(\gamma_i(q))$.
2. $\forall q \in Q_i : \gamma_i(q) \neq \texttt{NULL} \rightarrow (\forall obs \in Obs_i \cdot \forall q' \in Q_i : q' \in \delta_i(q, obs) \rightarrow \gamma_i(q') = \delta'_i(\gamma_i(q), obs))$.
3. $\forall q, q' \in Q'_i \cdot \forall obs \in Obs_i : q' = \delta'_i(q, obs) \rightarrow (\exists q^\star \in Q'_i \cdot \exists obs' \in Obs_i : q^\star = \delta'_i(q', obs))$.
4. *The language defined by $\mathcal{A}_\mathcal{G}$ contains the language defined by $A'_1 \times \ldots \times A'_n$.*

We explain the meaning of the above properties as follows.

- For (1), the mapping preserves the action defined in the vertex label.
- For (2), if a vertex in A_i is not dropped, then the transformation should preserve all edges. At the same time it keeps determinacy in the fix (as in $\gamma_i(q') = \delta'_i(\gamma_i(q), obs)$, $\delta'_i(\gamma_i(q), obs)$ is a function).
- For (3), it is used to ensure the non-terminating behavior based on the structure induced by (2).

Example We consider a fix over A_3 by using A'_3 in Figure 7.9 (A_1 remains the same), where γ_3 maps two vertices labeled e_3 in A_3 to a single vertex labeled e_3. It can be observed that it is a correct automata fix by examining the properties above. If G_1 and G_3 follow the strategies defined by A_1 and A'_3, it is guaranteed that $\mathcal{G}$ never reaches the set of risk states.

7.4.4.1 Complexity for automata fixing

One important observation over the fixing is that as both the product automaton $A'_1 \times \ldots \times A'_n$ and $\mathcal{A}_\mathcal{G}$ are deterministic, the checking of language inclusion can be done in polynomial time to the size of the automaton. This generates an intuitive `NP` algorithm by creating an instance and check for all properties: this `NP` setting leads to the potential of introducing SAT solvers for finding a fix.

7.5 Related Work

Previous work on distributed synthesis is mainly concerned with undecidability results [PR90, Jan07] or with identifying decidable architectural classes [PR90, FS05, GLZ04, GLZ05, MW03]. As these decidable subclasses are very restrictive, we are exploring a different route here by exploring algorithms for synthesizing distributed strategies with certain bounds on resources only.

Except the work mentioned above, we find two directions trying to draw the line for decidable results (for all architectures): Works from Gastin, Lerman and Zeitoun [GLZ04, GLZ05] derive decidable results by replacing strategies using local memory (i.e., each local game views its own location) to strategies using causal memory; this offers each controller a view including all actions (also from other local games) causally in the past. As stated in [GLZ05], it is an almost complete global view; under this setting, for series-parallel systems it is decidable [GLZ05] while undecidability remains for some other cases [GLZ04]. To fulfill the causal view, as stated in [GLZ05], external communications between processes should be added in the implementation. Overall, this approach might lead to implementation problems with concrete applications[5]. The algorithms mentioned in this chapter are all based on respecting local strategies defined in [MW03]. The second direction is from Madhusudan and Thiagarajan [MT02], where they look at the distributed synthesis problem, and propose three rules to derive decidable results; when three rules are all satisfied then distributed synthesis is decidable. For example, the R1 rule demands that a specification should be robust, i.e., if one linearization of an execution is contained in the specification, then all of its linearizations must also be [MT02]. R2 and R3 are constraints on the generated strategies, while these constraints differs from algorithms specified in this work. E.g., R2 demands that each local strategy can only remember the length of the local history but not the history itself [MT02]. As also mentioned in [MT02], rule R2 poses very strong restrictions on the applicability to allow decidability. We also find that our concept of applying constant depth memory is independently suggested in their potential extensions (which they claim to be worth studying). For this, our motivation comes from the fact that positional strategies are sometimes insufficient (in distributed games), while their motivation is to release the R2 constraint (for distributed synthesis). For them no concrete algorithm is proposed.

Our algorithms use, adapt, and generalize many known ingredients from the literature, including the concept of partitioning risk sets in and techniques for solving games of incomplete information [Rei84] using the lattice of antichains [DWDHR06, DWDR06]. The algorithm MP_c involving projection and partitioning of the risk set for synthesizing c-forgetful distributed strategies is a generalization of the risk set partitioning algorithm with priority scheduling and model checking. Also our algorithms for generating local observations for local games and the notion of distributed controllable predecessors, as a generalization of the controlled predecessor introduced in [DWDHR06], together with the global observation strategy automaton are novel.

To sum up, we consider our approach orthogonal / complementary to the above results. We seek for general algorithms to find strategies for both decidable and undecidable classes, and our starting model (distributed game) offers intuitive translation from concrete applications to distributed synthesis. Admittedly algorithms we propose are essentially incomplete, but these resource-bounded strategies can work as fundamental gadgets to bring distributed synthesis from theory to concrete applications, as emphasized in the introduction.

Table 7.2: Algorithms for strategy finding in distributed games

Resource-Bounded Algorithms	Time Complexities
Projection	P
Projection + Risk Partition	NP
Local observation + Antichain	EXPTIME

7.6 Outlook

Table 7.2 summarizes our results on algorithms of increasing complexity for synthesizing resource-bounded distributed strategies for distributed safety games. Although we have restricted ourselves to safety winning conditions, extensions of these results to reachability and Büchi winning conditions should be straightforward; the antichain method, for example, seems to be applicable for solving distributed reachability games using the dual of the antichain lattice with the Post image operator for unfolding [DWDHR06].

Indeed, the algorithms presented here are designed for tackling distributed scheduling problems for component-based distributed systems on multicore platforms and for priority synthesis of component-based BIP systems for realizing goal-directed behavior. Preliminary experience with our prototype implementations of (a subset of) these algorithms (see Appendix, which is provided for reference and is not part of the official submission) in our synthesis engines GAVS+ and VISSBIP shows interesting results. On the other hand, our experiments also point to necessary refinements and optimizations such as application-specific heuristics for guiding the process of partitioning risk states in the MPP algorithm, based, for example, on discovering invariants of the game or by applying machine learning techniques. An obvious extension is to use SMT backend solvers instead of SAT solvers for the symbolic representation and solving of distributed games, thereby handling certain classes of timed and hybrid systems. We also need to consider the synthesis of (Pareto-)optimal strategies, for example, to generate efficient (e.g. energy) strategies / schedules, and fault-tolerant distributed strategies in case one or more of the players fail to play according to the rule book.

Another interesting line of research involves predetermining upper bounds on the resources needed for distributed winning strategies for definable subclasses of distributed games.

[5]E.g., consider modeling an unreliable network to a distributed game, where the goal is to synthesize strategies for processes to overcome the loss of message. When using causal view, these added communications for implementing the causal view can also suffer from message loss. As the synthesized result is based on the assumption that the causal view can be carried out, this assumption can hardly be fulfilled.

Appendix A: Symbolic Algorithm for MP_c

We outline a symbolic version of the MP_c algorithm, which has been implemented in an experimental version of GAVS+ [CKLB11]. In the algorithm, we try to avoid constructing each abstraction game in isolation based on the definition. Given a distributed game graph $\mathcal{G} = (\mathcal{V}_s \uplus \mathcal{V}_e, \mathcal{E})$ constructed from n local games, together with the set of initial and risk states $\mathcal{V}_{ini}$, $\mathcal{V}_{risk}$, the algorithm proceeds with the following steps.

- Create a symbolic version of the augmented game $\mathcal{G}' = ({\mathcal{V}_s}' \uplus {\mathcal{V}_e}', \mathcal{E}')$ which contains memory slots and performs the memory update in each transition of $\mathcal{G}$.
  1. (Number of BDD variables) For $i = 1, \dots, n$, in local game $G_i = (V_{si} \uplus V_{ei}, E_i)$, we need to use $(c+1)\lceil log_2(|V_{si} \uplus V_{ei}| + 1)\rceil$ bits to represent the state. We use $|V_{si} \uplus V_{ei}| + 1$ rather than $|V_{si} \uplus V_{ei}|$ because we need to represent the empty memory token $\Box$. Thus the total number of BDD variables used is $2\sum_{i=1\dots n}(c+1)\lceil log_2(|V_{si} \uplus V_{ei}|+1)\rceil$ (a factor of 2 is for the preimage and the postimage).
     - We denote $pre(bddvar(i,j))$ as the preimage of the j-th history in local game G_i ($j = 0$ means the current state), similarly we use $post(bddvar(i,j))$ for the postimage.
  2. (Symbolic transition creation) The game construction requires an additional step, i.e., when a transition in $\mathcal{G}$ is performed, the memory should be updated in $\mathcal{G}'$. Therefore, define a memory update predicate $pred_{MEM} :=$

     $$\bigwedge_{i=1}^{n} \bigwedge_{j=0}^{c-1} pre(bddvar(i,j)) \Leftrightarrow post(bddvar(i,j+1))$$

     Thus transitions in $\mathcal{E}$ should be conjuncted with $pred_{MEM}$ to get $\mathcal{E}'$.
  3. (Transition pruning) We start from the initial state (with memory content equals empty) and perform forward reachability analysis to derive the set of reachable states $Reach$, where the memory update is correct. Then restrict $\mathcal{E}'$ generated in (2) to derive the set of outgoing transitions from $Reach$ (still call it $\mathcal{E}'$). This step is required, as when we later calculate the risk attractor (using backward computation), directly using transitions in (2) may overestimate the set of unsafe transitions. Also, eliminate any outgoing transitions from risk states by performing set differences.
- Calculate the *environment attractor* $\mathsf{Attr}_e(Reach \cap \mathcal{V}_{risk})$, which is the set of states from which environment can enforce to move to a risk state, regardless of moves done by player [GTW02]. Derive a set of *risk edges*, which are all the (player) edges leading to the risk attractor. We compute the set of risk edges $\mathcal{E}'_{risk}$ with the following formula: $pre(Reach) \cap \mathcal{E}'_s \cap post(Attr_1(Reach \cap \mathcal{V}_{risk}))$.
- For $i = 1 \dots n$, perform the following.
  1. For local game i, for $\mathcal{E}'_{risk}$ perform existential quantification over variables of other games indexed from $1, \dots, i-1, i+1, \dots, n$. By doing so, we have generated the risk edges $\bar{\mathcal{E}}_{i_{risk}}$ in the abstraction game $\bar{\mathcal{G}}_i$.

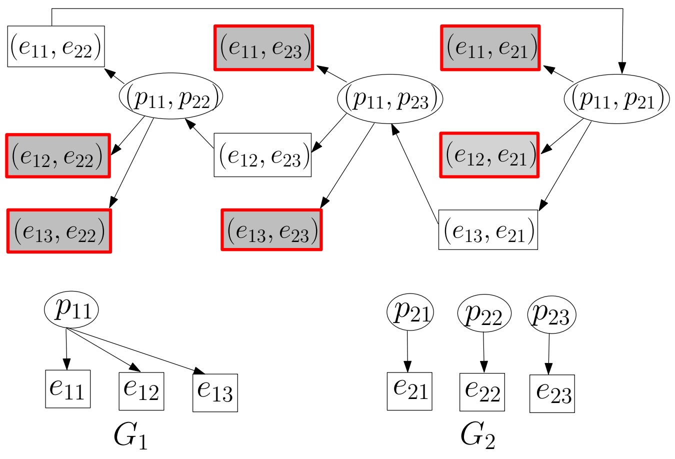


Figure 7.10: Example for distributed strategies using finite-memory.

2. Check if starting from the initial state, performing forward reachability analysis using $\mathcal{E}' \setminus \bar{\mathcal{E}}_{i_risk}$ (the set of transitions where local game G_i does not use $\bar{\mathcal{E}}_{i_{risk}}$) would lead to risk states or deadlock states (induced by restricting edges in $\bar{\mathcal{E}}_{i_{risk}}$). If not, then report a strategy of MP_c by using edges $E_i \setminus \bar{\mathcal{E}}_{i_{risk}}$ for local game G_i.

Appendix B: Proof of Lemma 5

For lemma 5, we prove the following statement: There exists an infinite set of distributed games $\{\mathcal{G}\}$ composed from two local games with the following property:

1. For $\mathcal{G}$, there exists a distributed strategy $\langle f_1, f_2 \rangle$ for safety winning.
2. For any distributed strategy $\langle f_1, f_2 \rangle$ specified in (1), the minimum size of the memory used in f_1 is at least $\frac{|V_{s1} \uplus V_{e1}|}{2}$.

Proof. (Sketch) Here we use two simplest cases to give an idea on how such family of distributed games is created.

- Consider again the distributed game in Figure 7.2a. A finite memory strategy is as follows: For local game G_1, if its previous location is e_1, then go to e_1', else go to e_1. Thus the memory size of the local strategy f_1 is at least 2.
- For the distributed game in Figure 7.10, the distributed strategy is as follows: For local game G_1, if its previous state is e_{11}, then go to e_{13}; if its previous location is e_{13} go to e_{12}; else go to e_{11}. As for each previous location, there exists a unique move to avoid entering risk state, for G_1 it should remember all (previous-location, move) pairs. Thus the memory size of any f_1 is at least 3.

We use the Algorithm 13 to create the family of distributed games. E.g., by applying $K = 3$ the algorithm returns a distributed game similar to Figure 7.10. Given an integer K, the number of vertices created in G_1 equals $K+1$, and the minimum size of memory for f_1 is at least K (following the argument similar to the case in Figure 7.10). □

Algorithm 13: Game Construction for Lemma 5

```
**Data**: Integer $K > 2$
**Result**: Distributed game $G = (\mathcal{V}_s \uplus \mathcal{V}_e, \mathcal{E}, Acc)$ from two local games $G_1$ and $G_2$
        satisfying Lemma 5
**begin**
    // Construct local game graph $G_1 = (V_{s1} \uplus V_{e1}, E_1)$
    $V_{s1} = \{p_{11}\}, V_{e1} = \bigcup_{i=1}^{K} e_{1i}, E_1 = \bigcup_{i=1}^{K} (p_{11}, e_{1i})$
    // Construct local game graph $G_2 = (V_{s2} \uplus V_{e2}, E_2)$
    $V_{s2} = \bigcup_{i=1}^{K} p_{2i}, V_{e2} = \bigcup_{i=1}^{K} e_{2i}, E_2 = \bigcup_{i=1}^{K} (p_{2i}, e_{2i})$

    // Construct $\mathcal{G}$ out of $G_1$ and $G_2$ $\mathcal{V}_s = \bigcup_{i=1}^{K} (p_{11}, p_{2i})$, $\mathcal{V}_e = \bigcup_{i=1}^{K} \bigcup_{j=1}^{K} (e_{1i}, e_{2j})$
    **create** work set $Set = \bigcup_{i=1}^{K} \bigcup_{j=1}^{K} (e_{1i}, e_{2j})$
    **create** empty set $\mathcal{V}_{risk}$
    // Construct edges for the control (player 0)
    $\mathcal{E} = \mathcal{E} \cup \bigcup_{i=1}^{K} \bigcup_{j=1}^{K} ((p_{11}, p_{2j}), (e_{1i}, e_{2j}))$
    // Construct environment edges and add risk states
    **let** integer $b = 1$
    **while** $Set \neq \emptyset$ **do**
        // $(e_{1a}, e_{2b})$ is a safe state
        Remove $(e_{1a}, e_{2b})$ from $Set$
        // Add the edge (safe state has an outgoing edge)
        $\mathcal{E} = \mathcal{E} \cup ((e_{1a}, e_{2b}), (p_{11}, p_{2c}))$ where $c \neq b$
        // Add risk states
        $\mathcal{V}_{risk} = \mathcal{V}_{risk} \cup \bigcup_{d=1;d\neq a}^{K} (e_{1d}, e_{2b})$
        $\mathcal{V}_{risk} = \mathcal{V}_{risk} \cup \bigcup_{d=1;d\neq b}^{K} (e_{1a}, e_{2d})$
        $Set = Set \setminus \bigcup_{d=1;d\neq a}^{K} (e_{1d}, e_{2b})$
        $Set = Set \setminus \bigcup_{d=1;d\neq b}^{K} (e_{1a}, e_{2d})$
        $b = c$ // Continue with the next round
    Let $Acc$ be defined by the set of risk states $\mathcal{V}_{risk}$
    **return** $(\mathcal{V}_s \uplus \mathcal{V}_e, \mathcal{E}, Acc)$
```

CHAPTER 8

Conclusion

This thesis summarizes our existing efforts in bringing algorithmic game solving from theoretical results to concrete applications, and most of our results have been published on conferences in verification. Figure 8.1 gives a road map which summarizes our results in this thesis.

Our major focus relies on an implementation of solver libraries realizing a broad spectrum of games. These games, e.g., turn-based / concurrent / probabilistic / incomplete information games, or games on pushdown game graphs, can naturally relate to problems in synthesis.

To connect games to concrete applications, we adapt an application-driven approach, and propose three synthesis techniques.

- *Priority synthesis* is a technique used for orchestrating component-based systems. Given a set of Behavior-Interaction-Priority (BIP) components together with their possible interactions and a safety property, the synthesis algorithm, based on automata-based (game-theoretic) notions, restricts the set of possible interactions in order to rule out unsafe states.
- *Behavioral-level synthesis using PDDL* offers a bridge between research in verification and artificial intelligence. We perform a slight extension over the language such that researchers in verification (algorithmic games) can profit from the rich collection of examples coming from the AI community.
- For the third synthesis technique, we present an approach for *HW/SW level fault-tolerant synthesis* by combining predefined patterns for fault-tolerance with algorithmic game solving. A non-fault-tolerant system, together with the relevant fault hypothesis and fault-tolerant mechanism templates in a pool are translated into a distributed game, and we perform an incomplete search of strategies to cope with undecidability.

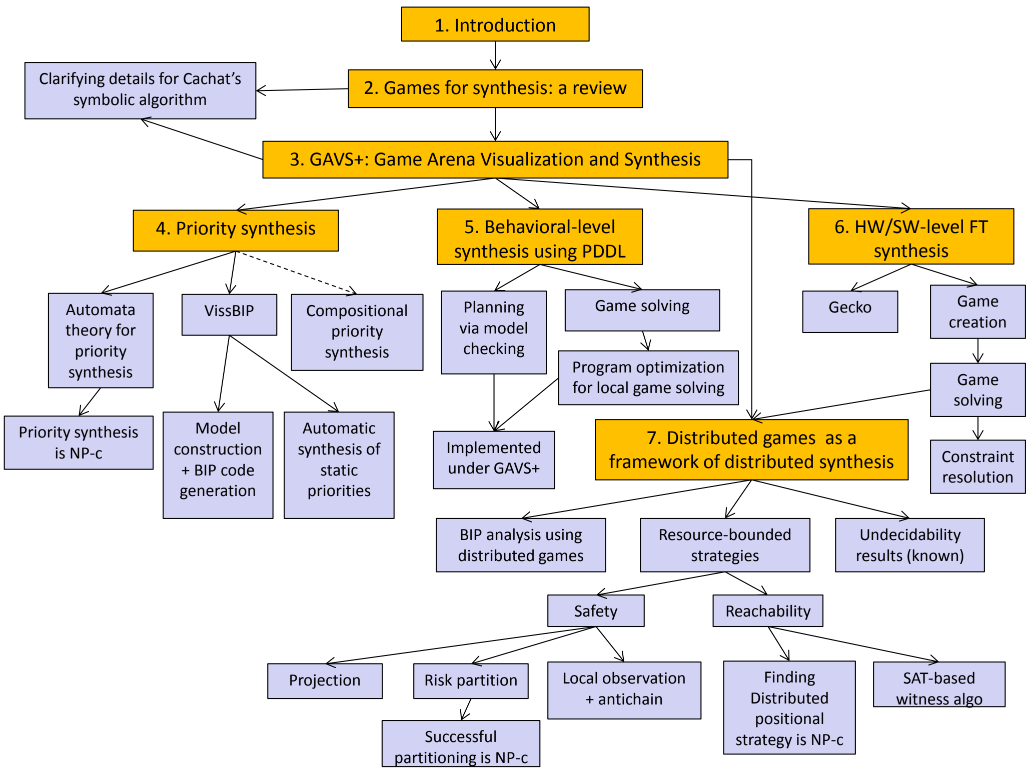

1. Introduction
2. Games for synthesis: a review
Clarifying details for Cachat's symbolic algorithm
3. GAVS+: Game Arena Visualization and Synthesis
4. Priority synthesis
5. Behavioral-level synthesis using PDDL
6. HW/SW-level FT synthesis
Automata theory for priority synthesis
VissBIP
Compositional priority synthesis
Planning via model checking
Game solving
Gecko
Game creation
Program optimization for local game solving
Game solving
Priority synthesis is NP-c
Model construction + BIP code generation
Automatic synthesis of static priorities
Implemented under GAVS+
7. Distributed games as a framework of distributed synthesis
Constraint resolution
BIP analysis using distributed games
Resource-bounded strategies
Undecidability results (known)
Safety
Reachability
Projection
Risk partition
Local observation + antichain
Finding Distributed positional strategy is NP-c
SAT-based witness algo
Successful partitioning is NP-c

Figure 8.1: A summary over results in this thesis and their relations.

For the above techniques, we have proposed algorithms and created academic tools (VISSBIP, GAVS+ for PDDL, and GECKO) to concretize our ideas.

By applying application-driven approaches, we found solving distributed games (an alternative formalism for distributed synthesis), although undecidable in general, important for concrete applications. We therefore elaborate on resource-bounded methods for reachability and safety games, including new algorithms and complexity bounds. As these resource-bounded methods reflect the design intention, our hope is to automate distributed synthesis to a level of assistance - what is considered as a potential fix by a designer can be found automatically using our methods. Most algorithms for solving distributed games with resource-bounded strategies have been implemented into GAVS+, our solver library.

Future Work

In the following, we summarize some further directions created by the work in the thesis; they are now under investigation by us and our collaborating researchers.

- Our initial result of priority synthesis has raised several interesting research problems, e.g., quality of priority synthesis or local starvation issues.

- For synthesis techniques to scale, an important factor is to exploit the structure (modularity) of the problem. Performing synthesis with compositional methods is an important research direction. Details include synthesizing priorities which maintain stronger locality or modularization (for the ease of implementation), or applying learning techniques in compositional verification [PGB$^+$08, CCF$^+$10] to achieve compositional synthesis.
- Concerning fault-tolerant synthesis and games, we consider reinvestigating timed games, expecting to design an open source implementation in our GAVS+ tool, preferably with new algorithms. By doing so, we hope to work on applications such as synthesizing monitors for control systems (e.g., model railway or flight control).

Bibliography

[AAE98] A. Arora, P.C. Attie, and E.A. Emerson. Synthesis of fault-tolerant concurrent programs. In *Proceedings of the 7th annual ACM symposium on Principles of distributed computing (PODC'98)*, pages 173–182. ACM, 1998.

[ABC$^+$06] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[ABK09] F. Abujarad, B. Bonakdarpour, and S.S. Kulkarni. Parallelizing deadlock resolution in symbolic synthesis of distributed programs. In *Proceedings of the 8th International Workshop on Parallel and Distributed Methods in Verification (PDMC'09)*, volume 14 of *EPTCS*, pages 92–106, 2009.

[AD94] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[AG11] K. R. Apt and E. Grädel. *Lectures in Game Theory for Computer Scientists*. Cambridge University Press, 2011.

[AINT07] M. Autili, P. Inverardi, A. Navarra, and M. Tivoli. SYNTHESIS: a tool for automatically assembling correct and distributed component-based systems. In *Proceedings of the 29th international conference on Software Engineering (ICSE'07)*, pages 784–787. IEEE Computer Society, 2007.

[AMN05] R. Alur, P. Madhusudan, and W. Nam. Symbolic computational techniques for solving games. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(2):118–128, 2005.

[AMS03] F.A. Aloul, I.L. Markov, and K.A. Sakallah. FORCE: a fast and easy-to-implement variable-ordering heuristic. In *Proceedings of the 13th ACM Great Lakes symposium on VLSI (GLSVLSI'03)*, pages 116–119. ACM, 2003.

[Ang87] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[apa] Commons Math: The Apache Commons Mathematics Library. `http://commons.apache.org/math/`.

[ATW06] C.S. Althoff, W. Thomas, and N. Wallmeier. Observations on determinization of Büchi automata. *Theoretical Computer Science*, 363(2):224–233, 2006.

[BBNS09] S. Bensalem, M. Bozga, T.H. Nguyen, and J. Sifakis. D-finder: A tool for compositional deadlock detection and verification. In *Proceedings of the 21st International Conference in Computer Aided Verification (CAV'09)*, volume 5643 of *LNCS*, pages 614–619. Springer, 2009.

[BBPS09] A. Basu, S. Bensalem, D. Peled, and J. Sifakis. Priority scheduling of distributed systems based on model checking. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV'09)*, volume 5643 of *LNCS*, pages 79–93. Springer, 2009.

[BBQ11] B. Bonakdarpour, M. Bozga, and J. Quilbeuf. Automated distributed implementation of component-based models with priorities. In *Proceedings of the 11th International conference on Embedded Software (EMSOFT'11)*, 2011. to appear.

[BBS06] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in BIP. In *Proceedings of the 4th IEEE International Conference on Software Engineering and Formal Methods (SEFM'06)*, pages 3–12. IEEE, 2006.

[BBSN08] S. Bensalem, M. Bozga, J. Sifakis, and T.H. Nguyen. Compositional verification for component-based systems and application. In *Proceedings of the 6th International Symposium in Automated Technology for Verification and Analysis (ATVA'08)*, volume 5311 of *LNCS*, pages 64–79. Springer-Verlag, 2008.

[BCD$^+$07a] G. Behrmann, A. Cougnard, A. David, E. Fleury, K.G. Larsen, and D. Lime. Uppaal-tiga: Time for playing games! In *Proceedings of the 19th international conference on Computer Aided Verification (CAV'07)*, volume 4590 of *LNCS*, pages 121–125. Springer-Verlag, 2007.

[BCD$^+$07b] G. Behrmann, A. Cougnard, A. David, E. Fleury, K.G. Larsen, and D. Lime. Uppaal-tiga: Time for playing games! In *Proceedings of the 19th international conference on Computer aided verification (CAV'07)*, volume 4590 of *LNCS*, pages 121–125. Springer-Verlag, 2007.

[BCDW$^+$09] D. Berwanger, K. Chatterjee, M. De Wulf, L. Doyen, and T. Henzinger. Alpaga: A tool for solving parity games with imperfect information. In *Proceedings on the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'10)*, volume 6015 of *LNCS*, pages 58–61. Springer-Verlag, 2009.

[BCFL04] P. Bouyer, F. Cassez, E. Fleury, and K. Larsen. Optimal strategies in priced timed game automata. In *Proceedings of the 24th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'04)*, volume 3328 of *LNCS*, pages 423–429. Springer-Verlag, 2004.

[BCP$^+$01] P. Bertoli, A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Mbp: a model based planner. In *IJCAI-2001 Workshop on Planning under Uncertainty and Incomplete Information (PRO-2)*, 2001.

[BDL04] G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT'04)*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[BF97] A.L. Blum and M.L. Furst. Fast planning through planning graph analysis. *Artificial intelligence*, 90(1-2):281–300, 1997.

[BFS00] C. Bernardeschi, A. Fantechi, and L. Simoncini. Formally verifying fault tolerant system designs. *The Computer Journal*, 43(3):191–205, 2000.

[BGL$^+$11] S. Bensalem, A. Griesmayer, A. Legay, T-H. Nguyen, J. Sifakis, and R-J. Yan. D-Finder 2: Towards Efficient Correctness of Incremental Design. In *Proceedings of the 3rd NASA Formal Methods Symposium (NFM'11)*, LNCS. Springer-Verlag, 2011.

[BHG$^+$93] S. Balemi, G.J. Hoffmann, P. Gyugyi, H. Wong-Toi, and G.F. Franklin. Supervisory control of a rapid thermal multiprocessor. *Automatic Control, IEEE Transactions on*, 38(7):1040–1059, 1993.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986.

[Buc08] C. Buckl. *Model-Based Development of Fault-Tolerant Real-Time Systems*. PhD thesis, Technische Universität München, Oct 2008.

[BWH$^+$03] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *Computer*, 36(4):45–52, april 2003.

[Cac02] T. Cachat. Symbolic strategy synthesis for games on pushdown graphs. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming (ICALP'02)*, volume 2382 of *LNCS*, pages 704–715. Springer-Verlag, 2002.

[Cac03a] T. Cachat. *Games on Pushdown Graphs and Extensions*. PhD thesis, RWTH Aachen, 2003.

[Cac03b] T. Cachat. Uniform solution of parity games on prefix-recognizable graphs. *Electronic Notes in Theoretical Computer Science*, 68(6):71–84, 2003.

[CBC$^+$11] C.-H. Cheng, S. Bensalem, Y.-F. Chen, R-J. Yan, B. Jobstmann, A. Knoll, C. Buckl, and H. Ruess. Algorithms for synthesizing priorities in component-based systems. In *Proceedings of the 9th International Symposium on Automated Technology for Verification and Analysis (ATVA'11)*, LNCS. Springer-Verlag, 2011.

[CBEK09] C.-H. Cheng, C. Buckl, J. Esparza, and A. Knoll. Modeling and verification for timing satisfaction of fault-tolerant systems with finiteness. In

Proceedings of the 13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications (DS-RT'09), pages 208–215. IEEE Computer Society, 2009.

[CBJ$^+$11] C.-H. Cheng, S. Bensalem, B. Jobstmann, R-J. Yan, A. Knoll, and H. Ruess. Model construction and priority synthesis for simple interaction systems. In *Proceedings of the 3rd NASA Formal Methods Symposium (NFM'11)*, volume 6617 of *LNCS*, pages 466–471. Springer-Verlag, 2011.

[CBK10] C.-H. Cheng, C. Buckl, and A. Knoll. Tool-based development for lightweight fault-tolerant systems (poster abstract). In *16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'10)*, April 2010.

[CBLK10] C.-H. Cheng, C. Buckl, M. Luttenberger, and A. Knoll. GAVS: Game arena visualization and synthesis. In *Proceedings of the 8th International Symposium on Automated Technology for Verification and Analysis (ATVA'10)*, volume 6252 of *LNCS*, pages 347–352. Springer-Verlag, 2010.

[CGR$^+$12a] C.-H. Cheng, M. Geisinger, H. Ruess, C. Buckl, and A. Knoll. Game solving for industrial automation and control. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'12)*, May 2012.

[CGR$^+$12b] C.-H. Cheng, M. Geisinger, H. Ruess, C. Buckl, and A. Knoll. MGSyn: Automatic synthesis for industrial automation. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV'12)*, LNCS. Springer-Verlag, 2012. to appear.

[CJBK11] C.-H. Cheng, B. Jobstmann, C. Buckl, and A. Knoll. On the hardness of priority synthesis. In *Proceedings of the 16th International Conference on Implementation and Application of Automata (CIAA'11)*, volume 6807 of *LNCS*. Springer-Verlag, 2011.

[CKLB11] C.-H. Cheng, A. Knoll, M. Luttenberger, and C. Buckl. GAVS+: An open framework for the research of algorithmic game solving. In *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'11)*, volume 6605 of *LNCS*, pages 258–261. Springer-Verlag, 2011.

[CCF$^+$10] Y.F. Chen, E. M. Clarke, A. Farzan, M.-H. Tsai, Y.-K. Tsay, and B.-Y. Wang. Automated assume-guarantee reasoning through implicit learning. In *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV'10)*, volume 6174 of *LNCS*, pages 511–526. Springer-Verlag, 2010.

[CCGR99] A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model verifier. In *Proceedings of the 11th Conference on Computer-Aided Verification (CAV'99)*, volume 1633 of *LNCS*, pages 495–499. Springer-Verlag, 1999.

[CBRZ01] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[CGP99] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT-Press, 1999.

[CGP03] J.M. Cobleigh, D. Giannakopoulou, and C.S. Păsăreanu. Learning assumptions for compositional verification. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *LNCS*, pages 331–346. Springer-Verlag, 2003.

[CHJR10] K. Chatterjee, T. Henzinger, B. Jobstmann, and A. Radhakrishna. GIST: A Solver for Probabilistic Games. In *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV'11)*, volume 6174 of *LNCS*, pages 665–669. Springer-Verlag, 2010.

[Con92] A. Condon. The complexity of stochastic games. *Information and Computation*, 96(2):203–224, 1992.

[Con93] A. Condon. On algorithms for simple stochastic games. *Advances in computational complexity theory*, 13:51–73, 1993.

[CRBK11] C.-H. Cheng, H. Ruess, C. Buckl, and A. Knoll. Synthesis of fault-tolerant embedded systems using games: from theory to practice. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'11)*, volume 6538 of *LNCS*, pages 118–133. Springer-Verlag, 2011.

[CUD] CUDD: CU decision diagram package. `http://vlsi.colorado.edu/~fabio/CUDD/`.

[DAHK07] L. De Alfaro, T.A. Henzinger, and O. Kupferman. Concurrent reachability games. *Theoretical Computer Science*, 386(3):188–217, 2007.

[DBBL07] R.I. Davis, A. Burns, R.J. Bril, and J.J. Lukkien. Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, 2007.

[DF09] R. Dimitrova and B. Finkbeiner. Synthesis of fault-tolerant distributed systems. In *Proceedings of the 7th International Symposium on Automated Technology for Verification and Analysis (ATVA'09)*, volume 5799 of *LNCS*, pages 321–336. Springer-Verlag, 2009.

[DHJP08] L. Doyen, T.A. Henzinger, B. Jobstmann, and T. Petrov. Interface theories with component reuse. In *Proceedings of the 8th ACM international conference on Embedded software (EMSOFT'08)*, pages 79–88. ACM, 2008.

[DWDHR06] M. De Wulf, L. Doyen, T. Henzinger, and J. Raskin. Antichains: A new algorithm for checking universality of finite automata. In *Computer Aided Verification (CAV'06)*, volume 4144 of *LNCS*, pages 17–30. Springer-Verlag, 2006.

[DWDR06] M. De Wulf, L. Doyen, and J.F. Raskin. A lattice theory for solving games of imperfect information. In *Proceedings of the 9th International Workshop in Hybrid Systems: Computation and Control (HSCC'06)*, volume 3927 of *LNCS*, pages 153–168. Springer-Verlag, 2006.

[EFW02] J. Eidson, M.C. Fischer, and J. White. IEEE 1588 standard for a precision clock synchronization protocol for networked measurement and control systems. In *34 th Annual Precise Time and Time Interval (PTTI) Meeting*, pages 243–254, 2002.

[Ehl11] R. Ehlers. Unbeast: Symbolic bounded synthesis. In *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'11)*, volume 6605 of *LNCS*, pages 272–275. Springer-Verlag, 2011.

[EHRS00] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00)*, number 1855 in LNCS, pages 232–247. Springer-Verlag, 2000.

[EJ91] E.A. Emerson and C.S. Jutla. Tree automata, mu-calculus and determinacy. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science (FOCS'91)*, pages 368–377. IEEE, 1991.

[EJL$^+$03] J. Eker, J.W. Janneck, E.A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[EK07] S. Edelkamp and P. Kissmann. Symbolic exploration for general game playing in PDDL. In *ICAPS-Workshop on Planning in Games*, 2007.

[emf] Eclipse Modeling Framework. `http://www.eclipse.org/modeling/emf/`.

[FKL08] D. Fisman, O. Kupferman, and Y. Lustig. On verifying fault tolerance of distributed protocols. In *Proceedings of the 14th international conference on Tools and algorithms for the construction and analysis of systems (TACAS'08)*, volume 4963 of *LNCS*, pages 315–331, Berlin, Heidelberg, 2008. Springer-Verlag.

[FL03] M. Fox and D. Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20(1):61–124, 2003.

[FL10] O. Friedmann and M. Lange. Local strategy improvement for parity game solving. In *Proceedings of the 1st International Symposium on Games, Automata, Logics and Formal Verification (GandALF'10)*, volume 25, pages 118–131. EPTCS, 2010.

[FS05] B. Finkbeiner and S. Schewe. Uniform distributed synthesis. In *Proceedings. 20th Annual IEEE Symposium on Logic in Computer Science (LICS'05)*, pages 321–330. IEEE, 2005.

[GAI$^+$98] M. Ghallab, C. Aeronautiques, C.K. Isi, S. Penberthy, D.E. Smith, Y. Sun, and D. Weld. PDDL-the planning domain definition language. Technical Report CVC TR-98003/DCS TR-1165, Yale Center for Computer Vision and Control, Oct 1998.

[GLZ04] P. Gastin, B. Lerman, and M. Zeitoun. Distributed games and distributed control for asynchronous systems. In *Proceedings of the 6th Latin American Symposium on Theoretical Informatics (LATIN'04)*, volume 2976 of *LNCS*, pages 455–465. Springer-Ver, 2004.

[GLZ05] P. Gastin, B. Lerman, and M. Zeitoun. Distributed games with causal memory are decidable for series-parallel systems. In *Proceedings of the 24th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'04)*, volume 3328 of *LNCS*, pages 275–286. Springer-Ve, 2005.

[GPQ10] S. Graf, D. Peled, and S. Quinton. Achieving distributed control through model checking. In *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV'10)*, volume 6174 of *LNCS*, pages 396–409. Springer-Verlag, 2010.

[GR09] A. Girault and É. Rutten. Automating the addition of fault folerance with discrete controller synthesis. *Formal Methods in System Design*, 35(2):190–225, 2009.

[GS03] G. Gößler and J. Sifakis. Priority systems. In *Proceedings of the 2nd International Symposium on Formal Methods for Components and Objects (FMCO'03)*, volume 3188 of *LNCS*, pages 314–329. Springer-Verlag, 2003.

[GTW02] E. Gradel, W. Thomas, and T. Wilke. *Automata, Logics, and Infinite Games*, volume 2500 of *LNCS*. Springer-Verlag, 2002.

[Han07] R. Hanmer. *Patterns for Fault Tolerant Software*. Wiley Software Patterns Series, 2007.

[Hay09] I.J. Hayes. Dynamically Detecting Faults via Integrity Constraints. In *Methods, Models and Tools for Fault Tolerance*, volume 5454 of *LNCS*, pages 85–103. Springer-Verlag, 2009.

[Hel06] M. Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26(1):191–246, 2006.

[HHK03] T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003.

[HK66] A.J. Hoffman and R.M. Karp. On nonterminating stochastic games. *Management Science*, 12(5):359–370, 1966.

[HO09] M. Hague and C. Ong. Winning regions of pushdown parity games: A saturation method. In *Proceedings of the 20th International Conference on Concurrency Theory (CONCUR'09)*, volume 5710 of *LNCS*, pages 384–398. Springer-Verlag, 2009.

[Hol04] G.J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2004.

[IMA02] M.V. Iordache, J. Moody, and P.J. Antsaklis. Synthesis of deadlock prevention supervisors using Petri nets. *Robotics and Automation, IEEE Transactions on*, 18(1):59–68, 2002.

[Jan07] D. Janin. On the (high) undecidability of distributed synthesis problems. In *Proceedings of the 33rd Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'07)*, volume 4362 of *LNCS*, pages 320–329. Springer-Verlag, 2007.

[Jav] JavaCC: The Java Parser Generator. `http://javacc.java.net/`.

[jdd] The JDD project. `http://javaddlib.sourceforge.net/jdd/`.

[JGB05] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05)*, volume 3576 of *LNCS*, pages 226–238. Springer-Verlag, 2005.

[jgr] JGraphX: Java Graph Drawing Component. `http://www.jgraph.com/jgraph.html`.

[JGWB07] B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem. Anzu: A tool for property synthesis. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07)*, volume 4590 of *LNCS*, pages 258–262. Springer-Verlag, 2007.

[KA00] S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. In *Proceedings of the 6th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'00)*, volume 1926 of *LNCS*, pages 339–359. Springer-Verlag, 2000.

[KE11] P. Kissmann and S. Edelkamp. Gamer, a general game playing agent. *KI-Künstliche Intelligenz*, 25(1):1–4, 2011.

[Koz83] D. Kozen. Results on the propositional [mu]-calculus. *Theoretical Computer Science*, 27(3):333–354, 1983.

[KS08] A.D. Kshemkalyani and M. Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press New York, NY, USA, 2008.

[KTV06] H. Kautz, W. Thomas, and M. Y. Vardi. 05241 executive summary – synthesis and planning. In Henry Kautz, Wolfgang Thomas, and Moshe Y. Vardi, editors, *Synthesis and Planning*, number 05241 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[LLV] The LLVM Compiler Infrastructure Project. `http://llvm.org/`.

[LM94] L. Lamport and S. Merz. Specifying and verifying fault-tolerant systems. In Hans Langmaack, Willem-Paul de Roever, and Jan Vytopil, editors, *Proceedings of the 3rd International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'94)*, volume 863 of *LNCS*, pages 41–76. Springer-Verlag, 1994.

[LZW08] Z.W. Li, M.C. Zhou, and N.Q. Wu. A survey and comparison of Petri net-based deadlock prevention policies for flexible manufacturing systems. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 38(2):173–188, 2008.

[MKB$^+$05] D. Marinov, S. Khurshid, S. Bugrara, L. Zhang, and M. Rinard. Optimizations for compiling declarative models into boolean formulas. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, volume 3569 of *LNCS*, pages 632–637. Springer-

Verlag, 2005.

[MPS95] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In *Proceedings of the 12th Annual Symposium on Theoretical Aspects of Computer Science (STACS'95)*, volume 900 of *LNCS*, pages 229–242. Springer-Verlag, 1995.

[MRL05] B. Marthi, S. Russell, and D. Latham. Writing stratagus-playing agents in concurrent alisp. In *Proceedings of the IJCAI-05 Workshop on Reasoning, Representation, and Learning in Computer Games*, 2005.

[MT02] P. Madhusudan and P. Thiagarajan. A decidable class of asynchronous distributed controllers. In *Proceedings of the 13th International Conference on Concurrency Theory (CONCUR'02)*, volume 2421 of *LNCS*, pages 445–472. Springer-Verlag, 2002.

[MW03] S. Mohalik and I. Walukiewicz. Distributed games. In *Proceedings of the 23rd International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'03)*, volume 2914 of *LNCS*, pages 338–351. Springer-Verlag, 2003.

[NRTV07] N. Nisan, T. Roughgarden, E. Tardos, and V.V. Vazirani. *Algorithmic Game Theory*. Cambridge University Press, 2007.

[NRZ11] D. Neider, R. Rabinovich, and M. Zimmermann. Solving muller games via safety games. Technical Report AIB-2011-14, RWTH Aachen, 2011.

[oaw] openArchitectureWare Project. `http://www.openarchitectureware.org/`.

[ORSvH95] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, 1995.

[PDD] PDDL4J: Front-end parser library for PDDL. `http://sourceforge.net/projects/pdd4j/`.

[PGB$^+$08] C. Pasareanu, D. Giannakopoulou, M. Bobaru, J. Cobleigh, and H. Barringer. Learning to divide and conquer: applying the l* algorithm to automate assume-guarantee reasoning. *Formal Methods in System Design*, 32:175–205, 2008.

[PPE$^+$08] T. Pop, P. Pop, P. Eles, Z. Peng, and A. Andrei. Timing analysis of the flexray communication protocol. *Real-Time Systems*, 39(1):205–235, 2008.

[PPS06] N. Piterman, A. Pnueli, and Y. Saar. Synthesis of reactive (1) designs. In *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'06)*, volume 3855 of *LNCS*, pages 364–380. Springer-Verlag, 2006.

[PR89] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'89)*, pages 179–190. ACM, 1989.

[PR90] A. Pneuli and R. Rosner. Distributed reactive systems are hard to synthesize. In *Proceedings of the 31st Annual Symposium on Foundations of Com-*

puter Science (FOCS'90), volume 0, pages 746–757 vol.2. IEEE Computer Society, 1990.

[Rei84] J.H. Reif. The complexity of two-player games of incomplete information. *Journal of computer and system sciences*, 29(2):274–301, 1984.

[RW89] P.J.G. Ramadge and W.M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.

[Sha53] L.S. Shapley. Stochastic games. In *Proceedings of the National Academy of Sciences of the United States of America*, volume 39, page 1095. National Academy of Sciences, 1953.

[SLRBE05] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu. Programming by sketching for bit-streaming programs. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI'05)*, pages 281–294. ACM, 2005.

[Sys] Open SystemC Initiative (OSCI). `http://www.systemc.org/`.

[TBW95] K. Tindell, A. Burns, and A.J. Wellings. Calculating controller area network (CAN) message response times. *Control Engineering Practice*, 3(8):1163–1169, 1995.

[TCT$^+$08] Y.K. Tsay, Y.F. Chen, M.H. Tsai, W.C. Chan, and C.J. Luo. Goal extended: Towards a research tool for omega automata and temporal logic. In *Proceedings of 14th international conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, volume 4963 of *LNCS*, pages 346–350. Springer-Verlag, 2008.

[THY93] S. Tani, K. Hamaguchi, and S. Yajima. The complexity of the optimal variable ordering problems of shared binary decision diagrams. In *Algorithms and Computation*, volume 762 of *LNCS*, pages 389–398. Springer-Verlag, 1993.

[VJ00] J. Vöge and M. Jurdziński. A discrete strategy improvement algorithm for solving parity games. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *LNCS*, pages 202–215. Springer-Verlag, 2000.

[Wal96] I. Walukiewicz. Pushdown processes: Games and model checking. In *Proceedings of the 8th International Conference on Computer Aided Verification (CAV'96)*, volume 1102 of *LNCS*, pages 62–74. Springer-Verlag, 1996.

[WW89] C.C. White and D.J. White. Markov decision processes. *European Journal of Operational Research*, 39(1):1–16, 1989.