Tesi di Dottorato in Informatica

Synthesis of Recursive State Machines from Libraries of Game Modules

Dipartimento di Scienze Matematiche, Fisiche e Naturali

Università degli Studi di Salerno

Candidata
Ilaria De Crescenzo

Tutor
prof. Salvatore La Torre

Coordinatore
prof. Giuseppe Persiano

Dottorato di Ricerca in Informatica XIII Ciclo - Nuova serie
Anno Accademico 2015-2016

Abstract

This thesis is focused on synthesis. In formal verification synthesis can be referred to the controller synthesis and the system synthesis. This work combines both this area of research.

First we focus on synthesizing modular controllers considering game on recursive game graph with the requirement that the strategy for the protagonist must be modular. A recursive game graph is composed of a set of modules, whose vertices can be standard vertices or can correspond to invocations of other modules and the standard and the set of vertices is split into two sets each controlled by one of the players. A strategy is modular if it is local to a module and is oblivious to previous module invocations, and thus does not depend on the context of invocation. We study for the first time modular strategies with respect to winning conditions that can be expressed languages of pushdown automata. We show that pushdown modular games are undecidable in general, and become decidable for visibly pushdown automata specifications. We carefully characterize the computational complexity of the considered decision problem. In particular, we show that modular games with a universal Büchi or co-Büchi visibly pushdown winning condition are EXPTIME-complete, and when the winning condition is given as a CARET or NWTL temporal logic formula the problem is 2EXPTIME-complete, and it remains 2EXPTIME-hard even for simple fragments of these logics. As a further contribution, we present a different synthesis algorithm that runs faster than known solutions for large specifications and many exits.

In the second part of this thesis, we introduce and solve a new component-based synthesis problem that subsumes the synthesis from libraries of recursive components introduced by Lustig and Vardi with the modular synthesis

introduced by Alur et al. for recursive game graphs. We model the components of our libraries as game modules of a recursive game graph with unmapped boxes, and consider as correctness specification a target set of vertices. To solve this problem, we give an exponential-time fixed-point algorithm that computes annotations for the vertices of the library components by exploring them backwards. We show a matching lower-bound via a direct reduction from linear-space alternating Turing machines, thus proving EXPTIME-completeness. We also give a second algorithm that solves this problem by annotating in a table the result of many local reachability game queries on each game component. This algorithm is exponential only in the number of the exits of the game components, and thus shows that the problem is fixed-parameter tractable.

Finally, we study a more general synthesis problem for component-based pushdown systems, the *modular synthesis from a library of components* (LMS). We model each component as a game graph with *boxes* as placeholders for *calls* to components, as in the previous model, but now the library is equipped also with a *box-to-component map* that is a partial function from boxes to components. An instance of a component C is essentially a copy of C along with a local strategy that resolves the nondeterminism of pl_0. An RSM S synthesized from a library is a set of instances along with a total function that maps each box in S to an instance of S and is consistent with the box-to-component map of the library. We give a solution to the LMS problem with winning conditions given as internal reachability objectives, or as external deterministic finite automata (FA) and deterministic visibly pushdown automata (VPA) (6). We show that the LMS problem is EXPTIME-complete for any of the considered specifications.

To my mother...

Acknowledgements

Looking an empty page after a long journey, sometimes it is difficult to turn back and find the right words to share feelings, but in this page I will try to express my gratitude to the persons that have shared this path with me and supported me thousand times.

I would like to thank my supervisor Salvatore La Torre for providing his experience, knowledge and guidance during these years, for his patience throughout our work together and for believing in me, even when I did not believe in myself.

Completing this PhD has been a difficult task and often I have relied greatly on my family and my friends. In particular I want to thank my husband that has been incredibly patient during my "no-moments", and my father, that always pushes me to do my best.

I would also like to thank the researchers who have provided many comments and suggestions about my work. A special thank goes to Jaron Velner for his contributions in this research.

The last but most important thought is for my mother. Everything I did, I do and I will do... it is always for you...

Contents

List of Figures **vii**

Glossary **ix**

1 Introduction **1**

2 Automata **11**

2.1 Preliminaries 12

2.1.1 Notation 12

2.1.2 Words and alphabets 12

2.1.3 Trees 13

2.2 Automata 14

2.2.1 Finite state automata and ω -automata 14

2.2.2 Pushdown automata 16

2.2.3 Visibly pushdown automata 17

2.2.4 Tree automata 18

3 Games on pushdown systems **21**

3.1 Games on graphs 22

3.2 Winning conditions as formulas of temporal logics 24

3.2.1 Linear Temporal Logic 24

3.2.2 Temporal Logic of Calls and Returns 25

3.2.3 Nested Word Temporal Logic 27

3.3 Algorithmic problems related to games 29

3.4 Pushdown games with regular winning conditions 30

3.5 Visibly pushdown games 33

4 Modular strategies **37**
4.1 Recursive game graph . . . 38
4.2 Modular strategies . . . 40
4.3 Winning conditions and modular games . . . 41
4.4 Solving modular games with reachability winning conditions . . . 41
4.5 Solving modular games with regular winning conditions . . . 44

5 Visibly modular pushdown games **49**
5.1 Contribution . . . 49
5.2 Pushdown modular games . . . 50
5.3 Solving modular games with VPA specifications . . . 52
5.4 Improved tree automaton construction . . . 55
5.4.1 General structure of the construction . . . 55
5.4.2 Strategy trees and the automaton $\mathcal{A}_G$ . . . 56
5.4.3 The automaton $\mathcal{A}_{\mathcal{B},\mathcal{C},CG}$ . . . 59
5.4.4 Checking $\mathcal{B}$ acceptance on strategy-tree plays with infinitely many unreturned calls . . . 69
5.4.5 Reducing modular synthesis to emptiness of tree automata . . . 70
5.5 Temporal logic winning conditions . . . 71
5.5.1 Solving modular CaRet and Nwtl games . . . 71
5.5.2 Path formulas . . . 71
5.5.3 Modular synthesis in simple fragments of Ltl . . . 72

6 Synthesis from libraries **79**
6.1 Synthesis from libraries of transducers . . . 80
6.1.1 Synthesis from Components Libraries . . . 81
6.1.2 Synthesis from Recursive Components Libraries . . . 83
6.2 Comparing synthesis of modular strategies with synthesis from libraries 85

7 Component-based synthesis of open systems **87**
7.1 Contribution . . . 87
7.2 From recursive components to open recursive components . . . 88
7.3 A General Modular Synthesis from Libraries . . . 90
7.3.1 Library of open components . . . 90

7.3.2 Instances and recursive state machines 92
7.3.3 A general synthesis problem 94
7.4 Other formulations of the modular synthesis 95

8 Modular synthesis with reachability conditions 97
8.1 Contribution 97
8.2 A simpler modular synthesis problem 99
8.3 Solving our modular synthesis problem 103
8.4 Computational complexity analysis 106
8.5 Solving Lms and component-based Lms problems with reachability winning conditions 110

9 Modular synthesis with other winning conditions 115
9.1 Contribution 115
9.2 Safety Lms 116
9.2.1 Overview of the construction. 116
9.2.2 Component and library trees. 117
9.2.3 The construction of $\mathcal{A}^{C}_{\mathcal{P},\mathcal{B}}$. 119
9.2.4 The construction of $\mathcal{A}$. 122
9.3 Lms with deterministic VPA specification 124
9.4 On component-based Lms problems 127

10 Discussion and Conclusion 129
10.1 On visibly modular pushdown games 129
10.2 On modular synthesis 130
10.2.1 Modular synthesis and synthesis from recursive-component libraries 131
10.2.2 Modular synthesis and program repair 132
10.2.3 Modular vs. global synthesis 133
10.3 Future research 136

References 137

List of Figures

3.1 Syntax and semantics of LTL 25
3.2 Syntax and semantics of CARET. 26
3.3 Syntax and semantics of NWTL 28

4.1 A sample RGG. 38
4.2 A fragment of a strategy tree T_{smpl}. 46

5.1 The module m_{in} and m 51
5.2 The module d_m. 53
5.3 A fragment of a strategy tree T_{smpl}. 57
5.4 The automaton $\mathcal{B}_{smpl}$. 60

7.1 A library (a) and RSMs from it: unrestricted (b), same local strategy for instances of the same component (c), and at most one instance for each component (d). 92

8.1 An example of modular synthesis 102
8.2 Graphical representation of the game components C_{main}, $C_{\forall}$ and C_{fin} . 107

9.1 Top fragments of (a) the component tree of C_1 and (b) the library tree from our running example. 116
9.2 The component C_{stack_i} for $i \in [n]$ 126

10.1 A faulty program (a) and a pre-existing function (b). 132
10.2 A sample library. 134

Glossary

CaRet	Call-Return Temporal Logic
FA	Finite-state Automaton
LMS	Modular synthesis from Libraries
LGS	Global modular synthesis from Libraries
LTL	Linear Temporal Logic
MVPG	Modular Visibly Pushdown Game
NFA	Nondeterministic finite-state Automaton
NWTL	Nested Word Temporal Logic
PDG	Standard Pushdown Game
RGG	Recursive Game Graph
RSM	Recursive State Machine
VPA	Visibly Pushdown Automaton
VPG	Visibly Pushdown Game
VPS	Visibly Pushdown System
VPRG	Visibly Pushdown Game on an Recursive Game Graph
ω-FA	ω-automaton

1

Introduction

In formal and automatic verification, the system synthesis is one of the most relevant areas of research. From its first definition, posed by Church in (14) to express the problem of synthesizing digital circuits from specifications written in a restricted logic of arithmetic, the synthesis problem has always attracted the attention of the researchers, because it is related to the long-term dream for computer scientists of realizing the automatic development of programs that are correct (i.e. that satisfy a given specification) by construction.

In spite of the rich theory developed for synthesis in the last two decades, little of this theory has been used in practice.

The main problem concerns the fact that in general the synthesis process can not be automatized and synthesis problems have usually very high complexities, that are critically determined by the size of the specification (43).

Another problem is that typically the synthesized systems are monolitic and the classical synthesis algorithms create flat system with subroutines or subsystems that may be repeated many times.

The natural structure of a complex hardware or software system in real-life is recursive in such way that the components are defined only once and then can be invoked many times. Complex systems are in fact usually composed by relatively simple modules that interacts with each other using procedure calls. For this reason, in the last years the research has focused its attention on component-based approaches to synthesis (see (1, 30, 37) for a sample of such research).

There is another remarkable observation that can be done about component-based synthesis. In real practice few programs are built by scratch and often the programmers develop complex system starting from a set of templates or of pre-existing reusable function, typically contained in a *library*.

Component-based design plays a key role in configurable and scalable development of efficient hardware as well as software systems. For example, it is current practice to design specialized hardware using some base components that are more complex than universal gates at bit-level, and programming by using library features and frameworks. Moreover sometimes the employment of standard preexisting components is unavoidable.

A *component* can be seen as a piece of hardware or software that can be directly plugged into a solution or a template that needs to be customized for a specific use. In the procedural-programming world, a general notion of component composition can be obtained by allowing to synthesize some modules from generic templates and then connect them along with other off-the-shelf modules via the call-return paradigm (*synthesis from libraries(28)*). In such synthesis from library, the main aim is to usually resolve the external game, finding a composition of the system such that it satisfies the given specification. However the constructed system is typically a close system that does not interact with the environment.

With the evolution of computer science, systems have become more and more complex and nowadays the large number of hardware and software systems as procedural and object-oriented programs, distributed systems, communication protocols and web services, acts not only with a recursive behaviour, but also with a *reactive* behaviour:"reactive" means that the considered system works in an open setting, where the system interacts with an external environment that introduces an uncontrollable nondeterminism. An execution is the product of the interaction between these two entities in opposition and in general the role of the system itself is to maintain the ongoing interaction with the environment. Therefore, an essential aim for such systems is to synthesize a controller which supplies input to the system such that the system is correct whatever will be the behaviour of the external environment. This problem is named *controller synthesis problem*.

In automata theory, the standard controller synthesis problem is studied on games on graphs that can serve as a general model for reactive system. A game graph is a

direct graph where the set of vertices is partitioned between positions that belong to the system (the protagonist) and positions that belong to environment (the adversary). A play is a possible execution on such graph and it is a sequence of vertices: it starts from an initial vertex and each round, if the play is in a vertex, the player that owns such vertex must choose the next move among those that are possible. A winning condition allows to define the goal of the game and to decide when a play is considered winning for a player.

Synthesizing a controller corresponds to computing winning strategies in two-player games. In general, a strategy is a function that associates a next move to each prefix of a play that ends in a vertex of the protagonist (*global history*) and a strategy is a winning strategy if each play obtained according to the strategy is winning for the protagonist.

This standard formulation of the controller synthesis problem lacks of a compositional point of view, regarding both the chosen model and the kind of considered strategy. Finite state game graph are not adequate for the analysis of component-based open systems and a better choice is to consider recursive game graphs as model.

A recursive game graph (RGG)(5) corresponds to a pushdown games but the emphasis is on the *modules* composing the system. An RGG is composed by a set of graph named *modules*. In a module the vertices can be ordinary state or represent call to other module of the RGG. As usual, the game is modelled splitting the vertices into two sets, each controlled by one of the players.

The compositional structure of RGG has inspired the definition of a component-based strategy, named *modular strategy*, introduced formally in (5). A modular strategy is formed of a set of strategies, one for each RGG module, that are *local* to a module and *oblivious* of the history of previous module activations, i.e., the next move in such strategies is determined by looking only at the local memory of the current module activation (by contrast, if one allows the local memory to *persist* across module activations, deciding these games becomes undecidable already with reachability specifications (5)). Note that in modular games it is required only that the strategy of one player must be modular and no restrictions are placed on the strategy of the other player: this choice is reasonable because a set of synthesized modular controllers must operate with no assumptions on the uncontrollable nondeterminism of the system. Moreover each

modular controller makes the related module correct and independent by the context where it is invoked and such trait matches perfectly the component-based approach.

If we consider recursive and reactive systems in component-based synthesis a natural question arises: what happens if we want to synthesize an *open system* from a library of *open components*, which adds also the resolution of the internal game to overcome the nondeterminism generated by the behaviour of the external environment?

The work presented in this thesis goes in the direction of providing a new framework for component-based synthesis that requests the composition of elements obtained from library of open components and the modularity of the solutions.

The first step in this direction requires to define a new model. On the one hand, the standard library models proposed as in (28) are not enough expressive to handle appropriately the internal game. On the other, in the modular synthesis of recursive game graphs the call-return structure is given and cannot be modified. Our model combines and subsumes the best features of both such approaches.

The game modules for our component-based synthesis are taken from a finite set (*library*) of *game components*. We model each component as a game graph with vertices split between player 0 (pl_0) and player 1 (pl_1), and the addition of *boxes* as place-holders for *calls* to components. The library can be equipped with a *box-to-component map* that is a partial function from boxes to components (in (16) we do not consider such partial mapping). An instance of a component C is essentially a copy of C along with a local strategy that resolves the nondeterminism of pl_0. A recursive state machine (2, 12) (in short RSM) S synthesized from a library is a set of instances along with a total function that maps each box in S to an instance of S and is consistent with the box-to-component map of the library.

In this thesis, we formalize the modular synthesis from library (in short Lms) problem and we prove that such problem is decidable if we consider winning conditions given as internal reachability objectives, or as external deterministic finite automata (FA) and deterministic visibly pushdown automata (VPA) (6). We show that the Lms problem is Exptime-complete for any of the considered specifications. In particular, for reachability we first present the algorithm that solves a simpler case of the Lms problem where the boxes of the components are all un-mapped (i.e., the library has no box-to-component map) and then we modify it to the general setting. For safety and VPA specifications, the lower bounds can be obtained by standard reductions from

alternating linear-space Turing machines. The upper bound for safety specifications is based on a reduction to tree automata emptiness that is based on the notion of *library tree*: an infinite tree that encodes the library along with a choice for a total box-to-component map where both the components and the total map are unrolled. The construction is structured into several pieces and exploits the closure properties of tree automata under concatenation, intersection and union. The upper bound for VPA specifications is obtained by a reduction to safety specifications that exploits the synchronization between the stacks of the VPA and the synthesized RSM.

A solution to the LMS problem can involve arbitrarily many instances of each library component with possibly different local strategies. Such a diversity in the system design is often not affordable or unrealistic, therefore we also consider restrictions of this problem by focusing on solutions with few component instances and designs. In our setting, a natural way to achieve this is by restricting the synthesized RSMs such that: 1) at most one instance of each library component is allowed (few component instances), or 2) all the instances of a same library component must be controlled by a same local strategy (few designs). We refer to the LMS problems with these restrictions as the *single-instance* LMS problem and the *component-based* LMS problem, respectively. Note that in the *component-based* LMS there is no restriction imposed on the local strategy to be synthesized for a component and two instances of the same component can still differ in the mapping of the boxes.

For the component-based LMS problem we get the same complexity as for the general LMS problem: the upper bounds are obtained by adapting the constructions given for the general case.

The single-instance LMS problem can be reduced to the modular synthesis on recursive game graphs by guessing a total box-to-component map for the library, and thus we immediately get that the problem is NP-complete for reachability (5), and EXPTIME-complete for FA (4) and VPA specifications. Changing the point of view, the synthesis of modular controllers becomes a specific case of the LMS problem, where the box-to-component maps is total and each component can be instantiated only once. Therefore, we also extend the previous results on modular synthesis considering several classes of non-regular specifications, expressed as pushdown automata. We show that in the general case, i.e., by allowing any pushdown automaton as a specification, the modular synthesis problem is undecidable. For this, we give a reduction from the problem

of checking the emptiness of the intersection of deterministic context-free languages. We thus focus on visibly pushdown automata (VPA) (7) specifications with Büchi or co-Büchi acceptance and we show that the corresponding problems are decidable.

The LMS problem also gives a general framework for program repair where besides the intra-module repairs considered in the standard approach (see (22, 23)) one can think of repairing a program by replacing a call to a module with a call to another module (*function call repairs*). This relation will be discussed in the last chapter of this thesis.

Relatex work. The contributions and result presented in this thesis have been published in (16, 17, 18). A preliminary work of (17) can be found in (15).

The problem of deciding the existence of a modular strategy in a recursive game graph has introduced in (5) and it has been already studied with respect to ω-regular specifications. The problem is known to be NP-complete for reachability specifications (5), EXPTIME-complete for specifications given as deterministic and universal Büchi or Co-Büchi automata, and 2EXPTIME-complete for LTL specifications (4).

We want to recall that, if we consider pushdown games with global winning strategy according to reachability specifications or parity conditions, the decision problem is known to be EXPTIME-complete (46). This result holds also for other regular winning conditions, as sample given by a Büchi / Rabin / Muller automaton and, in general, for each specification that can be translated into a parity condition (clearly the complexity of the reduction must be considered in the computation of the overall complexity).

Beside the already mentioned papers on the modular controller synthesis, the notion of modular strategy is also of independent interest and has recently found application in other contexts, such as, the automatic transformation of programs for ensuring security policies in privilege-aware operating systems (20).

Component-based synthesis of software is the subject of several papers in the last years. In (28, 29) is described the component-based synthesis problem based on libraries of components. In (28) the components are modelled with finite-state transducers, and the correctness specification is given as an LTL formula. The same synthesis problem with specification is given as a temporal logic formula over nested words and components modelled as transducers with call-return structures is addressed in (29). In (11), this problem is formulated for synthesizing hierarchical systems bottom-up with respect

to a different μ-calculus specification for each component in the hierarchy. All these synthesis problem turn out to be 2EXPTIME-complete. The synthesis from libraries of components with simple specifications has been also implemented in tools: an example is presented in (21) where it is described an oracle-guided learning from examples and constraint-based synthesis from components via SMT solvers are combined to achieve the automatic synthesis of loop-free programs.

The synthesis problem presented in (28) can be rephrased in terms of game modules: given a library of component and an LTL formula, we can construct a game graph and considering the same specification we obtain a modular game such that there exists a winning modular strategy in this modular game if and only if there exists a composition that fulfils the LTL formula.

In addition to the previously cited works on library of components, we also want to recall some examples of researches on component-based synthesis: the component-based construction in (39), the work (37), where modules are expressed as terms of the λY-calculus, the interface-based design (1) and the development of web services (38). Moreover, component-based synthesis has been implemented and incorporated (in simple cases) in tools. As a sample research we cite: (31) where code-fragments are automatically generated from simple queries describing the desired input and output using as components a set of API methods; and (24), where an AI planner is incorporated in a compiler to automatically generate a sequence of library calls for an abstract algorithm given by the user.

We want also to recall the synthesis problem from (30) that deals with the automatic development of a program composed by a bounded number of functions. This framework differs from our setting in that programs and not transition systems are dealt with, and the number of functions of a synthesized program is bounded a priory but no structure of the functions is given.

Organizzation of the thesis. This thesis is structured as follows.

In Chapter 2, we introduce the basic notions on automata, pushdown automata and tree automata.

In Chapter 3, we focus on game on graph, first presenting the " flat" games and then focusing on pushdown games (standard pushdown games and visibly pushdown games). For these games we recall some of the result about their decidability according

to regular and non-regular specifications. Most of these results will be used to prove the lower bounds of the problems introduced in the subsequent chapters.

In Chapter 4, we present the formal definition of RGG. We recall the notion of modular strategies introduced by Alur and al. and briefly the approaches to solve modular game problems. Some of such approaches will be extended and adapted into new algorithms. This chapter summarizes the state of art of the synthesis of modular controllers.

In Chapter 5 we complete the framework depicted by the previous works, introducing and solving modular games according to winning conditions that express non-regular requirements. We consider also simpler specifications, considering fragment of logic and analysing the decidability and the complexity of the proposed problems. Moreover, we give a quite accurate picture of the modular game problems, refining the complexity of the previous results to expose the critical and unavoidable issues that such setting determines.

In Chapter 6 we move our attention from synthesis of modular controllers to synthesis from libraries. The first section of this chapter deals with the works of Vardi and al. that are focused on the synthesis from libraries of plain or recursive transducers. We present the model and recall the complexity results of the related synthesis problems. Then, we analyse the connection between modular synthesis and synthesis of components.

In Chapter 7 we first introduce informally the ideas of the new model which combines modular synthesis and synthesis from libraries of components. We point out the features that we expect to realize through such new model proposed for the synthesis from library of open components. We give the formal definitions of a library of game components and composition from library. Therefore we introduce a general definition for the modular synthesis from library problem (Lms), that asks to decide if there is a system synthesized from the library such that there exists a winning modular strategy for the protagonist according to a given winning condition. We also introduce some restrictions (single-instance and component-based Lms problems) that are natural in this setting and we expose the connections between single-instance Lms problems and modular controller synthesis.

In Chapter 8 we solve a simpler modular synthesis problem, that starts from a library that is an evolution of the RGG and considers as correctness specification only

reachability objectives. We solve the proposed problem presenting a fix-point algorithm that decides the considered problem in exponential time. Moreover, we show that the computational complexity becomes PTIME when the number of exits of the input model is fixed. Then, we modify the proposed algorithm to solve the general Lms and component-based Lms reachability problems. Both problems turn out to be Exptime-complete.

In Chapter 9 we consider more complex winning condition, given as regular and non-regular languages. We prove that the algorithms that solve such problems have the same complexity of the corresponding solution for synthesis of modular controllers or synthesis from library of transducers. We also extended these results to the related component-based Lms problems.

The Lms problems has many connections with different well-studied problems. We discuss about these connections in Chapter 10. Moreover, we propose some ideas about the future directions of this research.

2

Automata

The formal verification requires to define three essential elements. The first of them is the mathematical model that is used to model the behaviour of the system that we want to study and verify. The second necessary element is the model to express the correctness requirements that our systems must fulfil. The third element consists in the algorithmic procedures that, given a modelled system and a given correctness requirement, decides if the input system fulfils the specification.

The automata theory represents an unifying framework for the formal verification. On the one hand, automata can be used as models and their study allows to develop methods that can describe and analyse the dynamic behaviour of discrete systems. On the other hand, automata are closely related to formal language theory, whose constructs are typically used to express the specifications. An automaton can be seen as a finite representation of a formal language that may be an infinite set and automata are often classified by the class of formal languages they can recognize.

In this chapter, we introduce the definitions of some basic automata. Such automata will be consider in the rest of this work to express specifications and, in few cases, as models. Moreover, we recall some fundamental results that will be used to prove the correctness of some of our automata-theoretic solutions .

2.1 Preliminaries

2.1.1 Notation

Given two positive integers i and j, $i \leq j$, we denote with $[i, j]$ the set of integers k with $i \leq k \leq j$, and with $[j]$ the set $[1, j]$.

2.1.2 Words and alphabets

Let Σ be a finite alphabet of symbols.

A *finite word* w over Σ is a finite sequence of elements of Σ. With Σ^* we denote the set of finite words on Σ. With Σ^+ we denote the set of finite nonempty words over Σ.

The *length* of a word w is denoted by $|w|$ and is the number of symbol composing the word, i.e. if $w = \sigma_1 ... \sigma_n$ then $|w| = n$.

The *empty word*, denoted by ϵ, is the string that consists of no symbols. The length of the empty word is consequently zero.

An *infinite word*, named also ω-word, w over Σ intuitively is an infinite sequence of elements of Σ. With Σ^ω we denote the set of the infinite words on Σ.

For an infinite sequence $\pi = \sigma_0, \sigma_1, ...$ from Σ^ω the *infinite set* $Inf(\pi)$ is the set of elements that repeat infinitely often, i.e. $Inf(\pi) = \{\sigma \in \Sigma |$ there exist infinitely many i such that $\sigma_i = \sigma\}$.

A *nested ω-word* is a tuple $\bar{w} = (w, \mu, call, ret)$ where $w \in \Sigma^\omega$ is an $\omega - word$, $(\mu, call, ret)$ is a matching on $\mathbb{N}$. For a nested word, a *matching* on $\mathbb{N}$ is formed by a binary relation μ and two unary relation *call* annd *ret* satisfying the following:

- (1) if $\mu(i, j)$ holds then $call(i)$ and $ret(j)$ and $i < j$;
- (2) if $\mu(i, j)$ and $\mu(i, j')$ hold then $j = j'$ and if $\mu(i, j)$ and $\mu(i', j)$ hold, then $i = i'$;
- (3) if $i \leq j$ and $call(i)$ and $ret(j)$ then there exists $i \leq k \leq j$ such that either $\mu(i, k)$ or $\mu(k, j)$.

We say that a position i in a nested word $\bar{w}$ is a *call position* if $call(i)$ holds; a *return position* if $ret(i)$ holds; and an internal position if it is neither a call nor a return. If $\mu(i, j)$ holds, we say that i is the *matching call* of j, and j is the *matching return* of

i Calls without matching returns are *unmatched calls* (or pending call), and returns without matching calls are *unmatched returns* (or pending return). A nested word is said to be well-matched if no calls or returns are pending. Note that for well-matched nested words, the unary predicates *call* and *ret* are uniquely specified by the relation μ.

2.1.3 Trees

Graphs and trees. A graph is a pair $G = (V, E)$ where V is a set of elements named vertices and E is a pairs of vertices, named edges. If a graph G is a directed graph if $(v_1, v_2) \in E$ then the edge is directed from v_1 to v_2.

A *tree* T is a connected graph with no cycles and a *forest* is a set of graphs with no cycle, i.e. it is a disjoint union of one or more trees.

Labeled k-trees. Let $k \in \mathbb{N}$ and Ω be a finite alphabet. A Ω-*labeled* k-*tree* T is a pair $([k]^*, \nu)$ where the set $[k]^*$ denotes the vertices and $\nu : [k]^* \to \Omega$ is a labeling function, that labels every vertex of the tree with a letter in Ω. To distinguish the vertices of a tree, we named them *tree vertices*. The symbol ϵ (denoting as usual the empty word) is the root and for each tree vertex $x \in [k]^*$, the tree vertex $x.i$ is the i^{th} child of x. We denote with $\mathcal{T}_{k,\Omega}$ the set of all the possible finite Ω-labeled k-trees and with $\mathcal{T}^{\omega}_{k,\Omega}$ the set of all the possible infinite Ω-labeled k-trees and with $\mathcal{T}^{\infty}_{k,\Omega} = \mathcal{T}_{k,\Omega} \cup \mathcal{T}^{\omega}_{k,\Omega}$ the set of all possible Ω-labeled k-trees.

Domain, frontier and concatenation of trees. We named *domain* of a tree T the non-empty set $dom(t) \subseteq \{1, ..., k\}^*$ that satisfies the rule that for $w \in dom(T)$ and $j \in [k]$, $w.j \in dom(T)$ if exists $i < j$ such that $w.i \in dom(T)$.

The *frontier* of a tree T is the set of word $frt(T) = \{w \in dom(T) |\ \nexists i \text{ such that } w.i \in dom(T)\}$.

A tree concatenation is an operation that allows to "glue"other trees to a finite tree, substituting the nodes of the frontier with the given trees. Let $\mathcal{T} \subseteq \mathcal{T}_{k,\Omega}$ $\mathcal{T}' \subseteq \mathcal{T}^{\omega}_{k,\Omega}$ and $c \in \Omega$. Then the *tree concatenation* $\mathcal{T}._c\mathcal{T}'$ contains all the trees which result from some $T \in \mathcal{T}$ by replacing each occurrence of c on $frt(T)$ by a tree from $\mathcal{T}'$. Different trees are admitted only for different occurrences of c.

Instead of a single symbol, we can have a tuple of concatenation symbols $c = \{c_1, ..., c_n\}$. For $\mathcal{T}, \mathcal{T}_1, ..., \mathcal{T}_n \subseteq \mathcal{T}_{k,\Omega}$ let $\mathcal{T}.{}_c\mathcal{T}_1, ..., \mathcal{T}_n$ be the set of trees obtained from trees $T \in \mathcal{T}$ by substituting each occurrence of c_i in $frt(T)$ by some tree in $\mathcal{T}_i$ for $i \in [n]$. The set $(\mathcal{T}_1, ..., \mathcal{T}_n)^{\omega_c}$ is named the *ω-fold concatenation* and contains all the trees obtained by ω-iteration of the standard tree concatenation.

A tree language $\mathcal{T} \subseteq \mathcal{T}_{k,\Omega}$ is called *regular* if and only if it can be express starting from a finite subsets of trees and applying the union, the concatenation $._c$ and the star $*_c$ operations. A tree language is recognizable if and only if $\mathcal{T}$ is regular (see (40)).

2.2 Automata

2.2.1 Finite state automata and ω -automata

A deterministic finite state automaton (FA) is a quintuple $A_{FA} = (\Sigma, Q, q_0, \delta, F)$ where:

- Σ is a set of symbols, named *alphabet*
- Q is a finite set of states
- $q_0 \in Q$ is a initial state
- $\delta : Q \times \Sigma \to Q$ is the transition function
- $F \subseteq Q$ is the set of final states

A *run* of A_{FA} is a finite sequence of states $\pi = \pi_0, \pi_1, \pi_2, ..., \pi_n \in Q^*$ such that $\pi_0 = \sigma_0$ and for each $i \in [n-1]$ such that σ_i is the i^{th} letter if σ_i, then $\pi_{i+1} \in \delta(\pi_i, \sigma_i)$. A word $w \in \Sigma^*$ is accepted if A_{FA} has a run on w and $\delta(q_0, w) = q_f$ with $q_f \in F$.

In nondeterministic finite state automaton (NFA) for some state and input symbol, the next state may be nothing or one or two or more possible states. Formally an NFA, is quintuple $A_{NFA} = (\Sigma, Q, q_0, \delta, F)$ where:

- Σ is a set of symbols of the alphabet
- Q is a finite set of states
- $q_0 \in Q$ is a initial state
- $\delta : Q \times \Sigma \to 2^P$ is the transition function and 2^P is the power set of Q

- $F \subseteq Q$ is the set of final states

The definition of *run* of an NFA is the same as run of an FA, but now the transition can lead to a subset of state of Q. A word w is accepted if A_{NFA} if $\delta(q_0, w) \cap F \neq \emptyset$.

A ω-automaton (ω-FA) is a quintuple $A = (\Sigma, Q, q_0, \delta, F)$ where:

- Σ is the finite alphabet
- Q is a finite set of states
- $q_0 \in Q$ is a initial state
- $\delta : Q \times \Sigma \to Q$ is the transition function
- F is the acceptance condition.

For an ω-automaton a *run* over an infinite word $\sigma \in \Sigma^\omega$ is a sequence of states $\pi = \pi_0, \pi_1\pi_2, ... \in Q^\omega$ if $\pi_0 = \sigma_0$ and for each i such that σ_i is the i^{th} letter if σ_i, then $\pi_{i+1} \in \delta(\pi_i, \sigma_i)$. The difference between ω-automaton and finite state automaton relies in the acceptance conditions and, consequently, in the accepted words. A run π is accepting by A if it satisfies the acceptance condition.

For either a deterministic or a nondeterministic ω-automata, an ω-word $w \in \Sigma^\omega$ is accepted if there exists an accepting run on it. For *universal* ω-automata an ω-word w is accepted if all the runs on it are accepting.

In the following subsection we discuss about the main acceptance conditions. We refer the reader to (40) for more details on ω-words automata.

Safety automata A *safety automaton* A is a deterministic ω-automaton with no final states, and the language accepted by A, denoted W_A, is the set of all ω-words on which A has a run. We denote a safety automaton by $(\Sigma, Q, q_0, \delta_A)$ where Σ is a finite set of input symbols, Q is a finite set of states, $q_0 \in Q$ is the initial state, and $\delta_A : Q \times \Sigma \to Q$ is a partial function (the transition function). The language accepted by a safety automaton A is the set of all ω-words such that A has a run on it.

Büchi and co-Büchi automata A *Büchi automaton* A is ω-automaton and the language accepted by A, denoted W_A, is the set of all ω-words such that a state in F repeats infinitely often. We denote a Büchi automaton by $(\Sigma, Q, q_0, \delta_A, F)$ where Σ is a finite set of input symbols, Q is a finite set of states, $q_0 \in Q$ is the initial state, and $\delta_A : Q \times \Sigma \to Q$ is the transition function and $F \subseteq Q$ is the set of final state. A run π over an infinite word $\sigma \in \Sigma^\omega$ is accepted by A if $Inf(\pi) \cap F \neq \emptyset$.

A *co-Büchi automaton* is defined by the same tuple $(\Sigma, Q, q_0, \delta_A, F)$, but in this case the set F represents the set of state that the automaton must visited finitely often. Consequently, the language accepted by a co-Büchi automaton is the set of all ω-words such that a state in $F \subseteq$ repeats finitely often, i.e. a run π of a word w is accepting if $Inf(\pi) \cap F \neq \emptyset$.

Parity automata A *parity automaton* A is an ω-automaton and the language accepted by A, denoted W_A, is the set of all the ω-words such that the smallest number that is visited infinitely often is even. Formally, a parity automaton is defined by the tuple $(\Sigma, Q, q_0, \delta_A, F)$ where Σ is a finite set of input symbols, Q is a finite set of states, $q_0 \in Q$ is the initial state, and $\delta_A : Q \times \Sigma \to Q$ is the transition function and $F : Q \to [c]$ for some $c \in \mathbb{N}$, i.e. a colouring function that maps each vertex with a natural number in $[c]$. A run π over an infinite word $\sigma \in \Sigma^\omega$ is accepted by A if $min\{c(q) | q \in Inf(\pi)\}$ is even.

For more information on infinite automata, we refer the reader to (41).

2.2.2 Pushdown automata

A *pushdown automaton* $\mathcal{P}$ is a tuple $(Q, q_0, \Sigma, \Gamma, \delta, \gamma^\perp, F)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, Σ is a finite alphabet, Γ is a finite stack alphabet, $\gamma^\perp$ is the bottom-of-stack symbol, $F \subseteq Q$ defines an acceptance condition, and $\delta : Q \times \{\Sigma \cup \epsilon\} \times \Gamma \to Q \times \Gamma^*$ is the transition function. A pushdown automaton is *deterministic* if it satisfies the following two conditions:

- $\delta(q, \sigma, \gamma)$ has at most one element for any $q \in Q$, $\gamma \in \Gamma$ and $\sigma \in \{\Sigma \cup \epsilon\}$;
- for any $q \in Q$, $\gamma \in \Gamma$ and $\sigma \in \Sigma$, if $\delta(q, \epsilon, \gamma) \neq \emptyset$ then $\delta(q, \sigma, \gamma) = \emptyset$.

The transition relation is interpreted as follows: $\delta(q, \sigma, \gamma) = (q', \gamma')$ where q and q' are state of $\mathcal{P}$, $\sigma \in \Sigma$, $\gamma \in \Gamma$ and $\gamma' \in \Gamma^*$ and it means that the pushdown automaton is in

the state q with γ as symbol of the top of the stack and, when the input symbol σ is read, the automaton enters in a state q' and it can replace the top symbol γ with the string γ'. A word $w \in \Sigma^*$ is accepted by $\mathcal{P}$ if there is some run of $\mathcal{P}$ on w, starting from q_0 with symbol on the top of the stack $\gamma_\perp$ and finishing at any control state with the empty stack having consumed all of w (acceptance by empty stack).

2.2.3 Visibly pushdown automata

Consider a finite alphabet Σ, and let *call*, *ret*, and *int* be new symbols. We denote $\Sigma_{call} = \Sigma \times \{call\}$, $\Sigma_{ret} = \Sigma \times \{ret\}$, $\Sigma_{int} = \Sigma \times \{int\}$, and $\widehat{\Sigma} = \Sigma_{call} \cup \Sigma_{ret} \cup \Sigma_{int}$.

A *visibly pushdown automaton* (VPA) (see (6)) P is a tuple $(Q, Q_0, \Sigma, \Gamma \cup \{\gamma^\perp\}, \delta, F)$ where Q is a finite set of states, $Q_0 \subseteq Q$ is a set of initial states, Σ is a finite alphabet, Γ is a finite stack alphabet, $\gamma^\perp$ is the bottom-of-stack symbol, $F \subseteq Q$ defines an acceptance condition, and $\delta = \delta^{int} \cup \delta^{push} \cup \delta^{pop}$ where $\delta^{int} \subseteq Q \times \Sigma_{int} \times Q$, $\delta^{push} \subseteq Q \times \Sigma_{call} \times \Gamma \times Q$, and $\delta^{pop} \subseteq Q \times \Sigma_{ret} \times (\Gamma \cup \{\gamma^\perp\}) \times Q$.

A *configuration* (or global state) of P is a pair (α, q) where $\alpha \in \Gamma^*.\{\gamma^\perp\}$ and $q \in Q$. Moreover, (α, q) is *initial* if $q \in Q_0$ and $\alpha = \gamma^\perp$. We omit the semantics of the transitions of P being quite standard (see (6) for details). It can be obtained similarly to that of RGG with the addition of the input symbols on transitions. Here we just observe that we allow pop transitions on empty stack (a stack containing only the symbol $\gamma^\perp$). In particular, a pop transition does not change the stack when $\gamma^\perp$ is at the top, and by the definition of δ^{push}, $\gamma^\perp$ cannot be pushed onto the stack.

A *run* ρ of P over the input $\sigma_0\sigma_1\ldots$ is an infinite sequence $C_0 \xrightarrow{\sigma_0} C_1 \xrightarrow{\sigma_1} \ldots$ where C_0 is the initial configuration and such that, for each $i \in \mathbb{N}$, C_{i+1} is obtained from C_i by applying a transition on input σ_i. Acceptance of an infinite run depends on the control states that are visited infinitely often. Fix a run $\rho = (\gamma^\perp, q_0) \xrightarrow{\sigma_0} (\alpha_1, q_1) \xrightarrow{\sigma_1} (\alpha_2, q_2) \ldots$. With a *Büchi acceptance condition*, ρ is accepting if $q_i \in F$ for infinitely many $i \in \mathbb{N}$ (*Büchi* VPA). With a *co-Büchi acceptance condition*, ρ is accepting if there is a $j \in \mathbb{N}$ such that $q_i \notin F$ for all $i > j$ (*co-Büchi* VPA).

A VPA P is *deterministic* if: (1) $|Q_0| = 1$, (2) for each $q \in Q$ and $\sigma \in \Sigma_{call} \cup \Sigma_{int}$ there is at most one transition of δ from q on input σ, and (3) for each $q_1 \in Q$, $\sigma \in \Sigma_{ret}$, $\gamma \in \Gamma \cup \{\gamma^\perp\}$ there is at most one transition from q on input σ and stack symbol γ.

Note that a deterministic VPA has at most one run over any given input word w.

For a word w, a *deterministic/nondeterministic* VPA accepts w if there exists an accepting run over w. A *universal* VPA accepts w if all runs over w are accepting.

2.2.4 Tree automata

Trees are a natural data structure and they can be use to model many objects or behaviours in computer science, for example to represent hierarchical/nested data structures or functional/imperative programs. When it is necessary to reason on these settings, it is crucial to have finite representation of infinite sets of trees and tree automata allows us to have such finite representation. If finite state automata recognize strings, tree automata deal with tree structure. Tree Automata are strictly related to regular tree grammars and both describe sets of trees, and have well known algorithms for veryfing inclusion.

A *one-way nondeterministic tree automaton* is defined by a tuple $A = (Q, Q_0, \delta, F)$, where Q is a set of states, the set of initial states $Q_f \subseteq Q$, and δ is a transition relation i.e. $\delta \subseteq Q \times \Omega \times Q^k$, and F is the acceptance condition. One way deterministic tree automaton has the same definition of nondeterministic tree automaton but the set of initial state is composed only by a single initial state q_0 and there are no two transition rules with the same left hand side. We want to remember that deterministic tree automata are strictly less powerful than nondeterministic ones.

A one-way nondeterministic tree automaton is a *nondeterministic Büchi* (resp. *co-Büchi*) *tree automaton* over Ω-labeled k-trees if F is a Büchi (resp. co-Büchi) acceptance condition i.e. $F \subseteq Q$.

A *run* R of $\mathcal{A}$ on a Ω-labeled k-tree $T = ([k]^*, \nu)$ is a Q-labeled k-tree $([k]^*, \tau)$ such that $\tau(\varepsilon) \in Q_0$ and for each $x \in [k]^*$, $(\tau(x), \nu(x), \tau(x.1), \ldots, \tau(x.k)) \in \delta$. The labeling of R is such that the first component tracks the current node of the input tree T and the second component the current state of the automaton.

A *path* in a k-tree is a sequence of vertices $x_1 x_2 \ldots$ where for all $i \in \mathbb{N}$, $x_{i+1} = x_i.j_{i+1}$ for $j_{i+1} \in [k]$. In a Büchi tree automaton a run R is *accepting* if each path is accepting, i.e., for every *infinite* path $x_1 x_2 \ldots, \tau(x_i) \in F$ for infinitely many $i \in \mathbb{N}$. In a co-Büchi tree automaton a run R is *accepting* if each path is accepting, i.e., for every *infinite* path $x_1 x_2 \ldots$, there is a $j \in \mathbb{N}$ s.t. $\tau(x_i) \notin F$ for every $i > j$.

A *universal Büchi* or *co-Büchi tree automaton* $\mathcal{A} = (Q, q_0, \delta, F)$ is defined as a nondeterministic tree automaton except that $\delta : Q \times \Sigma \to 2^{[k] \times Q}$, where $\delta(q, \sigma)$ means

that at a node x labeled with σ, from a state q, $\mathcal{A}$ moves to state q' on $x.i$ for each $(i, q') \in \delta(q, \sigma)$. Without loss of generality we can assume that $|\delta(q, \sigma)| = r$ for each $q \in Q$ and $\sigma \in \Sigma$. For a Ω-labeled k-tree $T = ([k]^*, \nu)$, a *run* R of $\mathcal{A}$ on T is a $([k]^* \times Q)$-labeled r-tree $([r]^*, \tau)$ such that $\tau(\varepsilon) = (\varepsilon, q_0)$, and for each $x \in [r]^*$, denoting $\tau(x) = (y, q)$, then for all $i \in [r]$, $\tau(x.i) = (y.j, q_i)$ such that $j \in [k]$, $\{(j, q_i) \mid i \in [r]\} = \delta(q, \nu(y))$. Acceptance is as for nondeterministic tree automata. An automaton A accepts a Ω-labelled k-tree T iff there is an accepting run of A on T; the language of A, denoted $\mathcal{L}(A)$, is the set of all Ω-labelled k-trees that $\mathcal{A}$ accepts.

Two-way Alternating tree automata. In nondeterministic automata each transition sends exactly one state to each successor node in the tree. Alternating automata relax this restriction: it is possible to send several states to the same successor or to ignore some subtrees by not sending states to the corresponding node at all. Two-way tree automata extend ordinary tree automata by allowing transitions that not only construct terms but also destruct terms.

A *two-way alternating parity tree automaton.* (see (44)) over Ω-labelled k-trees is tuple $\mathcal{A} = (Q, q_1, \delta, W)$, where Q is a finite set of states, $q_1 \in Q$ is the initial state, W is a parity condition on Q and $\delta : Q \times \Omega \to \mathcal{B}^+((\{-1, 0, 1, \ldots, k) \times Q)$. Intuitively, $\{-1, 0, \ldots, k\}$ code the directions from a tree-vertex, where $\{1, \ldots, k\}$ stand for the k children of the tree vertex, -1 stands for the parent of the tree vertex, and 0 stands for the current tree-vertex itself. Let us extend the definition of concatenation of words over $[k]^*$ as follows: $(xi.(-1)) = x$ and $x.0 = x$, for any $x \in [k]^*$, $i \in [k]$, i.e. when a word is concatenated with -1, it removes the last letter and concatenating with 0 is the identity function.

A run of $\mathcal{A}$ over a Ω-labelled k-tree (T_k, ν), where $T_k = (Z, E)$, is a labelled tree $T_\rho = (V_\rho, E_\rho)$ where each tree-vertex in V_ρ is labelled with a pair (x, q) where $x \in Z$ is a tree-vertex of the input tree and $q \in Q$ is a state of the automaton $\mathcal{A}$, such that: (a) the root of T_ρ is labelled (ϵ, q_1), and (b) if a tree-vertex y of T_ρ is labelled (x, q), then we require that there is a set $F \subseteq \{-1, 0, 1, \ldots, k\} \times Q$ such that F satisfies $\delta(q, \nu(x))$ and for each $(i, q') \in F$, y has a child labelled $(x.i, q')$. A run is *accepting*, if for every *infinite* path in the run tree, if one projects the second component of the labels along the path, then it is a sequence of states in Q that satisfies the winning condition of $\mathcal{A}$. Note that there is no condition for *finite* paths of the run tree. An automaton $\mathcal{A}$

accepts a Ω-labelled k-tree T iff there is an accepting run of $\mathcal{A}$ on T; the language of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set of all Ω-labelled k-trees that $\mathcal{A}$ accepts.

In (44), Vardi introduced and studied two-way alternating tree automata and its conversion to one-way nondeterministic tree automata:

- Let $\mathcal{A}$ be a two-way alternating parity tree automaton. Then there is a one-way nondeterministic parity tree automaton $\mathcal{A}'$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$, where the number of states in $\mathcal{A}'$ is exponential in the number of states in $\mathcal{A}$, and the number of colours in the parity condition of $\mathcal{A}'$ is linear in the number of colours in the parity condition of $\mathcal{A}$.

For the emptiness problem of one-way nondeterministic tree automata (see (42)), we recall that:

- The emptiness of a one-way parity tree automaton $\mathcal{A}$ can be checked in time that is polynomial in the number of states and exponential in the number of colours in the parity condition.

A one-way nondeterministic tree automaton can be seen as a two-way alternating tree automaton where the transition function is always a disjunction of formulas of the kind $\bigwedge_{j=1}^{k}(j, q_j)$, i.e. the automaton guesses nondeterministically to send exactly *one* copy of itself in each *forward* direction. Two-way automata, though related to pushdown automata, are quite different. In fact, for every pushdown automaton, it is easy to construct a two-way automaton which accepts the possible contents of the stack . However, two-way tree (resp. word) automata have the same expressive power as standard tree (resp. word) automata: they only accept regular languages, while pushdown automata accept context-free languages, which strictly contain regular languages.

3

Games on pushdown systems

The theory of two player game on graph is an important area in formal verification, automata theory and logics. Infinite games are useful in many different contexts. First, the model-checking problem for the μ-calculus is intimately related to solving parity games (19), the precise complexity of which is still open. The monadic second order logic are also related to infinite games and it is proven that MSO-formulas can define the winning region of a parity game. The theory of games finds, moreover, a natural application in many synthesis problems, as example they forms a natural abstraction of the synthesis and control-synthesis problems, where the aim is to synthesize a system that satisfies a given specification (35).

In the standard setting of verification and synthesis the study of games is focused on finite games that can represent to infinite computations. A finite game is given as a pair composed by an arena, a finite graph where the set of vertices is split between two players, and a winning condition, that defines when a play is considered winning for a player. Flat game graphs are a good choice to model plain open system, but if we consider to model system with recursive procedures calls, pushdown games attract our attention, because correctly capture the behaviour of reactive and recursive systems. In this model a graph of a game is given by a configuration graph of a pushdown automaton. Such games are more suitable to model phenomena like procedure invocations, because the stack and the operations on it are explicitly present in the model. Most results in formal verification of pushdown games involve problems on finite graphs which maintains the call-stack of the system. Against regular specification the pushdown game problems are proving to be decidable (46). However solving games on pushdown

system against non-regular specification is in general undecidable. To overcome this issue, the researchers have focused their attention on a class of automata, the visibly pushdown automata (7), that have a decidable model-checking. This decidability extends also to the corresponding visibly pushdown game problems (27).

3.1 Games on graphs

Informally, a two-player game is played on an arena, a graph where the vertices are split in position of Player0 (named pl_0) and Player1 (named pl_1). The game starts with a token in an initial position and, each turn, if the vertex where the token is positioned belongs to a player, then such player chooses a move between the possible moves defined by the arena and moves the token in such position. Intuitively a play is defined the sequence of vertices generates by the choices of both players.

Formally an arena (named *game graph*) is a triple (V_{pl_0}, V_{pl_1}, E). The set V_{pl_0} is the set of pl_0-vertices, V_{pl_1} is the set of pl_1-vertices. The set of all the vertices of the graph is $V = V_{pl_0} \cup V_{pl_1}$ and V_{pl_0} and V_{pl_1} are a partition of V, i.e. $V_{pl_0} \cap V_{pl_1} = \emptyset$. With $E \subseteq V \times V$ we represent the set of directed edges: if $(v, v') \in E$ then from the vertex v the token can be moved on a vertex v'.

Given initial vertex $v_0 \in V$, a play π is a (possible infinite) sequence of vertices $\pi = \pi_0\pi_1...$ with $\pi_i \in V$ for $i = 0, 1...$ such that I) $\pi_0 = v_0$, II) for each $i = 0, 1, ...$ $(v_i, v_{i+1}) \in E$. The player pl_l with $l = [0, 1]$ moves the token from v_i to v_{i+1} if and only if $v_i \in V_{pl_l}$. If the play is finite, the player that should do the next move loses. If the play is infinite, to decide the winner we must consider winning conditions.

Winning conditions can express different aims. In Section 2.2.1 we have yet discuss the main basic winning conditions, but related to automata. We briefly recall here some of them and we add some new.

- *Reachability:* Given a target set $F \subseteq V$, pl_0 wins if the play π reaches a vertex in the target set, i.e. $\pi = \pi_0\pi_1...$ and $\exists i \geq 0$ such that $\pi_i \in F$.

- *Safety:* Given a set $F \subseteq V$ of safe vertices, pl_0 wins if the play π is composed only of vertices in the safe vertices, i.e. $\pi = \pi_0\pi_1....$ and $\forall i \geq 0$ $v_i \in F$.

- *Büchi:* Given a set of vertices $F \subseteq V$, pl_0 wins the play π if π visits at least one vertex in F infinitely often.

- *Co-Büchi:* Given a set of vertices $F \subseteq V$, pl_0 wins the play π if in π visits the vertices in F only finitely often.

- *Parity:* Given a colouring function $F : V \to [c]$, pl_0 wins if in the play π the minimum number seen infinitely often is even.

Each winning condition defines a *winning set* $W \subseteq V^\omega$, i.e. the set of all the play that are winning for pl_0. Note that Büchicondition is more general that the reachability condition and we can always transform a reachability game into a Büchigame. The same holds between parity condition and Büchicondition, i.e. a Büchicondition can be always transform into an equivalent parity condition using only two colours.

Given a game, i.e. a game graph equipped with a winning condition, the decidability problem for games asks to determine if pl_0 has a winning strategy, i.e. whatever the other player decides to move, pl_0 can has a way to choice its next moves such that the resulting play is always winning for him. Informally, a *strategy* is a function that says how pl_0 must behave during a play against pl_1. If applying its strategy, pl_0 wins each play regardless of how pl_1 moves,we say that this strategy is a *winning strategy*. In general case, a *strategy* (named global strategy) for pl_0 is a function $str_g : V^*.V_{pl_0} \to V$ which associates to each prefix $\pi_0\pi_1...\pi_n$ (with $\pi_0 = v_0$ and $\pi_n \in V_{pl_0}$) of a play π (the *global history* of π) the next move. Different strategies can be defined according on the information that can be used to decide the next move. In this thesis, we are interested also in *memoryless* strategies, i.e. the choice of the next move does only depend on the current vertex. Formally a memoryless strategy is a function $str_m : V_{pl_0} \to V$ that associates to each prefix $\pi_0\pi_1...\pi_n$ (with $\pi_0 = v_0$ and $\pi_n \in V_{pl_0}$) of a play π a next move only looking at π_n. A third kind of strategy, named *modular strategy*, can be defined for models for recursive procedure systems and the next move is chosen according to the history of the current activation of the module. We give the formal definition of modular strategy and discuss about the related decidability problems in detail in Chapter 4.

The choice of amount of information that can be used by the strategy to decide the next move is crucial and it influences the decidability of the game. Winning strategies that remember a bounded amount of informations lead to simpler solutions and in general the complexity of the decidability game problem is easier, but the non-existence

of simple winning strategy does not implies the non-existence of a winning strategy with complete information.

3.2 Winning conditions as formulas of temporal logics

Logics have found an important application in formal verification because they allows to express requirements for hardware or software systems. Moreover, the equivalence between many logical formalisms and automata constitutes a solid theoretical basis to the development of powerful algorithms and software system for the verification of finite-state programs. In the eighties, temporal logics have been introduced to express correctness requirement referring to time and they represent a convenient formalism for specifying and verifying properties of reactive systems. Using temporal logic formulas we can express statements as "This property always holds "or "Finally this requirement becomes true". Their relevance in the theoretical computer science has motivate the researches of the last decades to build on an extensive literature, composed by many different formalisms and important results. In the following subsections, we introduce some temporal logics that are relevant in this work because we had used their formalisms to specify our correctness requirements.

3.2.1 Linear Temporal Logic

The Linear Temporal Logic (Ltl) is a modal temporal logic proposed by Amir Pnueli for the formal verification of computer programs. Ltl can encode formulae that can express requirements about the future, as example condition that eventually will be true or condition that are true until another fact becomes true. However, branching time, quantifiers or past modalities can not be express using the set of Ltl operators. (LTL) is a popular choice for specifying correctness requirements of reactive systems. LTL formulas are built from atomic propositions using temporal modalities such as next, always, until... and are interpreted over infnite sequences of states that assign values to atomic propositions. The syntax and the semantics of Ltl are reported in Fig. 3.1.

The basic temporal operators of Ltl are:

- $\bigcirc$ or **X** is the *next* operator and requires that a property holds in the next state of the path

Syntax	$\varphi := p \mid \varphi \vee \varphi \mid \neg\ \varphi \mid \bigcirc\varphi \mid \varphi\ \mathcal{U}\varphi$
Semantics	for a word $\alpha \in 2^{AP}$ and n $\in \mathbb{N}$: • $(\alpha, n) \models p$ iff $\alpha_0 = (X, d)$ and $p \in X$ or $p = d$ (where $p \in A$P) • $(\alpha, n) \models \varphi_1 \vee \varphi_2$ iff $(\alpha, n) \models \varphi_1$ or $(\alpha, n) \models \varphi_2$ • $(\alpha, n) \models \neg\varphi$ iff $(\alpha, n) \not\models \varphi$ • $(\alpha, n) \models \bigcirc\varphi$ iff $(\alpha, n + 1) \models \varphi$ • $(\alpha, n) \models \varphi_1\ \mathcal{U}\varphi_2$ iff there is a sequence of position $i_0, i_1, ..., i_k$, where $i_0 = n, (\alpha, i_k) \models \varphi_2$ and for every $0 \leq j \leq k - 1$ $(\alpha, i_j) \models \varphi_1$

Figure 3.1: Syntax and semantics of Ltl .

- $\mathcal{U}$ or **U** is the *until* operator there is a state on the path where the second property holds and, at every state before that, the first property holds

Additional temporal operators are:

- $\Diamond$ or **F** is the *eventually* operator and it is used to assert that a property will hold at some state in the path
- $\Box$ or **G** is the *always* operator and it specifies that a property holds in every state of the path
- $\mathcal{R}$ or **R** is the *release* operator and it asserts that the second property will hold until the first will become true.
- $\mathcal{W}$ or **W** is the *weak until* operator and it is similar to the until operator, but the second property is not required to occur.

Such additional temporal operators (and the logical operators $\wedge, \rightarrow, \leftrightarrow$ can be defined in terms of the fundamental operators to write LTL formulas succinctly.

3.2.2 Temporal Logic of Calls and Returns

Model checking Ltl specifications with respect to pushdown systems has been shown to be a useful tool for analysis of programs with potentially recursive procedures. LTL, however, can specify only regular properties. If we must express properties such as correctness of procedures with respect to pre and post conditions, that require matching of

Syntax	$\varphi := p \mid \varphi \vee \varphi \mid \neg\varphi \mid \bigcirc^g \varphi \mid \bigcirc^a \varphi \mid \bigcirc^- \varphi \mid \varphi\ \mathcal{U}^g \varphi \mid \varphi\ \mathcal{U}^a \varphi \mid \varphi\ \mathcal{U}^- \varphi$
Semantics	for a word $\alpha \in 2^{AP} \times \{call, ret, int\}$ and n $\in \mathbb{N}$: • $(\alpha, n) \models p$ iff $\alpha_0 = (X, d)$ and $p \in X$ or $p = d$ (where $p \in A$P) • $(\alpha, n) \models \varphi_1 \vee \varphi_2$ iff $(\alpha, n) \models \varphi_1$ or $(\alpha, n) \models \varphi_2$ • $(\alpha, n) \models \neg\varphi$ iff $(\alpha, n) \not\models \varphi$ • $(\alpha, n) \models \bigcirc^g \varphi$ iff $(\alpha, succ^g_\alpha(n)) \models \varphi$, i.e., iff $(\alpha, n+1) \models \varphi$ • $(\alpha, n) \models \bigcirc^a \varphi$ iff $(\alpha, succ^a_\alpha(n)) \neq \bot$ and $(\alpha, succ^a_\alpha(n)) \models \varphi$ • $(\alpha, n) \models \bigcirc^- \varphi$ iff $(\alpha, succ^-_\alpha(n)) \neq \bot$ and $(\alpha, succ^-_\alpha(n)) \models \varphi$ • $(\alpha, n) \models \varphi_1\ \mathcal{U}^b \varphi_2$ (for any $b \in \{g, a, -\}$) iff there is a sequence of position $i_0, i_1, ..., i_k$, where $i_0 = n, (\alpha, i_k) \models \varphi_2$ and for every $0 \leq j \leq k-1, i_j + 1 = succ^b_\alpha(i_j))$ and $(\alpha, i_j) \models \varphi_1$

Figure 3.2: Syntax and semantics of CaRet.

calls and returns and are not regular, Ltl is not enough expressive to formalize such requirements and we must consider richer specification languages. One of such languages is the Temporal Logic of Calls and Returns (CaRet), a temporal logic that can express requirements about matching calls and returns, along with the necessary tools for algorithmic reasoning. The formulas of such logic are interpreted over structured computations . A structured computation is an infnite sequence of states, where each state assigns values to atomic propositions, and can be additionally tagged with *call* or *ret* symbols. Besides the global temporal modalities (as Ltl), CaRet admits their abstract counterparts, that capture the local computations within a module removinng the computation fragments corresponding to calls to other modules. Moreover, CaRet introduces the notion of the caller position which gives the most recent unmatched call position.These modality allows to espress the specification of properties that require inspection of the call-stack.

The syntax and the semantics of CaRet are reported in Fig. 3.2. We refer the reader to (3) for a detailed definition of CaRet.

In this logic, three different notions of successor are used:

- the global-successor ($succ^g$) which is the usual successor function. It points to next node, whatever module it belongs;

- the abstract-successor ($succ^a$) which, for internal moves, corrensponds to the global successor and for calls corresponds to the matching returns;

- the caller successor ($succ^-$) which is a "past" modality that points to the innermost unmatched call.

Typical properties that can be expressed by the logic CaRet are pre and post conditions. An example is the formula $G[(call \wedge p \wedge p_{p_1}) \rightarrow X^a q]$. If we assume that all calls to procedure p_1 are characterized by the proposition p_{p_1}, the formula expresses that if the pre-condition p holds when the procedure p_1 is invoked, then the procedure terminates and the post-condition q is satisfied upon the return. This is the requirement of total correctness. Observe the use of the abstract next operator to refer to the return associated with the call. In (3) it is proven that given a CaRet formula φ it is possible to construct a nondeterministic Büchi VPA of size exponential in $|\varphi|$ that accepts exactly all the words that satisfy φ.

3.2.3 Nested Word Temporal Logic

Nested word Temporal Logic (Nwtl) is an other formalism that is well suited for systems that work with a call/return paradigm. As CaRet , Nwtl has abstract and previous modalities. Its operators are:

- not ($\neg$), or ($\vee$), next ($\bigcirc$), as Ltl

- abstract next ($\bigcirc_\mu$), previous ($\ominus$), abstract previous ($\ominus_\mu$) as CaRet

- summary until (U^σ) and summary since (S^σ) and they are interpreted over a summary path that is the unique shortest directed path one can take between a position in a run and some future position, if one is allowed to use both successor edges and matching call-return summary edges.

The syntax and the semantics of Nwtl are reported in Fig. 3.3. We refer the reader to (8) for a detailed definition of Nwtl .

In (8) it is shown that even a Nwtl formula φ can be translate in a nondeterministic Büchi VPA of size exponential in $|\varphi|$ that accepts exactly all the words that satisfy φ.

Syntax	$\varphi := p \mid \varphi \vee \varphi \mid \neg\, \varphi \mid \bigcirc\varphi \mid \bigcirc_\mu\varphi \mid \ominus\varphi \mid \ominus_\mu\varphi \mid \varphi\ \mathcal{U}^\sigma\varphi \mid \varphi\ \mathcal{S}^\sigma\varphi$
Semantics	for a word w in 2^{AP} and the related nested word $\alpha = (w, \mu, call, ret)$ and $n \in \mathbb{N}$: • $(\alpha, n) \models p$ iff $\alpha_0 = (X, d)$ and $p \in X$ or $p = d$ (where $p \in A\mathrm{P}$) • $(\alpha, n) \models \varphi_1 \vee \varphi_2$ iff $(\alpha, n) \models \varphi_1$ or $(\alpha, n) \models \varphi_2$ • $(\alpha, n) \models \neg\varphi$ iff $(\alpha, n) \not\models \varphi$ • $(\alpha, n) \models \bigcirc\varphi$ iff $(\alpha, n+1) \models \varphi$ • $(\alpha, n) \models \bigcirc_\mu\varphi$ iff n is a call and j is the matching return (i.e.$\mu(n, j)$ holds) and $(\alpha, j) \models \varphi$ • $(\alpha, n) \models \ominus\varphi$ iff $n < 1$ and $(\alpha, n-1) \models \varphi$ • $(\alpha, n) \models \ominus_\mu\varphi$ iff n is a return and j is its matching call (i.e. $\mu(j, n)$ holds) and $(\alpha, j) \models \varphi$ • $(\alpha, n) \models \varphi_1\ \mathcal{U}^\sigma\varphi_2$ iff there exists a $j \geq n$ for which $(\alpha, j) \models \varphi_1$ and for the summary path $n = n_0 < n_i < \cdots < n_k = j$ between n and j, we have for every $p < k$ that $\alpha, n_p) \models \varphi_2$ • $(\alpha, n) \models \varphi_1 \mathcal{S}^\sigma \varphi_2$ iff there exists a $j < n$ for which $(\alpha, j) \models \varphi_2$ and for the summary path $j = n_0 < n_i < \cdots < n_k = n$ between j and n, we have for every $p < k$ that $\alpha, n_p) \models \varphi_1$

Figure 3.3: Syntax and semantics of NWTL .

3.3 Algorithmic problems related to games

Verification of reactive systems with recursive procedures The natural application of games is the modelling of reactive systems. A *reactive (or open) system* is a system whose role is to maintain an ongoing interaction with an external environment. In fact, in an open system, some of the input are controllable by the system itself, and other choices are uncontrollable and represent the behaviour of an external environment. The external environment can change the execution within the limits, that are defined by model. The system can react to the behaviour of the environment using one move taken from a set of available choices represented in the model. Typical examples of reactive system are programs that control mechanical devices or that interact with users. In general we consider a reactive system each ongoing and controller programs.

This setting can be nicely represented by a two player game. One player (named *protagonist* or pl_0) represents the controller. The other player (named *adversary* or pl_1) models the environment. The game graph defines the limits of the behaviour of both players: if a node of the graph belongs to pl_0 the next move is under the controll of the system itself,

The winning condition of the controller defines its goal. Determining the winner of the game answers the question whether there exists a controller, whereas computing a winning strategy realizes the *controller synthesis*.

Game graphs can be used to model "flat "system. However, if we consider systems that act using recursive procedure calls, we need to consider richer models. Pushdown games, visibly pushdown games and recursive game graphs allows to study the controller synthesis problem for such kind of open systems.

The μ-calculus The modal μ-calculus is an extension of the basic formalisms of standard logics, with a great number of attractive logical properties. For instance, it is the bisimulation invariant fragment of second order logic, it enjoys uniform interpolation, and the set of its validities admits a transparent, finitary axiomatization, and has the finite model property. The modal μ-calculus naturally generalizes all the properties of ordinary modal logic. The completeness theory of modal logic is an undeveloped field and there are still a set of open problems related to μ-calculus, as example the exact

expressiveness of the logic and the exact complexity of the model checking problem are still not known.

There is a direct connection between μ-calculus model checking and games on graphs. Given a graph G and a μ-calculus formula φ, in (48) it is shown that the formula can be translated in an alternating parity automaton and, taken the product of the graph with the alternating parity automaton, we construct a parity game and there exists a winning strategy for the protagonist for a vertex v_0 of G if and only if the μ-calculus formula φ is satisfied in v_0. The same reduction can be applied in the reverse: starting from a given parity game, it is always possible to write an equivalent μ-calculus formula that specify that a vertex is winning for the protagonist (see (46)). Therefore, parity games and model-checking for the μ-calculus are very close to each other and they are inter-reducible in linear time. Due to these transformations, it is sufficient to focus on solving games instead of model checking problem for μ-calculus.

Monadic Second Order Logic The determinacy theorem for games and the resolvability of infinite games on finite graphs are closely related to the decidability of the monadic second-order logic on trees. In fact, it is shown that there is a correspondence between automata on infinite trees and MSO-formulas. As consequence the monadic second-order theory turns out to be decidable. MSO-formulas define the winning region of a parity game where the alternation depth is constant and independent of the number of colours.

3.4 Pushdown games with regular winning conditions

In (46) the author considers pushdown parity games, i.e. pushdown processes where the set of the state is split among two players and equipped with a specification given as a colouring function that maps the states of the automaton to natural numbers. The pushdown process is a transition system that is form of pushdown automaton with emphasis on graph configurations. The author proves the EXPTIME-completeness of the model checking problem for pushdown parity games and consequently also the $\mu - calculus$ model checking problem. The upper bound is proved reducing the model checking from the pushdown process to the model checking problem for finite state transition system. The size of the resulting transition system after the reduction is

exponential in the number of states of the pushdown process and in the size of the formula. The exponential lower bound is proven presenting a reduction to the alternating linear Turing Machine, constructing a simulating automaton as the one presented in (13). These reductions are quite standard, bet we recall them because we will apply such similar approach to prove the lower bounds of the solutions presented in Chapters 5 and 9. The decidability of the model checking problem of pushdown parity games can be easily extended even to pushdown games with different winning conditions, provided that the specification could be translated into a parity automaton.

Pushdown processes Consider a finite alphabet Γ, let $\Gamma_\perp$ the finite alphabet such that $\Gamma_\perp = \Gamma \cup \gamma^\perp$ and let $\Gamma^c_\perp = \{skip, pop\} \cup \{push(\gamma) | \gamma \in \Gamma_\perp\}$ be the set of *stack commands* over $\Gamma_\perp$.

In this setting, a pushdown automaton $\mathcal{P}$ is a tuple $(Q, \Sigma, q_0, \gamma^\perp, \delta)$ where Q is the finite set of states, $\Gamma_\perp$ is the stack alphabet, $q_0 \in Q$ is the initial state, $\gamma^\perp$ is the initial stack symbol, and $\delta : Q \times \Gamma_\perp \to 2^{Q \times \Gamma^c_\perp}$ is the transition function. A configuration of $\mathcal{P}$ is a pair (α, q) where $\alpha \in \Sigma^+_\perp$ and $q \in Q$. The bottom-stack symbol can not be pushed or popped.

A *pushdown tree* $T_\mathcal{P}$ is defined by a pushdown automaton $\mathcal{P}$ according to the following rules: (i) the root of the tree is labelled with $(\gamma^\perp, q_0)$ and (ii) for every node $(\alpha_0, q_0), ..., (\alpha_i, q_i)$ if $(\alpha, q) \in \delta(\alpha_i, q_i)$ then the node has a son labelled with $(\alpha_0, q_0), ..., (\alpha_i, q_i), (\alpha, q)$.

Pushdown parity games Given an automaton $\mathcal{P}$, where the set of state Q is split in two subsets Q_{pl_0} and Q_{pl_1}, and a colouring function $\mathcal{F}$ (as in a parity automaton) we can define a *pushdown game* $G_\mathcal{P} = (V, E, \mathcal{F})$, where (V, E) defines the pushdown tree $T_\mathcal{P}$ and $\mathcal{F} : V \to [0, n]$ such that for each $v \in V$ then $\mathcal{F}(v) = \mathcal{F}(q)$ if q is the state in the label of v. We have a partition of V in two subsets V_{pl_0} and V_{pl_1} according to the partition of the set Q: we say that $v \in V_{pl_0}$ if and only if the state that occurring in the label of v belongs to Q_{pl_0}, otherwise $v \in V_{pl_1}$.

From a state labelled with a vertex $v \in V_{pl_0}$ (resp. $v \in V_{pl_1}$) a play in such game means that the player pl_0 (resp. pl_1) chooses the next state. To ease the presentation and without loss of generality, the visit of vertex of pl_0 and pl_1 is considered as strictly alternated. The play goes on indefinitely unless one of the players cannot make a move

and in such case that player loses. If the the projection of the resulting plays on the labelled vertices is an infinite sequence $v_1, v_2, ...$ and if in the corresponding infinite word $F(v_0)F(v_1)...$ the smallest number that is seen infinitely often is even, then this play is *winning* for pl_0.

A *pushdown strategy* is a function that, reading the moves of pl_1, defines the according moves of pl_0. The product of the moves done by both players defines a path on $T_{\mathcal{P}}$. The choice of the next move for pl_0 can be done only looking at the current state and the current symbol on the top of the stack.

A move is pair that consists of a state of $\mathcal{P}$ and a stack command, i.e. it is an element of $Q \times \Gamma_{\perp}^{c}$. A path on $T_{\mathcal{P}}$ determines a sequence of moves that automaton made on this path and a sequence of moves may determine a sequence of configurations. Not al the sequences of moves determines paths because they could contain invalid moves. A strategy automaton is a deterministic finite state automaton with input and output $\mathcal{P}_{str} = (Q_{\mathcal{P}_{str}}, \Sigma_i, \Sigma_o, \Gamma_{\mathcal{P}_{str}}, q_0, \gamma_{\perp}, \delta_{\mathcal{P}_{str}})$ where $Q_{\mathcal{P}_{str}}$ is a finite set of states, Σ_i is the input alphabet, Σ_o is the output alphabet, $\Gamma_{\mathcal{P}_{str}}$ is the stack alphabet, $\gamma_{\perp}$ is the initial stack symbol , and $\delta_{\mathcal{P}_{str}}$ is the transition function $\delta_{\mathcal{P}_{str}} : Q_{\mathcal{P}_{str}} \times \Gamma_{\perp} \times (\Sigma_i \times \{\tau\}) \rightarrow Q_{\mathcal{P}_{str}} \times \Gamma_{\perp}^{c} \times (\Sigma_o \times \{\tau\})$. Intuitively, if $\delta_{\mathcal{P}_{str}}(q, \gamma, \alpha) = (q', c, \beta)$ this means that when the automaton is the state q with γ on the top of the stack, the automaton changes its state in q', performs the stack command c and returns as output the symbol β. If $\gamma = \tau$ then the automaton does not read the input and if $s = \tau$ the automaton does not return an output symbol. For each pushdown strategy in $G_{\mathcal{P}}$, then there is a strategy automaton for $\mathcal{P}$.

Let $\mathcal{P}$ be a pushdown automaton and let $\mathcal{F}$ be a colouring function. As we said, they define a pushdown parity game $G_{\mathcal{P}}$. The problem that is considered is to establish the existence of a winning strategy in such game $G_{\mathcal{P}}$ for pl_0. The solution of such problem relies in the reduction to the problem of deciding the existence of a winning strategy in game on finite graphs. Such reduction is handled as a kind of power-set construction, which leads the complexity immediately to be exponential in size of the game.

Lower bound. The exponential time lower bound follows from a reduction from alternating Turing machine. Let M be an alternating linear space Turing Machine and let Q be the set of state of M partitioned in two sets $Q_{\exists}$ and $Q_{\forall}$, respectively the set of existential states and universal states. Let Γ be the tape alphabet and

$\delta : Q \times \Gamma \to (Q \times \Gamma \times \{L, R\})$ be the transition function. Assume that M has only one tape and let $n-1$ be the number of cells used by M on an input word w of size n.

Remember that a configuration of M is a word $\sigma_1 \dots \sigma_{i-1} q, \sigma_i \dots \sigma_n$ where $\sigma_1 \dots \sigma_n$ is the content of the tape cells, q is a state of M and q, σ_i denotes that the tape head is on cell i.

For a give word w it is possible to construct a pushdown automaton $\mathcal{P}$ with the state partitioned in Q_{pl_0} and Q_{pl_1} such that pl_0 has a strategy to reach a leaf in the game if and only if the word w is recognized by M. Initially, $\mathcal{P}$ must guess the a sequence of n letters of the configuration. To produce a correct configuration, the automaton must guess one and only one letter $q \in Q$ in such sequence. Then the automaton pushes such letters on the stack. After such pushing, the automaton arrives in a state in Q_{pl_0} if $q \in Q_\exists$, or in a state in Q_{pl_1} if $q \in Q_\forall$. From this state, the automaton simulates the transition function of M and that $\mathcal{P}$ pushes on the stack the another sequence that represents the previous configuration and the process repeats. At this point the automaton has encoded two configurations and a selected transition and it must verify that such triple simulates a legal move of M and, to do that, the automaton enters a state named $Check$. This state is a pl_1-state and the adversary here can choose to verify that the two configuration are legal according to the transition of M or to assume that the simulation has been done correctly and move on. If pl_1 forces the verification, the automaton enters a state $Check_1$ and tests the consistency of the two guessed configurations according to the selected move. If the test succeeds $\mathcal{P}$ stops, otherwise $\mathcal{P}$ goes in an infinite loop. If pl_1 decides to do not check the consistency of two consecutive configuration, the process of choose transitions and guess the configurations proceeds until the first configuration is reached. At this point pl_1 checks if this configuration is an initial one and if the test succeeds $\mathcal{P}$ stops, otherwise it goes in an infinite loop. Then the following holds:

Theorem 1. *The problem of deciding the existence of a winning strategy in a pushdown parity game is* EXPTIME*-complete.*

3.5 Visibly pushdown games

Formal verification of pushdown games against regular specification is in general tractable, because the emptiness of pushdown automata is decidable. However, solving pushdown

games with pushdown winning conditions is undecidable. Checking software models against regular specifications is useful, but there are important context-free requirements as specification of pre-post conditions for procedures, security properties,etc... that require the stack inspection. The logic CaRet and Nwtl allow to express specification of such context-free properties and preserves decidability of pushdown model-checking. Visibly pushdown automata (VPA) can recognize the corresponding languages and have a decidable model-checking. In (27) it is shown that the decidability can be extended also to Visibly Pushdown Games (VPG) with winning condition given as a VPA .

Infinite Two-Player Games A game graph over a finite alphabet Σ is a graph $G = (V, V_{pl_0}, V_{pl_1}, E)$ where (V, E) is a graph with its edges labeled with letters of Σ and (V_{pl_0}, V_{pl_1}) is a partitions of V between the two players.

An infinite two player game is a pair (G, W) where the winning condition W can be an internal winning condition, i.e. W is a subset of V^ω, or an external winning condition, i.e. W is a subset of Σ^ω.

Intuitively a play in this setting starts from an initial node v_0 and then proceeds as follows: for $i \in \mathbb{N}$ if $v_i \in V_{pl_j}$ with $j \in [0, 1]$ then pl_j moves the token and it can move it on a vertex v_{i+1} such that exists and edge $e = (v_i, \sigma, v_{i+1}) \in E$. If a players cannot make a move, the other player wins, otherwise the play $\pi = v_0\sigma_0v_1\sigma_1...$ is an infinite sequence and if the winning condition is internal (resp.external) pl_0 wins if $v_0v_1... \in W$ (resp.$\sigma_0\sigma_1... \in W$), otherwise pl_1 wins.

Visibly Pushdown Games A *visibly pushdown game* (VPG) is a tuple $(S, Q_{pl_0}, Q_{pl_1}, P)$ where S is a visibly pushdown system and a VPA P, both over a common alphabet $\hat{\Sigma}$, and a partition $\langle Q_{pl_0}, Q_{pl_1}\rangle$ of the state of S between the two players.

A *visibly pushdown system* (VPS) over the alphabet $\hat{\Sigma}$ is a tuple $S = (Q, q_{in}, \Gamma, \delta)$ where Q is a finite set of states, $Q_{in} \subseteq Q$ is a set of initial states, Γ is a finite stack alphabet that contains the symbol $\perp$, which represents the bottom of the stack, and $\delta \subseteq Q \times \Sigma_{call} \times Q \times (\Gamma \backslash \{\perp\})) \cup (Q \times \Sigma_{ret} \times \Gamma \times Q) \cup (Q \times \Sigma_{int} \times Q)$ is the transition relation.

The stack content w_γ is a sequence of symbol of Γ or the bottom of the stack, i.e. $w_\gamma \in (\Gamma \backslash \{\bot)\} * \cup \{\bot\}$. We refer to the set of all possible stack contents with the notation Stk.

A transition (q, σ, q', γ) where $\gamma \in \Sigma_{call}$ and $\gamma \neq \bot$ is a push transition, i.e. on reading σ the stack symbol γ is pushed onto the stack and the control passes to q'. Similarly (q, σ, q', γ) is a pop transition if $\sigma \in \Sigma_{ret}$ and and $\gamma \in \Sigma_{ret}$: when γ is read from the top of the stack, it is popped and the control passes from q to q'. Note that in the case of a pop transition, if the stack is empty, the stack is read but not popped. On the internal operation, the stack is left unchanged.

A configuration graph of a VPS S is a graph $G_S = (V_S, E_S)$ where $V_S = \{(q, w_\gamma) | q \in Q$ and $w_\gamma \in Stk\}$ and E_S is the set composed by all triples of the form $((q, w_\gamma), \sigma, (q', w'_\gamma))$ that satisfy the following rules:

- **Push:** If σ is a call, then $\exists \gamma \in \Gamma$ such that $(q, \sigma, q', \gamma) \in \delta$ and $w'_\gamma = \gamma . w_\gamma$
- **Pop:** If σ is a return, then $\exists \gamma \in \Gamma$ such that $(q, \sigma, q', \gamma) \in \delta$ and either $\gamma \neq \bot$ and $w_\gamma = \gamma . w'_\gamma$ or $\gamma = \bot$ and $w_\gamma = w'_\gamma$
- **Internal:** If σ is an internal action, then $(q, \sigma, q') \in \delta$ and $w_\gamma = w'_\gamma$

For a word $w = w_0 w_1 w_2 ... \in \Sigma^\omega$, a run of S is a sequence of configurations $\pi = (q_0, w^0_\gamma)(q_1, w^1_\gamma)...$ where $q_0 \in Q_{in}$, $\sigma_0 = \bot$ and $((q_i, w^i_\gamma), \sigma, (q_{i+1}, w^{i+1}_\gamma))$

The VPA P is define as in Section 2.2.3 and it represent the specification automaton. The language of the VPA P is the set of words accepted by P and we denote it with L_P.

The visible pushdown game problem asks if, given a visibly pushdown game (S, Q_{pl_0}, Q_{pl_1}) and a state p_{in} in S, there exists a (global) strategy for pl_0 such that is winning from the position $(p_{in}, \bot)$.

In (7) it is proven that such problem is decidable and it is 2Exptime-complete. The main challenge to overcome is that in general specification automata given as VPA are not determinazable and this prevents from taking the product of the game with the specification automaton and reduce it to a pushdown game with internal winning condition. The authors resolve such problem passing through a different kind of VPA, named stair VPA. In such automata, the acceptance condition is not evaluated on the whole run, but on a subsequence of it. Such subsequence is obtained from the whole

run discarding the configurations for which a future configurations has a smaller stack height. Such restriction does not decrease the expressive power of the automata and the author proves that for each nondeterministic Büchi VPAP it is possible construct an equivalent nondeterministic stair VPA P' with a number of states exponential in the number of the states of P.

Using this result, it is possible internalize the winning condition, transforming it into a deterministic stair VPA and, then, taking its product with the game graph defined by the visibly pushdown game. The resulting game has now a stair parity winning condition and it can be solved adapting a variant of the classical techniques for pushdown parity game that leads to the 2Exptime-completeness. Due to the fact that a CaRet is a subclass of context-free languages which is contained in the languages expressible by VPAs and, given a CaRet formula φ over 2^{AP}, it is possible to construct an equivalent Büchi visibly pushdown automaton on a partition of 2^{AP} with calls, returns and internal action, and the size of the constructed automaton is $2^{O(|\varphi|)}$. Hence it follows that solving visibly pushdown games with specification given as a CaRet formula is in 3Exptime. Therefore, in (7) the following results are proven:

- Given a visibly pushdown game a winning condition given as a nondeterministic Büchi automaton, checking whether pl_0 has a winning strategy is 2Exptime-complete.
- Given a visibly pushdown game and a nondeterministic Büchi VPA, checking if pl_0 has a winning strategy is is 2Exptime-complete.
- Given a visibly pushdown game with specification given as a Ltl formula, checking if pl_0 has a winning strategy is is 3Exptime-complete.
- Given a visibly pushdown game with specification given as a CaRet formula, checking if pl_0 has a winning strategy is is 3Exptime-complete.

4

Modular strategies

As we said, the original motivation for studying games in the context of formal analysis of systems comes from the controller synthesis problem. Given a description of the system where some of the choices depend upon the input and some of the choices represent uncontrollable internal non-determinism, designing a control ler that supplies inputs to the system so that the product of the controller and the system satisfies the correctness specification corresponds to computing winning strategies in two-player games.

In traditional model checking, the model is a finite state machine whose vertices correspond to states, and whose edges correspond to transitions. To define two-player games in this model, the vertices are partitioned into two sets corresponding to the two players, where a player gets to choose the transition when the current state belongs to its own partition.

In this framework, the work (4) introduces the notion of *modular strategies*. A modular strategy is a strategy that can remember only the history of the current activation of a module, and thus, the resolution of choices within a module does not depend on the context in which the module is invoked, but only its local history. This notion allows to synthesize controllers that are indipendent from the context where the modules are invoked.

In this chapter we present the definition of RGG , the model that is chosen to represent the recursive system, and then we formilize the notion of modular strategy. In the remaining sections, we recall the results presented in (4, 5), that consider modular games where the acceptance condition is given as a reachability winning condition

or as deterministic/universal Büchi automata or expressed using LTL formulas. The first problem is solved using a fixed-point computation algorithm that generalizes the symbolic solution to reachability games. For the remaining decidability problem, the solution is presented as an automata-theoretic construction.

4.1 Recursive game graph

A recursive game graph (RGG) is composed of *game modules* that are essentially two-player graphs (i.e., graphs whose vertices are partitioned into two sets depending on the player who controls the outgoing moves) with *entry* and *exit* nodes and two different kind of vertices: the nodes and the boxes. A *node* is a standard graph vertex and a *box* corresponds to invocations of other game modules in a potentially recursive manner (in particular, entering into a box corresponds to a module *call* and exiting from a box corresponds to a *return* from a module). As an example consider the RGG in Figure 4.1, where the vertices of player (pl_0) are denoted with circles, those of adversary (pl_1) with squares and the rectangles denote the vertices where there are no moves that can be taken by either players and correspond to calls and exits. Atomic propositions p_a, p_b, p_c, p_d and p_e label the vertices.

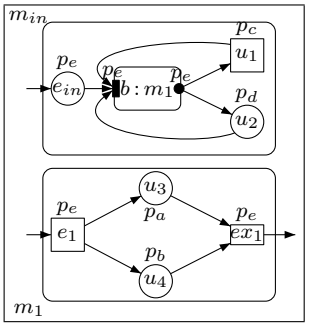

Figure 4.1: A sample RGG.

Each RGG has a distinct game module which is called the *main* module (module m_{in} in the figure). In analogy to many programming languages, we require that the main module cannot be invoked by any other module. A typical play starts in vertex e_{in}. From this node, there is only one possible move to take and thus the play continues at the call to m_1 on box b, which then takes the play to the entry e_1 in m_1. This is a vertex of the adversary, who gets to pick the transition and thus can decide to visit either u_3 (generating p_a) or u_4 (generating p_b). In either of the cases, the play will reach the exit and then the control will return to module m_{in} at the return vertex on box b. Here pl_0 gets to choose if generating p_c or p_d and so on back to the call to m_{in}. Essentially, along any play alternately pl_1 chooses one between p_a and p_b, and pl_0 chooses one between p_c and p_d. Formally, we have the following definitions.

Definition 1. *(*Recursive Game Graph*)* *A* recursive game graph G *over* AP *is a triple* $(M, m_{in}, \{S_m\}_{m \in M})$ *where* M *is a finite set of module names,* $m_{in} \in M$ *denotes the main module and for each* $m \in M$, S_m *is a game module. A* game module S_m *is* $(N_m, B_m, Y_m, En_m, Ex_m, \delta_m, \eta_m, P_m^0, P_m^1)$ *where:*

- N_m *is a finite set of nodes and* B_m *is a finite set of boxes;*
- $Y_m : B_m \to (M \setminus \{m_{in}\})$ *maps every box to a module;*
- $En_m \subseteq N_m$ *is a non-empty set of entry nodes;*
- $Ex_m \subseteq N_m$ *is a (possibly empty) set of exit nodes;*
- $\delta_m : N_m \cup Retns_m \to 2^{N_m \cup Calls_m}$ *is a transition function where* $Calls_m = \{(b, e) | b \in B_m, e \in En_{Y_m(b)}\}$ *is the set of calls and* $Retns_m = \{(b, e) | b \in B_m, e \in Ex_{Y_m(b)}\}$ *is the set of returns;*
- $\eta_m : V_m \to 2^{AP}$ *labels in* 2^{AP} *each vertex from* $V_m = N_m \cup Calls_m \cup Retns_m$*;*
- P_m^0 *and* P_m^1 *form a partition of* $(N_m \cup Retns_m) \setminus Ex_m$*;* P_m^0 *is the set of the positions of* pl_0 *and* P_m^1 *is the set of the positions of* pl_1*.*

In the rest of the thesis, we denote with: G an RGG as in the above definition; $V = \bigcup_m V_m$ (set of vertices); $B = \bigcup_m B_m$ (set of boxes); $Calls = \bigcup_m Calls_m$ (set of calls); $Retns = \bigcup_m Retns_m$ (set of returns); $Ex = \bigcup_m Ex_m$ (set of exits); $P^\ell = \bigcup_m P_m^\ell$ for $\ell \in [0, 1]$ (set of all positions of pl_ℓ); and $\eta : V \to 2^{AP}$ the function such that $\eta(v) = \eta_m(v)$ where $v \in V_m$.

To ease the presentation and without loss of generality, we make the following assumptions (with $m \in M$):

– there is only one entry point to every module S_m and we refer to it as e_m (we can reduce a module with multiple entries to a set of modules with single entry by duplicating this module one for each entry point and changing the calls and returns appropriately);

– From within the same module, there are no transitions to an entry, i.e., $e_m \notin \delta_m(u)$ for every u;

– From within the same module, there are no transitions from an exit, i.e., $\delta_m(x)$ is empty for every $x \in Ex_m$;

– a module is not called immediately after a return from another module, i.e.,

$\delta_m(v) \subseteq N_m$ for every $v \in \mathit{Retns}_m$.

A (global) *state* of an RGG is composed of a call stack and a vertex, that is, each state of G is of the form $(\alpha, u) \in B^* \times V$ where $\alpha = b_1 \dots b_h$, $b_1 \in B_{m_{in}}$, $b_{i+1} \in B_{Y(b_i)}$ for $i \in [h-1]$ and $u \in V_{Y(b_h)}$.

A *play* of G is a (possibly finite) sequence of states $s_0 s_1 s_2 \dots$ such that $s_0 = (\epsilon, e_{in})$ and for $i \in \mathbb{N}$, denoting $s_i = (\alpha_i, u_i)$, one of the following holds:

- **Internal move:** $u_i \in (N_m \cup \mathit{Retns}_m) \setminus \mathit{Ex}_m$, $u_{i+1} \in \delta_m(u_i)$ and $\alpha_i = \alpha_{i+1}$;
- **Call to a module:** $u_i \in \mathit{Calls}_m$, $u_i = (b, e_{m'})$, $u_{i+1} = e_{m'}$ and $\alpha_{i+1} = \alpha_i.b$;
- **Return from a call:** $u_i \in \mathit{Ex}_m$, $\alpha_i = \alpha_{i+1}.b$, and $u_{i+1} = (b, u_i)$.

Fix an infinite play $\pi = s_0 s_1 \dots$ of G where $s_i = (\alpha_i, u_i)$

With π_k we denote $s_0 \dots s_k$, i.e., the prefix of π up to s_k. For a finite play $\pi'.s$, with $\mathit{ctr}(\pi'.s)$ we denote the module m where the control is at s, i.e., such that $u \in V_m$ where $s = (\alpha, u)$. We define a predicate μ_π such that $\mu_\pi(i, j)$ holds iff for some $m \in M$, $u_i \in \mathit{Calls}_m$ and j is the smallest index s.t. $i < j$, $u_j \in \mathit{Retns}_m$ and $\alpha_i = \alpha_j$ (μ_π captures the matching pairs of calls and returns in π).

4.2 Modular strategies

Fix $\ell \in [0, 1]$. A *strategy* of pl_ℓ is a function f that associates a legal move to every play ending in a node controlled by pl_ℓ.

A modular strategy constrains the notion of strategy by allowing only to select legal moves depending on the "local memory"of a module activation (every time a module is re-entered the local memory is reset).

Formally, a *modular strategy* f of pl_ℓ is a set of *local* functions $\{f_m\}_{m \in M}$, one for each module $m \in M$, where $f_m : V_m^*.P_m^\ell \to V_m$ is such that $f_m(\pi.u) \in \delta_m(u)$ for every $\pi \in V_m^*, u \in P_m^\ell$.

The local successor of a position in a play π is: the successor according to the matching relation μ_π at matched calls, undefined at an exit or an unmatched call, and the next position otherwise. Formally, the *local successor* of j, denoted $\mathit{succ}_\pi(j)$, is: h if $\mu_\pi(j, h)$ holds; otherwise, is undefined if either $u_j \in \mathit{Ex}$ or $u_j \in \mathit{Calls}$ and $\mu_\pi(j, h)$ does not hold for every $h > j$; and $j + 1$ in all the remaining cases.

For each $i \leq |\pi|$, the *local memory* of π_i, denoted $\lambda(\pi_i)$, is the maximal sequence $u_{j_1} \dots u_{j_k}$ such that $u_{j_k} = u_i$ and $j_{h+1} = succ_\pi(j_h)$ for each $h \in [k-1]$. (Note that since the sequence is maximal, $u_{j_1} = e_m$ where $m = ctr(\pi_i)$.)

A play π *conforms to* a modular strategy $f = \{f_m\}_{m \in M}$ of pl_ℓ if for every $i <| \pi |$, denoting $ctr(\pi_i) = m$, $u_i \in P_m^\ell$ implies that $u_{i+1} = f_m(\lambda(\pi_i))$.

Consider again the example from Figure 4.1. A strategy of pl_0 that chooses alternately to generate p_c and p_d is modular, in fact it requires as memory just to store the last move from the return of b, and thus is local to the current (sole) activation of module m_{in}. Instead, a strategy that attempts to match each p_a with p_c and each p_b with p_d is clearly non-modular.

We remark that modular strategies are oblivious to the previous activations of a module. In the RGG of Figure 4.1, a modular strategy for pl_1 would only allow either one of the behaviors: "pl_1 always picks p_a"or "pl_1 always picks p_b".

4.3 Winning conditions and modular games

A modular game on RGG is a pair $\langle G, L \rangle$ where G is an RGG and L is a winning condition. A winning condition is a set L of ω-words over a finite alphabet $\Sigma = 2^{AP}$, where AP is a set of propositions.

Given an RGG G, for a play $\pi = s_0 s_1 \dots$ of G, with $s_i = (\alpha_i, u_i)$, we define the *trace* of π, denoted w_π, as the word $\eta(u_0)\eta(u_1)\dots$ that maps each position to the corresponding symbol from Σ. A (modular) strategy f is *winning* if $w_\pi \in L$ for every play π of G that conforms to f.

The *modular game problem* asks to determine the existence of a winning (modular) strategy of pl_0 in a given modular game.

In the following sections, we will consider modular game with reachability winning conditions, or L given by deterministic/universal Büchi automata and by LTL formulas.

4.4 Solving modular games with reachability winning conditions

In (4) the authors consider modular games with reachability winning conditions. The approach to solve this problem is an evolution of the attractor set algorithm in flat

games. The algorithm starts from the target set, that is marked as reachable, and it updates the attractor set step by step, adding new vertices that are discovered "backward" from the vertices that are it the set yet.

The algorithm is formulate as a fixed -point computation that generalizes the symbolic solution to reachability games. The fixed-point algorithm starting with a set of target vertices and iteratively grows the set of vertices from which winning is ensured. In the case proposed in (4), when a node is found to be winning, the algorithm keeps track of the strategies that are used within different module activations. The labeling make sure that the same set of module strategies is used consistently everywhere and, in this way, the algorithm guarantees that the strategy will be modular. When a play enters a module, the strategy used for the module always drives the play such that the play exits only in a subset E_{reach} of exit nodes. The strategy only allows that a subset M_{reach} of modules to be called from a current module. This two set are the only relevant aspects of the strategy that needs to be recorded.

Fix a RGG $G = (M, 1, \{S_i\}_{i\in[n]})$ where each module S_i is $(N_i, B_i, Y_i, En_i, Ex_i, \delta_i, \eta_i, P_i^0, P_i^1)$ with $i \in [n]$ and a target set $X \subseteq Ex_1$. The first key observation is that if there is a winning strategy for a reachability game, we can reduce this to a hierarchical modular strategy. Intuitively, if a play reaches a target node executing one or more cycles, this means that we can omit to go through the cycles, going directly forward, and the resulting play will still reach the target node.

Now the authors can describe their solution to the considered problem. The proposed algorithm is a labeling algorithm that iteratively labels vertices of the RGG with tuples of sets of exit nodes according to some initialization and update rules. The algorithm halts when it can not add any other label. Initially each exit node $x \in X$ is labeled by a tuple $\langle E_1, ..., E_n\rangle$ where $E_1 = x$ and $E_i = \{\top\}$ for every $i \in [n]$ All the other vertices are unlabeled. The next updates are done according to the following rules.

- For a node $v \in P_i^0$, if $\langle E_1, ..., E_n\rangle$ labels $u \in \delta_i(v)$ then add $\langle E_1, ..., E_n\rangle$ to the labels of v.

- For a node $v \in P_i^1$ and $\delta_i(v) = \{v_1, ..., v_k\}$ if (i)$\langle E_1^z, ..., E_n^z\rangle$ labels v_z for $z \in [k]$ and (ii) for every $j \neq i$ $\langle E_1^y, ..., E_n^y\rangle$ are pairwise consistent, then add $\langle E_1', ..., E_n'\rangle$ to the labels of v, where $E_j' = \bigcup_{z=1}^{k} E_j^z$ for $j \in [n]$.

- For $(b, e) \in Retns_i$ labeled by $\langle E_1, ..., E_n \rangle$ where $Y_i(b) = j$, add $\langle E'_1, ..., E'_n \rangle$ to labels of x where $E'_j = \{x\}$ and $E'_z = \top$ for $z \neq j$.
- For $(b, e) \in Calls_i$ such that $Y_i(b) = j$, let $(b, e_1), ..., (b, e_k)$ be any k distinct returns of box b. Suppose that for $z \in [k]$, $\langle E^z_1, ..., E^z_n \rangle$ labels (b, x_h) and $\langle E^0_1, ..., E^0_n \rangle$ labels $e \in En_j$. If $E^0_j = \{x_1, ..., x_k\}$ and $E^0_l, ..., E^k_l$ are pairwise consistent for every $l \neq i$, then add $\langle E_1, ..., E_n \rangle$ to the labels of (b, e) where $E_l = \bigcup_{z=1}^{k} E^z_l$ for $l \in [n]$.

Each rule has a specific aim:

- The first rule allows to the pl_0 nodes to inherit a label of the successor.
- The second rule allows to label the pl_1 nodes. The algorithm must check the consistency of all the labels of the successors.
- The third rules activates an exit node of the j^{th} module, when there is a box b in the i^{th} module that is a call to the j^{th} module and has a return (b, e) that has a label.
- The fourth rules allows to label the calls. If the module $i^t h$ calls a module $j^t h$, such call can get the label of the entry of the j^{th} module if the algorithm ensures that there are a strategy which forces the plays that enter in the j^{th} module to reaches some exit nodes and the return of the i^{th} module correspond to these exit and they have the same assumption on all the modules.

The proposed algorithm requires exponential time, since it can at most generate exponentially many labels. The algorithm works in time exponential in the number of the exit nodes and in time linear in the size of the recursive game graph. The following theorem holds:

Theorem 2. *For an RGG G and a target set X, deciding if pl_0 has a winning modular strategy in such game is* NP *-complete.*

Undecidability with incomplete information The choice of consider modular strategies is motivated by the fact that, in a system composed by separated modules, it is natural that even the controllers that have the role to keep the system correct are implemented to represent the strategy of its related module. This feature can be

relaxed and in (4) it is considered a different approach, where a module can remember the history of the previous invocations and can use it to decide in each pl_0 vertex the next move. This approach is named *persistent strategy*, allowing to remember all the parts of the play in a module.

In (4) it is proven that this subtle change in the definition a modular strategies leads the problem to become undecidable. The proof of this claim is based on a reduction from the undecidability of solving multi-player games with incomplete information. In a multiplayer game, there are two players, pl_0^α e pl_0^β, that form a team 0 that plays against a team 1, represented by a single player, the adversary. The adversary choices the name of a player of team 0 and challenge him with a symbol and, then, the chosen player must respond with a symbol. In the multi-player game is with incomplete information, after a play π if it is the turn of the player pl_0^α (respectively pl_0^β), the pl_0^α (respectively pl_0^β) can decide the next move only using the portion of history that concern its choices and the choices of the adversary.

To solve a multi-player game with incomplete information, it can be reduce to a RGG composed of only three modules, m_{in}, m_1 and m_2, where m_{in} is the main module and it calls m_1 and m_2. In these two last modules, there are no call to other modules. The module m_1 has an entry node for each letter of the pl_1 alphabeth and an exit for each pl_0^α symbol, and module m_2 has an entry node for each letter of the pl_1 alphabeth and an exit for each pl_0^β symbol. Intuitively, the persistent strategy in the module m_1 corresponds directly to an incomplete information information for the player pl_0^α and the persistent strategy in the module m_2 corresponds directly to an incomplete information for the player pl_0^β. If in the RGG game graph a persistent winning strategy exists, then it exists a winning strategy in the starting multi-player game with incomplete information. Due to the fact that in (36) it is proven the undecidability of the multiplayer with incomplete information, in (4) the authors have:

Theorem 3. *Solving a modular game with persistent strategy is undecidable.*

4.5 Solving modular games with regular winning conditions

In (5) games on recursive game graphs with winning conditions given as an ω-regular specification and with the requirement that the strategy must be modular are consid-

ered.

The first considered case is when the specification is given as a deterministic Büchi automaton and the problem is shown to be decidable. The same holds if specifications given as universal co-Büchi automata or LTL formulas are considered. For the deterministic Büchi automaton case, the presented solution is an automata-theoretic construction. First, the authors introduce the definition of *strategy tree*, a tree that encode modular strategies in the RGG. There exists an effectively constructible tree automaton linear in the size of the game that recognizes strategy tree. Then the authors describe the construction of a two way alternating tree automaton, that accepts winning strategy trees.

In the following paragraphs we describe the strategy trees and the details of the construction. Such approaches and results will be evolved in our following works to solve different synthesis problems.

Strategy trees A strategy tree intended to encode a modular strategy. In a strategy tree, the special label *root* is associated with the root of the tree. The children of the root are labeled with the entries of each module in G. The subtree rooted in i^{th} of these nodes corresponds to the unrolling of the i^{th} module. The root of such subtrees is labelled by the entry point and the symbol $\top$. The subtree for a module m encodes the strategy for a module m, unrolling the module and annotating the nodes with the corresponding name of the RGG vertex and $\top$ or $\bot$. Such symbols encode if a move to the corresponding node is possible or not.

If a vertex x of a subtree for m is labelled with $(u, \top)$, then:

- If $u \in (N_m \backslash Ex_m) \cup Retns_m$, the children of x are labelled by the successors of u along with a $\top/\bot$ annotation. Further, if $u \in P^0$, i.e. is a pl_0 vertex, then we must choose one successor of u, that corresponds to the move selected by the strategy, and annotate it with the symbol $\top$. If $u \in P^1$, that all the successors are enabled and this means that all children of x are labelled with $\top$

- If $u \in Calls_m$, the call to the other module is not unrolled and the vertex x has its children that correspond to the returns from the called module. The annotation $\top/\bot$ encodes that the call to the other module will end in that return (if tagged with $\top$) or the call will definitely not end in that return (if labelled with $\bot$)

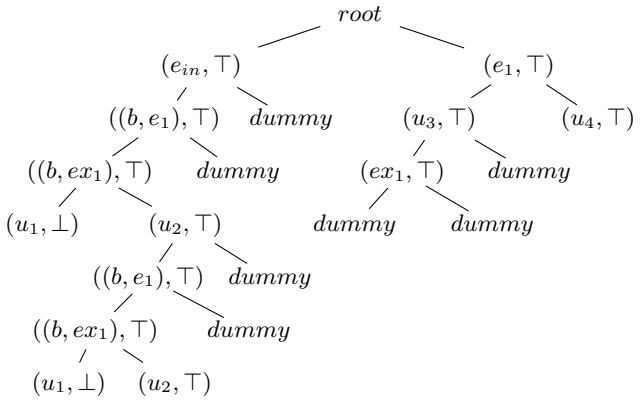

Figure 4.2: A fragment of a strategy tree T_{smpl}.

If a tree-vertex is labelled $(u, \bot)$, this denotes a move that is disabled by the strategy. The successors of this tree-vertex do not encode any strategy and are labelled with *dummy*.The *dummy* nodes are also used to complete the k-tree.

In Figure 4.2 we show the top fragment of a strategy tree T_{smpl} for pl_0 of the RGG from Figure 4.1.

Proposition 4. *There exists an effectively constructible Büchi (resp. co-Büchi) tree automaton of size $O(|G|)$ that accepts a Ω_G-labeled k-tree if and only if it is a strategy tree.*

Recognizing winning strategy tree The second step is define the algorithm to construct the two-alternating tree automaton $\mathcal{A}_{win}$ such that the automaton $\mathcal{A}_{win}$ accepts a strategy tree if and only if it encodes a winning modular strategy. The automaton $\mathcal{A}_{win}$ must guarantee the consistency of the guess done on the reachable/unreachable exits in the strategy tree, i.e. that the exits associated with a tree-vertices that are marked with the symbol $\bot$ are avoided by the strategy encoded by the strategy tree. Moreover, the construction must ensure that all the possible plays, that can be executed according to the encoded strategy, are winning according to the given specification. The automata-theoretic construction is presented considering specification given by a safety automaton, but it can be extended to specification given as deterministic Büchi or co-Büchi automata, universal Büchi or co-Büchi automata and LTL formulas.

Safety automata Consider an RGG $G = (M, m_{in}, \{S_m\}_{m \in M})$ and a safety automaton $\mathcal{A} = (Q, q_0, \delta_A)$. The authors construct an alternating tree automaton $\mathcal{A}_{win}$ that:

- i-Simulates $\mathcal{A}$ to ensure that the specification is satisfied by the strategy tree
- ii-Ensures that the plays are generated from the entry point of the initial module and they are infinite (on any finite play $pl_0 loses$)
- iii-Guarantees that if a return is marked with $\perp$, the calls to the module will definitely not end in such return

The fulfilment of these tasks is obtained using avoid components. An *avoid component* is a couple $(u, q) \in Ex \times Q$, and corresponds to the assumption that a play must not end at the exit u with the specification state q. Another kind of avoid component is denoted by the symbol $\$_m$ and it corresponds to the assumption that no play must exit from the current invocation of the module m. In $\mathcal{A}_{win}$, the correct fulfilment of the task is guaranteed by the following behaviours:

- i-The transition function $\delta_{\mathcal{A}}$ is simulated, whenever a node is read. If an exit is read, the automaton must check the assumption on reachable/unreachable exit, using the set of avoid components.
- ii-From the first transition the exits of the module m_{in} are never reached and this mean that the automaton considers as avoid component each couple (u, q) such that $u \in Ex_{m_{in}}$ and $q \in Q$
- iii-Using its alternating behaviour, the automaton checks the requirement sending a copy of itself to the root of the subtree corresponding to the called module and verifying that the exit that corresponds to the unreachable return, will be

The winning condition for $\mathcal{A}_{win}$, is a parity condition, and the colouring of the states is 0 for all of them. The automaton $\mathcal{A}_{win}$ s converted in a one-way nondeterministic tree automaton, with an exponential blow-up, and its intersection with the automaton that recognizes strategy tree accepts a strategy tree if and only if it encodes a winning strategy for the considered game. Since the EXPTIME -hardness can be proved using a direct reduction from linear-space Turing machines and, combining with Proposition 12, the following holds:

Theorem 5. *For an RGG G and a safety automaton $\mathcal{A}$, deciding if pl_0 has a winning modular strategy in such game is* EXPTIME *-complete.*

The authors extend the results of this main construction to games with deterministic/universal Büchi or co-Büchiautomata. The main modification to handle deterministic Büchi automaton (the co-Büchi is dual) is provide a way to expose, when a call that will return will be executed, if in the all possible plays of the caller module that reaches a specific exit, a final state of the specification automata is visited or not. This goal is achieved forcing $\mathcal{A}_{win}$, when the automaton sends a copy to simulate the call that is guessed to visit a final state, to signal a Büchi final state before continuing the play in the current module. If a universal Büchi or co-Büchi specification is considered, in the construction of the automaton when we were updating the specification state, the automaton creates a copy of itself for each possible update of the specification state. Them, the authors get:

Theorem 6. *Deciding recursive game graph with deterministic/universal Büchi /co-Büchi automaton is* EXPTIME *-complete.*

From (45) we know that, given an LTL formula φ, it is possible to construct a nondeterministic Büchi automaton, and its size is exponential in the size of the formula, that accepts a word if and only if it satisfies $\neg\varphi$. Combining this construction and using the Theorem 5 and recalling the 2EXPTIME-hardness of the LTL games for normal graphs, the authors have:

Theorem 7. *Deciding recursive game graphs with specification given as* LTL *formula is 2*EXPTIME*-complete.*

5

Visibly modular pushdown games

In Chapter 4 we have recalled that the results on the decidability problems connected the existence of a modular strategy in a recursive game graph. This results has been studied only with respect to ω-regular specifications. However in recursive systems many interesting properties are expressed by non-regular specifications. In the following sections, we introduce and solve modular games $\langle G, L \rangle$ where G is an RGG and L is a winning condition given by pushdown, visibly pushdown automata and by LTL, CARET or NWTL formulas.

5.1 Contribution

The main contributions present in this chapter are:

- We show a polynomial time reduction from the MVPG problem with deterministic or universal VPA specifications to recursive modular games over ω-regular specifications. By (4), we get that this problem is EXPTIME-complete. We then use this result to show the membership to 2EXPTIME for the MVPG problem with nondeterministic VPA specifications.
- We show that when the winning condition is expressed as a formula of the temporal logics CARET(3) and NWTL(8) the MVPG problem is 2EXPTIME-complete, and hardness can be shown also for very simple fragments of the logics. In particular, we show a 2EXPTIME lower bound for the fragment containing only conjunctions of disjunctions of bounded-size path formulas (i.e., formulas expressing either the requirement that a given finite sequence is a subsequence of a word or

its negation), that is in contrast with the situation in finite game graphs where PSPACE-completeness holds for larger significant fragments (see (9, 10)). On the positive side, we are able to show an exponential-time algorithm to decide the MVPG problem for specifications given as conjunctions of temporal logic formulas that can be translated into a polynomial-size VPA (such formulas include the path formulas).

- We also give a different solution for recursive games with finite-state automata specifications. Our approach yields an upper bound of $|G|\, 2^{O(d^2(k+\log d)+\beta)}$ for the MVPG problem, where d is the number of P (the VPA) states, k is the number of G (the RGG) exits, and β is the number of *call edges* of G, i.e., the number of module pairs (m, m') such that there is a call from m to m'. The known solution (4) yields an $|G|\, 2^{O(kd^2 \log(kd))}$ upper bound. Thus, our solution is faster when k and d are large, and matches the known EXPTIME lower bound (4). In addition we use one-way nondeterministic/universal tree automata instead of two-way alternating tree automata, thus we explicitly handle aspects that are hidden in the construction from (4).

5.2 Pushdown modular games

Pushdown modular game A *pushdown modular game* is a pair $\langle G, \mathcal{P} \rangle$ where G is an RGG and $\mathcal{P}$ is a pushdown automaton, whose accepted language defines the winning condition in G.

Undecidability of pushdown specification The modular game problem becomes undecidable if we consider winning conditions expressed as standard (deterministic) pushdown automata. This is mainly due to the fact that the stack in the specification pushdown automaton is not synchronized with the call-return structure of the recursive game graph.

We prove the undecidability of our problem with pushdown specification by presenting a reduction from the problem of checking the emptiness of the intersection of two deterministic context-free languages.

Consider two deterministic context-free languages L_1 and L_2 on an alphabet Σ which are accepted by two pushdown automata $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively. We want to

construct an instance $\langle G, \mathcal{P} \rangle$ of a deterministic pushdown modular game problem such that a winning modular strategy for pl_0 exists in $\langle G, \mathcal{P} \rangle$ if and only if the intersection of L_1 and L_2 is not empty.

The basic idea of the reduction is to construct a game where pl_1 challenges pl_0 to generate a word from either L_1 or L_2, and pl_0 must match the choice of pl_1 without knowing it in order to win. We construct an RGG G with two modules m_{in} and m (see the Figure 5.1).

The module m_{in} is the main module and is composed of an entry e_{in}, two internal nodes u_1 and u_2, and one box b labeled with m. The entry e_{in} belongs to pl_1 and has two transitions, one to each internal node. From u_1 and u_2 there is only one possible move that leads to b. The labeling function associates the symbol a_1 to u_1 and the symbol a_2 to u_2 with $a_1, a_2 \notin \Sigma$. The node e_{in} and the call (b, e_m) are both labeled with $\sharp \notin \Sigma \cup \{a_1, a_2\}$. Observe that since the only choice of pl_1 is at e_{in}, for any strategy f of pl_0 there are only two plays conforming to it: one going through u_1 and the other through u_2.

The module m is essentially a deterministic generator of any word in $\{\sharp\}.\Sigma^*.\{\sharp\}$. The module m has one entry e_m, n internal nodes $v_1, \ldots, v_n$ and a sink node s (i.e., a node with only ingoing edges). All the vertices of m belong to pl_0. There are only outgoing edges from e_m, which go to each of the other vertices of m. Moreover, there is a transition from v_i to v_j for any $i, j \in [n]$, and from any node there is a move to s. Each node v_i is labeled with σ_i, for $i \in [n]$. The symbol $\sharp$ labels e_m and s.

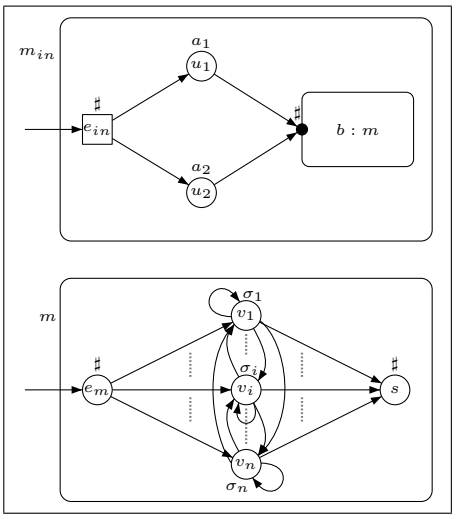

Figure 5.1: The module m_{in} and m

As winning condition, we construct a deterministic pushdown automaton $\mathcal{P}$ that is constructed on the top of the disjoint union of $\mathcal{P}_1$ and $\mathcal{P}_2$ as follows. We add new states $q_0, q_1, q_2, q_1', q_2'$, and an initial state q_{st} and a final state q_f. We take q_f as the only final state. From q_{st}, $\mathcal{P}$ reads $\sharp$ and moves into q_0. For $i \in [2]$, from q_0 and on input a_i, $\mathcal{P}$ enters q_i, and then after reading two occurrences of $\sharp$, enters the initial state of $\mathcal{P}_i$ (stepping through q_i'). From any $\mathcal{P}_i$ state,

$\mathcal{P}$ behaves as $\mathcal{P}_i$ and in addition, from each final state of $\mathcal{P}_i$, it has a move on input $\sharp$ that goes to q_f. Thus, $\mathcal{P}$ accepts the language $(\{\sharp\, a_1\, \sharp\, \sharp\}.L_1.\{\sharp\}) \cup (\{\sharp\, a_2\, \sharp\, \sharp\}.L_2.\{\sharp\})$.

Since the strategy must be modular, in the module m player pl_0 has no information about the choice of pl_1 in m_{in}. Also, the local strategy in module m generates one specific word (there are no moves of pl_1 allowed in m) and this is the same independently of the moves of pl_1 in module m_{in}. Thus, in order to win, pl_0 must generate a word in the intersection of L_1 and L_2, and therefore, the following theorem holds.

Theorem 8. *The (deterministic) pushdown modular game problem is undecidable.*

5.3 Solving modular games with VPA specifications

Visibly pushdown modular games A *visibly pushdown game* on an RGG (VPRG) is a pair $\langle G, P\rangle$ where G is a recursive game graph and P is a visibly pushdown automaton.

Consider a VPRG $\langle G, P\rangle$ where G is an RGG and P is a VPA. The visible trace of a play of an RGG is essentially its trace where each symbol is augmented with the annotation *call*, *ret* or *int* depending on whether the corresponding vertex of G is a call, a return or a node. This trace allows to syncronize the RGG and VPA call-return structure. Formally, for a play $\pi = s_0 s_1 \ldots$ of G, with $s_i = (\alpha_i, u_i)$, we define the *visible trace* of π, denoted v_π, as the word $\sigma_0\sigma_1\ldots$ such that for $i \in \mathbb{N}$, $\sigma_i = (\eta_m(s_i), t_i)$ where $ctr(\pi_i) = m$ and t_i is *call* if $u_i \in Calls$, *ret* if $u_i \in Retns$, and *int* otherwise.

The *visibly pushdown* (*modular*) *game problem* asks to determine the existence of a winning (modular) strategy of pl_0 in a given VPRG such that v_π is accepted by P for every play π that conforms to f. We denote the visibly pushdown modular game problem as the MVPG problem.

Solving games with VPA specifications We consider games with winning conditions that are given by a VPA with different acceptance conditions. We present a reduction from recursive games with VPA specifications to recursive games with specifications that are given as finite state automata.

The reduction is almost independent of the acceptance condition, and works for reachability and safety conditions as well as for Büchi and co-Büchi acceptance conditions. It transforms a recursive game graph with a visibly pushdown automaton

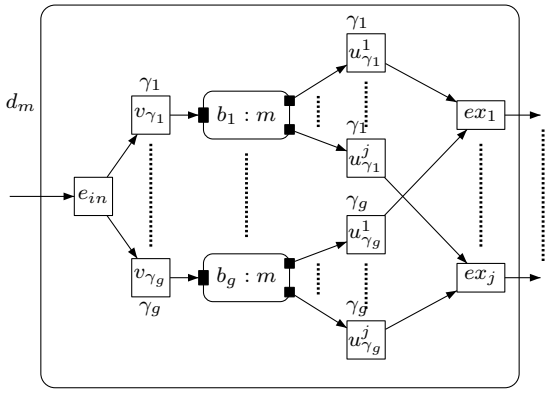

Figure 5.2: The module d_m.

specification (with some acceptance condition) to a slightly different recursive game graph with a finite-state automaton specification (with the same kind of acceptance condition).

The key idea is to embed the top stack symbol of a VPA P in the states of a finite-state automaton A. In addition, the states of A will simulate the corresponding states of P and thus we will get that the winning conditions are equivalent. Clearly, a finite-state automaton cannot simulate an unbounded stack: while it is easy to keep track of the top symbol after a *push* operation, extracting the top symbol after a *pop* operation requires unbounded memory.

To keep track of the stack of P in the RGG, we exploit the fact that the stacks of the VPA P and the game graph G are synchronized. Thus, we introduce a new *dummy* module d_m for every module in G and replace every invocation of m by a call to d_m (recall that an invocation of a module m in G corresponds to a *push* operation in P).

Denote by $\gamma_1, \ldots, \gamma_g$ the stack symbols of P and let m be a module with j exits. In d_m (see Figure 5.2), pl_1 first has to *declare* the top stack symbol in P by choosing a node among $v_{\gamma_1}, \ldots, v_{\gamma_g}$, and A can verify that pl_1 is honest since it keeps track of the current top symbol (if the player is not honest then A goes to a sink accepting state and pl_1 loses). After this declaration, the module invokes the actual module m and when m terminates, we must restore the symbol that was at the top of the P stack before the call to m. This is done by forcing a visit of vertex $u^l_{\gamma_i}$ on returning at the l-th return of b_i and letting A change the stored symbol accordingly.

Denoting with k the overall number of exits of the starting RGG, with g the number of stack symbols of P, and d the number of states of P, we get that the resulting RGG G has $2k$ exits and the resulting finite automaton A has $O(dg)$ states. Thus combining this with the algorithm from (4), we get an upper bound linear in $|G|$ and exponential in $2k(d\,g)^2 \log(2\,k\,d\,g)$, and hence we get:

Theorem 9. *The MVPG problem with winning conditions expressed as a deterministic Büchi or co-Büchi VPA is* EXPTIME*-complete.*

The proposed reduction can be extended to handle universal VPA specifications. W.l.o.g we can assume that in the VPA for every state, stack symbol and input symbol there are exactly two possible transitions. Then, we add a dummy module d', that comprises only pl_1 nodes and has only one exit. Each transition from a node v to a node u is split into two transitions, $v \to c_{uv}$ and $r_{uv} \to u$ where c_{uv} and r_{uv} are respectively the call and the return of a new box that is mapped to d'. In module d', pl_1 selects one of the two possible transitions for the VPA specification and exits. The choices of pl_1 in d are oblivious to pl_0. Hence, the universal VPA accepts if and only if pl_0 has a strategy that wins against all pl_1 choices in d', and we get:

Theorem 10. *The MVPG problem with winning conditions expressed as a universal Büchi or co-Büchi VPA is* EXPTIME*-complete.*

We can handle nondeterministic VPAs in the following way: Let P be a nondeterministic Büchi VPA P. By (6), we can construct a nondeterministic Büchi VPA P' that accepts a word w iff P does not accept it, and such that the size of P' is exponential in the size of P. Complete P' with transitions that go to a rejecting state so that P' has at least one run over each word. Let P'' be the dual automaton of P', i.e., P has the same components of P' except that acceptance is now universal and the set of accepting states is now interpreted as a co-Büchi condition. Clearly, P'' accepts exactly the same words as P and has size exponential in $|P|$. Similarly, we can repeat the above reasoning starting from a co-Büchi VPA P. Therefore, we have:

Theorem 11. *The MVPG problem with winning conditions expressed as a nondeterministic Büchi or co-Büchi VPA is in* 2EXPTIME.

5.4 Improved tree automaton construction

In this section, we give a new solution to the modular synthesis problem with winning conditions expressed as standard Büchi or co-Büchi automata (i.e., finite automata equipped with a Büchi or co-Büchi acceptance).

For the notions of tree and automata accepting trees, we refer the reader to Chapter 1.

5.4.1 General structure of the construction

For the rest of this section we fix a modular game $\langle G, \mathcal{B} \rangle$ where $\mathcal{B} = (Q, q_0, \Sigma, \delta, F)$ is a deterministic automaton and $G = (M, m_{in}, \{S_m\}_{m \in M})$ is an RGG (for each S_m we use the same notation as in Definition 1).

For $\mathcal{B}$ we consider both a Büchi and a co-Büchi acceptance condition, and depending on this, we construct a Büchi or a co-Büchi tree automaton $\mathcal{A}_{G,\mathcal{B}}$ that accepts a tree if and only if pl_0 has a winning modular strategy in the game $\langle G, \mathcal{B} \rangle$. We will give a single construction and point out the differences between the Büchi and the co-Büchi cases whenever this will be needed.

The trees accepted by $\mathcal{A}_{G,\mathcal{B}}$ must encode G and a modular strategy on it (*strategy trees*). Each such tree essentially has a subtree rooted at a child of the root for each module of G and each such subtree is the unwinding of the corresponding module along with a labeling that encodes a local function.

Assume that the input is a strategy tree T, $\mathcal{A}_{G,\mathcal{B}}$ nondeterministically guesses: a) a call graph CG that expresses a call relation for the modules and marks a subset of its edges as meaningful for acceptance, and b) an extended pre-post requirement $\mathcal{C} = \langle \mathcal{C}_{pre}, \mathcal{C}_{post}, Fin \rangle$ which summarizes the effects of $\mathcal{B}$ executions in each module of G as a relation of states at the module entry and states at each of its exits (Fin just tells whether any state state in the acceptance set is visited).

Then, $\mathcal{A}_{G,\mathcal{B}}$ checks that the guessed call graph CG and extended pre-post requirement $\mathcal{C}$ are consistent with T and the specification $\mathcal{B}$, and by using $\mathcal{C}$ and CG, that the traces of all the plays of the strategy encoded in T are accepted by $\mathcal{B}$.

We split the construction of $\mathcal{A}_{G,\mathcal{B}}$ into an automaton $\mathcal{A}_G$ which checks that the input tree is a valid strategy tree for G, and an automaton $\mathcal{A}_{\mathcal{B},\mathcal{C},CG}$ for each guessed CG and $\mathcal{C}$, for checking the remaining properties.

In the rest of this section, we give more details on these automata and argue their correctness according to the stated properties.

5.4.2 Strategy trees and the automaton $\mathcal{A}_G$

Denote with $\#(u)$ the out-degree of a vertex u. Let k be the maximum over the number of exits of G modules and the out-degree of G vertices. Denote with Ω_G the set $\{dummy, root\} \cup (V \setminus P^0) \cup \Omega'$ where $\Omega' = \{(u,i) | u \in P^0, i \in [\#(u)]\}$ (recall V and P^0 denote respectively the set of vertices and pl_0 vertices of G). Note that $|\Omega_G| = O(|G|)$.

We construct $\mathcal{A}_G$ s.t. it recognizes the set of strategy trees for G, i.e. the set of Ω_G-labeled k-trees that encode modular strategies of pl_0.

Strategy trees Intuitively, in a strategy tree, the label *root* is associated with the root of the tree. The children of the root are labeled with the entries of each module in G. A subtree rooted in one of these vertices corresponds to the unrolling of a module. If a vertex is labeled with a node that belongs to pl_0, the move according to the encoded strategy is annotated with the index of the selected successor. If a node is associated to a call, then its children are labeled with the matching returns. The *dummy* nodes are used to complete the k-tree.
Formally, an Ω_G-labeled k-tree T is a *strategy tree* of G (for player pl_0) if:

- the root of T is labeled with *root*;
- for $i \in [|M|+1, k]$, the i^{th} child of the root is labeled with *dummy*;
- for $i \in [1, |M|]$, the subtree T_i rooted at the i^{th} child of the root corresponds to an unrolling of module m_i; the nodes of T_i are labeled with the corresponding vertices of the module m_i; thus, in particular, the root of T_i is labeled with e_{m_i} and the calls have as children the matching returns;
- all the other nodes are labeled with *dummy*, meaning that they are not meaningful in the encoding; in particular, for each T_i all the descendants of the nodes labeled with an exit of m_i and all the other nodes that do not correspond to a vertex in the unrolling of m_i are labeled with *dummy*;

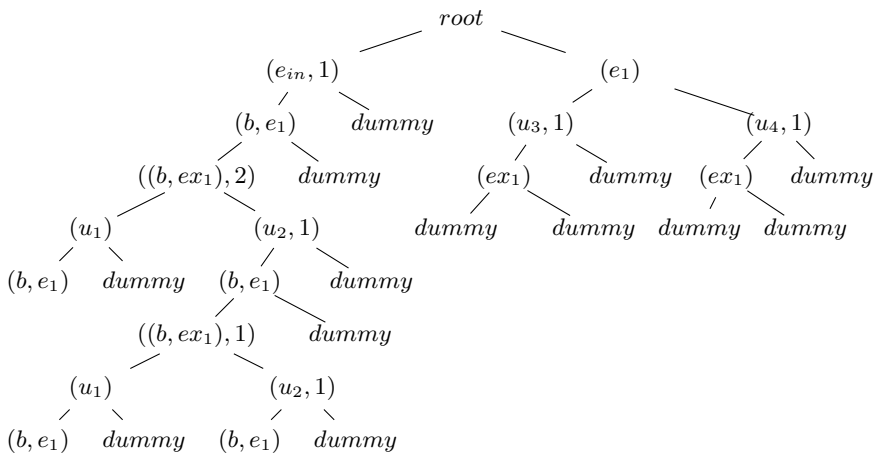

Figure 5.3: A fragment of a strategy tree T_{smpl}.

- each node x labeled with a vertex of pl_0 is also labeled with $i \in [k]$ such that $x.i$ is not labeled with *dummy*, and $x.i$ is the *selected* child of x (selected by the encoded strategy).

In Figure 9.1 we show the top fragment of a strategy tree T_{smpl} for pl_0 of the RGG from Figure 4.1. Module m_{in} is unrolled from the first child of the root and m_{ad} from the second one. Note that from the dummy leaves of the fragment, T_{smpl} contains only dummy nodes, and from the remaining leaves it continues with the unrolling of the corresponding module. In particular, T_{smpl} can be obtained from the considered fragment by iteratively replacing the non-dummy leaves with any finite subtree rooted at an internal node labelled with (b, e_1) in Figure 9.1 where we possibly vary the selected move at the nodes labelled with (b, e_1) (which is the only vertex of the considered RGG where pl_0 has more that just one alternative).

Regularity of strategy trees Given a tree T, the automaton $\mathcal{A}_G$ is constructed s.t. it accepts T iff T is a strategy tree for G. $\mathcal{A}_G$ can be easily constructed from G, and thus we omit it (see (4) for a similar construction).

Proposition 12. *There exists an effectively constructible Büchi (resp. co-Büchi) tree automaton of size $O(|G|)$ that accepts a Ω_G-labeled k-tree if and only if it is a strategy tree.*

Strategy trees and modular strategies Directly from the definitions, one can show that a strategy tree identifies a modular strategy.

Proposition 13. *For an RGG G, there exists a one-to-one mapping between the modular strategies of G and the strategy trees of G.*

From the above proposition, the modular strategy corresponding to a strategy tree T is well defined and we will refer to it as the T-strategy in the rest of Section 5.4.

To identify the plays of the T-strategy in the game, we introduce the notion of play of T. For this, we denote with α_i the call stack at the i^{th} step, i.e., the stack of the nodes corresponding to the unmatched calls within the π prefix up to x_i.

For a strategy tree T of G, a *play of T from module m* is an ω-sequence of T-nodes $x_1x_2\dots$ such that x_1 is the child of the root corresponding to the entry of m, $\alpha_1 = \varepsilon$ (*call-stack*) and for $i \in \mathbb{N}$: (1) if x_i is labeled with a call to m', then x_{i+1} is the child of the T root corresponding to module m' (and thus is labeled with the entry $e_{m'}$), and $\alpha_{i+1} = \alpha_i.x_i$; (2) if x_i is labeled with an exit ex and $\alpha_i = \alpha_{i+1}.x_j$ (with $j < i$), then x_j is a node labeled with a call (b, e_m) and x_{i+1} is the child of y labeled with the return (b, ex); (3) otherwise, $\alpha_{i+1} = \alpha_i$ and x_{i+1} is the selected child of x_i, if x_i is labeled with a pl_0 vertex, and any child of x_i in all the other cases.

As example, considering the fragment of the strategy tree T_{smpl} proposed in Figure 9.1. A play π_1 from the module m_{in} starts at the node labelled $(e_{in}, 1)$, then continues with the node labelled with the call (b, e_1), say x, then jumps to the second child of the root of T_{smpl} (x is pushed on the call stack) and descends on the leftmost path up to the node labelled with exit ex_1 (note that the second child of the root is labelled with a pl_1 vertex thus also descending on its second child would give a play), then jumps back to the first child $x.1$ of x (which is popped from the call stack), then continues on its second child ($x.1$ is labelled with a return from ex_1 and the encoded strategy at this point selects its second child), and so on.

In the following, when we refer to any play of T from any module m we will omit m. Also, when the case (2) above applies for a node x_j we say that along $x_1x_2\dots$ the x_j is a *returned call*, and is an *unreturned call* otherwise. Further, the labels of a play π of a strategy tree T identifies a sequence of G vertices, we refer to a *trace of π* as the trace of this sequence of G vertices. As example, the trace of G vertices along π_1 is $e_1(b, e_1)e_1u_3ex_1(b, ex_1)u_2...$ and the trace of π_1 is $p_e^3p_ap_e^2p_d...$.

From the definitions, it is simple to verify that a play π of a strategy tree from m_{in} identifies a play of G (given by the sequence of π labels).

Let $\pi = x_1 x_2 \dots$ be any play of a strategy tree T from m as above.

We say that x_i and x_j *correspond to the same invocation* of a module, if $\alpha_i = \alpha_j$ (same call-stack) and α_i is a prefix of α_l for all $l \in [i, j]$. A *module play* of T from m is any prefix π' of a play π from m s.t. π' ends at a node x_j, and x_1 and x_j correspond to the same invocation of m. For example, the portion of the play π_1 described before is indeed a module play from m_{in}, instead any prefix of this module play up to a node in the subtree corresponding to the unrolling of m_{ad} is not a module play.

We observe that the sequence of labels of a play either is an ω-sequence of vertices of G or has a suffix formed of only occurrences of symbol *dummy*. In the first case, we say that the play is *non-terminating*. Note that π_1 is non-terminating. A terminating play is the leftmost path of the subtree corresponding to the unrolling of m_{ad} in T_{smpl}.

5.4.3 The automaton $\mathcal{A}_{\mathcal{B},\mathcal{C},CG}$

We start introducing the notions of pre-post requirement and call graph.

Pre-post requirements A *pre-post requirement* on the graph G is a pair $\langle \mathcal{C}_{pre}, \mathcal{C}_{post} \rangle$ where $\mathcal{C}_{pre} \subseteq M \times Q$ (set of *pre-conditions*), $\mathcal{C}_{post} \subseteq M \times Q \times Ex \times Q$ (set of *pre-post conditions*), and such that for each $(m, q, ex_j, q') \in \mathcal{C}_{post}$, also $(m, q) \in \mathcal{C}_{pre}$ (i.e., tuples of $\mathcal{C}_{post}$ add a post-condition to some of the pre-conditions of $\mathcal{C}_{pre}$).

Intuitively, for a strategy tree T of G, a pre-post requirement is meant to cover all the states q of $\mathcal{B}$ that can be reached on entering each module m of G along any play of T, and for each reachable exit ex of a module m and each such state q, all the pairs (q, q') of $\mathcal{B}$ states s.t. there exists a play of T from m where $\mathcal{B}$ starts at q and exits m from ex at q'.

An *extended pre-post requirement* $\mathcal{C} = \langle \mathcal{C}_{pre}, \mathcal{C}_{post}, Fin \rangle$ is a pre-post requirement $\langle \mathcal{C}_{pre}, \mathcal{C}_{post} \rangle$ along with a function $Fin : \mathcal{C}_{post} \to \{true, false\}$.

We now formalize the intended meaning of the extended pre-post requirements by the notion of consistency.

For a play $\pi = x_1 x_2 \dots$ denote with $\rho_\pi(q)$ the only run of $\mathcal{B}$ over w_π starting from q (recall $\mathcal{B}$ is deterministic).

An extended pre-post requirement $\mathcal{C} = \langle \mathcal{C}_{pre}, \mathcal{C}_{post}, Fin \rangle$ is *consistent with a strategy tree* T if for each play $\pi = x_1 x_2 \dots$ of T from module m, for each $(m, q) \in \mathcal{C}_{pre}$, and for each x_i that is labeled with a call to module m', denoting q' the state visited in $\rho_\pi(q)$ after reading the label at x_i:
(1) (m, q') belongs to $\mathcal{C}_{pre}$ and (2) if $\pi' = x_{i+1} \dots x_j$ is a module play for m', x_j is labeled with exit ex (of m') and q'' is the ending state in $\rho_{\pi'}(q')$, then (m', q', ex, q'') belongs to $\mathcal{C}_{post}$.

An extended pre-post requirement $\mathcal{C} = \langle \mathcal{C}_{pre}, \mathcal{C}_{post}, Fin \rangle$ is *consistent with* $\mathcal{B}$ *acceptance* if for each module play $\pi = x_1 x_2 \dots x_j$ where x_j is labeled with exit ex, denoting q' the end state of $\rho_\pi(q)$:

1. for a Büchi automaton $\mathcal{B}$, we require that a state in F must be visited along $\rho_\pi(q)$ whenever $Fin(m, q, ex, q') = true$;

2. for a co-Büchi automaton $\mathcal{B}$, we require that no state in F is visited along $\rho_\pi(q)$ whenever $Fin(m, q, ex, q') = false$.

An extended pre-post requirement that is consistent with T and $\mathcal{B}$ acceptance is said to be $(T, \mathcal{B})$-*consistent*.

To illustrate the above definitions, we enrich our running example with a specification automaton. Let $\mathcal{B}_{smpl}$ be the Büchi deterministic automaton shown in Figure 5.4, where q_0 is the starting state and q_2 is the only state in the acceptance set. This automaton accepts the language of all ω-words where both p_c and p_d occur infinitely often. We define the extended pre-post requirement $\mathcal{C}' = \langle \mathcal{C}'_{pre}, \mathcal{C}'_{post}, Fin' \rangle$ on the RGG from Figure 4.1 as follows: $\mathcal{C}'_{pre} = \{(m_{in}, q_0), (m_1, q_0), (m_1, q_1)\}$, $\mathcal{C}'_{post} = \{(m_1, q_0, ex_1, q_0),\ (m_1, q_1, ex_1, q_0)\}$ and $Fin'(m_1, q_0, ex_1, q_0) = Fin'(m_1, q_1, ex_1, q_0) = false$. The pre-post requirement $\mathcal{C}'$ is not consistent with T_{smpl}. Intuitively, when a call to m_1 is executed, at the entry of m_1 $\mathcal{B}_{smpl}$ is in a state q_0 or q_1. All the plays reaches ex_1 and when such vertex is visited, the state of $\mathcal{B}_{smpl}$ is not changed, because in m_1 there is no vertex labeled with p_c or p_d. In particular, if a call to m_1 is executed and the state of $\mathcal{B}_{smpl}$ is q_1 after reading the label of the call, then q_1 is the ending state of $\mathcal{B}_{smpl}$ at ex_1. However, (m_1, q_1, ex_1, q_1) does not belongs to $\mathcal{C}'_{post}$ and this means that

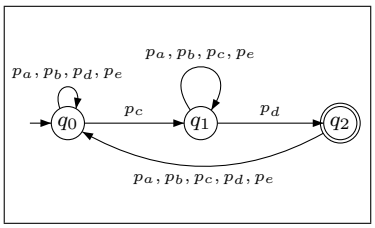

Figure 5.4: The automaton $\mathcal{B}_{smpl}$.

$\mathcal{C}'$ is not consistent with the strategy tree T_{smpl}. Consider a different extended pre-post requirement $\mathcal{C}'' = \langle \mathcal{C}''_{pre}, \mathcal{C}''_{post}, Fin'' \rangle$, where $\mathcal{C}''_{pre} = \mathcal{C}'_{pre}$, $\mathcal{C}''_{post} = \{(m_1, q_0, ex_1, q_0), (m_1, q_1, ex_1, q_1)\}$, $Fin''(m_1, q_0, ex_1, q_0) = \mathit{false}$ and $Fin''(m_1, q_1, ex_1, q_1) = \mathit{true}$. The pre-post requirement $\mathcal{C}''$ is consistent with T_{smpl}, but is not consistent with $\mathcal{B}_{smpl}$ acceptance, because when each play moves across m_1, along each run $\mathcal{B}_{smpl}$ visits or always q_0 or always q_1, and such states are not in the acceptance set of $\mathcal{B}_{smtp}$. If we consider $\mathcal{C}''' = \langle \mathcal{C}''_{pre}, \mathcal{C}''_{post}, Fin''' \rangle$, where $Fin'''(m_1, q_0, ex_1, q_0) = Fin'''(m_1, q_1, ex_1, q_1) = \mathit{false}$, such pre-post requirement is $(T_{smpl}, \mathcal{B}_{smpl})$-consistent.

Call graphs A call graph is a directed graph that captures the sequences of states at the unreturned calls in the runs of $\mathcal{B}$, and is a useful abstraction to deal with the $\mathcal{B}$ acceptance over traces of plays with infinitely many unreturned calls. Intuitively, a play π of this kind can be seen as the concatenation $\pi_1\pi_2\ldots$ of infinitely many portions, where each π_i starts at the entry of a module m_i and ends at an unreturned call to a module m_{i+1} in the same invocation of m_i. Thus, we can abstract the run ρ of $\mathcal{B}$ over the trace of π, by replacing each portion ρ_i of ρ over each π_i with an edge $(m_i, q_i) \to (m_{i+1}, q_{i+1})$ where q_i and q_{i+1} are respectively the starting and ending states of ρ_i, and reporting also if a state of the acceptance set of $\mathcal{B}$ is visited along ρ_i. This will suffice to witness the existence of ρ and check the fulfillment of the acceptance condition of $\mathcal{B}$ for plays with infinitely many unreturned calls. We formalize this intuition below.

A *call graph* of G is a directed graph $CG = (U, \to, \overset{\checkmark}{\to})$ where $U \subseteq M \times Q$ is the set of vertices, $\to \subseteq U \times U$ is the set of edges and $\overset{\checkmark}{\to} \subseteq \to$ denotes a subset of marked edges.

Fix a call graph $CG = (U, \to, \overset{\checkmark}{\to})$ and a strategy tree T.

A *call witness* of T from a module m to a module m' is a module play from m that ends with a node labeled with a call to m'. For example, the module play π_{smpl} corresponding to the trace $e_{in}(b, e_1)$ is a call witness of T_{smpl} from m_{in} to m_1.

Let $\mathcal{C}_{pre}$ be a set of preconditions as above.

We say that CG is $(T, \mathcal{B}, \mathcal{C}_{pre})$-*consistent* if for each call witness π from m to m' and for each $(m, q) \in \mathcal{C}_{pre}$, denoting with ρ the $\mathcal{B}$ run over the trace of π starting from q and ending at q':

1. $(m, q) \to (m', q')$ holds;

2. if $\mathcal{B}$ is a Büchi automaton and $(m, q) \xrightarrow{\checkmark} (m', q')$ also holds, then a state in the acceptance set of $\mathcal{B}$ must be visited in ρ;

3. if $\mathcal{B}$ is co-Büchi automaton and $(m, q) \xrightarrow{\checkmark} (m', q')$ does not hold, then none of the states in the acceptance set of $\mathcal{B}$ must be visited in ρ.

In our running example consider the call graph $CG_{smpl} = (U_{smpl}, \rightarrow, \xrightarrow{\checkmark})$ where $U_{smpl} = \{(m_{in}, q_0), (m_1, q_0), (m_1, q_1)\}$ and $(m_{in}, q_0) \rightarrow (m_1, q_0)$, $(m_{in}, q_1) \rightarrow (m_1, q_1)$, $(m_{in}, q_2) \rightarrow (m_1, q_0)$ hold and $(m_{in}, q_0) \xrightarrow{\checkmark} (m_1, q_0)$ holds. We say that CG_{smpl} is not $(T_{smpl}, \mathcal{B}_{smpl}, \mathcal{C}'_{pre})$-consistent. Consider the module play π_{smpl}. As mention previously, π_{smpl} is a call witness from m_{in} to m_1 and, moreover, we know that $(m_{in}, q_0) \in \mathcal{C}''_{pre}$. It is easy to see that the run ρ of $\mathcal{B}_{smpl}$ over the trace of π starts from q_0, ends at q_0 and no other state is visited. From the definition of CG_{smpl}, we know that $(m_{in}, q_0) \rightarrow (m_1, q_0)$ holds, but also $(m_{in}, q_0) \xrightarrow{\checkmark} (m_1, q_0)$ holds. Due to the fact that ρ of $\mathcal{B}_{smpl}$ visits only q_0 and such state is not in the acceptance set of $\mathcal{B}_{smpl}$, the second rules of consistency is violated and we said that CG_{smpl} is not $(T_{smpl}, \mathcal{B}_{smpl}, \mathcal{C}'_{pre})$-consistent.

Construction of $\mathcal{A}_{\mathcal{B},\mathcal{C},CG}$ The automaton $\mathcal{A}_{\mathcal{B},\mathcal{C},CG}$ is parameterized over the automaton $\mathcal{B}$, an extended pre-post requirement $\mathcal{C}$ and a call-graph CG. It is quite complex and its tasks are:

1. to simulate $\mathcal{B}$ on a strategy tree (it uses $\mathcal{C}$ to update the state of $\mathcal{B}$ when moving from calls to matching returns);

2. to check the correctness of the pre-post requirement $\mathcal{C}$ (i.e., that $\mathcal{C}$ is $(T, \mathcal{B})$-consistent);

3. to check that CG is $(T, \mathcal{B}, \mathcal{C}_{pre})$-consistent;

4. to check the fulfillment of the acceptance conditions of $\mathcal{B}$ on nonterminating plays with finitely many unreturned calls.

We first construct an automaton $\mathcal{A}^{\mathcal{C}}_{\mathcal{B}}$ which ensures task 1. Then on the top of $\mathcal{A}^{\mathcal{C}}_{\mathcal{B}}$ we construct three different automata $\mathcal{A}_{\mathcal{B},\mathcal{C}}$, $\mathcal{A}_{\mathcal{B},CG}$ and $\mathcal{A}_{\mathcal{B}_{win}}$, one for each of the remaining three tasks respectively. We then get $\mathcal{A}_{\mathcal{B},\mathcal{C},CG}$ by taking the usual cross product for the intersection of these automata (note that an efficient construction can be obtained by discarding all the states that do not agree on the $\mathcal{A}^{\mathcal{C}}_{\mathcal{B}}$ part, thus avoiding a

cubic blow-up in the size of $\mathcal{A}^{\mathcal{C}}_{\mathcal{B}}$). Under the assumption that the input tree is a strategy tree, we get that $\mathcal{A}_{\mathcal{B},\mathcal{C},CG}$ accepts only trees T s.t. $\mathcal{C}$ is $(T,\mathcal{B})$-consistent and CG is $(T,\mathcal{B},\mathcal{C}_{pre})$-consistent, and that satisfy the winning condition $\mathcal{B}$ on all the plays that have a finite number of unreturned calls. In the rest of this section we give more details on all these automata.

Construction of $\mathcal{A}^{\mathcal{C}}_{\mathcal{B}}$ Let m_i be the module mapped to the i^{th} child of the root of a strategy tree. Fix an extended pre-post requirement $\mathcal{C} = \langle \mathcal{C}_{pre}, \mathcal{C}_{post}, Fin\rangle$.

We construct a universal automaton $\mathcal{A}^{\mathcal{C}}_{\mathcal{B}}$ to simulate $\mathcal{B}$ on an input strategy tree T by using $\mathcal{C}$. In particular, starting from the i^{th} child, the automaton $\mathcal{A}^{\mathcal{C}}_{\mathcal{B}}$ runs in parallel a copy of $\mathcal{B}$ from each state q such that $(m_i, q) \in \mathcal{C}_{pre}$. When reading a node labeled with a call, $\mathcal{A}^{\mathcal{C}}_{\mathcal{B}}$ starts at each matching return a copy of $\mathcal{B}$ according to the applicable tuples in $\mathcal{C}$ and performs updates according to Fin. On all the other enabled nodes, the state of $\mathcal{B}$ is updated for each copy according to $\mathcal{B}$ transitions.

The states of $\mathcal{A}^{\mathcal{C}}_{\mathcal{B}}$ are: an initial state q_0, an accepting state q_a, a rejecting state q_r, and states of the form $(q, d, f, q_{m_i}, \mathcal{C})$ where $q, q_{m_i} \in Q$, q is the state which is updated in the simulation of $\mathcal{B}$, q_{m_i} is the current pre-condition, and $d, f \in \{0,1\}$ are related to acceptance. Namely, f is used to signal that a state in the acceptance set F of $\mathcal{B}$ was seen between a call and its matching return, and d is used to expose that a state from F occurred since the beginning of the current module invocation. A task of $\mathcal{A}^{\mathcal{C}}_{\mathcal{B}}$ is to handle the correct update of these bits but it does not check if the plays fulfill the acceptance condition of $\mathcal{B}$. This will be the task of $\mathcal{A}_{\mathcal{B}_{win}}$ that will use the bit f for this task. The states q_a and q_r are sinks, i.e., once reached, the automaton cycles forever on them.

We give an informal (though detailed) description of the transition rules. In the description of the transition rules, we omit to refer to children marked with *dummy* and we assume that transition rules mark them with q_a except when the parent is marked with q_r. If not differently stated, in a transition, f is always set to 0 and d keeps its value.

The automaton $\mathcal{A}^{\mathcal{C}}_{\mathcal{B}}$ starts from q_0 and enters on the i^{th} child of the root, for $i \in [|M|]$, a state $(q, 0, 0, q, \mathcal{C})$, for each q such that $(m_i, q) \in \mathcal{C}_{pre}$. If no such tuple exists, meaning that the module is never invoked in this modular strategy, then q_a is entered. (Note that $\mathcal{C}$ is the same for all the states.)

Now, consider a run at a tree-node x. Let $s = (q, d, f, q_m, \mathcal{C})$ be the $\mathcal{A}^{\mathcal{C}}_{\mathcal{B}}$ state at x, $u \in V$ be the G vertex that labels x and m_i be the current module.

If $u \in (N_{m_i} \cup Retns_{m_i}) \setminus Ex_{m_i}$, $u \in P^0$ and $h \in [k]$ labels x, then on its child $x.h$ the component q of s is updated to q' where $(q, \eta_{m_i}(u), q') \in \delta$. Additionally, if $q' \in F$, then also the component d of s is set to 1. On the other children $\mathcal{A}^{\mathcal{C}}_{\mathcal{B}}$ moves to q_a.

If either $u \in Ex_{m_i}$ or $u \in (N_{m_i} \cup Retns_{m_i}) \setminus Ex_{m_i}$ and $u \in P^1$, on each child y of x the component q of s is updated to q' if $(q, \eta_{m_i}(u), q') \in \delta$. Again, if $q' \in F$, then the d component is set to 1.

If $u = (b, e_{m_j}) \in Calls_{m_i}$ (a call to module m_j), let y be the child of x that corresponds to the return from exit ex of m_j:

- If there is a transition $(q, \eta_{m_i}(b, e_{m_j}), q') \in \delta$, $(m_j, q') \in \mathcal{C}_{pre}$ and $ex' \neq ex$ for all tuples of the form $(m_j, q', ex', q'') \in \mathcal{C}_{post}$, then on y the automaton enters q_a.

- for each $(q, \eta_{m_i}(b, e_{m_j}), q') \in \delta$ and $(m_j, q', ex, q'') \in \mathcal{C}_{post}$, the automaton sends on y a copy of $\mathcal{B}$ for each such q'' and the components d and f are set both to 1 if $Fin(m_j, q', ex, q'') = true$ holds, and d stays unchanged otherwise.

- In all the other cases, the automaton moves to q_r. Note that if for each $q' \in Q$, $(m_j, q') \notin \mathcal{C}_{pre}$, i.e., the guess is that m_j should not be called, then the automaton correctly moves to q_r on all the children.

As a Büchi automaton, for $\mathcal{A}^{\mathcal{C}}_{\mathcal{B}}$ we choose as acceptance set the set of all the states except q_r, and as a co-Büchi automaton, we choose $\{q_r\}$ as the acceptance set. Since $\mathcal{C}$ is fixed, the size of $\mathcal{A}^{\mathcal{C}}_{\mathcal{B}}$ is quadratic in $|\mathcal{B}|$ and linear in the number of exits of G.

The following lemma states that $\mathcal{A}^{\mathcal{C}}_{\mathcal{B}}$ simulates $\mathcal{B}$ on the strategy trees provided that $\mathcal{C}$ is consistent with the input.

Lemma 14. *Let $\mathcal{C} = \langle \mathcal{C}_{pre}, \mathcal{C}_{post}, Fin \rangle$ be a pre-post requirement and $\mathcal{B}$ be a deterministic Büchi (resp., co-Büchi) automaton.*
The Büchi (resp., co-Büchi) version of the tree automaton $\mathcal{A}^{\mathcal{C}}_{\mathcal{B}}$ is s.t. if $A^{\mathcal{C}}_{\mathcal{B}}$ accepts a strategy tree T and $\mathcal{C}$ is $(T, \mathcal{B})$-consistent then the following holds.
For each module play π of T from a module m_i that does not end with a node labeled either with a call or with an exit, and $(m_i, q) \in \mathcal{C}_{pre}$:

1. *$\mathcal{A}^{\mathcal{C}}_{\mathcal{B}}$ reaches a state of the form $(q', d, f, q, \mathcal{C})$ at the end of π iff the only run ρ of $\mathcal{B}$ starting at q and over the trace of π ends at q';*

2. further, $d = 1$ iff a state of the acceptance set is visited in ρ.

The size of $\mathcal{A}^{\mathcal{C}}_{\mathcal{B}}$ is quadratic in $|\mathcal{B}|$ and linear in the number of G exits.

Proof. The size of $\mathcal{A}^{\mathcal{C}}_{\mathcal{B}}$ can be determined by a simple counting of the states.

Let π be a module play and q as in the statement of the lemma. Note that π is of the form $\alpha_0\beta_1 \dots \beta_l\alpha_l$ where $\alpha_0 \dots \alpha_l$ are the portions of π that contain only and all the T nodes in π of the current invocation of m_i. Also, each β_i is a module play of T for a different module or another invocation of m_i.

By construction, $A^{\mathcal{C}}_{\mathcal{B}}$ updates the first component of its states of the form $(q', d, f, q, \mathcal{C})$ by mimicking the transitions of $\mathcal{B}$ over each α_i and using $\mathcal{C}$ to jump across each β_i. Since π is a module play, its first node x is the root of the subtree T_i corresponding to m_i and its last node y is a node of this subtree.

Now, suppose that the run of $\mathcal{B}$ over the trace of π and starting at q ends at q'. Since $\mathcal{C}$ is consistent with T and at a node of subtree T_i a state $(p', d', f', p, \mathcal{C})$ can be reached only starting from a state $(p, 0, 0, p, \mathcal{C})$ at x, we get that: $A^{\mathcal{C}}_{\mathcal{B}}$ reaches a state of the form $(q', d, f, q, \mathcal{C})$ at the successor of y selected by the T-strategy, whenever y is marked with a pl_0 vertex of G, and at all the successors, otherwise. Conversely, if $A^{\mathcal{C}}_{\mathcal{B}}$ reaches a state of the form $(q', d, f, q, \mathcal{C})$ at the end of π (i.e., at the successors of the last node as before), we can define a run of $\mathcal{B}$ from the transitions of $A^{\mathcal{C}}_{\mathcal{B}}$ on the α_i portions, and using the fact that $\mathcal{C}$ is $(T, \mathcal{B})$-consistent, over the β_i portions. Clearly, the resulting run of $\mathcal{B}$ is from q to q'.

Further, observe that the d component of $A^{\mathcal{C}}_{\mathcal{B}}$ states gets updated to 1 in the transitions as soon as a state in the acceptance set of $\mathcal{B}$ is met along any α_i or this is signaled by the pre-post requirement $\mathcal{C}$ over any β_i. Moreover, once it is set to 1, this component is never reset. Thus, since $\mathcal{C}$ is consistent with $\mathcal{B}$ acceptance we get that a state in the acceptance set of $\mathcal{B}$ is visited along ρ iff $d = 1$ holds. □

Construction of $\mathcal{A}_{\mathcal{B},\mathcal{C}}$ The automaton $\mathcal{A}_{\mathcal{B},\mathcal{C}}$ is in charge of checking that $\mathcal{C}$ is consistent with the input tree and the automaton $\mathcal{B}$. We construct it from $\mathcal{A}^{\mathcal{C}}_{\mathcal{B}}$ by modifying the transitions from a state of the form $(q', d, f, q, \mathcal{C})$ at tree-nodes labeled with an exit.

Suppose the automaton reaches a state $(q', d, f, q, \mathcal{C})$ at a node labeled with the exit ex of module m.

For a Büchi automaton $\mathcal{B}$, $\mathcal{A}_{\mathcal{B},\mathcal{C}}$ enters q_a, whenever $(m, q, ex, q') \in \mathcal{C}_{post}$ and if $Fin(m, q, ex, q') = true$ then also $d = 1$ must hold. Otherwise $\mathcal{A}_{\mathcal{B},\mathcal{C}}$ enters q_r.

For a co-Büchi automaton $\mathcal{B}$, $\mathcal{A}_{\mathcal{B},\mathcal{C}}$ enters q_a, whenever $(m, q, ex, q') \in \mathcal{C}_{post}$ and if $Fin(m, q, ex, q') = false$ then also $d = 0$ must hold, and q_r otherwise.

The acceptance set for the Büchi (resp. co-Büchi) version of $\mathcal{A}_{\mathcal{B},\mathcal{C}}$ is the same as for the Büchi (resp. co-Büchi) version of $\mathcal{A}^{\mathcal{C}}_{\mathcal{B}}$. We get:

Lemma 15. *Let $\mathcal{B}$ be a Büchi (resp. co-Büchi) deterministic automaton and $\mathcal{C}$ be an extended pre-post requirement. The universal Büchi (resp. co-Büchi) tree automaton $\mathcal{A}_{\mathcal{B},\mathcal{C}}$ is s.t. if $A^{\mathcal{C}}_{\mathcal{B}}$ accepts a strategy tree T then:*

$$\mathcal{A}_{\mathcal{B},\mathcal{C}} \text{ accepts } T \text{ iff } \mathcal{C} \text{ is } (T,\mathcal{B})\text{-consistent.}$$

Proof. We only discuss the case when $\mathcal{B}$ is a Büchi automaton, the co-Büchi case being similar.

(*if*) By construction, $\mathcal{A}^{\mathcal{C}}_{\mathcal{B}}$ and $\mathcal{A}_{\mathcal{B},\mathcal{C}}$ differ only on the transitions that can be taken at the nodes that are labeled with an exit when starting from states of the form $(q', d, f, q, \mathcal{C})$. Moreover, by Lemma 14, from such state and input node $\mathcal{A}_{\mathcal{B},\mathcal{C}}$ enters the accepting state q_a if and only if $\mathcal{C}$ is $(T,\mathcal{B})$-consistent. Therefore, starting from an accepting run of $\mathcal{A}^{\mathcal{C}}_{\mathcal{B}}$ we can construct a run of $\mathcal{A}_{\mathcal{B},\mathcal{C}}$ by replacing only the transitions involving the scenario described above, and the resulting run is accepting for $\mathcal{A}_{\mathcal{B},\mathcal{C}}$ since by hypothesis $\mathcal{C}$ is $(T,\mathcal{B})$-consistent.

(*only if*) We use an assume-guarantee style argument to prove this direction. On each subtree T_i of the root that corresponds to the unrolling of a module m_i, $\mathcal{A}_{\mathcal{B},\mathcal{C}}$ assumes that $\mathcal{C}$ is $(T,\mathcal{B})$-consistent possibly except for the part corresponding to module m_i. Now, $\mathcal{A}_{\mathcal{B},\mathcal{C}}$ on T_i uses this assumption to move from a call to its matching return within T_i and witnesses the fulfillment of the part of the $(T,\mathcal{B})$-consistency properties concerning to module m_i by mimicking the transitions of $\mathcal{B}$ on all the nodes of T_i that do not correspond to either a call or an exit. Then, it accepts if and only if these properties are fulfilled. Therefore, if $\mathcal{A}_{\mathcal{B},\mathcal{C}}$ accepts T then the above assumptions must hold for all modules, and thus $\mathcal{C}$ is $(T,\mathcal{B})$-consistent. □

Construction of $\mathcal{A}_{\mathcal{B},CG}$ The purpose of $\mathcal{A}_{\mathcal{B},CG}$ is to check that CG is $(T,\mathcal{B},\mathcal{C}_{pre})$-consistent where T is an input strategy tree. We construct it from $\mathcal{A}^{\mathcal{C}}_{\mathcal{B}}$ by modifying the transitions from the calls as follows.

For an input tree T, suppose that:

- the current node u is in the subtree of the i^{th} child of the root (i.e., in the subtree corresponding to the unrolling of module m_i) and is labeled with a call to a module m_j;
- the current state of $\mathcal{A}_{\mathcal{B},CG}$ is of the form $(q', d, f, q, \mathcal{C})$;

- there is a transition $(q', \eta_{m_i}(u), q'') \in \delta$.

For a Büchi automaton $\mathcal{B}$, $\mathcal{A}_{\mathcal{B},CG}$ must enter the rejecting state q_r if either:

- $(m_i, q) \to (m_j, q'')$ does not hold (i.e., the call graph does not account for a run of $\mathcal{B}$ from q to q' on a call witness from the entry of m_i to a call to m_j), or
- $(m_i, q) \xrightarrow{\checkmark} (m_j, q'')$ holds in CG and $d = 0$ (i.e., the call graph requires that a state in the acceptance set of $\mathcal{B}$ must be reached along the run of $\mathcal{B}$ on a call witness for $(m_i, q) \to (m_j, q'')$, but this is not the case).

Analogously, for a co-Büchi automaton $\mathcal{B}$, $\mathcal{A}_{\mathcal{B},CG}$ must enter the state q_r if either $(m_i, q) \to (m_j, q'')$ does not hold, or $(m_i, q) \xrightarrow{\checkmark} (m_j, q'')$ does not hold in CG and $d = 1$.

In all the other cases $\mathcal{A}_{\mathcal{B},CG}$ behaves as $\mathcal{A}^{\mathcal{C}}_{\mathcal{B}}$. The acceptance set for the Büchi (resp. co-Büchi) version of $\mathcal{A}_{\mathcal{B},CG}$ is the same as for the Büchi (resp. co-Büchi) version of $\mathcal{A}^{\mathcal{C}}_{\mathcal{B}}$. We get:

Lemma 16. *Let $\mathcal{B}$ be a Büchi (resp. co-Büchi) deterministic automaton, $\mathcal{C}$ be an extended pre-post requirement and CG be a call graph.*
The universal Büchi (resp. co-Büchi) tree automaton $\mathcal{A}_{\mathcal{B},CG}$ is s.t. if $A^{\mathcal{C}}_{\mathcal{B}}$ accepts a strategy tree T and $\mathcal{C}$ is $(T, \mathcal{B})$-consistent then:

$$\mathcal{A}_{\mathcal{B},CG} \text{ accepts } T \text{ iff } CG \text{ is } (T, \mathcal{B}, \mathcal{C}_{pre})\text{-consistent.}$$

Proof. We only discuss the case when $\mathcal{B}$ is a Büchi automaton, the co-Büchi case being similar.

Fix $(m, q) \in \mathcal{C}_{pre}$ and a module play $\pi.x$ from m.

We first claim that if the properties stated in the notion of $(T, \mathcal{B}, \mathcal{C}_{pre})$-consistency of a call-graph hold for each prefix of $\pi.x$ that is a call witness, then it is possible to show the property stated in Lemma 14 also for $\mathcal{A}_{\mathcal{B},CG}$. Namely, we can show that: $\mathcal{A}_{\mathcal{B},CG}$ reaches a state of the form $(q', d, f, q, \mathcal{C})$ at x iff the only run ρ of $\mathcal{B}$ starting at q and over the trace of π ends at q'; further, $d = 1$ iff a state of the acceptance set is visited in ρ.

A proof by induction can be structured by splitting $\pi.x$ as $x_0.\pi_1.x_1 \dots \pi_j.x_j$ where $x_1, \dots, x_j$ are all the nodes that correspond to the same occurrence of m as the start node x_0 and that are labeled with calls to modules $m'_1, \dots, m'_j$. Then, the induction is on the prefixes $x_0.\pi_1.x_1 \dots \pi_i$ for $i \in [1, j]$.

For the base case (i.e., $i = 1$), it suffices to apply Lemma 14 since $\mathcal{A}_{\mathcal{B},CG}$ and $A^{\mathcal{C}}_{\mathcal{B}}$ differ only in the transitions from nodes labeled with calls, and there are no such nodes

in $x_0.\pi_1$ that correspond to the same occurrence of m as x_0. For the induction step, we first apply the induction hypothesis to $x_0.\pi_1.x_1 \dots \pi_i$ with $i \in [1, j-1]$, and let $(q', d, f, q, \mathcal{C})$ be the state of $\mathcal{A}_{\mathcal{B},CG}$ at x_i. Then, by hypothesis the properties stated in the notion of $(T, \mathcal{B}, \mathcal{C}_{pre})$-consistency of a call-graph hold for $x_0.\pi_1.x_1 \dots \pi_i.x_i$ and by induction hypothesis $d = 1$ iff the run of $\mathcal{B}$ from q and over the trace of $x_0.\pi_1.x_1 \dots \pi_i$ visits a state in the acceptance set. Thus, $\mathcal{A}_{\mathcal{B},CG}$ from $(q_i, d_i, f_i, q, \mathcal{C})$ and at x_i can take exactly the same transitions as $A^{\mathcal{C}}_{\mathcal{B}}$. Therefore, we can apply again Lemma 14 to show that the property holds for $i+1$, that concludes the proof of the claim.

We conclude the proof by addressing the two directions of the lemma separately.

(*only if*) Suppose that CG is not $(T, \mathcal{B}, \mathcal{C}_{pre})$-consistent. Let $\pi.x$ be a call witness from m to m' s.t. the properties stated in the notion of $(T, \mathcal{B}, \mathcal{C}_{pre})$-consistency of a call-graph do not hold for it and do hold instead for each of its proper prefixes that forms a call witness.

Thus, by the above claim, $\mathcal{A}_{\mathcal{B},CG}$ reaches a state of the form $(q', d, f, q, \mathcal{C})$ at x iff the only run ρ of $\mathcal{B}$ starting at q and over the trace of π ends at q'; further, $d = 1$ iff a state of the acceptance set is visited in ρ.

Now, let q'' the ending state of the $\mathcal{B}$ run ρ over $\pi.x$. If the $(T, \mathcal{B}, \mathcal{C}_{pre})$-consistency of CG is violated because $(m, q) \to (m', q'')$ does not hold, then the first case of the added transitions in the construction of $\mathcal{A}_{\mathcal{B},CG}$ applies. Otherwise, i.e., if the $(T, \mathcal{B}, \mathcal{C}_{pre})$-consistency of CG is violated because $(m, q) \xrightarrow{\checkmark} (m', q'')$ and ρ does not visit states in the acceptance set of $\mathcal{B}$, by the above claim, the second case of the added transitions in the construction of $\mathcal{A}_{\mathcal{B},CG}$ applies. Thus, in both cases $\mathcal{A}_{\mathcal{B},CG}$ enters q_r and since q_r is a sink state, we get that $\mathcal{A}_{\mathcal{B},CG}$ does not accept T.

(*if*) We prove this direction by arguing that under the assumption that CG is $(T, \mathcal{B}, \mathcal{C}_{pre})$-consistent, any run of $A^{\mathcal{C}}_{\mathcal{B}}$ is also a run of $\mathcal{A}_{\mathcal{B},CG}$, and since from the hypothesis of the lemma $A^{\mathcal{C}}_{\mathcal{B}}$ accepts T this will conclude the proof.

First observe that since CG is $(T, \mathcal{B}, \mathcal{C}_{pre})$-consistent, the hypothesis of the claim shown at beginning of this proof are fulfilled for each module play of T. Also, by construction, $\mathcal{A}_{\mathcal{B},CG}$ differ from $A^{\mathcal{C}}_{\mathcal{B}}$ only in the transitions from some states of the form $(q', d, f, q, \mathcal{C})$ at some nodes labeled with a call. Thus let us consider the case of any state of the form $(q', d, f, q, \mathcal{C})$ at a node x labeled with a call from m to m'. From the mentioned claim, for any call witness $\pi.x$, $\mathcal{A}_{\mathcal{B},CG}$ reaches $(q', d, f, q, \mathcal{C})$ at x iff the only run ρ of $\mathcal{B}$ starting at q and over the trace of π ends at q'; further, $d = 1$ iff a state of the acceptance set is visited in ρ. Since CG is $(T, \mathcal{B}, \mathcal{C}_{pre})$-consistent, $\mathcal{A}_{\mathcal{B},CG}$ from $(q', d, f, q, \mathcal{C})$ at x moves as $A^{\mathcal{C}}_{\mathcal{B}}$, that concludes the proof. □

Construction of $\mathcal{A}_{\mathcal{B}_{win}}$ The purpose of $\mathcal{A}_{\mathcal{B}_{win}}$ is to check that the winning conditions of $\mathcal{B}$ are satisfied along all non-terminating plays of the input strategy tree with a finite number of unreturned calls. Thus, when $\mathcal{B}$ is a Büchi (resp. co-Büchi) automaton, we choose as the Büchi (resp. co-Büchi) acceptance set of $\mathcal{A}_{\mathcal{B}_{win}}$ the state q_a (resp. q_r) and all the states of the form $(q, d, f, q', \mathcal{C})$ such that either $q \in F$ or $f = 1$. We get:

Lemma 17. *Let $\mathcal{B}$ be a Büchi (resp. co-Büchi) deterministic automaton and $\mathcal{C}$ be an extended pre-post requirement. The universal Büchi (resp. co-Büchi) tree automaton $\mathcal{A}_{\mathcal{B}_{win}}$ is s.t. if $A^{\mathcal{C}}_{\mathcal{B}}$ accepts a strategy tree T and $\mathcal{C}$ is $(T, \mathcal{B})$-consistent then:*
$\mathcal{A}_{\mathcal{B}_{win}}$ accepts T iff $\mathcal{B}$ accepts the traces of all the non-terminating plays of T with finitely many unreturned calls.

Proof. We recall that in the construction of $A^{\mathcal{C}}_{\mathcal{B}}$ and thus $\mathcal{A}_{\mathcal{B}_{win}}$, f is always 0 except when starting from a state of the form $(q, d, f, q_m, \mathcal{C})$ at a node labeled with call from m to m' to to a node labeled with a return that matches the call, and the pre-post tuple (m', q', ex, q'') that summarizes this module invocation is marked true by function *Fin* of of the extended pre-post requirement $\mathcal{C}$. Thus, by Lemma 14 and since $\mathcal{C}$ is $(T, \mathcal{B})$-consistent, for each module play $x_1 x_2 \dots$ from m, each matching call x_i and return x_j corresponding to the same invocation of m as x_1 and each $(m, q_m) \in ppre$, the f-component of the state at x_j is 1 iff the a state in the acceptance set F of $\mathcal{B}$ is visited in the run of $\mathcal{B}$ from q_m over the trace of $x_{i+1} \dots x_{j-1}$. Therefore, denoting with ξ_π the path of T that is formed by all the nodes x_i that correspond to the same invocation of m as x_1, for each non-terminating play π with finitely many unreturned calls: starting from $(q_m, 0, 0, q_m, \mathcal{C})$ at x_1, states of the form $(q, d, f, q_m, \mathcal{C})$ with either $q \in F$ or $f = 1$ are visited infinitely often over ξ_π iff the only run of $\mathcal{B}$ from q_m over the trace of π visits infinitely often at least a state in the acceptance set. Hence, the lemma holds. □

5.4.4 Checking $\mathcal{B}$ acceptance on strategy-tree plays with infinitely many unreturned calls

From part 2 of Lemma 14, if a call graph $CG = (V, \rightarrow, \overset{\checkmark}{\rightarrow})$ is $(T, \mathcal{B})$-consistent for a strategy tree T, then each edge in $\rightarrow$ that admits a call witness π summarizes the (only) run of $\mathcal{B}$ over the trace of π, and the relation $\overset{\checkmark}{\rightarrow}$ carries the information whether a state of the acceptance set of $\mathcal{B}$ is visited along this run. Since a play of a strategy tree that contains infinitely many unreturned calls is the concatenation of infinitely many call witnesses, a call graph summarizes all the information that is needed to check for the fulfillment of the acceptance condition of $\mathcal{B}$ along such plays. In fact,

by replacing the call witnesses with the corresponding edges, we get an infinite path in the call graph (we are assuming that the call graph is $(T, \mathcal{B})$-consistent). Since call graphs are finite, all the edges that repeat infinitely often on this path are parts of a strongly connected component of the call graph where the path gets trapped. Thus, for a Büchi acceptance condition it is sufficient to require that each loop of the call graph has at least a marked edge (i.e., the edge also belongs to $\xrightarrow{\checkmark}$). For a co-Büchi acceptance condition, it is sufficient to require that each strongly connected component of the call graph has no marked edges.

Denote with BÜCHI (resp. CO-BÜCHI) the set of all call graphs s.t. each loop has at least a (resp. none) edge that is marked. Thus, by Lemma 17, we get the following lemma:

Lemma 18. *Let $\mathcal{B}$ be a Büchi (resp. co-Büchi) deterministic automaton, $\mathcal{C}$ be an extended pre-post requirement and $CG \in$ BÜCHI (resp. $CG \in$ CO-BÜCHI). The universal Büchi (resp. co-Büchi) tree automaton $\mathcal{A}_{\mathcal{B}_{win}}$ is s.t. if $A^{\mathcal{C}}_{\mathcal{B}}$ accepts a strategy tree T, $\mathcal{C}$ is $(T, \mathcal{B})$-consistent and CG is $(T, \mathcal{B}, \mathcal{C}_{pre})$-consistent then:*

$$\mathcal{A}_{\mathcal{B}_{win}} \text{ accepts } T \text{ iff } \mathcal{B} \text{ accepts all the traces of the } T \text{ plays.}$$

5.4.5 Reducing modular synthesis to emptiness of tree automata

We construct the automaton $\mathcal{A}_{G,\mathcal{B}}$ as the intersection of $\mathcal{A}_G$ and an automaton $\mathcal{A}'$ that does the following: at the root, $\mathcal{A}'$ nondeterministically guesses an extended pre-post requirement $\mathcal{C}$ and a call graph $CG \in$ BÜCHI (resp. $CG \in$ CO-BÜCHI) if $\mathcal{B}$ is a Büchi (resp. co-Büchi) automaton; then it behaves as $\mathcal{A}_{\mathcal{B},\mathcal{C},CG}$.

We observe that $\mathcal{A}_{\mathcal{B},\mathcal{C},CG}$ can be translated into an equivalent nondeterministic Büchi (resp. co-Büchi) tree automaton with $2^{O(|Q|^2 \log |Q|)}$ states (33). Denoting with k the number of G exits and β the number of *call edges* of G, the number of different choices for an extended pre-post requirement is $2^{O(k\,|Q|^2)}$ and for a call graph is $2^{2\beta}$. Since $\mathcal{A}_G$ is of size $O(|G|)$, the automaton $\mathcal{A}_{G,\mathcal{B}}$ (obtained as described earlier in this section) is of size $|G|\, 2^{O(|Q|^2(k+\log |Q|)+\beta)}$. Thus, by Propositions 12 and 13, and Lemmas 15, 16 and 18, we get:

Theorem 19. *For an RGG G and a deterministic Büchi (resp. co-Büchi) automaton $\mathcal{B}$, pl_0 has a winning modular strategy in $\langle G, \mathcal{B}\rangle$ iff the nondeterministic Büchi (resp. co-Büchi) tree automaton $\mathcal{A}_{G,\mathcal{B}}$ accepts a non-empty language. Moreover, each*

tree accepted by $\mathcal{A}_{G,\mathcal{B}}$ encodes a winning modular strategy and the size of $\mathcal{A}_{G,\mathcal{B}}$ is $|G|\, 2^{O(|Q|^2(k+\log|Q|)+\beta)}$, where k is the number of G exits.

5.5 Temporal logic winning conditions

5.5.1 Solving modular CaRet and Nwtl games

As we said in Chapter 2, CaRet and Nwtl are temporal logics that extend Ltl with new operators that allow to express properties on ordinary words and also on the matching call-return structure, in the future and in the past. As example, CaRet and Nwtl formulas can express specifications as stack inspection properties, partial correctness and local properties.

By (3), we know that given a CaRet formula φ it is possible to construct a nondeterministic Büchi VPA of size exponential in $|\varphi|$ that accepts exactly all the words that satisfy φ. From (8), we know that the same holds for the temporal logic Nwtl. Thus, given a formula φ in any of the two logics, we construct a Büchi VPA P for its negation $\neg\varphi$. By dualizing as in the case of nondeterministic VPA specifications, we get a co-Büchi VPA that accepts all the models of φ and whose size is exponential in $|\varphi|$. Since both CaRet and Nwtl subsume Ltl (34), and Ltl games are known to be 2Exptime-hard (35) already on standard finite game graphs, we get:

Theorem 20. *The MVPG problem with winning conditions expressed as* CaRet *and* Nwtl *formulas is* 2Exptime*-complete.*

In the rest of this section we discuss the complexity of the modular synthesis in simple fragments of temporal logic.

5.5.2 Path formulas

We consider as winning condition for the MVPG a fragment of the logic Ltl (34) that contains Boolean combinations of bounded-size path formulas.

A *path formula* is formula expressing either the requirement that a given sequence appears as a subsequence in an ω-word or its logical negation. Path formulas are captured by Ltl formulas of the form $\Diamond(p_1 \wedge \Diamond(p2 \wedge \ldots \Diamond(p_{n-1} \wedge \Diamond p_n)\ldots))$ and by their logical negation, where each p_i is state predicate, $\Diamond\psi$ (*eventually* ψ) denotes that

ψ holds at some future position, and $\wedge$ is the Boolean conjunction. We denote such a fragment of LTL as PATH-LTL.

Note that in this logic negation is allowed only at the level of atomic propositions or at the top level of a formula.

We interpret the formulas on an ω-word over 2^{AP}. At each position i of w: a state predicate holds true if it evaluates to true on σ_i; the Boolean connectives are interpreted as usual, i.e., a formula $\varphi_1 \wedge \varphi_2$ holds true at i iff both φ_1 and φ_2 hold true at i, and $\neg\varphi$ holds true at i iff φ does not hold at i; and $\Diamond\varphi$ holds true if there is a $j > i$ such that φ holds true at j.

It is known that each formula φ from PATH-LTL admits a deterministic Büchi word automaton accepting all the models of φ and that is linear in its size (9). The same can be shown for Büchi VPA, by extending PATH-LTL allowing the versions of the $\Diamond$ operator of CARET and NWTL, that include top-down call-stack inspection and local future (where calls to other modules are skipped moving from a call directly to its matching return). For more details on CARET and NWTL operators see (3) and (8).

By the closure properties of universal visibly pushdown automata we can easily extend Theorem 10 to winning conditions given as intersection of deterministic VPAs and thus:

Theorem 21. *The MVPG problem with winning conditions expressed as a conjunction of* CARET *and* NWTL *formulas that admit a deterministic Büchi or co-Büchi VPA generator of polynomial size is* EXPTIME*-complete.*

5.5.3 Modular synthesis in simple fragments of LTL

The complexity of the temporal logic MVPG problem remains 2EXPTIME-hard even if we consider simple fragments.

To simplify our description, first we present the reduction from polynomial-space alternating Turing machines for disjunctions of bounded-size PATH-LTL formulas and then we extend this approach to prove the 2EXPTIME lower bound for conjunctions of disjunctions of bounded-size PATH-LTL formulas, by a reduction from the acceptance problem for exponential-space alternating Turing machines.

Alternating Turing machine An alternating Turing machine extends the standard Turing machine as follows. The states are partitioned into *existential* and *universal*. A run is a tree of configurations where the root is mapped to an initial configuration, and each child of a node mapped to a configuration C is mapped to a configuration that can be reached in one step from C. For existential configurations only one child configuration is selected nondeterministically and for each universal configuration all the possible child configurations are selected. Thus, an input word is accepted if there is a run that reaches a final configuration on all of its paths.

Formally an alternating Turing machine is $M = (\Sigma, Q, Q_\exists, Q_\forall, \delta, q_0, q_f)$, where Σ is the alphabet, Q is the set of states, $(Q_\exists, Q_\forall)$ is a partition of Q, $\delta : Q \times \Sigma \times \{D_1, D_2\} \longrightarrow Q \times \Sigma \times \{L, R\}$ is the transition function, and q_0 and q_f are respectively the initial and the final states. (We assume that for each pair $(q, \sigma) \in Q \times \Sigma$, there are exactly two transitions that we denote respectively as the D_1-transition and the D_2-transition.)

A d-transition of M is $\delta(q, \sigma, d) = (q', \sigma', L/R)$ meaning that if q is the current state and the tape head is reading the symbol σ on cell i, M writes σ' on cell i, enters state q' and moves the head tape to the left/right on cell $(i-1)/(i+1)$.

Let n be the number of cells used by M on an input word w. A configuration of M is a word $\sigma_1 \dots \sigma_{i-1}(q, \sigma_i) \dots \sigma_n$ where $\sigma_1 \dots \sigma_n$ is the content of the tape cells, q is a state of M and (q, σ_i) denotes that the tape head is on cell i. The *initial configuration* contains the word w and the initial state. An *outcome* of M is a sequence of configurations, starting from the initial configuration, constructed as a play in the game where the $\exists$-player picks the next transition when the play is in a state of $Q_\exists$, and the $\forall$-player picks the next transition when the play is in a state of $Q_\forall$. A *computation* of M is a strategy of the $\exists$-player, and an input word w is accepted iff there exists a computation that reaches a configuration with state q_f on all the possible plays. A polynomial-space alternating Turing machine M is an alternating Turing machine that on an input word w uses a number of tape cells that is at most polynomial in $|w|$.

EXPTIME lower bound for disjunctions of bounded-size PATH-LTL formulas We sketch a reduction directly from polynomial-space alternating Turing machines. Let $\mathcal{A}$ be a polynomial-space alternating Turing machine with set of control states Q and input alphabet Σ, and let N be the number of cells used by $\mathcal{A}$ on an input word w. An *encoding of a configuration* of $\mathcal{A}$ is a sequence of $\bigcup_{i=1}^{N}(\Sigma_1 \dots \Sigma_{i-1}(Q \times \Sigma_i)\Sigma_{i+1} \dots \Sigma_N)$

where each Σ_j is $\Sigma \times \{j\}$. The transition function of $\mathcal{A}$ is given as tuples of the form (a, b, c, d, e) where a, b, c, e are symbols used in the configuration encoding, d is the transition name, and e encodes the effect of d on the middle cell of three cells containing respectively a, b, c.

A path of a computation is encoded as a sequence $C_0 d_0 \$ \dots C_i d_i \$ \dots$ where each C_i is a configuration encoding (C_0 is initial) and d_i denotes the transition taken from C_i to C_{i+1}.

We construct an RGG G_{exp} with two modules M_{in} and M_1. In M_{in}, initially, pl_0 generates an encoding of an initial configuration, then, a transition is selected by pl_0, if the initial state is existential, or by pl_1, otherwise. In both cases, an end-of-configuration marker \$ is generated and then pl_0 is in charge to generate again a configuration encoding, and so on. A call to M_1 is placed before generating the first cell encoding of each configuration and after generating each cell encoding. We omit further details on M_{in} which are quite standard.

In M_1, pl_1 selects one among $ok, obj_1, \dots, obj_6$. If obj_6 is selected, then an infinite sequence of a fresh symbol ∂ is generated and the call is not returned. In all the other cases, the call is returned through the only exit of M_1.

The goal of pl_0 is to build an encoding of an accepting run of $\mathcal{A}$ on input w, while the goal of pl_1 is to point out errors in such encoding by generating objections. The symbol ok is used to denote that no objection is raised. The objections $obj_1, \dots, obj_4$ are used to delimit three consecutive cells of a configuration, say cells $i-1, i, i+1$ (we can assume that the first and the last cell of each configuration are never changed and their consistency through the encoding is ensured by the construction of M_{in}), and obj_5 and obj_6 are used to delimit the cell i in the next configuration. Every other use of these objections will make pl_1 lose.

We construct a formula φ_{exp} as $\psi_{wr1} \vee \psi_\Delta \vee \Diamond F$ where: (1) F denotes a state predicate that is true only on the final sink state ; (2) ψ_{wr1} captures all the illegal uses of the objections by pl_1; and (3) ψ_Δ checks the transition relation between two consecutive configurations on the cells selected by the objections raised by pl_1.

Note that a path formula $\Diamond(a_1 \wedge \Diamond(a_2 \wedge \dots \Diamond a_r) \dots)$, where for each i the state predicates a_i and a_{i+1} cannot be satisfied at the same position, is satisfied on all the sequences where $a_1 a_2 \dots a_r$ appears as a subsequence. Formula ψ_Δ is quite standard and is a dis-

junction of formulas such that check a subsequence $obj_1.a.obj_2.b.obj_3.c.obj_4.d.obj_5.e.obj_6$ for each $\mathcal{A}$ transition (a, b, c, d, e).

Formula ψ_{wr1} is a disjunction of formulas that check subsequences corresponding to violations in the use of the objections by pl_1, that is, such that: (1) obj_j precedes obj_i for some $i < j$; (2) obj_i repeats for some i; (3) there are two symbols among those generated in M_{in} between obj_i and obj_{i+1} for some $i \in [3]$ or $i = 5$; (4) there are two occurrences of \$ between obj_4 and obj_5; (5) the cell number between obj_2 and obj_3 is not the same as that between obj_5 and obj_6.

Note that both ψ_Δ and ψ_{wr1} are disjunctions of bounded-size PATH-LTL formulas (the longest sequence to check has eleven symbols). Furthermore, by the construction of M_{in} we ensure besides the already mentioned requirements also that each configuration has exactly one cell containing a state (which denotes also the position of the tape head). Finally, in order to win with a modular strategy in the game $\langle G_{\text{exp}}, \varphi_{\text{exp}} \rangle$, pl_0 has to provide a correct encoding of the computations without knowing where pl_1 raises the objections (since that happens in a different module).

The proposed modular game has a winning strategy for pl_0 if and only if $\mathcal{A}$ accepts the input. On the one hand, if $\mathcal{A}$ accepts the input, then, by construction, pl_0 can generate the right sequences of encodings that form an accepting run. On these plays, pl_0 wins for $\Diamond F$ since the final sink state is reached. If pl_1 tries to cheat, making any objection obj_6 thus forcing the play to be trapped in M_1, then pl_0 wins for either ψ_{wr1}, because either the sequence of objections is illegal, or ψ_Δ, because the run is encoded correctly. On the other, if pl_0 has a winning strategy in this game, we take as run of the TM $\mathcal{A}$ the one that is encoded by all the plays that do not get trapped in M_1. To see that this run is accepting, first observe that for each play that gets trapped into M_1, since the strategy is winning, either ψ_{wr1} or ψ_Δ must hold, and thus the plays that do not get trapped correctly encode a run of $\mathcal{A}$. Thus, since these plays are winning for $\Diamond F$, the final sink state is reached and thus the final state is reached on all the paths of the run of $\mathcal{A}$ that thus is accepting. Therefore, we get:

Lemma 22. *The MVPG problem with winning conditions expressed as a disjunction of bounded-size* PATH-LTL *formulas is* EXPTIME*-hard.*

2EXPTIME **lower bound** The reduction given to show Lemma 22 can be adapted to show 2EXPTIME-hardness when the formula is a conjunction of disjunctions of bounded-size PATH-LTL formulas.

The reduction is now from exponential-space alternating Turing machines, and thus each configuration uses 2^N cells, where N is the length of the input. Since we cannot encode the cell number along with the cell content as before (we would have exponentially many symbols), we explicitly encode it as a sequence of bits that precedes the encoding of the cell content. We recall that a similar encoding is used in (10). We use new atomic propositions $d_i^\top$ and $d_i^\bot$ to denote that the i^{th} bit of the cell number is respectively 1 and 0. Thus a configuration encoding now is a sequence of the form $\langle 0\rangle\sigma_0 \dots \langle 2^N\rangle\sigma_{2^N}$ where there is an i s.t. $\sigma_i \in Q \times \Sigma$ (this denotes the current state, the symbol of cell i and that the tape head is on cell i), $\sigma_j \in \Sigma$ for all $j \neq i$ (symbol in cell j), and $\langle h\rangle$ is the binary encoding of h (cell number) over the new symbols $d_r^\top$ and $d_r^\bot$ for $r \in [N]$.

Module M_{in} is modified such that in each iteration pl_0 selects the cell number (bit-by-bit) and then the encoding of the cell content. Also module M_1 is modified by adding three new objections (denoted obj'_1, obj'_2, obj'_3) that are used to check that the cell numbering is correct in each configuration. By the construction of M_{in} we also ensure that the first and the last cells of each configuration are numbered respectively $d_1^\bot \dots d_N^\bot$ and $d_1^\top \dots d_N^\top$.

The winning condition is a formula $\varphi_{2\,\mathrm{exp}}$ where we add $obj = \bigvee_{i=1}^{6} obj_i$ and the sub-formulas ψ_{wr2} and $\psi_\sharp$. Formula ψ_{wr2} checks the violations in the use of the newly added objections and can be constructed similarly to ψ_{wr1}. Formula $\psi_\sharp$ checks that the encoding of the cell numbers is correct, and in particular, for each two consecutive cells, whose numbers are encoded respectively as $\hat{d} = d_1 \dots d_N$ and $\hat{d'} = d'_1 \dots d'_n$, it holds that $\langle \hat{d'}\rangle = \langle \hat{d}\rangle + 1$ (with $\langle\cdot\rangle$ we have denoted the number corresponding to the binary encoding). Denote with $same_j$ the formula checking for a subsequence either of the form $obj'_1.d_j^\top.obj'_2.d_j^\top.obj'_3$ or of the form $obj'_1.d_j^\top.obj'_2.d_j^\top.obj'_3$ (the j^{th} bits of two consecutive cells are the same), with up_j the formula checking for a subsequence of the form $obj'_1.d_j^\bot.obj'_2.d_j^\top.obj'_3$ (the j^{th} bits of two consecutive cells are in the increasing order), and with $down_j$ the formula checking for a subsequence of the form $obj'_1.d_j^\top.obj'_2.d_j^\bot.obj'_3$ (the j^{th} bits of two consecutive cells are in the decreasing order). The formula $\psi_\sharp$ is then $\bigvee_{i=1}^{N}(\bigwedge_{j=1}^{i-1} same_j \wedge up_i \wedge \bigwedge_{j=i+1}^{N} down_j)$.

Precisely, $\varphi_{2\,\mathrm{exp}}$ is $(\neg\Diamond obj \vee ((\psi_{wr1} \vee \psi_{\Delta}) \wedge (\psi_{wr2} \vee \psi_{\sharp}))) \wedge (\Diamond obj \vee \Diamond F)$.

Observe that $\psi_{\sharp}$ is the only formula in $\varphi_{2\,\mathrm{exp}}$ that would give an exponential blow up in the conversion into a conjunction of disjunctions (CNF) of $\textsc{Ltl}_{\Diamond,\wedge}$ formulas. We can avoid this blow-up if we modify M_{in} such that pl_1 starts the numbering of each cell declaring at which bit there will be the first difference with the encoding of the number of the following cell. Denoting with p_i the symbol used to declare the first difference at the bit i for $i \in N$, the formula $\psi_{\sharp}$ would be written as $\bigwedge_{i=1}^{N}(\bigwedge_{j=1}^{i-1} same_{i,j} \wedge up_{i,j} \wedge \bigwedge_{j=i+1}^{N} down_{i,j})$ where: (1) $same_{i,j}$ is the formula checking for a subsequence of either one of the forms $obj'_1.p_h.obj'_2$ for $h \neq i$, or $obj'_1.p_i.d_j^{\top}.obj'_2.d_j^{\top}.obj'_3$, or $obj'_1.p_i.d_j^{\bot}obj'_2.d_j^{\bot}.obj'_3$; (2) $up_{i,j}$ is the formula checking for a subsequence of either one of the forms $obj'_1.p_h.obj'_2$ for $h \neq i$, or $obj'_1.p_i.d_j^{\bot}.obj'_2.d_j^{\top}.obj'_3$; and (3) $up_{i,j}$ is the formula checking for a subsequence of either one of the forms $obj'_1.p_h.obj'_2$ for $h \neq i$, or $obj'_1.p_i.d_j^{\top}.obj'_2.d_j^{\bot}.obj'_3$.

All the above formulas are written with disjunctions of path formulas except for $\psi_{\sharp}$ that is a conjunction of disjunctions of path formulas. The overall formula can be transformed into an equivalent formula of size polynomial in $|\varphi|$, which is a conjunction of disjunctions of path formulas. All the used path formulas are of bounded size (the most complex one uses eleven occurrences of $\Diamond$). For the correctness of the reduction, we can argue as in the previous case. Essentially, in a modular strategy pl_0 cannot use the fact that pl_1 has raised an objection to decide the next move since the objections are raised in a different module (which has just one exit). Therefore, in order to win, pl_0 must correctly generate the computations of the TM. We get the following:

Lemma 23. *The MVPG problem with winning conditions expressed as a conjuction of disjunctions of bounded-size* PATH-Ltl *formulas is* 2Exptime*-hard.*

6

Synthesis from libraries

Component-based design plays a key role in configurable and scalable development of efficient hardware as well as software systems. For example, it is current practice to design specialized hardware using some base components that are more complex than universal gates at bit-level, and programming by using library features and frameworks. In this framework, the synthesis is still the automatic construction of a system from a specification, but such construction is not done from scratch, The final system will be obtained by a composition of reusable elements, named components. This problem was extensively studied by Lusting and Vardi in (28, 29) and their works introduced and solved the problem of synthesized a system from a set of recursive components (named *library*). In Section 6.1 we discuss about the model and the problems discussed in the works (28, 29). Synthesis from libraries of recursive components and modular synthesis are strictly related, and in Section 6.2 we analize the connection between these problems.

A library of recursive components, however, resolves only the external compositional game and does not allow to guide the composition. Our research focus on one hand to extend the synthesis also to the internal game, considering components where the set of vertices was split between two player, and on the other to allows to restrict or relax the constraints on the composition of the final system. In Chapter 7 we present formally our models and the general synthesis problem. The solution to a set of new modular synthesis problems will be described in Chapter 8 and Chapter 9.

6.1 Synthesis from libraries of transducers

In (28) the authors starts giving a first overview of the possible composition of system. They define two notions of component composition. One relates to data-flow and is motivated by hardware, while the other relates to control-flow and is motivated by software. The authors show that whether or not synthesis is computable depends crucially on the notion of composition.

The first composition notion is *data-flow composition*, in which the outputs of a component are fed into the inputs of other components. In data-flow composition the synthesizer controls the flow of data from one component to the other. In this case the problem of LTL synthesis from libraries is undecidable. This claim is proven showing that the LTL synthesis from libraries is undecidable even if we restrict ourselves to pipeline architectures, where the output of one component is fed into the input of the next component.

The second notion of composition is the *control-flow composition*, which is motivated by software and web services. In the software context, when a function is called, the function is given control over the machine. The computation proceeds under the control of the function until the function calls another function or returns. Therefore, it seems natural to consider components that gain and relinquish control over the computation. A control-flow component is a transducer in which some of the states are designated as exit states [1]. Intuitively, a control-flow component receives control when entering an initial state and relinquish control when entering an exit state. Composing control-flow components amounts to deciding which component will resume control when the control is relinquished by the component that currently is in control.

When a component is in control the entire system behaves as the component and the system composition plays no role. The composition comes into play, however, when a component relinquishes control. Choosing the next component to be given control is the essence of the control-flow composition. A control-flow component relinquishes control by entering one of several exit states. A suitable notion of composition should specify, for each of the exit states, the next component the control will be given to. Thus, a control-flow composition is a sequence of components, each paired with an

[1] The authors in their work named them *final states*. We preferred to change them in exit states, to avoid confusion with the final states of the specification automata, that are presented in the previous section of this thesis

interface function that maps the various exit states to other components in system. We refer to these pairs of a component coupled with an interface function as interfaced component. Note that a system synthesized might choose to reuse a single component from the library several times, each with a different interface. Therefore, the number of interfaced components might differ from the number of components is the library.

In (28, 29) the basic model to represent a component is using the *transducer*, a finite-state machine with outputs. The transducers allow to abstract the internal architecture and to focus on the input/output behaviour. In (28) the authors prove that the LTL synthesis from libraries of transducers is 2EXPTIME -complete. In (29) the authors extend the previous work, considering the call/return structure of the control flow. This study requires the definition of the recursive components. A recursive component must have exit points, i.e. nodes where the control flow exits from the transducer to call an other transducer or to return to a caller transducer.

6.1.1 Synthesis from Components Libraries

A *transducer* is a deterministic finite state automaton with outputs. Formally, a transducer $\mathcal{TR} = \langle \Sigma_I, \Sigma_O, Q, q_0, \delta, F, \rangle$ where Σ_I is the finite input alphabet, Σ_O is the finite output alphabet, Q is the finite set of states, $q_0 \in Q$ is the initial state, $\delta : Q \times \Sigma_I \to Q$ is the transition function, F is the set of exit states, and $L : Q \to \Sigma_0$ is the output function labelling states with output letters. For a transducer $\mathcal{TR}$ and an input word $w = w_1 w_2 ... w_n \in \Sigma_I^n$ a *run* is a sequence of states $s = s_0, s_1, ..., s_n \in Q^n$ such that $s_0 = q_0$ and for every $i \in [n]$, $s_i = \delta(r_{i_1}, w_i)$. The *trace* of the run s is the word $u = u_1, u_2 ... u_n \in \Sigma_O^n$ where for each $i \in [n]$ we have $u_i = L(r_{i-1})$. The notion of run and trace are extended to infinite words in the natural way.

A transducer, for every input letter, returns as output an output letter. Therefore, for an input word w_I the transducer induces a word $w \in (\Sigma_I \times \Sigma_O)^\omega$, that interleaves each input letter with the corresponding output letter, generating an input-output word. A transducers satisfies an LTL formula φ if for every input word $w_i \in \Sigma_I^\omega$, the induced input-output word $w \in (\Sigma_I \times \Sigma_O)^\omega$ satisfies φ.

The control-flow components receives the control when entering the initial state and returns such control when entering a exit state. When a control-flow component is in control, the input/output interaction with the environment is done by component, and this means that any component in the system must use the same input and output

alphabets. In a control-flow mode, the composition plays its role when a component releases the control, and at this point the system must choose the next component that will receive the control. This means that a composition in a control-flow mode is a sequence of components, each paired with an interfaces function, i.e. a function that maps the exit states to call to other components in the system. Intuitively, a single component can be reused several time, changing each time its interface, and this means that the final system could be formed by a number of transducers higher than the number of reusable components in the given library.

Formally, in this setting a *composition* from control-flow components library is a finite sequence of pairs $\langle C_1, f_1\rangle, \langle C_2, f_2\rangle, ..., \langle C_n, f_n\rangle$ where $C_i = \langle \Sigma_I, \Sigma_O, Q_i, q_0^i, \delta_i, F_i, \rangle$ for $i \in [n]$ and $f_i : F_i \to \{1, ..., n\}$ is the interface function. Each pair $\langle C_i, f_i\rangle$ is named *interfaced component*. For each interfaced component, when the component C_i is in control and enters an exit state $q \in F_i$.

The control-flow library LTL synthesis is, given a library of components and an LTL -formula φ, find if exists or does not a composition that realizes φ. In (28) the authors prove the following theorem:

Theorem 24. *The control-flow library synthesis problem is 2*EXPTIME *-complete.*

The lower bound is proven reducing the classical synthesis problem to the control-flow library synthesis. The classical synthesis problem consists to construct a transducer such that for every sequence of input signals, the sequence of input and output signals induced by the transducer computation satisfies φ. It is simply to provide a library of control-flow components which implement a set of basic functionality, and then combine them to produce any possible transducers. Each components with basic functionality is named atomic transducer and that has only an initial state and a set of exit states. Each atomic transducer differs from each other only in its output function. The set of all such possible transducers will represent the library and it is easy to see that every transducer can be composed out of this library. Such synthesis is possible from this library of atomic control-flow components if and only if the classical synthesis is possible.

To prove the upper bound, the authors propose an automata theoretic construction. Fixing a library of components, the idea is to model a type of labelled trees, which represents compositions, such that every composition would induce a tree, and

every regular tree would induce a composition. Then, the authors define control-flow trees. Control-flow trees represent all possible flows of control during computations of a composition and a single path in a control-flow tree represents the flow of control between components in the system. A regular control-flow tree can be used to define a composition of control-flow components from the library. It is possible to construct a tree automaton whose language is the set of execution trees in which the LTL formula is satisfied, and the realizability problem reduces to checking emptiness of this automaton.Such tree automaton $\mathcal{A}$ accepts a infinite tree compositions if it satisfies the formula φ. If the language of $\mathcal{A}$ is empty then the formula cannot be satisfied by any control-flow composition. If, on the other, the language of $\mathcal{A}$ is not empty, then there exists a regular tree in the language of $\mathcal{A}$, from which we can extract a finite composition.

6.1.2 Synthesis from Recursive Components Libraries

In (29) the authors shift the focus of their research from the "go to" control flow to the "call and return" control flow. To model such control flow structure, the authors must introduce small variation on the basic transducer model.

A recursive transducer is a transducer in which some of the states are designed as *call states* and *exit states*. The recursive component receives the control when entering its initial state and relinquishes the control when entering in a call state or a exit state. When a call is entered, the control is transferred from the current component to the called component. When an exit is entered, the control is transferred from the current component to the caller component. To model the values passed to the caller, each transducer has several exit states, and each of them is associated with a re-entry state in the caller module. In this setting, a *recursive component* is transducer $\mathcal{TR}_{rec} = \langle \Sigma_I, \Sigma_O, Q, q_0, Q_E, Q_C, Q_X, \delta, F \rangle$ where Σ_I is the finite set of states, $q_0 \in Q$ is the initial state (when called by another component, the $\mathcal{TR}_{rec}$ component enters q_0). The set $Q_E \subseteq Q$ is a set of re-entry states and when the control returns from a call to another component, $\mathcal{TR}_{rec}$ enters one of the re-entry states. The set $Q_C \subseteq Q$ represents the set of call states and when a component $\mathcal{TR}_{rec}$ in a call state, and the control is transferred to the called module until the control is returned. The set $Q_X \subseteq Q$ represents the set of exit states and when the execution in the module $\mathcal{TR}_{rec}$ reaches the i^{th} exit state, the control is passed to the caller module. The function $\delta : Q \to \Sigma_I \to Q$

is the transition function and the function $F : S \to \Sigma_O$ is the output function, which labels each state by an output symbol.

A *library* $\mathcal{L}ib$ of recursive component, intuitively, is a set of recursive components i.e. $\mathcal{L}ib = \{\mathcal{TR}^1_{rec}, ..., \mathcal{TR}^l_{rec}\}$.

In this setting composing recursive transducers means to matching call states with entry states and exit states with re-entry states. A *composition* over a library $\mathcal{L}ib$ is a tuple $\langle(1, \mathcal{TR}^1_{rec}, f_1), ...(k, \mathcal{TR}^k_{rec}, f_k)\rangle$ of a finite number of composition elements. Each element is described by the triple $(i, \mathcal{TR}^i_{rec}, f_i\rangle)$, where i is an index, $\mathcal{TR}^i_{rec}$ is a recursive component in the given library and $f_i : \mathcal{TR}_{rec} \to [k]$ is a interface function that maps each call state of the component $\mathcal{TR}^i_{rec}$ to an index of a recursive component (remember that the control from the call state will be passed to the entry of the related called module). Note that a same transducer can instantiate different elements of the composition, with different interface function.

A run begins in the state q_0 of the component $\mathcal{TR}^1_{rec}$ and such component is in control until a call/exit state is reached. In this case the control will be passed to the called/caller component, according to the interface function. A composition fulfills a specification given as a NWTL formula φ if all the computation induced by such composition satisfy φ.

The recursive library component synthesis problem asks if, given a library of recursive component $\mathcal{L}ib$ and a NWTL specification, exists a composition such that it satisfies φ.

The solution to this synthesis problem is again an automata theoretic construction. First, the authors construct a tree automaton that accepts composition trees (i.e. a labeled tree that represents a possible composition obtained from the given library) that do not satisfy the specification tree. The automaton can be complemented to get an automaton which accepts composition trees that do satisfy the specification. Finally, the author checks if the language expressed by such automaton is empty and, if it is not, they construct the solution using a witness to identify the correct system. Then the following holds:

Theorem 25. *The recursive library component synthesis problem is 2*EXPTIME *-complete.*

6.2 Comparing synthesis of modular strategies with synthesis from libraries

Synthesis of modular strategies and synthesis from recursive component libraries are problems strictly related. The synthesis problem from a library of components introduced in (28) uses a different formulation by modeling components as game modules instead of finite state transducer. We rephrase the formulation in terms of game modules.

A *component* is a game module with a single entry and a single exit where there are only vertices of pl_1 and each vertex is a node (i.e., no boxes). A *library* of components is any finite set of components. For a library of components $\mathcal{L}$ and an LTL formula φ, the *synthesis problem from a library of components* asks to determine the existence of a finite sequence of components $C_1, \ldots, C_h$ from $\mathcal{L}$ such that for each play π_i that goes from the entry of C_i to its exit, for $i \in [h]$, the sequence $w_{\pi_1} \ldots w_{\pi_h}$ fulfills φ.

Fix a library of components $\mathcal{L}$ and an LTL formula φ. We construct a recursive game graph G formed of a main module and the components of $\mathcal{L}$. The main module has exactly one box b_C for each component $C \in \mathcal{L}$ such that b_C is mapped to C. The entry e of the main module is a pl_0 node and is connected to the call of each box b_C and the only return of each b_C is connected back to e. Formula φ is translated to an equivalent universal visibly pushdown automaton P that stutters on calls, returns and the entry of the main module. Such automaton P is exponential in the size of φ. Thus it is simple to verify that there exists a winning modular strategy of pl_0 in $\langle G, P \rangle$ if and only if there exists a finite sequence of components from $\mathcal{L}$ that fulfills φ. Thus, from Theorem 10 we get an alternative proof of the result stated in (28):

Theorem 26. *The synthesis problem from a library of components is* 2EXPTIME*-complete.*

As we said, in the realizability problem studied in (29), each component is modeled as a finite state transducer with entry, call, return and exit states, and a solution to the related synthesis problem is a composition of components (taken from a library of finitely many template components) that realizes a specification given as an NWTL formula. Composing components in this setting means to connect them by matching calls and returns of a component respectively with entries and exits of components in

the composition such that a call-return structure as in standard procedural programs is obtained.

This realizability problem is a form of a pushdown game and by fixing a bound on the number of components in a composition, can be easily modeled as an MVPG problem with NWTL specifications by using game modules instead of transducers as in the case of non-recursive component libraries introduced above. Clearly, this gives a semi-decision algorithm for the original problem and we need to show a bound to turn it into a decision algorithm. The existence of such theorem is guaranteed by the results from (29).

7

Component-based synthesis of open systems

In this chapter we formally introduce a new model that combines the synthesis from libraries of recursive components introduced by Lustig and Vardi with the modular synthesis introduced by Alur et al. for recursive game graphs.

In the following sections we limit the discussion only to the introduction and definition of library of open components, systems synthesized from such kind of library and the general modular synthesis problem. The problems related to specific winning conditions and the proposed solutions will be discussed extensively in the remaining chapters.

7.1 Contribution

The main contributions present in this chapter are:

- We introduce and formalize a new model that allows to sythesized open systems. We consider libraries equipped with a *box-to-component map*. This map is a partial function from boxes to components and, in this setting, an instance of a component C is essentially a copy of C along with a local strategy that resolves the nondeterminism of pl_0. An RSM S synthesized from a library is a set of instances along with a total function that maps each box in S to an instance of S and is consistent with the box-to-component map of the library.

- We define the *modular synthesis from a library of components* (LMS) in its general formulation. Moreover, we introduce some restrictions to such general LMS problem. We refer to the LMS problems with these restrictions as the *single-instance* LMS problem and the *component-based* LMS problem, respectively. Finally, we show the relation between the modular synthesis and the single-instance LMS problems.

7.2 From recursive components to open recursive components

The connections between modular synthesis and synthesis from library inspired us to dwell deeper on possible combinations of these two problem.

In the modular synthesis for recursive game graphs, the call-return structure is given and cannot be modified. Therefore, the synthesis process concerns only the internal structure of each module and the modules cannot be freely composed. On the other hand, in the synthesis from library the call-return structure can be modified and the synthesis concerns the external structure of the system, but we can not model a possible behaviour against an external environment. As we said, the internal game is used to model the uncontrollable nondeterminism caused by the interaction of the open system with an external environment. The natural question that arise is: what happened if we try to synthesize an open system from a library?

The first step to approach such problem is to find the right model to represent a library of *open components*. Transducers are not a good choice to model elements of this setting, therefore we must focus our attention on a different approach to define a new model.

We propose a model based on a variation of the recursive game graphs, modifing the definition of game module to meet the requirements that a component has to implement in the synthesis from library setting. As we said in Chapter 4, a module is a two-player finite game graphs with two kinds of vertices: standard *nodes* and *boxes*. Each box has *call* and *return* points, and each module has distinguished *entry* and *exit* nodes. The edges are *from* a node or a return *to* a node or a call within the same module. Moreover, the nodes and the returns are split among the two players (pl_0 and pl_1).

The call-return structure of a RGG is fixed and the invocations that can be executed for a module are well defined by the mapping of its boxes. Our first idea was to break these links to allow the free composition of the game modules. In the our component-based synthesis, our game modules are taken from a finite set (*library*) of *game components*. Game components differ from game modules in that the boxes are not mapped to any module (as an empty position in a code where we could insert a function call).

In this setting, we must define how we construct a system. A *composition* is obtained duplicating a subset of open components from the library and defining a global mapping that models the call-return mechanics between the generated modules.

Defining the composition, however, is not sufficient to obtain the final system. We must also handle with the internal game of the modules. Each game module must be coupled with a *local strategy* which defines how the module must behave according to the moves done by external environment. The choices of providing modular strategy are quite natural in this context. Due to the fact that the modules are independent, realizing an independent controller is an evident consequence, even more if we consider to realize system that works in distributed or security setting, where usually the single elements have no access to the memory of the entire system.

This basic model can be extended with different features.

First, we can note that often in a given set of reusable components there are some dependencies that can not be avoided, for example a component that can be called only by a specific component that has the role to guarantee a preliminary execution. This means that we want to realize a *guided composition* and we must provide our model of a way to express and handle this additional feature. For this reason, we consider that the library is equipped with a partial mapping, named *box-to-component* mapping, that can express dependency between a caller and a called components. Intuitively if the function is undefined for a box, no restriction is imposed on caller module, that can invoke any module of the system.

A composition in such setting can involve arbitrarily many modules of each component with possibly different local strategies. Such a diversity in the system design is often not affordable or unrealistic. Therefore we also consider restrictions of this problem by focusing on solutions with few component instances and designs. In our setting, a natural way to achieve this is by restricting the synthesized RSMs such that: 1) at

most one instance of each library component is allowed (few component instances), or 2) all the instances of a same library component must be controlled by a same local strategy (few designs).

7.3 A General Modular Synthesis from Libraries

7.3.1 Library of open components

For $k \in \mathbb{N}$, a *k-component* is a finite game graph with two kinds of vertices, the standard *nodes* and the *boxes*, and with an *entry* node and k *exit* nodes. Each box has a *call* point and k *return* points, and each edge takes from a node/return to a node/call in the component. Nodes and returns are split into player 0 (pl_0) positions and player 1 (pl_1) positions.

For a box b, we denote with $(1, b)$ the only call of b and with (b, i) the i^{th} return of b for $i \in [k]$. A *k-component* C is a tuple $(N_C, B_C, e_C, Ex_C, \eta_C, \delta_C, P_C^0, P_C^1)$ where N_C is a finite set of nodes, B_C is a finite set of boxes, $e_C \in N_C$ is the entry, $Ex_C : [k] \to N_C$ is an injection that maps each i to the i^{th} exit, $\eta_C : V_C \to \Sigma$ is a labeling map of the *set of C vertices* $V_C = N_C \cup Calls_C \cup Retns_C$, $\delta : N_C \cup Retns_C \to 2^{N_C \cup Calls_C}$ is a transition function with $Retns_C = \{(b, i) \mid b \in B_C, i \in [k]\}$ (*set of C returns*) and $Calls_C = \{(1, b) \mid b \in B_C\}$ (*set of C calls*), and P_C^0 (the pl_0 positions) and P_C^1 (the pl_1 positions) form a partition of $N_C \cup Retns_C$.

We introduce the notion of *isomorphism* between two k-components. Intuitively, two components are isomorphic if and only if their game structures are equivalent, that is: the properties of standard isomorphism of labeled graphs must hold, and additionally isomorphic vertices must be assigned to the same player and be of the same kind.
Formally, the k-components C and C' are *isomorphic*, denoted $C \overset{\text{iso}}{\equiv} C'$, if there exists a bijection $iso : V_C \cup B_C \to V_{C'} \cup B_{C'}$ s.t.: (1) for all $u, v \in V_C$, $v \in \delta_C(v)$ iff $iso(v) \in \delta_{C'}(iso(u))$ and (2) for $u \in V_C \cup B_C$ and $u' \in V_{C'} \cup B_{C'}$, we get $u' = iso(u)$ iff u and u'

- *have the same labeling*, i.e. $\eta_C(u) = \eta_{C'}(u')$;
- *are assigned to the same player*, i.e., $u \in P_C^j$ iff $u' \in P_{C'}^j$ for $j \in [0, 1]$;
- *are of the same kind*, i.e.:

- u is an entry/box of C iff u' is an entry/box of C';
- for $i \in [k]$, u is the i^{th} exit of C iff u' is the i^{th} exit of C';
- $u = (1, b)$ iff $u' = (1, iso(b))$ and for $i \in [k]$, $u = (b, i)$ iff $u' = (iso(b), i)$ (*calls and i^{th}-returns of isomorphic boxes must be isomorphic*).

For $k > 0$, a *k-library* is a tuple $\mathcal{L}ib = \langle \{C_i\}_{i\in[0,n]}, Y_{\mathcal{L}ib} \rangle$ where:

- $\{C_i\}_{i\in[0,n]}$ is a finite set of k-components;
- C_0 is the *main component*;
- let $B_{\mathcal{L}ib} = \bigcup_{i\in[0,n]} B_{C_i}$ be the set of all boxes of the library components, $Y_{\mathcal{L}ib} : B_{\mathcal{L}ib} \to \{C_i\}_{i\in[n]}$ is a partial function (*box-to-component map*).

Running Example. We illustrate the definitions with an example. In Fig.7.1(a), we give a library $\mathcal{L}ib$ of four components C_0, C_1, C_2 and C_3. Each component has two exits. In the figure, we denote the nodes of pl_0 with circles and the nodes of pl_1 with squares. Rounded squares are used to denote the boxes. Entries (resp., exits) are denoted by nodes intersecting the frame of the component on the left (resp., on the right). For example, C_0 has entry e_0 and two exits x_1 and x_2, one internal node u_1 and two boxes b_1 and b_2. With "$b_1 : C_1$" we denote that box b_1 is mapped to component C_1. The only unmapped box is b_3. To keep the figure simple, we only show the labeling of vertices with labels α, β and γ, and hide the labeling for all the remaining vertices (meaning that they are labeled with any other symbol).

Notes. For the ease of presentation, we have imposed a few restrictions. First, in the definition of library, $Y_{\mathcal{L}ib}$ can map a box to each component but the main component C_0. We observe that this is in accordance with the choice of many programming languages where the main function cannot be called by other functions and is without loss of generality of our results. Second, multiple entries can be handled by making for each component as many copies as the number of its entries, and accommodating calls and returns accordingly. Third, all the components of a library have the same number of exits that also matches the number of returns for each box. This can be relaxed at the cost of introducing a notion of *compatibility* between a box and a component, and map boxes to components only when they are compatible. We make a further assumption that is standard: in the components there are no transitions leaving from exits (assigning them to pl_0 or pl_1 is thus irrelevant).

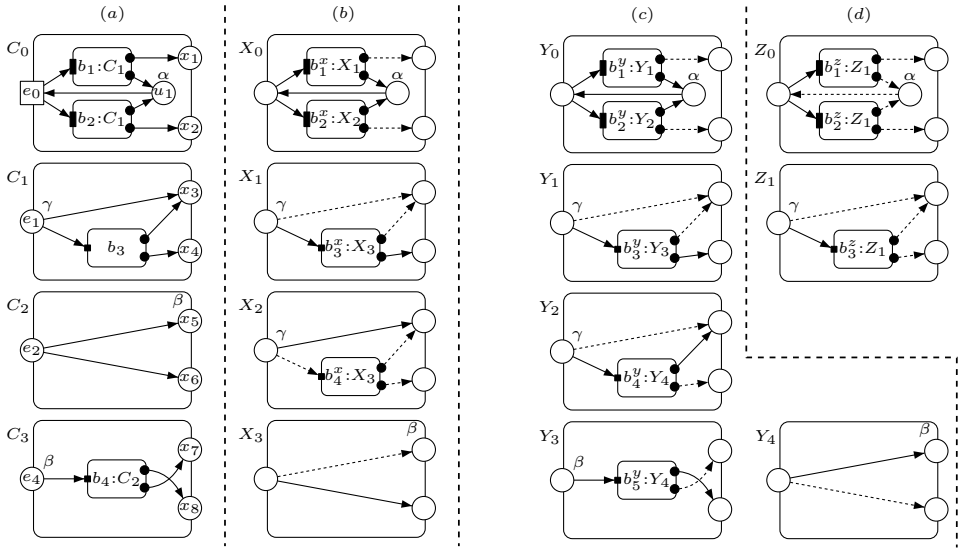

Figure 7.1: A library (a) and RSMs from it: unrestricted (b), same local strategy for instances of the same component (c), and at most one instance for each component (d).

7.3.2 Instances and recursive state machines

We are interested in synthesizing a *recursive state machine* (RSM) (2) from a library of components. Such a machine is formed by a finite number of *instances* of library components, where each instance is isomorphic to a library component and resolves the nondeterminism of pl_0 by a finite-state local strategy. The boxes of each instance are mapped to instances in the machine with the meaning that when a call of a box b is reached then the execution continues on the entry of the mapped instance and when the i^{th} exit of such instance is reached then it continues at the i^{th} return of b (as in the recursive call-return paradigm). The box-to-instance map of an RSM must agree with the box-to-component map of the library when this is defined.

We observe that our definition of RSM differs from the standard one in that (i) each finite-state machine is implicitly given by a component and a finite-state local strategy, and (ii) the nodes are split between pl_0 and pl_1. (However the last is immaterial since the nondeterminism of pl_0 is completely resolved by the local strategies.)

For a component C, a *local strategy* is $S : V_C^*.P_C^0 \to Calls_C \cup N_C$ such that $S(w.u) \in \delta_C(u)$. The strategy is *finite-state* if it is computable by a finite automaton (we omit a formal definition here, see (42)).

An *instance of* C is $I = (G, S)$ where G is s.t. $G \overset{\text{iso}}{\equiv} C$ holds and S is a finite-state local strategy of G. For example, in Fig. 7.1, X_1 and X_2 are two instances of C_1 that differ on the local strategy (we have denoted with dashed edges the transitions that cannot be taken because of the local strategies). Also, Y_1 is an instance of C_1 and has the same local strategy as X_1. Note that, though the local strategies used in this example are memoryless, this is not mandatory and thus the number of instances of each component with different local strategies is in general unbounded.

Fix a library $\mathcal{L}ib = \langle \{C_i\}_{i\in[0,n]}, Y_{\mathcal{L}ib}\rangle$. A *recursive state machine (RSM) from* $\mathcal{L}ib$ is $S = \langle \{I_i\}_{i\in[0,m]}, Y_S\rangle$ where:

- for $i \in [0, m]$, $I_i = (G_i, S_i)$ is an instance of a component C_{j_i} from $\mathcal{L}ib$;
- I_0 is an instance of the main component C_0;
- the *box-to-instance* map $Y_S : \bigcup_{i\in[0,m]} B_{G_i} \to \{I_i\}_{i\in[m]}$ is a total function that is consistent with $Y_{\mathcal{L}ib}$, i.e., for each $i \in [0, m]$ and $b \in B_{G_i}$, denoting with b' the box of C_{j_i} that is isomorphic to b, it holds that if $Y_{\mathcal{L}ib}(b') = C_{j_h}$ then $Y_S(b) = G_h$.

Examples of RSM for the library from Fig. 7.1(a) are given in Fig.7.1(b)–(d).

We assume the following notation: $V_S = \bigcup_{i\in[0,m]} V_{G_i}$ (set of all vertices); $B_S = \bigcup_{i\in[0,m]} B_{G_i}$ (set of all boxes); $En_S = \bigcup_{i\in[0,m]} \{e_{G_i}\}$ (set of all entries); $Ex_S = \bigcup_{i\in[0,m]} Ex_{G_i}$ (set of all exits); $Calls_S = \bigcup_{i\in[0,m]} Calls_{G_i}$ (set of all calls); $Retns_S = \bigcup_{i\in[0,m]} Retns_{G_i}$ (set of all returns); and $P^j_S = \bigcup_{i\in[0,m]} P^j_{G_i}$ for $j = 0, 1$ (set of all positions of pl_j).

A *state* of S is (γ, u) where $u \in V_{Y_S(b_h)}$ is a vertex and $\gamma = \gamma_1 \dots \gamma_h$ is a finite sequence of pairs $\gamma_i = (b_i, \mu_i)$ with $b_i \in B_S$ and $\mu_i \in V^*_{Y_S(b_i)}$ for $i \in [h]$ (respectively, *calling box* and *local memory* of the called instance).

In the following, for a state $s = (\gamma, u)$, we denote with $V(s)$ its vertex u. Moreover, we define the *labeling map of* S, denoted η_S, from the labeling η_{G_i} of each instance I_i in the obvious way, i.e., $\eta_S(s) = \eta_{G_i}(V(s))$ for each $V(s) \in V_{G_i}$ and $i \in [0, m]$. η_S naturally extends to sequences.

A *run* of S is an infinite sequence of states $\sigma = s_0 s_1 s_2 \dots$ such that $s_0 = ((\epsilon, e_{G_0}), e_{G_0})$ and for $i \in \mathbb{N}$, denoting $s_i = (\gamma_i, u_i)$ and $\gamma_i = (b_1, \mu_1) \dots (b_h, \mu_h)$, one of the following holds:

– **Internal** pl_1 **move:** $u_i \in (N_S \cup Retns_S) \setminus Ex_S$, and $u_i \in P^1_S$, then $u_{i+1} \in \delta_S(u_i)$ and $\gamma_{i+1} = (b_1, \mu_1) \dots (b_h, \mu_h.u_{i+1})$;

– **Internal pl_0 move:** $u_i \in (N_S \cup Retns_S) \setminus Ex_S$, $u_i \in P_S^0$ and $u_i \in V_{G_j}$ with $j \in [0, m]$, then $u_{i+1} = S_j(\mu_h)$ and $\gamma_{i+1} = (b_1, \mu_1) \dots (b_h, \mu_h.u_{i+1})$.

– **Call to an instance:** $u_i = (1, b) \in Calls_S$, $u_{i+1} = e_{Y_S(b)}$ and $\gamma_{i+1} = \gamma_i.(b, e_{Y_S(b)})$;

– **Return from a call:** $u_i \in Ex_S$ and u_i corresponds to the j^{th} exit of an instance I_h, then $u_{i+1} = (b_h, j)$ and $\gamma_{i+1} = (b_1, \mu_1) \dots (b_{h-1}, \mu_{h-1}.u_{i+1})$.

An *infinite* RSM is defined as an RSM where we just relax the request that the set of instances is finite. We omit a formal definition and retain the notation. Note that the definitions of state and run given above still hold in this case.

7.3.3 A general synthesis problem

Fix a library $\mathcal{L}ib = \langle \{C_i\}_{i \in [0,n]}, Y_{\mathcal{L}ib} \rangle$ with alphabet Σ.

A *library game* is $(\mathcal{L}ib, W)$ where $\mathcal{L}ib$ is a library of components and W is a *a winning set*, i.e., a language $W \subseteq \Sigma^\omega$.

The *modular synthesis from libraries* (LMS, for short) is the problem of determining if for a given library game $(\mathcal{L}ib, W)$ there is an RSM $S = \langle \{I_i\}_{i \in [0,m]}, Y_S \rangle$ from $\mathcal{L}ib$ that *satisfies* W, i.e., $\eta_S(\sigma) \in W$ for each run σ of S.

As an example, consider the LMS queries $\mathcal{Q}_i = (\mathcal{L}ib, W_i)$, $i \in [3]$, where $\mathcal{L}ib$ is from Fig. 7.1(a) and denoting $\Sigma = \{\alpha, \beta, \gamma\}$: W_1 is the set of all ω-words whose projection into Σ gives the word $(\gamma\alpha)^\omega$, W_2 is the set of all words whose projection into Σ gives a word in $(\gamma\beta\alpha + \gamma\beta^2\alpha)^\omega$, and W_3 is the set of all ω-words with no occurrences of β. The RMSs from Fig. 7.1(b)–(d) are solutions of the LMS queries $\mathcal{Q}_1, \mathcal{Q}_2$ and $\mathcal{Q}_3$ respectively. In the figure, we use circles to denote all the nodes, this is to stress that the splitting between the two players is not meaningful any more.

We recall that a solution in the considered synthesis problems over a library game $(\mathcal{L}ib, W)$ has two levels: the game within each component and the compositional game among instances. Consequently, on the one hand, we must solve the internal game, determining how many instances will compose our system and how they will be controlled. This means that the first feature of the solution is to generate the set of instances obtained from $\mathcal{L}ib$ components and, in particular, the set of finitely representable local strategies associated to each element of the set. On the other hand,to construct the system, we must decide how these instances interact using the procedure calls. This means that the second feature of our solution is to determine the external compositional

game, i.e., defining a box mapping compatible with the mapping of $\mathcal{L}ib$, and, in such way, constructing the recursive state machine.

In the following, we refer to the set of local strategies of a synthesized RSM that fulfills the winning condition as a *winning modular strategy*.

7.4 Other formulations of the modular synthesis

We introduce two variations of the Lms problem based on the two restrictions for the RSMs that can be synthesized. The idea is to constrain our algorithms to synthesize, when possible, "simpler" RSMs. For example, in the function call repair we can imagine that it is not good to fix a fault introducing or duplicating an arbitrarily large number of new instances and we could be interested to construct a repaired system that implements at most one instance of each library component.

Fix a library $\mathcal{L}ib$. An RSM S from $\mathcal{L}ib$ is *component-based* if for any two S instances $I = (G, f)$ and $I' = (G', f')$ of a component C from $\mathcal{L}ib$, the local strategies f and f' *coincide* (up to a renaming). Moreover, S is *single-instance* if it has *at most* one instance of each library component.

The *component-based* (resp. single-instance) Lms problem is the Lms problem restricted to component-based (resp. single-instance) RSMs.

Denote with P_{single} (resp. P_{comp}, P_{LMS}) the set of Lms queries ($\mathcal{L}ib$, W_A) for which the single-instance Lms problem (resp. component-based Lms problem, Lms problem) admits a positive answer. Directly from the definitions, a single-instance RSM is also component-based. Thus we get that $P_{single} \subseteq P_{comp} \subseteq P_{LMS}$. These inclusions are indeed strict.

Let $\mathcal{L}ib$ be the library from Fig.7.1(a). The RSM in Fig.7.1(b) is not component-based (and thus not single-instance): X_1 and X_2 are instances of C_1 and use two different local strategies. The RSM in Fig. 7.1(c) instead is component-based but not single-instance since Y_1 and Y_2 are two instances of C_1 (note that even if they have the same local strategy, they differ on the reachable vertices because the box is mapped differently). The RSM from Fig. 7.1(d) is clearly single-instance.

Let W_1, W_2 and W_3 be the winning conditions given at the end of Section 7.3. Observe that they are all expressible by safety automata. Moreover, there is no component-

based RSM from $\mathcal{L}ib$ that satisfies W_1 and no single-instance RSM from $\mathcal{L}ib$ that satisfies W_2. Thus, we get the following lemma:

Lemma 27. $P_{single} \subset P_{comp} \subset P_{LMS}$.

The single-instance LMS problems and the synthesis of modular strategies on recursive game graphs are strictly related: a modular game is a single-instance LMS game where the box-to-component map is total. Given an instance of single-instance LMS game, we guess a total box-to-component map for the library and then we can solve all the considered single-instance LMS problems applying the algorithms proposed in (4, 5) and in Section 5.3. We get:

Theorem 28. *The safety and VPA single-instance* LMS *problems are* EXPTIME*-complete. The reachability single-instance* LMS *problem is* NP*-complete.*

8

Modular synthesis with reachability conditions

In this chapter we formally introduce and solve the simpler version of the component-based synthesis problem seen in Chapter 7. In this case we want to synthesize a system from a library of open components without guiding the composition and considering only reachability winning conditions. To solve this problem, we give an exponential-time fixed-point algorithm that computes annotations for the vertices of the library components by exploring them backwards. We also show a matching lower-bound via a direct reduction from linear-space alternating Turing machines, thus proving Exptime-completeness. therefore, we give a second algorithm that solves this problem by annotating in a table the result of many local reachability game queries on each game component. This algorithm is exponential only in the number of the exits of the game components, and thus shows that the problem is fixed-parameter tractable.

In the last part of this chapter, we modify the proposed algorithm to solve the Lms and the component-based Lms problem with reachability winning condition.

8.1 Contribution

The main contributions present in this chapter are:

- We introduce and solve the simpler version of a component-based synthesis problem. We model the components of our libraries as game modules of a recursive game graph with unmapped boxes (the synthesis is not guided by a box-

to-component mapping), and we limit us to consider as correctness specification only a target set of vertices. This problem is a restriction of a reachability LMS problem, where there is no partial mapping.

- We show that the proposed problem (*modular synthesis problem*) is decidable and we present a fixed-point algorithm $\mathcal{A}_1$ that decides in exponential time such modular synthesis problem. This algorithm iteratively computes a set Φ of tuples of the form $(u, E, \{\mu_b\}_{b \in B})$ where u is a vertex of a game component C, E is a set of C exits, B is the set of C boxes and for each box $b \in B$, μ_b is either a set of exits of another component C_b or undefined. Each such tuple summarizes for vertex u a reachable *local target* E (via a modular strategy of pl_0) and a set of *assumptions* $\{\mu_b\}_{b \in B}$ that are used to get across the boxes in order to reach the local target. We start from the tuples of the target exits $\mathcal{T}$ and then propagate the search backwards in the game components. Internally to each component, the search proceeds as in the standard attractor set construction (32) and it is propagated through calls to other components from the returns to the exits and then back from the entries to the calls. In this, tuples that have incompatible assumptions or refer to a different local target are treated as belonging to different searches and thus are not used together in the update rules.

- We show the matching lower bound by a reduction from linear-space alternating Turing machines. In the reduction, we use only four game components and $O(n)$ exits, where n is the number of cells used in the configurations of the Turing machine.

- We give a second decision algorithm $\mathcal{A}_2$ and we introduce it to show that the computational complexity of the proposed problem becomes PTIME when the number of exits is fixed. The main idea here is to solve many reachability game queries "locally" to each game component and maintain a table with the obtained results to avoid recomputing. Each table entry corresponds to a game component and a set of its exits (used as targets in the query), and for the successful queries, contains a link to each table entry that has been used to reach the target (we look up into the table to propagate the search across the boxes). We observe that $\mathcal{A}_2$ takes time exponential only in the number of exits, while $\mathcal{A}_1$ takes time

exponential also in the number of boxes. This is due mainly to the fact that $\mathcal{A}_1$ may compute and store exponentially many different ways of assigning the boxes to modules, in contrast, $\mathcal{A}_2$ computes and stores just one of them. Therefore, since alternating reachability in finite game graphs is already PTIME-hard, by algorithm $\mathcal{A}_2$ we get that the considered problem is PTIME-complete when the number of exits is fixed.

- We consider the general reachability LMS problem and we modify the proposed algorithm to handle the box-to-component mapping. We also present the solutions for the reachability component-based LMS problems.

8.2 A simpler modular synthesis problem

In this section, we consider a simplified version of the LMS problem, the *modular synthesis problem*. In this setting, our model *does not allow* to guide the composition of the instances and as winning condition we consider only the *reachability condition*. The solution of the general case, but still limited to reachability conditions, will be presented in the last part of this chapter. We discuss about complex winning conditions in Chapter 9.

Library of (game) components. For $h, k \in \mathbb{N}$, a (h,k)-*component* is a finite graph with two kinds of vertices, the standard *nodes* and the *boxes*, and with h *entry* nodes and k *exit* nodes. Each box has h *call* points and k *return* points, and the edges take from a node/return to a node/call in the component.

Formally, for a box b, we denote with (i,b) the i-th call of b for $i \in [h]$, and with (b,i) the i-th return of b for $i \in [k]$. A (h,k)-*component* is a tuple (N, B, En, Ex, δ) where N is a finite set of nodes, B is a finite set of boxes, $En \subseteq N$ is the set of entries, $Ex \subseteq N$ is the set of exits, and $\delta : N \cup Retns \to 2^{N \cup Calls}$ where $Retns = \{(b,i) \mid b \in B, i \in [k]\}$ and $Calls = \{(i,b) \mid b \in B, i \in [h]\}$. The calls, returns and nodes of a component form its set of vertices. In the following, when we do not need to specify h and k, we simply write component.

A *game component* is a component whose nodes and returns are split into two sets P^0 and P^1, where P^0 is the set of player 0 (pl_0) positions and P^1 is the set of player 1 (pl_1) positions. We denote it as a tuple $(N, B, En, Ex, \delta, P^0, P^1)$.

For $h, k > 0$, a *library* of (game) components is a finite set $\mathcal{L}ib = \{C_i\}_{i\in[n]}$ where each C_i is a (game) (h, k)-component.

To ease the presentation we make the following standard assumptions:

- there is only one entry node for every (game) component and thus just one call for each box, i.e., we refer to (game) $(1, k)$-components;
- in each (game) component there are no transitions taking to its entry and no transitions leaving from its exits, i.e., the entries are sources and the exits are sinks in the graph representation of the component;
- there is no transition from a return to a call, i.e., two boxes are not directly connected by a single transition.

Instances from a library. Intuitively, an instance of a (game) component C is a copy A of C where each box is mapped to an instance of a (game) component (possibly A itself). Depending on whether we consider a library of components or of game components, the instances define a *recursive state machine* (2) or a *recursive game graph* (5).

Fix a library $\mathcal{L}ib = \{C_1, \ldots, C_n\}$ of game components.

A *recursive game graph* from $\mathcal{L}ib$ is $G = (M, m_{in}, \{S_m\}_{m\in M})$ where M is a finite set of module names, $m_{in} \in M$ is the name of the initial module and for each $m \in M$, S_m is a game module. A *game module* S_m is defined as $(N_m, B_m, Y_m, \{e_m\}, Ex_m, \delta_m, P^0_m, P^1_m)$ where:

- $Y_m : B_m \to (M \setminus \{m_{in}\})$ is a labeling function that maps every box to a game module;
- $(N_m, B_m, \{e_m\}, Ex_m, \delta_m, P^0_m, P^1_m)$ is equal to a component C of $\mathcal{L}ib$ up to a renaming of nodes and boxes such that calls and returns of a box b are 1-to-1 mapped to the entries and the exits of $M_{Y_m(b)}$, that is, denoting $Ex_{Y_m(b)} = \{x_1, \ldots, x_k\}$: the call of b is renamed to $(e_{Y_m(b)}, b)$ and each return (b, i) is renamed to (b, x_i).

The calls, returns and vertices of S_m are denoted respectively $Calls_m$, $Retns_m$ and V_m. We also assume the following notation: $V = \bigcup_m V_m$ (set of all vertices); $B = \bigcup_m B_m$ (set of all boxes); $Calls = \bigcup_m Calls_m$ (set of all calls); $Retns = \bigcup_m Retns_m$ (set of all returns); and $P^i = \bigcup_m P^i_m$ for $i = 0, 1$ (set of all positions of pl_i).

The definition of a *recursive state machine* from $\mathcal{L}ib$ can be obtained from that of recursive game graph by ignoring the splitting among pl_0 and pl_1 nodes.

A (global) *state* of G is composed of a call stack and a vertex. Formally, the states are of the form $(\gamma, u) \in B^* \times V$ where $\gamma = b_1 \dots b_h$, $b_1 \in B_{m_{in}}$, $b_{i+1} \in B_{Y(b_i)}$ for $i \in [h-1]$ and $u \in V_{Y(b_h)}$. In the following, for a state $s = (\gamma, u)$, we denote with $V(s)$ its vertex, that is $V(s) = u$.

A *play* of G is a (possibly finite) sequence of states $s_0 s_1 s_2 \dots$ such that $s_0 = (\epsilon, e_{m_{in}})$ and for $i \in \mathbb{N}$, denoting $s_i = (\alpha_i, u_i)$, one of the following holds:

- **Internal move:** $u_i \in (N_m \cup Retns_m) \setminus Ex_m$, $u_{i+1} \in \delta_m(u_i)$ and $\alpha_i = \alpha_{i+1}$;
- **Call to a module:** $u_i \in Calls_m$, $u_i = (b, e_{m'})$, $u_{i+1} = e_{m'}$ and $\alpha_{i+1} = \alpha_i.b$;
- **Return from a call:** $u_i \in Ex_m$, $\alpha_i = \alpha_{i+1}.b$, and $u_{i+1} = (b, u_i)$.

Modular strategies. A *strategy* of a player pl is a function f that associates a legal move to every play ending in a node controlled by pl. A *modular strategy* (5) for G consists of a set of local strategies, that are used together as a global strategy for a player. A local strategy for a game module S can only refer to the local memory of S, i.e. the sequence of S vertices that are visited in the play in the current invocation of S.

Formally, fix $j \in \{0, 1\}$. A *modular strategy* f of pl_j is a set of functions $\{f_m\}_{m \in M}$, one for each game module, where for every m, $f_m : V_m^*.P_m^j \to V_m$ such that $f_m(\pi.u) \in \delta_m(u)$ for every $\pi \in V_m^*, u \in P_m^j$.

Fix a play $\pi = s_0 s_1 ... s_n$ where $s_i = (\gamma_i, u_i)$ for any i. Denote with $\pi_i = s_0 s_1 ... s_i$, i.e., the prefix of π up to index i. With $ctr(\pi_i)$ we denote $m \in M$ such that $u_i \in V_m$, that is the name of the game module where the control is after π_i. The *local history* at π_i, denoted $\lambda(\pi_i)$, is the maximal sequence of S_m vertices u_j, $j \leq i$, starting with the most recent occurrence of entry e_m where $m = ctr(\pi_i)$.

A play π *conforms to* a modular strategy $f = \{f_m\}_{m \in M}$ of pl_j if for every $i <| \pi |$, denoting $ctr(\pi_i) = m$, $u_i \in P_m^j$ implies that $u_{i+1} = f_m(\lambda(\pi_i))$.

Modular synthesis from libraries of game components. A *modular game over a library* is $(\mathcal{L}ib, C_{main}, \mathcal{T})$ where $\mathcal{L}ib$ is a library of game components, $C_{main} \in \mathcal{L}ib$ and $\mathcal{T}$ is a set of exits of C_{main}.

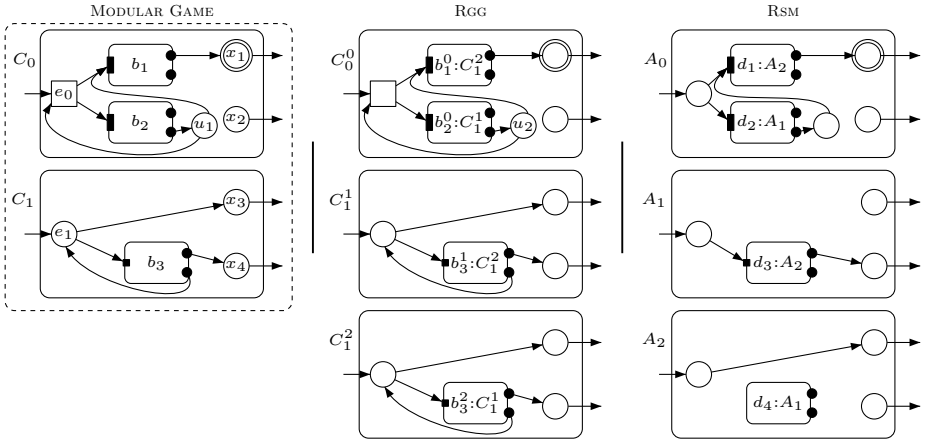

Figure 8.1: An example of modular synthesis

Given an instance $(\mathcal{L}ib, C_{main}, \mathcal{T})$ of a modular game over a library, the *modular synthesis problem* is the problem of determining whether: for some recursive game graph G from $\mathcal{L}ib$ whose initial module is an instance of C_{main}, there exists a modular strategy f for pl_0 in G such that all the maximal plays that conform to f reach an exit of the initial module of G that corresponds to an exit in $\mathcal{T}$.

Such a strategy f for pl_0 is called a *winning modular strategy.*

Example. We illustrate the definitions with an example. In the first column of Fig. 8.1, we give $(\mathcal{L}ib, C_0, \{x_1\})$, an instance of a modular game over a library of game components. Each game component has two exits, and $\mathcal{L}ib$ is composed of two game components C_0 and C_1. In the figure, we denote the nodes of pl_0 with circles and the nodes of pl_1 with squares. Rounded squares are used to denote the boxes. The target is marked with a double circle. C_0 has one entry e_0, two exits x_1 and x_2, and two boxes b_1 and b_2. C_1 has one entry e_1, two exits x_3 and x_4, and one box b_3.

In the second column of the figure, we show one of the possible recursive game graphs that can be obtained from $\mathcal{L}ib$ and whose initial module C_0^0 is an instance of C_0. Note that we have marked as target the vertex of C_0^0 that corresponds to (i.e., is a copy of) x_1. The other modules C_1^1 and C_1^2 are instances of C_1. Note that each box now is mapped to a game module, for example b_1^0 is mapped to C_1^2. Also, the box b_3^1 of C_1^1 is mapped to C_1^2 and the box b_3^2 of C_1^2 is mapped to C_1^1 thus forming a cycle in the chain of recursive calls.

Consider a modular strategy for pl_0, where the local strategy of C_0^0 selects the call from u_2, the local strategy of C_1^1 selects the call from its entry and the local strategy for C_1^2 selects the upper exit from its entry. This strategy is winning and modular. In the third column of the figure, we show a recursive state machine, obtained from the considered recursive game graph by resolving the moves of pl_0 according to this modular strategy. To simplify the figure, we have deleted all the unreachable transitions. Clearly, each run of this machine reaches the target. Also, note that in the considered game it is not possible to win if we do not instantiate at least two instances of C_1.

8.3 Solving our modular synthesis problem

In this section, we describe an exponential-time fixed-point algorithm to solve the modular synthesis problem.

We fix a library of game components $\mathcal{L}ib = \{C_{main}, C_1, \ldots, C_n\}$ and a target set $\mathcal{T}$ of C_{main} exits.

Intuitively, our algorithm iteratively computes a set Φ of tuples of the form $(u, E, \{\mu_b\}_{b\in B})$ where u is a vertex of a game component C, E is a set of C exits, B is the set of C boxes and for each box $b \in B$, μ_b is either a set of exits of a game component or undefined (we use $\perp$ to denote this). The intended meaning of such tuples is that: there is a local strategy f of pl_0 in C such that starting from u, each maximal play conforming to f reaches an exit within E, under the assumption that: for each box $b \in B$, if μ_b is defined, then from the call of b the play continues from one of the returns of b corresponding to a $x \in \mu_b$ (if μ_b is undefined means that no play conforming to f visits b starting from u). Thus, each tuple $(u, E, \{\mu_b\}_{b\in B})$ summarizes for vertex u a reachable *local target* E and a set of *assumptions* $\{\mu_b\}_{b\in B}$ that are used to get across the boxes.

For computing Φ, we use the concept of compatibility of the assumptions. Namely, we say that two assumptions μ and μ' are *compatible* if either $\mu = \mu'$, or $\mu' = \perp$, or $\mu = \perp$ (i.e., there is at most one assumption that has been done). Moreover, we say that the assumptions $\mu_1, \ldots, \mu_m$ are *passed to* μ if $\mu = \bigcup_{i\in[m]} \mu_i$ (we assume that $\perp \cup X = X \cup \perp = X$ holds for each set X).

The set Φ is initialized with all the tuples of the form $(u, \mathcal{T}, \{\perp\}_{b\in B_{main}})$ where $u \in \mathcal{T}$ and B_{main} is the set of boxes of C_{main}. Then, Φ is updated by exploring the components backwards according to the game semantics, and in particular: within the components,

tuples are propagated backwards as in an attractor set construction, by preserving the local target and passing to a node the assumptions of its successors (provided that multiple assumptions on the same box are are passed they are pairwise compatible); the exploration of a component is started from the exits with no assumptions on the boxes, whenever the corresponding returns of a box b have been discovered with no assumptions on b; the visit of a component is resumed at the call of a box b, whenever (1) there is an entry of a component that has been discovered with local target X and (2) there is a set of b returns corresponding to the exits X with no assumptions on b (thus, that can be responsible for discovering the exits in X as in the previous case) and with compatible assumptions on the remaining boxes; if this is the case, then the call is discovered with the assumption X on box b and passing the local target and the assumptions on the other boxes as for the above returns.

Below, we denote with b_x the return of a box b corresponding to an exit x (recall that all game components of a library have the same number of exits, and so do the boxes). The update rules are formally stated as follows:

UPDATE 1: For a pl_0 vertex v, we add $(v, E, \{\mu_b\}_{b\in B})$ provided that there is a transition from v to u and $(u, E, \{\mu_b\}_{b\in B}) \in \Phi$ (the local target and the assumptions of a v successor are passed on to a pl_0 vertex v).

UPDATE 2. For a pl_1 vertex v, denote $u_1, \ldots, u_m$ all the vertices s.t. there is a transition from v to u_i, $i \in [m]$, then we add $(v, E, \{\mu_b\}_{b\in B})$ to Φ provided that for each $i, j \in [m]$ and $b \in B$: (1) there is a $(u_i, E_i, \{\mu^i_b\}_{b\in B}) \in \Phi$, (2) $E_i = E_j$, (3) μ^i_b and μ^j_b are compatible, and (4) $\mu_b = \bigcup_{i\in[m]} \mu^i_b$ (all the v successors must be discovered under the same target and with compatible assumptions; target and assumptions are passed on to a pl_1 vertex v).

UPDATE 3. For an exit u, we add a tuple $(u, E, \{\bot\}_{b\in B'})$ to Φ provided that $u \in E$ and for a box b' it holds that there are tuples $(b'_x, E_x, \{\mu^x_b\}_{b\in B}) \in \Phi$, one for each $x \in E$, such that for all $x, y \in E$ and $b \in B$, (1) $\mu^x_{b'} = \bot$, (2) $E_x = E_y$, and (3) μ^x_b and μ^y_b are compatible (the discovery of the exits follows the discovery of the corresponding returns under compatible assumptions and the same local target).

UPDATE 4. For a call u of a box b', we add a tuple $(u, E_u, \{\mu^u_b\}_{b\in B})$ to Φ provided that (i) there is an entry e s.t. $(e, E_e, \{\mu^e_b\}_{b\in B'}) \in \Phi$, (ii) for each return b'_x,

$x \in E_e$, there is a tuple $(b_x, E, \{\mu_b^x\}_{b \in B}) \in \Phi$ s.t. all these tuples satisfy (1), (2) and (3) of UPDATE 3, and moreover, (iii) $E_u = E$, $\mu_b^u = \bigcup_{x \in E_e} \mu_b^x$ for $b \neq b'$, and $\mu_{b'}^u = E_e$ (the discovery of a call u of box b' follows the discovery of an entry e from exits E_e that in turn have been discovered by matching returns b'_x, $x \in E$; thus on u we propagate the local target and the assumptions on the boxes $b \neq b'$ of the returns b'_x and make an assumption E_e on box b').

We compute Φ as the fixed-point of the recursive definition given by the above rules and outputs "YES" iff $(e, \mathcal{T}, \{\mu_b\}_{b \in B_{main}}) \in \Phi$ for the entry e of C_{main}.

Observe that, the total number of tuples of the form $(u, E, \{\mu_b\}_{b \in B})$ is bounded by $|\mathcal{L}ib|\, 2^{O(k\beta)}$ where k is the number of exits of each game component in $\mathcal{L}ib$ and β is the maximum over the number of boxes of each game component. Therefore, the algorithm always terminates and takes at most time exponential in k and β, and linear in the size of $\mathcal{L}ib$.

Soundness of the algorithm is a consequence of the fact that each visit of a game component is done according to the standard attractor set construction, and repeated explorations of each component are kept separate by allowing to progress backwards in the graph only with the same local target and compatible assumptions on the boxes. By not allowing to change the box assumptions (when defined), we ensure that we cannot cheat by using different assumptions in repeated visits of a box within the same exploration. The computed strategy is clearly modular since we compute it locally to each graph component. Note that we can end up computing more than a local strategy for each graph component, but this does not break the modularity of the solution since this happens when in the computed solution we use different instances of the component. Also, observe that for each game component we construct at most a local strategy for each possible subset of its exits, thus we bound the search of a solution to modular strategies of this kind.

To prove completeness, we first observe that using standard arguments one can show that:

Lemma 29. *If there is a modular winning strategy for an instance of the modular synthesis problem over a library $\mathcal{L}ib$, then there is a winning modular strategy f for a recursive game graph G from $\mathcal{L}ib$ such that: for each two instances S and S' of a same game component in $\mathcal{L}ib$, the sets of exits visited along any play conforming to f in S and S' differ.*

Observe that by the above lemma, we can restrict the search for a solution within the modular strategies of the instances of a $\mathcal{L}ib$ that have at most 2^k copies of each game component, where k is the number of exits for the components. Therefore, combining this with the results from (5) we get a simple argument to show membership to NEXPTIME of the considered problem.

The next step in the completeness argument is to show that if there is a winning modular strategy f as for Lemma 29, then our algorithm outputs YES. Denoting with G the recursive game graph from $\mathcal{L}ib$ for which f is winning, this can be shown by proving by induction on the structure of G that: if on a game module S of G that is an instance of $C \in \mathcal{L}ib$, f forces to visit a set of exits corresponding to the exits X of C, then the algorithm adds to Φ the tuples $(x, X, \{\bot\}_{b \in B})$ for each $x \in X$ and eventually discovers the entry of C with local target X. We omit the proof of this here.

Therefore, we get that the algorithm is a solution of the modular synthesis problem from game component libraries, and the following theorem holds.

Theorem 30. *The modular synthesis problem from libraries of game components with k exits and at most β boxes can be solved in time linear in the size of $\mathcal{L}ib$ and exponential in k and β.*

8.4 Computational complexity analysis

Lower-bound. We reduce the membership problem for linear-space alternating Turing machines to the modular synthesis problem for libraries of game components, thus showing EXPTIME-hardness for this problem.

Consider a linear-space alternating Turing machine A and an input word $w = a_1 \dots a_n$. Without loss of generality, we assume that the transition function δ of A is the union of two functions δ_1 and δ_2 where $\delta_i : Q \times \Sigma \to \{L, R\} \times Q$ for $i \in [2]$, and Q is the set of control locations, Σ is the tape alphabet, and L/R cause to move the tape head to left/right. A configuration of A is represented as $b_1 \dots (q, b_i) \dots b_n$ where b_j is the symbol at cell j of the input tape for $j \in [n]$, q is the control state and the tape head is on cell i. The control states are partitioned into states where the $\exists$-player can move, and states where the $\forall$-player can move. A computation of M is a strategy of the $\exists$-player, and an input word w is accepted iff there exists a computation ρ that reaches a configuration with a final state on all the plays conforming to ρ.

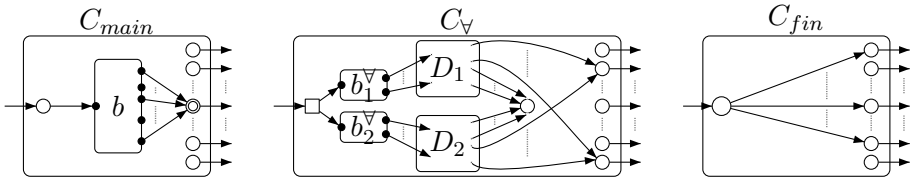

Figure 8.2: Graphical representation of the game components C_{main}, $C_\forall$ and C_{fin}

Denoting $h = n\,|\Sigma|\,(|Q|+1)$, fix two sets $X = \{x_1, \ldots, x_h\}$ and $Y = \{y_1, \ldots, y_h\}$ such that each x_i and y_i correspond exactly to a symbol and a position in a configuration of A (i.e., for each symbol in $\Sigma \cup Q \times \Sigma$ we have exactly n variables from X and n from Y, one for each position on the tape). We can encode each configuration $\sigma_1 \ldots \sigma_n$ of A by setting to true a variable x_j (resp. y_j) iff it corresponds to a σ_i for $i \in [n]$ (that is, to a configuration symbol and its position in the configuration). It is well-known that for each δ_i, we can construct a Boolean circuit (using only the logical gates AND and OR) with inputs $\bar{x} = x_1 \ldots x_h$ and outputs $\bar{y} = y_1, \ldots, y_h$, such that if $\bar{x}$ is an encoding of a configuration, then $\bar{y}$ is the next configuration after the application of the only possible transition of δ_i.

From each such circuit we can construct a game graph by replacing each AND gate with a node of pl_1 and each OR gate with a node of pl_0. We denote with D_1 and D_2 the game graphs corresponding to the above circuits for δ_1 and δ_2, respectively. The encoding of the bits is done by reachability, that is, true at an input x_i corresponds to connecting it to a vertex that can lead to the target, and false otherwise. Since the circuits compute a next configuration, from each output wire y_i that evaluates to true we will be able to get to the target by a strategy that resolves the choices on the pl_0 nodes (and thus the OR gates), and this will not be possible for those y_i that evaluates to false.

We construct a library $\mathcal{L}ib$ containing exactly the game components C_{main}, $C_\forall$, $C_\exists$, and C_{fin} (see Fig. 8.2). Each component has exactly h exits, each one corresponding to a variable x_i for $i \in [h]$. In C_{main}, we arbitrarily select an exit as the only vertex in the target $\mathcal{T}$, and link to it all the returns of the box that encode the initial configuration (we can assume that A has only one initial state). In $C_\forall$, all the exits are wired as inputs to both D_1 and D_2 except for those that correspond to states of the $\exists$-player. We add a pl_0 node that has no out-going edges and is wired as input to D_1 and D_2 for

the remaining inputs. The outputs of D_1 and D_2 are wired respectively to the boxes $b_1^\forall$ and $b_2^\forall$, and the calls of these boxes are connected to the entry, that is a pl_1 node. $C_\exists$ is as $C_\forall$ except that the entry is a pl_0 node and the exits that are not connected correspond to $\forall$-player states. The component C_{fin} has just the entry and the exits. The entry is a pl_0 node and is connected to all the exits that correspond to a final state.

It is simple to verify that if, starting from an instance of C_{main}, we map the boxes such that to reproduce an accepting computation of A, then we get a recursive game graph that admits a modular winning strategy of pl_0. Vice-versa, suppose that there is a modular winning strategy of pl_0 in the synthesis problem $(\mathcal{L}ib, C_{main}, \mathcal{T})$. First, observe that since the returns from which we reach the target encode a legal initial configuration, each game module to which we map the box b will have the corresponding exits with the same property. Moreover, in order to reach backwards the entries of all the used instances of C_{main}, $C_\forall$, and $C_\exists$, at some point we need to use a copy of C_{fin}. Now, if the initial state is a $\forall$-player state and we map b to an instance of $C_\exists$, since the exit encoding the head position and the state will not be wired to D_1 and D_2, in all the modules below in the hierarchy of calls, none of such exits will be connected to the target. Thus, also the entry of each copy of C_{fin} in this hierarchy would not be connected to the target, and so all the entries up to the entry of the copy of C_{main}, thus contradicting the hypothesis. A contradiction can be shown also in the dual case. Thus, at any point we must have mapped each box to an instance of either $C_\exists$ or $C_\forall$ depending on whether the next move is of the $\exists$-player or the $\forall$-player. Since, the graphs D_1 and D_2 ensure the correct propagations of the reachability according to the computed configurations, we can correctly reconstruct a computation ρ of A from the modular strategy. Moreover, since a winning modular strategy ensures that each maximal sequence of module calls ends with a call to an instance of C_{fin}, then each play of ρ ends in a final configuration and thus ρ is accepting, that concludes the proof.

Lemma 31. *There is a polynomial-time reduction from the membership problem for linear-space alternating Turing machines to the modular synthesis problem for libraries of game components. Moreover, the resulting library has four game components each one with at most two boxes and a number of exits which is linear in the size of the input word.*

Complexity and fixed-parameter tractability. The algorithm from the previous section, say $\mathcal{A}_1$, shows membership to EXPTIME for the modular synthesis problem. Therefore, by Lemma 31, we get:

Theorem 32. *The modular reachability problem is* EXPTIME*-complete.*

Note that $\mathcal{A}_1$ takes time exponential in both the number of boxes β and the number of exits k. We sketch a different algorithm that shows that this problem is indeed in PTIME when the number of exits for each game component is fixed.

The main idea is to solve many reachability game queries on standard finite game graphs, where each query asks to determine for a game component C and a subset of its exits E: if there exists a modular strategy f of pl_0 such that all the maximal plays, which conform to f and start from the entry of C, reach one of the exits from E. To avoid recomputing, the results of such queries are stored in a table T, and the algorithm halts when no more queries can be answered positively.

To solve the query for a component C and a set of its exits E, we extend the standard attractor set construction. Namely, we accumulate the winning set for pl_0 as usual for nodes and returns. To add the call of a box b, we look in the table for a positively answered query whose target set correspond to returns of b that are already in the winning set. If the entry of C is added to the winning set, then we update the T entry for E and C to YES, and store the links to the table entries that have been used to add the calls (observe that we just need to store exactly a link for each box that is traversed to win in the game query in order to synthesize the recursive game graph and the winning modular strategy).

With similar arguments as those used in Section 8.3, we can show that pl_0 has a winning modular strategy in the input modular synthesis problem if and only if the T entry for the target set $\mathcal{T}$ is set to YES. Since the size of the table is exponential in k and linear in β, and that solving the "local" reachability games is linear in the size of the game component and in the size of the table, we get that the whole algorithm takes time exponential in k and linear in β (and the size of the library). Since already alternating reachability is PTIME-hard, we get:

Theorem 33. *The modular reachability problem for a fixed number of exits is* PTIME*-complete.*

We observe that $\mathcal{A}_1$ computes all the solutions of the kind as from Lemma 29, by trying all the possible ways of assigning each box with all the game components. This causes the exponential in the number of boxes, but also gives a quite simple and direct way to show completeness. Moreover, the fixed-point updates of $\mathcal{A}_1$ can be implemented quite efficiently and only the sets of exits from which we can reach the target (in a series of calls) are used in the computation.

Algorithm $\mathcal{A}_2$ arbitrarily computes, for each game component and each set of exits, only one assignment of each box with a game module. Moreover, it computes (several times) all the game queries, even those with exits that cannot reach the global target $\mathcal{T}$.

Both algorithms can be used to synthesize the winning modular strategy as a recursive state machine. Also, we can modify them to compute optimal winning modular strategies with respect to some criteria, such as minimizing the number of modules, the depth of the call stack or the number of used exits.

8.5 Solving LMS and component-based LMS problems with reachability winning conditions

Reachability LMS problem: The exponential-time fixed-point algorithm that we have proposed solves the LMS problem with reachability winning conditions but the considered model does not provide the box-to-component map.

In this section, we modify the algorithm presented in Section 8.3 to handle the partial mapping function of our model.

Fix a library of components $\mathcal{L}ib = \langle \{C_i\}_{i\in[0,n]}, Y_{\mathcal{L}ib}\rangle$ and a target set $\mathcal{T}$ of C_0 exits. The algorithm iteratively computes a set Φ of tuples of the form $(u, E, \{\mu_b\}_{b\in B_C})$ where u is a vertex of a component C, E is a set of C exits, B_C is the set of C boxes and for each box $b \in B_C$, μ_b is either a set of exits of a component or undefined (we use $\perp$ to denote this). The intended meaning of such tuples is that: there is a local strategy f of pl_0 in C such that starting from u, each maximal play conforming to f reaches an exit within E, under the assumption that: for each box $b \in B_C$, if μ_b is defined, then from the call of b the play continues from one of the returns of b corresponding to a $x \in \mu_b$ (if μ_b is undefined means that no play conforming to f visits b starting from u).

Thus, each tuple $(u, E, \{\mu_b\}_{b \in B_C})$ summarizes for vertex u a reachable *local target* E and a set of *assumptions* $\{\mu_b\}_{b \in B_C}$ that are used to get across the boxes.

For computing Φ, we use the concept of compatibility of the assumptions. Namely, we say that two assumptions μ and μ' are *compatible* if either $\mu = \mu'$, or $\mu' = \bot$, or $\mu = \bot$ (i.e., there is at most one assumption that has been done). Moreover, we say that the assumptions $\mu_1, \ldots, \mu_m$ are *passed to* μ if $\mu = \bigcup_{i \in [m]} \mu_i$ (we assume that $\bot \cup X = X \cup \bot = X$ holds for each set X).

The set Φ is initialized with all the tuples of the form $(u, \mathcal{T}, \{\bot\}_{b \in B_{C_0}})$ where $u \in \mathcal{T}$ and B_{C_0} is the set of boxes of C_0. Then, Φ is updated by exploring the components backwards according to the game semantics, and in particular: within the components, tuples are propagated backwards as in an attractor set construction, by preserving the local target and passing to a node the assumptions of its successors (provided that multiple assumptions on the same box are are passed they are pairwise compatible); the exploration of a component is started from the exits with no assumptions on the boxes, whenever the corresponding returns of a box b have been discovered with no assumptions on b; the visit of a component is resumed at the call of a box b, whenever (1) there is an entry of a component that has been discovered with local target X and (2) there is a set of b returns corresponding to the exits X with no assumptions on b (thus, that can be responsible for discovering the exits in X as in the previous case) and with compatible assumptions on the remaining boxes; if this is the case, then the call is discovered with the assumption X on box b and passing the local target and the assumptions on the other boxes as for the above returns. Moreover the algorithm must verify that the association between a box and an instance is done according to the partial local function of the library $Y_{\mathcal{L}ib}$.

Below, we denote with b_x the return of a box b corresponding to an exit x (recall that all components of a library have the same number of exits, and so do the boxes). The update rules are formally stated as follows:

UPDATE 1: For a pl_0 vertex v, we add $(v, E, \{\mu_b\}_{b \in B_C})$ provided that there is a transition from v to u and $(u, E, \{\mu_b\}_{b \in B_C}) \in \Phi$ (the local target and the assumptions of a v successor are passed on to a pl_0 vertex v).

UPDATE 2. For a pl_1 vertex v, denote $u_1, \ldots, u_m$ all the vertices s.t. there is a transition from v to u_i, $i \in [m]$, then we add $(v, E, \{\mu_b\}_{b \in B})$ to Φ provided that

for each $i, j \in [m]$ and $b \in B_C$: (1) there is a $(u_i, E_i, \{\mu_b^i\}_{b \in B_C}) \in \Phi$, (2) $E_i = E_j$, (3) μ_b^i and μ_b^j are compatible, and (4) $\mu_b = \bigcup_{i \in [m]} \mu_b^i$ (all the v successors must be discovered under the same target and with compatible assumptions; target and assumptions are passed on to a pl_1 vertex v).

UPDATE 3. For an exit u, we add a tuple $(u, E, \{\bot\}_{b \in B_{C'}})$ to Φ provided that $u \in E$ and for a box b' it holds that there are tuples $(b'_x, E_x, \{\mu_b^x\}_{b \in B_C}) \in \Phi$, one for each $x \in E$, such that for all $x, y \in E$ and $b \in B_C$, (1) $\mu_{b'}^x = \bot$, (2) $E_x = E_y$, and (3) μ_b^x and μ_b^y are compatible (the discovery of the exits follows the discovery of the corresponding returns under compatible assumptions and the same local target).

UPDATE 4. For a call u of a box b', we add a tuple $(u, E_u, \{\mu_b^u\}_{b \in B_C})$ to Φ provided that (i) there is an entry e s.t. $(e, E_e, \{\mu_b^e\}_{b \in B_{C'}}) \in \Phi$, (ii) for each return b'_x, $x \in E_e$, there is a tuple $(b_x, E, \{\mu_b^x\}_{b \in B_C}) \in \Phi$ s.t. all these tuples satisfy (1), (2) and (3) of UPDATE 3, and moreover, (iii) $E_u = E$, $\mu_b^u = \bigcup_{x \in E_e} \mu_b^x$ for $b \neq b'$, and $\mu_{b'}^u = E_e$ (the discovery of a call u of box b' follows the discovery of an entry e from exits E_e that in turn have been discovered by matching returns b'_x, $x \in E$; thus on u we propagate the local target and the assumptions to the boxes $b \neq b'$ of the returns b'_x and make an assumption E_e on box b'), (iii) if $Y_{\mathcal{L}ib}(b') = C''$, then $e = e_{C''}$.

We compute Φ as the fixed-point of the recursive definition given by the above rules and outputs "YES" iff $(e, \mathcal{T}, \{\mu_b\}_{b \in B_{C_0}}) \in \Phi$ for the entry e of C_0.

Observe that, the total number of tuples of the form $(u, E, \{\mu_b\}_{b \in B_C})$ is bounded by $|\mathcal{L}ib|\, 2^{O(k\beta)}$ where k is the number of exits of each component in $\mathcal{L}ib$ and β is the maximum over the number of boxes of each component. Therefore, the algorithm always terminates and takes at most time exponential in k and β, and linear in the size of $\mathcal{L}ib$.

Theorem 34. *The reachability* LMS *problem is* EXPTIME*-complete.*

Reachability component-based LMS problem: For an instance of LMS problem, if there is a set of winning local strategies for a reachability condition, then a modular memoryless strategy exists such that it is winning according to the same reachability condition.

Proof. Consider a winning modular strategy f, a winning local strategy f_m of f and two plays $\pi_1 = \pi'.u$ and $\pi_2 = \pi''.u$ such that $\pi', \pi'' \in V^*$ and $u \in P_m^0$. Let $f(\lambda(\pi_1)) = v'$ and $f(\lambda(\pi_2)) = v''$ with $v' \neq v''$ be two different moves defined by the winning local strategy f_m. Intuitively, if the two different moves v' and v'' of f_m are executed in a same vertex u, this means that such vertex is reachable with two different local histories. We have two cases:

- *O*ne local history is a prefix of the other, i.e. $\lambda(\pi_1) = \lambda(\pi_2).\lambda(\pi).u$ or $\lambda(\pi_2) = \lambda(\pi_1).\lambda(\pi).u$ with $\pi \in V^*$.
- *T*he local histories as no prefix in common and $\lambda(\pi_1) \neq \lambda(\pi_2)$.

In the first case, suppose that $\lambda(\pi_2) = \lambda(\pi_1).\lambda(\pi).u$. We know that $\pi_1 = \pi'.u$ and we replace it in the expression, obtaining $\lambda(\pi_2) = \lambda(\pi'.u).\lambda(\pi).u = \lambda(\pi').u.\lambda(\pi).u$ (because u is a vertex of the current module m). This means that, after visiting the vertex u the first time, the run does a local loop $\lambda(\pi)$ and reaches again the vertex u. The modular memoryless strategy in u chooses always the move on v'', avoiding to execute the cycle $\lambda(\pi)$, and all the resulting runs are still winning (acceptance does not depend on the specific sequence of vertices to the target). In the second case we have that the different moves are done according to two different and incomparable local histories. In such case, the modular memoryless strategy in the vertex u chooses or always v' or always v''. Such memoryless strategy is still winning, because, in both cases, all the resulting runs will reach the target (acceptance does not depend on the context where the instance was invoked). □

Due to the fact that all the instances obtained by a same component must share the same local strategy, we pair each component with a specific modular memoryless strategy. We guess a modular memoryless strategy $\bar{f}_C$ for each component. We must force the algorithm to select always the same move defined by the strategy and we change the Update 1 rule to ensure this feature in the following way:

UPDATE 1: For a pl_0-vertex $v \in V_C$, we add $(v, E, \{\mu_b\}_{b \in B})$ provided that $\bar{f}_C(v) = u$ and $(u, E, \{\mu_b\}_{b \in B}) \in \Phi$.

This new rule guarantees that, even if we have instances with a different mapping on boxes, they are still controlled in the same way.

The following holds:

Theorem 35. *The reachability component-based* LMS *problem is* EXPTIME*-complete.*

9

Modular synthesis with other winning conditions

In this chapter, we solve the *modular synthesis from a library of components* (LMS) according to more complex winning conditions. We consider both regular and non-regular specifications.

The first proposed problem assumes that the winning conditions are given as safety automata. The proposed solution is an automata-theoretic construction, that is based on the notions of *library tree*, *box summary* and *pre-post conditions*. Due to the complexity of the complexity of the construction, we split it in several pieces that guarantees the fulfilment of specific subtasks.

We consider the LMS problem also with VPA specification and we prove its decidability proposing a reduction from VPA LMS problem to safety LMS problem. In this chapter we will also solve the corresponding component-based LMS problems.

9.1 Contribution

The main contributions present in this chapter are:

- We give a solution to the LMS problem with winning conditions given as external deterministic finite automata (FA) and deterministic visibly pushdown automata (VPA) (6). We show that the LMS problem is EXPTIME-complete for any of the considered specifications.

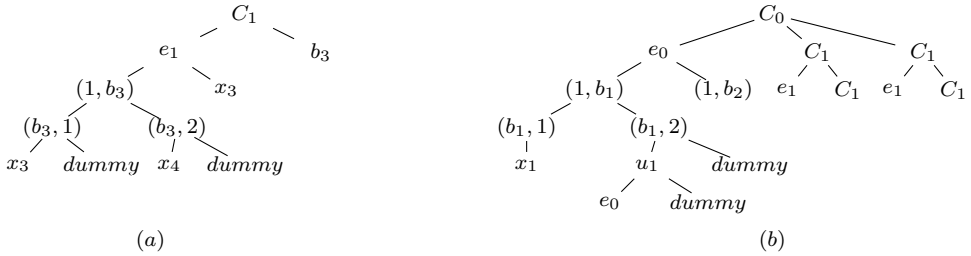

Figure 9.1: Top fragments of (a) the component tree of C_1 and (b) the library tree from our running example.

- We again consider the restrictions to the general LMS problem with winning condition expressed as FA and VPA and we the prove that this problems are all EXPTIME-complete.

9.2 Safety LMS

In the *safety* LMS problem the winning set is given by the set of words accepted by a safety automaton (see Section 2.2.1). In this section we show that deciding this problem is EXPTIME-complete. Our decision procedure consists of reducing the problem to checking the emptiness of tree automata. We assume familiarity with tree automata and refer the reader to 2.2.4 for the main definitions and to (40) for further details.

9.2.1 Overview of the construction.

Fix a safety LMS query $(\mathcal{L}ib, W_A)$ where $\mathcal{L}ib = \langle \{C_i\}_{i\in[0,n]}, Y_{\mathcal{L}ib}\rangle$ is a library and $A = (\Sigma, Q, q_o, \delta_A)$ is a safety automaton. We aim to construct an automaton $\mathcal{A}$ that accepts the trees that *encode* an RSM S synthesized from $\mathcal{L}ib$ iff S satisfies W_A.

For the RSM encoding we introduce the notions of component tree and library tree. Intuitively, a component tree corresponds to the unrolling of a library component, and a library tree is a concatenation of component trees that encodes a choice of the box-to-instance map and of the components for the synthesis of the instances.

For a library tree t, denote with $Roots(t)$ the set of all nodes of t that correspond to a root of a component tree. A set $\mathcal{I} = \{I_x\}_{x\in Roots(t)}$ is *compatible* with t if I_x is an instance of the component corresponding to the component tree rooted at x. Such a set $\mathcal{I}$ and the total box-to-instance map defined by the concatenation of component

trees in t define a possibly infinite RSM (it is infinite iff $Roots(t)$ is infinite). Denote $S_{\mathcal{I},t}$ such RSM.

Intuitively, the automaton $\mathcal{A}$ checks that the input tree t is a library tree of $\mathcal{L}ib$ and that there is a set of instances $\mathcal{I}$ that is compatible with t s.t. $S_{\mathcal{I},t}$ satisfies W_A. For this, $\mathcal{A}$ simulates the safety automaton A on the unrolling of each component and on pl_0 nodes also guesses a move of the local strategy (in this way we also guess an instance of the component). To move across the boxes, $\mathcal{A}$ uses a *box summary* that is guessed at the root of each component tree. For $x \in Roots(t)$, denoting with C_x the corresponding component and with x_b the child of x corresponding to a box b of C_x, the box summary guessed at x essentially tells for each such b (recall that Q is the set of states of A):

1. the associated component C_{x_b} in t, and
2. a non empty set $Q' \subseteq Q$, and for $i \in [k]$ and $q \in Q'$, sets $Q^b_{q,i} \subseteq Q$ s.t. for any run π of $S_{\mathcal{I},t}$ that starts at the entry of the instance I_{x_b} and ends at its i^{th} exit, if the safety automaton A starts from q and reads the sequence of input symbols along π then it must reach a state of $Q^b_{q,i}$.

The above assumption 2 is called a *pre-post condition* for C_{x_b}. The correctness of the pre-post condition for each such C_{x_b} is checked in the simulation of $\mathcal{A}$ on the unrolling of C_{x_b}.

We give $\mathcal{A}$ as the composition of several tree automata: $\mathcal{A}_{\mathcal{L}ib}$ checks that the input tree is a library tree, and each $\mathcal{A}^C_{\mathcal{P},\mathcal{B}}$ checks on the unrolling of C that the pre-post condition $\mathcal{P}$ holds provided that the box-summary $\mathcal{B}$ holds.

9.2.2 Component and library trees.

For a component C of $\mathcal{L}ib$, the *component tree* of C is a tree where the subtree rooted at the first child of the root is essentially the unrolling of C from its entry node and the other children of the root are leaves s.t. each box of C is mapped to exactly one of them.

Consider a library $\mathcal{L}ib = \langle \{C_i\}_{i\in[0,n]}, Y_{\mathcal{L}ib} \rangle$. Let $B_{\mathcal{L}ib} = \bigcup_{i\in[0,n]} B_{C_i}$ be the set of all boxes and $V_{\mathcal{L}ib} = \bigcup_{i\in[0,n]} V_{C_i}$ be the set of all vertices (i.e. nodes, calls and returns) of the library components.

Let d be the maximum over the number of exits, the number of boxes in each component and the out-degree of the vertices of the library components.

Denote with $\widehat{\Omega}$ the set $\{dummy, C_0, ..., C_n\} \cup B_{\mathcal{L}ib} \cup V_{\mathcal{L}ib}$. A *component tree* of some component C_i in $\mathcal{L}ib$ is an $\widehat{\Omega}$-labeled d-tree such that its first subtree encodes the unrolling of C_i and the children of its root, from the second through the $(\ell+1)^{th}$, are leaves corresponding respectively to each of the ℓ boxes of C_i. We make use of *dummy* nodes to complete the d-tree.

Precisely, an $\widehat{\Omega}$-labeled d-tree T_{C_i} is a *component tree* of C_i in $\mathcal{L}ib$, if:

- the root of T_{C_i} is labeled with C_i;
- the subtree $T^1_{C_i}$ that is rooted at the first child of the root corresponds to the unrolling of the component C_i; the nodes of $T^1_{C_i}$ are labeled with the corresponding vertices of the component C_i; thus, in particular, the root of $T^1_{C_i}$ is labeled with e_{C_i} and the calls have as children the matching returns; a tree-node labeled with an exit has no children; in $T^1_{C_i}$ all the nodes that do not correspond to a vertex in the unrolling of C_i are labeled with *dummy*, meaning that they are not meaningful in the encoding;
- for $i \in [2, \ell+1]$, the j^{th} child of the root is labeled with $b \in B_{C_i}$ and for any $j, z \in [2, \ell+1]$ with $j \neq z$ the labels of the j^{th} child and the z^{th} child must be different;
- the tree-nodes labeled with $b \in B_{C_i}$ have no children;
- the remaining tree-nodes are labeled with *dummy*.

As an example, in Fig. 9.1(a) we show a fragment of the component tree of the component C_1 from the library given in Fig. 7.1(a).

A *library tree* is a tree obtained by starting with the component tree of the main component and then iteratively gluing at each leaf corresponding to a box b: any component tree, if $Y_{\mathcal{L}ib}(b)$ is not defined, and the component tree of $Y_{\mathcal{L}ib}(b)$, otherwise.

One can formally define a library tree t as the ω-fold concatenation over languages of component trees. (We refer the reader to (**?**) for the main definitions and (40) for a detailed definition of ω-fold concatenation.) For this, let T_C be the component tree of C for each component C of $\mathcal{L}ib$ and denote $\mathbf{b} = (b_1, \ldots, b_n)$ where $B_{\mathcal{L}ib} = \{b_1, \ldots, b_n\}$

(recall that with $B_{\mathcal{L}ib}$ we denote the union of the set of boxes over all the components of $\mathcal{L}ib$). For each $i \in [n]$, we let $\mathcal{T}_i$ be the language $\{T_C\}$, if $Y_{\mathcal{L}ib}(b_i) = C$, and $\{T_{C'} \mid C' \text{ is a component of } \mathcal{L}ib\}$, otherwise.

A library tree for $\mathcal{L}ib$ is thus any tree $t \in \mathcal{T}_0 \cdot_{\mathbf{b}} (\mathcal{T}_1, \ldots, \mathcal{T}_n)^{\omega \mathbf{b}}$ where $\mathcal{T}_0 = \{T_{C_0}\}$.

In Fig. 9.1(b) we show the initial fragment of the library tree for the library from Fig. 7.1(a). Note that the second and the third child of the root correspond respectively to the boxes b_1 and b_2 of C_0 and thus each of them is replaced by a copy of T_{C_1} in the sample library tree.

The construction of $\mathcal{A}_{\mathcal{L}ib}$ can be obtained from the automata accepting the component trees for $\mathcal{L}ib$ using the standard construction for the ω-fold concatenation (see (40)). Thus, we get:

Proposition 36. *There exists an effectively constructible Büchi tree automaton $\mathcal{A}_{\mathcal{L}ib}$ of size linear in the size of $\mathcal{L}ib$, that accepts a tree if and only if it is a library tree of $\mathcal{L}ib$.*

9.2.3 The construction of $\mathcal{A}^C_{\mathcal{P},\mathcal{B}}$.

We first formalize the notions of pre-post condition and box summary that we have informally introduced earlier in this section. Intuitively, box summaries are composed of pre-post conditions and each postcondition summarizes the states of the safety automaton A that can be reached along a play of a strategy at the exits of a corresponding component instance.

Formally, a *pre-post condition* $\mathcal{P}$ is a set of tuples $(q, [Q_1, \ldots, Q_k])$ where $q \in Q$ and $Q_i \subseteq Q$ for each $i \in [k]$, and s.t. for any pair of tuples $(q, [Q_1, \ldots, Q_k])$, $(q', [Q'_1, \ldots, Q'_k]) \in \mathcal{P}$: (1) $q \neq q'$, and (2) $Q_i = \emptyset$ implies $Q'_i = \emptyset$ for each $i \in [k]$ (i.e., for each q there is at most a tuple with q as first component and each other component is either the empty set for all the tuples or it is non-empty for all of them). For such a pre-post condition $\mathcal{P}$, each q is a *precondition* and each tuple $[Q_1, \ldots, Q_k]$ is a *postcondition*. Note that according to the above intuition, part (2) above captures the fact that all the postconditions of a pre-post condition must agree on the assumption on whether the i^{th} exit is reachable (i.e., $Q_i = \emptyset$) or not (i.e., $Q_i \neq \emptyset$).

A *box summary* of an instance of C is a tuple $\mathcal{B}_C = \langle \hat{Y}_C, \{\mathcal{P}_b\}_{b \in B_C} \rangle$, where $\hat{Y}_C : B_C \to \{C_i\}_{i \in [n]}$ is a total map that is consistent with the library box-to-component map $Y_{\mathcal{L}ib}$ and for each box $b \in B_C$, $\mathcal{P}_b$ is a pre-post condition.

Fix a component C, a pre-post condition $\mathcal{P} = \{(q_i, [Q_{i_1}, ..., Q_{i_k}])\}_{i \in [h]}$ and a box summary $\mathcal{B} = \langle \hat{Y}_C, \{\mathcal{P}_b\}_{b \in B_C} \rangle$.

Denote T_C the component tree of C and T_C^1 the subtree rooted at the first child of T_C. Recall that T_C^1 corresponds to the unrolling of C from the entry node. For a local strategy f for C, a path $x_1 \dots x_j$ of T_C^1 *conforms to* f if the corresponding sequence of C vertices $v_1 \dots v_j$ is s.t. for $i \in [j-1]$ if v_i is a node of pl_0 then $v_{i+1} = f(v_1 \dots v_i)$.

For each path π of T_C^1, a run of the safety automaton A on π according to box summary $\mathcal{B}$ is a run where a state q is updated (1) according to a transition of A, from a tree-node corresponding to a node or a return of C, and (2) by nondeterministically selecting a state from Q_i with $(q, [Q_1, \dots, Q_k]) \in \mathcal{P}_b$ (i.e., a state from the postcondition for box b in $\mathcal{B}$), from a tree-node corresponding to a call $(1, b)$ to one corresponding to a return (b, i). Note that, we do not consider the case of an empty postcondition for a return. This is fine for our purposes since we need to simulate the safety automaton A only on the returns (b, i) that can be effectively reached in a play (according to the guessed box summary).

We construct $\mathcal{A}^C_{\mathcal{P},\mathcal{B}}$ s.t. it rejects any tree other than T_C and accepts T_C iff (recall h is the number of tuples in the pre-post condition $\mathcal{P}$):

(P1) There is a local strategy f for C s.t. for each $i \in [h]$, $j \in [k]$, and path π of T_C^1 from the root to the j^{th} exit that conforms to f, each run of A on π according to $\mathcal{B}$ that starts from q_i ends at a state in Q_{i_j} (i.e., the *pre-post condition* $\mathcal{P}$ *holds*).

For this, we define $\mathcal{A}^C_{\mathcal{P},\mathcal{B}}$ such that it summarizes for each precondition of $\mathcal{P}$ the states of the safety automaton A that can be reached at a given node.

The states of $\mathcal{A}^C_{\mathcal{P},\mathcal{B}}$ are:

- an *initial state* q_s,
- an *accepting sink state* q_a,
- a *rejecting sink state* q_r, a state q_e,
- a state q_b for each box b of C,
- *summary states* of the form $(R_1, \dots, R_h)$ where $R_i \subseteq Q$ for $i \in [h]$.

$\mathcal{A}^C_{\mathcal{P},\mathcal{B}}$ accepts on a finite path if it ends at q_a upon reading its sequence of labels. No condition is required in order to accept on infinite paths (the existence of a run suffices in this case).

At the root of T_C, from q_s the automaton enters q_e on the first child and for each box b of C, q_b on the child corresponding to b. From q_b, it then accepts entering q_a if the node is labeled with b. From q_e, it behaves as from $(\{q_1\}, \ldots, \{q_h\})$ if the current node corresponds to the entry of C (where $q_1, \ldots, q_h$ are the preconditions of $\mathcal{P}$).

In each run of $\mathcal{A}^C_{\mathcal{P},\mathcal{B}}$, for a state of the form $(R_1, \ldots, R_h)$ at a tree-node x, we keep the following invariant: for $i \in [h]$, R_i is the set of all the states that end any run of A starting from q_i on the path from the root of T^1_C up to x (according to the box summary $\mathcal{B}$).

From a tree-node corresponding to a node or a return of C, the transitions of $\mathcal{A}^C_{\mathcal{P},\mathcal{B}}$ update each R_i as in a standard subset construction provided that there is a transition of A from all the states in $\bigcup_{j\in[h]} R_j$ (we recall that a run is unsafe if A halts), thus maintaining the invariant. The updated state is entered on all the children from pl_1 vertices, and on only one nondeterministically selected child from pl_0 vertices (this correspond to guessing a local strategy in C).

The update on tree-nodes corresponding to a call $(1, b)$ of C is done according to the pre-post condition $\mathcal{P}_b$ from the box summary $\mathcal{B}$. In particular, denoting $\mathcal{P}_b = \{(q'_i, [Q'_{i,1}, \ldots, Q'_{i,k}])\}_{i\in[h']}$, from $(R_1, \ldots, R_h)$ we enter q_a on the tree-node corresponding to any return (b, j) that is not reachable according to $\mathcal{P}_b$, i.e., each $Q'_{i,j} = \emptyset$ (we accept since the guessed local strategy excludes such paths and thus the condition $\mathcal{P}$ does not need to be checked). On the reachable returns (b, j), we enter the state $(R'_1, \ldots, R'_h)$ where $R'_i = \bigcup_{q'_d \in R_i} Q'_{d,j}$ for $i \in [h]$, i.e., according to the above invariant, for each position i in the tuple we collect the postconditions of the j^{th} exit for each precondition of $\mathcal{P}_b$ that applies.

At a tree-node corresponding to the i^{th} exit of C, $\mathcal{A}^C_{\mathcal{P},\mathcal{B}}$ accepts by entering q_a iff $\mathcal{P}$ is fulfilled, i.e., $\mathcal{A}^C_{\mathcal{P},\mathcal{B}}$ is in a state $(R_1, \ldots, R_h)$ s.t. $R_i \subseteq Q_i$ for $i \in [h]$.

The state q_r is entered in all the remaining cases.

By a simple counting, we get that the size of $\mathcal{A}^C_{\mathcal{P},\mathcal{B}}$ is linear in the number of boxes and exponential in the number of states of the specification automaton A. Thus, we get:

Lemma 37. *$\mathcal{A}^C_{\mathcal{P},\mathcal{B}}$ accepts T_C iff property P1 holds. Moreover, the size of $\mathcal{A}^C_{\mathcal{P},\mathcal{B}}$ is linear in the number of C boxes and exponential in the number of A states.*

9.2.4 The construction of $\mathcal{A}$.

We first construct an automaton $\mathcal{A}'$. For this, we extend the alphabets such that $\mathcal{A}^C_{\mathcal{P},\mathcal{B}}$ accepts the trees that are obtained from the component tree T_C of C by labeling the leaf corresponding to b, for each box b of C, with any tuple of the form $(\hat{Y}(b), \mathcal{P}_b, \mathcal{B}_b)$ where $\hat{Y}$ is the total map of the box summary $\mathcal{B}$ and $\mathcal{B}_b$ is any box summary for component $\hat{Y}(b)$. Denote $\mathcal{L}^C_{\mathcal{P},\mathcal{B}}$ the set of all trees accepted by any such automaton.

Let $\mathcal{P}_0 = \{(q_0, [\emptyset, \ldots, \emptyset])\}$ where q_0 is the initial state of A and Lab be the set of all labels $(C, \mathcal{P}, \mathcal{B})$ s.t. C is a component, $\mathcal{P}$ is a pre-post condition of C, and $\mathcal{B}$ is a box summary of C. For each box summary $\mathcal{B}_0$ for C_0 denote $\mathcal{T}_{\mathcal{B}_0}$ the language $\mathcal{L}^{C_0}_{\mathcal{P}_0,\mathcal{B}_0} \cdot_{\bar{c}} (\langle \mathcal{L}^C_{\mathcal{P},\mathcal{B}} \rangle_{(C,\mathcal{P},\mathcal{B})\in Lab})^{\omega \bar{c}}$ where $\bar{c} = \langle (C, \mathcal{P}, \mathcal{B}) \rangle_{(C,\mathcal{P},\mathcal{B})\in Lab}$, i.e., the infinite trees obtained starting from a tree in $\mathcal{L}^{C_0}_{\mathcal{P}_0,\mathcal{B}_0}$ and then for all $(C, \mathcal{P}, \mathcal{B}) \in Lab$ iteratively concatenating at each leaf labeled with $(C, \mathcal{P}, \mathcal{B})$ a tree from $\mathcal{L}^C_{\mathcal{P},\mathcal{B}}$ until all such leaves are replaced.

By standard constructions (see (40)), we construct the automaton $\mathcal{A}'$ that accepts the union of the languages $\mathcal{T}_{\mathcal{B}}$ for each box summary $\mathcal{B}$ of the main component.

The automaton $\mathcal{A}$ is then taken as the intersection of $\mathcal{A}_{\mathcal{L}ib}$ and $\mathcal{A}'$. Thus, from Proposition 36, Lemma 37 and known results on tree automata (40), we get that the size of $\mathcal{A}$ is exponential in the sizes of $\mathcal{L}ib$ and A. Recall that the emptiness of (Büchi) nondeterministic tree automata can be checked in linear time and if the language is not empty then it is possible to determine a finite witness of it (*regular tree*) (40). The finiteness of a regular tree ensures both the finiteness of the local strategies and of the number of instances. Moreover, it encodes an RSM and thus starting from the automaton $\mathcal{A}$, we can use standard algorithms for tree automata to synthesize an RSM that fulfills the specification A. Note that from Proposition 36 and Lemma 37 we also get that the encoded strategy for each instance is local and, consequently, the set of synthesized strategies is modular.

Further, we can show an EXPTIME lower bound, proving the EXPTIME-hardness, with a direct reduction. Such reduction is from the membership problem for alternating linear-space Turing machines.

Proof. An alternating Turing machine is $M = (\Sigma, Q, Q_\exists, Q_\forall, \delta, q_0, q_f)$, where Σ is the alphabet, Q is the set of states, $(Q_\exists, Q_\forall)$ is a partition of Q, $\delta : Q \times \Sigma \times \{D_1, D_2\} \longrightarrow Q \times \Sigma \times \{L, R\}$ is the transition function, and q_0 and q_f are respectively the initial and the final states. (We assume that for each pair $(q, \sigma) \in Q \times \Sigma$, there are exactly two transitions that we denote respectively as the D_1-transition and the D_2-transition.) A d-transition of M is $\delta(q, \sigma, d) = (q', \sigma', L/R)$ meaning that if q is the current state and the tape head is reading the symbol σ on cell i, M writes σ' on cell i, enters state q' and moves the read head to the left/right on cell $(i-1)/(i+1)$. Let n be the number of cells used by M on an input word w. A configuration of M is a word $\sigma_1 \dots \sigma_{i-1}(q, \sigma_i) \dots \sigma_n$ where $\sigma_1 \dots \sigma_n$ is the content of the tape cells and q is a state of M. The *initial configuration* contains the word w and the initial state. An *outcome* of M is a sequence of configurations, starting from the initial configuration, constructed as a play in the game where the $\exists$-player picks the next transition when the play is in a state of $Q_\exists$, and the $\forall$-player picks the next transition when the play is in a state of $Q_\forall$. A *computation* of M is a strategy of the $\exists$-player, and an input word w is accepted iff there exists a computation that reaches a configuration with state q_f. A polynomial-space alternating Turing machine M is an alternating Turing machine that on an input word w uses a number of tape cells that is at most polynomial in $|w|$.

Let $M = (\Sigma, Q, Q_\exists, Q_\forall, \delta, q_0, q_f)$ be polynomial-space alternating Turing machine, and n be the number of cells used by M on an input word w. In the following, we define a library of recursive component $\mathcal{L}ib_\mathcal{A}$ and a safety automaton $A_\mathcal{A}$ such that an RSM S from $\mathcal{L}ib_\mathcal{A}$ exists and each its possible run is winning according to $A_\mathcal{A}$ if and only if M accepts w. The library $\mathcal{L}ib_\mathcal{A}$ has two components: C_0 and C_1. Let Σ' be $\Sigma \cup (Q \times \Sigma) \cup \{D_1, D_2\}$, C_0 generates sequences from $(\Sigma^* \cdot (Q \times \Sigma) \cdot \Sigma^* \cdot \{D_1, D_2\})^* \cdot \Sigma'^\omega$ interspersed with a new symbol \$. [1] C_0 has a cyclic structure, that repeatedly executes a call using a box b. The box-to-component map of $\mathcal{L}ib_\mathcal{A}$ relates such box b to C_1. After the execution of each call, the component C_0 enters a node labeled with a symbol of $\Sigma \cup (Q \times \Sigma)$. Also, it ensures that symbols from $\{D_1, D_2\}$ are selected according to a move of pl_0 (resp. pl_1) if the last pair from $Q \times \Sigma$ which is generated on the current play has a state from $Q_\exists$ (resp. $Q_\forall$). In C_1, there is only one exit, each node is controlled by pl_1 and two new labels *ok* and *obj* are used. Intuitively pl_1 simply chooses between "let the play continue" (*ok* is generated) or "raise an objection" (*obj* is generated), then the control is returned to C_0 that generates the next symbol in the sequence. The objection *obj* along with the specification automaton $A_\mathcal{A}$ is used to check that on each

[1] This symbol is only needed for labeling entry and exit nodes, and has no particular meaning in this encoding.

run according to the local strategies for pl_0, if we delete all the occurrences of *ok*, *obj* and \$ we obtain a sequence $w \cdot w'$ such that $w \in (\Sigma^* \cdot (Q \times \Sigma) \cdot \Sigma^* \cdot \{D_1, D_2\})^*$, $w' \in \Sigma'^{\omega}$, and w encodes an outcome of a halting computation of M on w. In particular, $A_{\mathcal{A}}$ is a safety automaton that checks the following:

1. the first n symbols of w encode the starting configuration;
2. each subsequence w'' of w such that $d' \cdot w'' \cdot d''$, for $d', d'' \in \{D_1, D_2\}$, and $w'' \in \Sigma^* \cdot (Q \times \Sigma) \cdot \Sigma^*$, contains exactly n symbols;
3. while generating a configuration c, if the content of cell i is generated right after objection *obj* is raised, and on the current play this is the first time that *obj* is raised, then the $(i+1)^{th}$ cell of the next configuration is consistent with the transition selected after generating c and the contents of cells i, $(i+1)$ and $(i+2)$ of c. [1]

Whenever $A_{\mathcal{A}}$ finishes reading the configuration containing the final state q_f, if the above part 2 holds, it enters a state where it stays on each input (and thus accepts). Also when *obj* is raised, if the above part 3 holds then $A_{\mathcal{A}}$ accepts. If one of the above checks fails $A_{\mathcal{A}}$ halts, thus rejects all the the plays that are obtained as a continuation of the word it has read. Automaton $A_{\mathcal{A}}$ also checks whether the configuration sequence ends at a halting configuration.

Due to the box-to-component map and the lack of boxes in C_1, an RSM S from $\mathcal{L}ib_{\mathcal{A}}$ is forced to have only two instances, I_0 that is an instance of C_0, and I_1 that is an instance of C_1. The choices done by pl_1 can be done only in I_1 and, consequently, they are hidden to the other player. Since strategies of pl_0 are local and the strategy for I_0 is oblivious to the objection being raised in I_1, pl_0 needs to generate an accepting run of $\mathcal{A}$ in order to win. Hence, each run of S is winning according to $A_{\mathcal{A}}$ if and only if M accepts its input, and the theorem holds. □

Therefore, we get:

Theorem 38. *The safety* LMS *problem is* EXPTIME*-complete.*

9.3 LMS with deterministic VPA specification

As we said in Chapter 2, a *visibly pushdown automaton* (VPA) is a pushdown automaton where the stack operations are determined by the input symbols: a call symbol causes

[1] Here, we skip the description of the limit cases concerning the cases when there are only two or one cells left to the end of the current configuration.

a push, a return symbol causes a pop and an internal symbol causes just a change of the finite control (6).

In the *VPA* LMS *problem* the specification is giving as a deterministic VPA. The labeling of the library components is synchronized with the usage of the stack of the VPA: calls are labeled with call symbols, returns with return symbols and nodes with internal symbols.

VPAs are strictly more expressive than finite state automata and they allow to express many additional specifications, as stack inspection properties, partial correctness or local properties (see (3)). For example, with a VPA we could express the requirement that along any run of an RSM M, every γ must be followed by at least an α in the same instance invocation where γ occurs. The RMS in Fig. 7.1(b) does not satisfy this requirement. In fact, though in any run each occurrence of γ is always followed by an occurrence of α, indeed each γ occurs during an invocation of either X_1 or X_2 while α always occurs in the only invocation of X_0 (when the invocations of X_1 and X_2 have already returned).

We give a reduction from the VPA LMS problem to the safety LMS problem. The idea is to achieve the synchronization on the stack symbols between automaton and specification using the mechanics of the game, such that the specification can be considered as a finite state automaton. The top symbol of the stack is embedded in the states of the specification automaton. Before every invocation of an instance, the adversary pl_1 has to declare the top symbol pushed by the specification automaton and the specification automaton has to verify that the adversary is honest (otherwise, pl_1 loses). After such declaration, the instance is invoked and, when its execution terminates, the adversary repeats the declared top-of-the-stack symbol such that the finite state automaton can update the simulated top symbol accordingly.

Consider a VPA LMS query with library $\mathcal{L}ib = \langle \{C_i\}_{i\in[0,n]}, Y_{\mathcal{L}ib}\rangle$ and deterministic VPA $\mathcal{A}_v$. We define a new library game $(\widehat{\mathcal{L}ib}, W_{\mathcal{A}})$, where $\widehat{\mathcal{L}ib} = \langle \{C_i\}_{i\in[0,n]} \cup \{C_{stack_i}\}_{i\in[n]}, Y_{\widehat{\mathcal{L}ib}}\rangle$ and $W_{\mathcal{A}}$ is the language recognized by a finite state automaton $\mathcal{A}$.

The structure of a component C_{stack_i} with $i \in [1, n]$ is given in Fig. 9.2 where with g we denote the number of stack symbols. Recall that k denotes the number of exits of any possible component C. Also, note that all the vertices are controlled by pl_1 and all the boxes are mapped to component C_i.

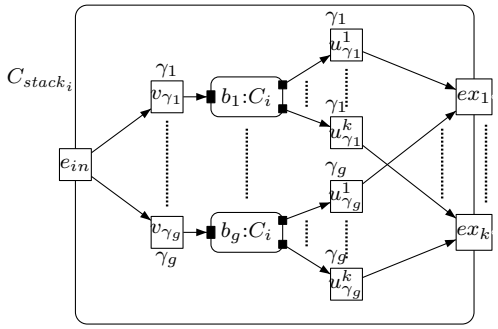

Figure 9.2: The component C_{stack_i} for $i \in [n]$

The main purpose of the new components is to store the symbol that is pushed onto the stack in the $\mathcal{A}_v$ pushes. This is achieved by letting pl_1 to guess a stack symbol γ_j, then call the corresponding $\mathcal{L}ib$ component and on returning from exit x of such component, restore γ_j before exiting from the exit corresponding to x (thus reporting to the caller the exit of the callee).

We encode the stack of the specification in the library by enforcing each call to a component C_i of $\mathcal{L}ib$ to occur through a call to the new component C_{stack_i}, for $i \in [n]$. For this, we define the box-to-component map $Y_{\widehat{\mathcal{L}ib}}$, such that it preserves the original box-to-component of the input library and partially guarantees the interleaving of calls of components and calls of stack components. Namely, for $i \in [n]$, if $Y_{\mathcal{L}ib}$ maps a box b to a component C_i, then $Y_{\widehat{\mathcal{L}ib}}$ maps such box to the new component C_{stack_i}. Then, $Y_{\widehat{\mathcal{L}ib}}$ maps all the boxes of C_{stack_i} to C_i. In all the other case, i.e., if $Y_{\mathcal{L}ib}$ is undefined for a box b, also $Y_{\widehat{\mathcal{L}ib}}$ is undefined for it.

The winning condition is given as a finite state automaton $\mathcal{A}$ given as the intersection of two finite state automata $\mathcal{A}_1$ and $\mathcal{A}_2$. We embed the top stack symbol of the deterministic VPA in the states of $\mathcal{A}_1$. Moreover, the states of $\mathcal{A}_1$ simulate the corresponding states of $\mathcal{A}_v$, and the winning condition is equivalent. On calls, $\mathcal{A}_1$ must mimic a push transition t from the current state, by first storing in the control the pushed symbol γ and the next control state according to t, then, if the next input is γ, it continues, otherwise it enters an accepting sink state. Returns are handled similarly (the popped symbol occurs after the return, and the fact that this corresponds to the symbol actually pushed in the current run on the matching call is ensured by the instance of a component C_{stack_i}).

The alternation of calls to instances of components C_{stack_i} and calls to instances of the components C_i is guaranteed by the box-to-component map except for the case of the calls from unmapped boxes of instances of the input library. In fact, since these are unmapped also in the library $\widehat{\mathcal{L}ib}$, in an RSM from $\widehat{\mathcal{L}ib}$, we could map them to instances of both kinds of components C_{stack_i} and C_i. Thus, in order to enforce the alternation, and thus prevent them to be mapped directly to instances of C_i, we use the second finite state automaton $\mathcal{A}_2$. This automaton cycles on a state q_{in} until it reads a node labeled with the call of an unmapped box. Then, the automaton enters a state q_{wait} and cycles on it, waiting for an entry. If the first encountered entry e is the entry of a component C_{stack_i}, then the automaton enters again q_{in}, otherwise it enters a rejecting sink state.

Note that pl_0 has no moves in the C_{stack_i} instances, and the moves of pl_1 there are not visible to her in the other instances. Thus, the local strategies for pl_0 in the original game are exactly the same in the new game.

Denote with g the number of stack symbols. Each of the new components C_{stack_i} has $O(g\,k)$ size. Since there are only n additional components, the resulting library $\widehat{\mathcal{L}ib}$ has $O(|\mathcal{L}ib| + n\,g\,k)$ size. Also, the constructed automaton $\mathcal{A}$ has $O(g\,|\mathcal{A}_v|)$ size. By Theorem 38, we thus have:

Theorem 39. *The VPA* LMS *problem is* EXPTIME*-complete.*

9.4 On component-based LMS problems

The construction given in Section 9.2 is based on the notion of library tree that essentially encodes the components and the box-to-instance map of an RSM. The local strategies are guessed on-the-fly by the tree automaton. To constrain the RSM to be component-based we should guess a strategy for each component C and then use it while visiting each component tree of C in the input library tree. This requires to prove first boundedness of the local strategies if there is a component-based RSM that satisfies the winning condition.

A simpler solution can be obtained by adapting the solution given in (4) for the synthesis of *modular strategies*. This problem is a particular case of the single-instance LMS problem where the box-to-component map is total, i.e., each box is pre-assigned. The solution given in (4) is based on the notion of *strategy tree* that unrolls each

component as a subtree of the root and encodes in the labels of this encoding a local strategy. To adapt their automaton construction for the component-based LMS we just need to guess the mapping for the boxes that are not mapped by the box-to-component map of the library, every time a component subtree is visited. The following holds:

Theorem 40. *The safety component-based* LMS *problem is* EXPTIME*-complete.*

Cleary the reduction presented in Section 9.3 applies also to component-based and we get the same complexity. Thus, we say:

Theorem 41. *The VPA component-based* LMS *problem is* EXPTIME*-complete.*

10

Discussion and Conclusion

10.1 On visibly modular pushdown games

In Chapter 5 we have considered modular games with winning conditions expressed by pushdown, visibly pushdown and temporal logic specifications. The winning strategies that we compute are *modular* and thus the local strategy of an instance that is called several times is oblivious of previous calls, i.e., it does not keep the memory of previous invocations. It is known that non-oblivious modular games are undecidable (5).

We have proved that the modular game problem with respect to standard pushdown specifications is undecidable. Then we have presented a number of results that give a quite accurate picture of the computational complexity of the MVPG problem with visibly pushdown winning conditions.

With some surprise, we have found that MVPG with temporal logic winning conditions becomes immediately hard. In fact, the complexity for LTL specifications is 2EXPTIME-complete both for MVPG and games on finite graphs. However, for the fragment consisting of all the Boolean combinations of PATH-LTL formulas, solving the corresponding games on finite graphs is PSPACE-complete while the MVPG problem is already 2EXPTIME-complete. As a consequence, the computational complexity of many interesting fragments of LTL, that have a better complexity than full LTL on finite game graphs, collapses at the top of the complexities (see (9, 10)). This also differs with the scenario of the complexities of model-checking RSMs in LTL fragments (see (26)).

The tree automaton construction proposed in Section 5.4 can be easily adapted

to handle visibly pushdown winning conditions to get a direct solution of the MVPG problem. We only need to modify the transition rules to synchronize the calls and returns of the RGG with the pushes and pops of the specification automaton, and this would be possible since they share the same visibly alphabet. The change does not affect the overall complexity, however it will slightly improve on the approach presented in Section 5.3 that causes doubling the number of exits and gives a complexity with an exponential dependency in the number of stack symbols.

We have also shown that the synthesis problem from a library of components with respect to LTL specification (28) can be reduced to an MVPG problem with a universal Büchi automaton winning condition. Since the size of the resulting RGG is linear in the size of the library and the universal Büchi automaton is exponential in the size of the LTL formula, we get an alternative proof of the 2EXPTIME-completeness of this component-based synthesis problem. Such reduction points out the connections between game with modular strategies and the synthesis from library and it inspired us to the following works.

10.2 On modular synthesis

In Chapter 7 we have introduced a new model and related problems, that generalize both the modular synthesis of recursive game graph and the synthesis from component libraries.

In Chapter 8 we have considered the reachability winning condition and we have shown that it is EXPTIME-complete and is fixed-parameter tractable when the number of exits is fixed.

In Chapter 9 we have presented a decidable synthesis problem for an expressive class of systems. Our decision algorithms for reachability specifications are fixed-point labeling computations and can be easily turned into an automatic synthesis of RSMs. All the other decision algorithms that we have presented are based on a reduction to tree automata, and thus also can be turned into automatic synthesis using standard results of this theory (see (40)).

Our model and its related problems are still strictly entangled with many interesting area of research. It can easily be imagined that LMS problem has connections with the synthesis in (29) and in particular it is a generalization of this setting. It is more difficult

to see relations between the LMS problem and the setting of the program repair. The last interesting remark is that even if in our setting we allow to duplicate components change modular strategies and compose the final system with no restriction the final system does not act as it could see the entire history of the system. We can also show that there are LMS queries for which a global strategy exists while a modular one does not. Moreover if we extend the class of solutions in the LMS problems by allowing to resolve the internal nondeterminism of pl_0 by a global strategy, we can show that the resulting problem is still decidable (this does not contradict the previous undecidability result since each pl_1 move is now observable by pl_0 in all the instances).

In the following subsections we discuss about these three relations.

10.2.1 Modular synthesis and synthesis from recursive-component libraries

As we said in Chapter 6, in the synthesis from recursive-component libraries (see Chapter 6 or (29) for more details), the library of reusable components is modeled using a set of transducers with call-return structures. The related synthesis problem asks to find a composition of these elements such that the synthesized system fulfills a given LTL specification. This problem is a specific restriction of our LMS problem. In fact, an instance of the problem proposed in (29) can be considered as an instance of the component-based LMS problem, where the library only has standard (non-game) components, i.e. components without the internal game between two players. In this case, a potential solution must determine only the correct external composition, as in (29). However, note that the problem solved in (29) is a model checking problem, i.e. it asks that all the possible runs of the resulting system must be winning. This means that if we want to have the equivalence, the input library for the component-based LMS library game must be formed by components with all pl_1 vertices.

Rephrasing an LMS problem in terms of synthesis from library of recursive-components determines a significant difference. Consider a node where the local strategy, according to two different local histories, can choose two different moves. If we want to model such behavior with the recursive (non-game) components presented in (29), we must solve the choices of pl_0. In this particular example, we should define two distinct recursive components, one that models the first choice, and, an other that models the second choice, splitting the considered node in two distinct nodes. This means that, if

```
1   main(){
2    const int n=4;
3    bool done=false;
4    int a[n]={7,4,5,1}
5    MergeSort1(a,0,n-1);
6    done=true;
7   }

8   void Merge(int a[],
    int left,int center,
    int right){ ...
9   //code with no errors
10  ...}
```

(a)

```
11  void MergeSort1(int a[],
    int left, int right){
12   int center=
          (left+right)/2;
13   MergeSort1(a,left,
          center);
14   MergeSort1(a,center+1,
          right);
15   Merge(a,left,
          center,right);
16  }
```

```
20  void MergeSort2(int a[],
    int left, int right){
21   if (left<left)
22    {
23   int center=
          (left+right)/2;
24   MergeSort2(a,left,
          center);
25   MergeSort2(a,center+1,
          right);
26   Merge(a,left,
          center,right);
27   }
28  }
```

(b)

Figure 10.1: A faulty program (a) and a pre-existing function (b).

we want to use a library of recursive-components to model a library of components, we could need to generate a library with a potentially infinite number of transducers.

10.2.2 Modular synthesis and program repair

We can extend the results presented in Chapter 8 in a complete different direction. In fact, The LMS problem also gives a general framework for program repair where besides the intra-module repairs considered in the standard approach (see (22, 23)) one can think of repairing a program by replacing a call to a module with a call to another module (*function call repairs*).

Given a misbehaving program according to a correctness specification, the program repair looks for the fault localization and a possible small modification of the program such that the repaired program satisfies its specification. The repair problem is closely related to the synthesis problem. In (23) the fault localization and correction of the problem are achieved using infinite games: the system chooses which component is incorrect, then, if for any input sequence, the system can select a behavior of this component such that the specification is satisfied, the replacement behavior for the component can be used to make the system correct.

Consider the program in Fig. 10.1(a) and the correctness specifications requiring that statement `(done=true)` is reachable (*termination*) and condition `(a[0]<=a[1])`

`&& (a[1]<=a[2]) && (a[2]<=a[3])` holds at the end of the program execution (*correct result*). This program does not fulfill both specifications. In fact, it contains an error that causes an infinite cycle of unreturned function calls: in function `MergeSort1` there is no control over the values of `left` and `right`, and no `return` statement before executing the recursive calls.

Note that this error cannot be repaired, because there is no assignment or condition on which we can set a diagnosis. However, `MergeSort1` is a sorting algorithm. Thus, we could look within an available library for a different function implementing a sorting algorithm, and possibly this function is either correct or could be repaired.

In our example, suppose now that we can use a library that contains the function `MergeSort2` given in Fig. 10.1(b). This function is faulty, but repairable, and the location of the error and its correction can be found using the approach in (23): by assuming that in `main` we call `MergeSort2`, the algorithm suggests to change the left-hand side of the condition in Line 21 from `left<left` to `left<right`. Therefore, by fixing this error and replacing the call in Line 5 with a call to `MergeSort2`, the repaired program will now satisfy the given specification.

We can generalize this approach and apply it directly using the modular synthesis. Given a program P and a correctness specification, we construct a library game. Intuitively, we use the internal game to find and repair fixable faults and the external compositional game to substitute the components that can not be repaired (*function call repair*). As library we consider a given set of standard pre-existing components and the components of the program P, both modeled as game components to find and fix possible bugs as in the standard program repair approach. All the call assignments of the boxes that invoke suspected faulty functions are modeled as unassigned boxes. The correctness specification is unchanged. If there is a solution to such library game, we can obtain a repaired version of the program P that fulfills the given specification.

10.2.3 Modular vs. global synthesis

Fix a library $\mathcal{L}ib = \langle \{C_i\}_{i\in[0,n]}, Y_{\mathcal{L}ib} \rangle$. A *global RSM* $S = \langle \{G_i\}_{i\in[0,m]}, Y_S, S_g \rangle$ where:

- for $i \in [m]$, $G_i \overset{\text{iso}}{\equiv} C_{j_i}$ holds,
- $G_0 \overset{\text{iso}}{\equiv} C_0$ holds,

- the *box-to-instance* map $Y_S : \bigcup_{i\in[0,m]} B_{G_i} \to \{G_i\}_{i\in[m]}$ is a total function that is consistent with $Y_{\mathcal{L}ib}$, i.e. for each $i \in [0, m]$ and $b \in B_{G_i}$, denoting with b' the box of C_{j_i} that is isomorphic to b, it holds that if $Y_{\mathcal{L}ib}(b') = C_{j_h}$ then $Y_S(b) = G_h$

- S_g is a *global strategy*, i.e. $S_g : V_S^*.P_S^0 \to Calls_S \cup N_S$ such that $S_g(w.u) \in \delta_S(u)$ and S_g is computable by a pushdown automaton.

Given a library and an ω-regular specification, the global Lms (Lgs for short) problem asks to construct a global RSM S such that S satisfies the given specification.

We introduce the Lgs problem to show that, even if the Lms problem allows to generate a solution with a potentially unbounded number of instances and, consequently, new local strategies for each of them, the constructed system can not act as it knew the complete history of the plays (as using a global strategy). To support this claim, we present an example (Figure 10.2).

Consider the depicted library and a winning condition that asks to see alternatively often the symbols α and β. It is easy to see that a solution S exists if we admit a global strategy: S is composed of two instances, one from C_0 and one from C_1, and, each time the play reaches e_1, looking at the global memory of the play, the strategy can see if x_1 was the last visited exit, and then x_2 is selected, or it is x_2, and then x_1 is selected. However, if only local strategies can be used, there is no solution for the proposed problem: in fact, the box b can be assigned only one time to a fixed instance of C_1. In this case, each time the play reaches e_1, the strategy is forced to move always on x_1 or always on x_2, because at the entry point, the instance has no memory about the previous moves/calls of the play.

The Lgs problem is decidable and can be solved introducing some changes to the algorithm for the Lms problem presented in Section 9.2. We give only the idea for this solution. The key intuition is that in an RSM synthesized using the Lms restrictions, if in an activation of an instance two different paths reach a same box, they must call the same instance. The behavior of the called instance is summarized using a pre-post condition. Using the global strategy, even if in an activation of an instance two paths

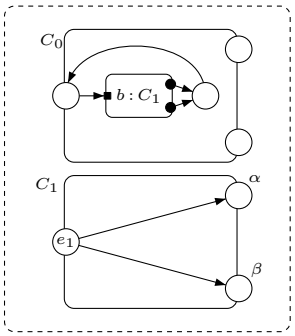

Figure 10.2: A sample library.

reach a same box, the invoked instance can implement different behaviors according to the different global histories. This means that an instance can implement a set of behaviors that can be modeled by a set of single pre-post conditions. Each single pre-post condition is independent and it can be verified using a different strategy.

A *single pre-post condition* $\bar{p}$ is a tuple of the form $(q, [Q_1, \ldots, Q_k])$ where $q \in Q$, for each $i \in [k]$, $Q_i \subseteq Q$.

A *global pre-post condition* is a set $\bar{\mathcal{P}} = \{\bar{p}_i\}_{i \in [z]}$ of single pre-post conditions such that for any $i, j \in [z]$ with $i \neq j$ then $\bar{p}_i \neq \bar{p}_j$. Due to the fact that the set Q is finite and, fixed a precondition q, there are $2^{|Q|k}$ distinct postconditions, this means that $z \leq 2^{|Q|^2 k}$.

A *global box summary* of an instance of C is a tuple $\bar{\mathcal{B}_C} = \langle \hat{Y}_C, \{\bar{\mathcal{P}}_{\hat{Y}_C(b)}\}_{b \in B_C} \rangle$, where $\hat{Y}_C$ is a a box mapping and $\{\bar{\mathcal{P}}_{\hat{Y}_C(b)}\}_{b \in B_C}$ is a set of global pre-post conditions, one for each box.

If we consider safety winning condition, we slightly change the structure of the solution for the Lms problem to solve the Lgs problem, handling these different assumptions. We introduce a set of automata of the form $\mathcal{A}^C_{\bar{\mathcal{P}},\bar{\mathcal{B}}}$ that starts an automaton $\mathcal{A}^C_{\bar{p},\bar{\mathcal{B}}}$ for each $\bar{p} \in \bar{\mathcal{P}}$. An automaton of the form $\mathcal{A}^C_{\bar{p},\bar{\mathcal{B}}}$ is a simplified form of $\mathcal{A}^C_{\mathcal{P},\mathcal{B}}$. In fact, the states are: an initial state q_s, an accepting sink state q_a, a rejecting sink state q_r, a state q_e, a state q_b for each box b of C, and states of the form (R) where $R \subseteq Q$. If $\bar{p} = (q, [Q_1, \ldots, Q_k])$ from q_e, it behaves as from $(\{q\})$ if the current node corresponds to the entry of C (remember that q is the precondition of $\bar{p}$). On the update on tree-nodes corresponding to a call $(1, b)$, $\mathcal{A}^C_{\bar{p},\bar{\mathcal{B}}}$ must nondeterministically choose a particular single pre-post condition $\bar{p}$ in $\bar{\mathcal{P}}_{\hat{Y}_C(b)}$ and, then, use it to execute the update of its state (R). The choice of the single pre-post condition can change each time a tree-nodes corresponding to a call is read. In the remaining cases, the automaton $\mathcal{A}^C_{\bar{p},\bar{\mathcal{B}}}$ acts as $\mathcal{A}^C_{\mathcal{P},\mathcal{B}}$, with the simplification that it works only on the single set R.

In $\mathcal{A}'$ we extend the alphabets such that $\mathcal{A}^C_{\bar{\mathcal{P}},\bar{\mathcal{B}}}$ accepts each tree that differs from the component tree T_C of C only for the labeling of the leaves corresponding to the boxes of C. The labels are of the form $(C, \bar{\mathcal{P}}, \bar{\mathcal{B}})$. The working of $\mathcal{A}'$ is exactly the same, up to a renaming from $\mathcal{P}/\mathcal{B}$ to $\bar{\mathcal{P}}/\bar{\mathcal{B}}$.

Further, the global Lms problem can be reduced to a standard pushdown game (PDG) with an exponential blow-up and vice-versa a PDG can be polynomially trans-

lated to a global LMS with a total box-to-component map (see also (2, 5)). We can also show that there are LMS queries for which a global strategy exists while a modular one does not.

10.3 Future research

The setting that we have introduce still presents several future directions that could be investigated.

Synthesis is an important area of research in formal verification, but in spite of a rich set of results in the theory, there are few practical solutions that were implemented. The reason of this imbalance relies in the high complexity of the algorithms that solve problems related to the synthesis. Hierarchical system are a "special"case of recursive system where the nesting of calls forces the stack-depth to be bounded. This represents a very interesting feature in formal verification because it limits the state-space explosion when we compute solutions to problems that involves hierarchical systems and in many cases the complexities of such solutions are not exponentially higher than the solutions of the same problems for flat systems.

It could be interesting to investigate the effect of a hierarchical labeling such as in (11) and (25) on the complexities of our proposed problem and could be meaningful in the LMS setting. Positive results in this setting could also motivate efforts in the development of a software that automatically and efficiently implements the modular synthesis approaches.

For the same reason, we think that it deserves further investigations the analysis of synthesis with simpler specification that could be meaningful in this setting and could lead to lower complexities.

Program repair is an appealing area of interest and we think that in this setting the notion of modular strategy and modular synthesis could have an interesting impact on the development of new models and techniques. We have given an example on how our formalism can be used to include in program repair a notion of function call repair and we believe that this direction deserves further investigation and leave it for future research.

References

[1] L.de Alfaro and T.A. Henzinger. **Interface-based design**. In *Engineering Theories of Software Intensive Systems*, **195** of *NATO Science series*, pages 83–104. Springer, 2005. 1, 7

[2] R.Alur, M. Benedikt, K.Etessami, P.Godefroid, T.Reps, and M.Yannakakis. **Analysis of Recursive State Machines**. *ACM Trans. Program. Lang. Syst.*, **27**(4):786–818, July 2005. 4, 92, 100, 136

[3] R.Alur, K.Etessami, and P. Madhusudan. **A Temporal Logic of Nested Calls and Returns**. In Kurt Jensen and Andreas Podelski, editors, *TACAS*, **2988** of *Lecture Notes in Computer Science*, pages 467–481. Springer, 2004. 26, 27, 49, 71, 72, 125

[4] R.Alur, S.La Torre, and P. Madhusudan. **Modular Strategies for Infinite Games on Recursive Graphs**. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *CAV*, **2725** of *Lecture Notes in Computer Science*, pages 67–79. Springer, 2003. 5, 6, 37, 41, 42, 44, 49, 50, 54, 57, 96, 127

[5] R.Alur, S.La Torre, and P. Madhusudan. **Modular strategies for recursive game graphs**. In **354** of *Theor. Comput. Sci.*, pages 230–249. 2006. 3, 5, 6, 37, 44, 96, 100, 101, 106, 129, 136

[6] R.Alur and P. Madhusudan. **Adding nesting structure to words**. In **56** of *J.ACM*, pages 230–249. 2006. iv, 4, 17, 54, 115, 125

[7] R.Alur, and P. Madhusudan. **Visibly Pushdown Languages**. In Proceeding of STOC'04, pages 202–211. ACM, 2004. 6, 22, 35, 36

[8] R.Alur, M.Arenas, P.Barceló, K.Etessami, N.Immerman and L.Libkin. **First-Order and Temporal Logics for Nested Words**. *Logical Methods in Computer Science*, **4**(4), 2008. 27, 49, 71, 72

[9] R.Alur and S.La Torre. **Deterministic generators and games for Ltl fragments**. *ACM Trans. Comput. Log.*, **5**(1):1–25, 2004. 50, 72, 129

[10] R.Alur, S.La Torre, and P. Madhusudan. **Playing Games with Boxes and Diamonds**. In Roberto M. Amadio and Denis Lugiez, editors, *CONCUR*, **2761** of *Lecture Notes in Computer Science*, pages 127–141. Springer, 2003. 50, 76, 129

[11] B.Aminof, F.Mogavero and A.Murano. **Synthesis of hierarchical systems**. In **83** of *Sci.Comput.Program.*, pages 56–79. 2014. 6, 136

[12] M.Benedikt, P.Godefroif and T.Reps. **Model checking of unrestricted hierarchical state machines**. In *ICALP*, **2076** of *Lecture Notes in Computer Science*, pages 652–666. Springer, 2001. 4

[13] A.K.Chandra, D.C.Kozen and L.J.Stockmeyer. **Alternation**. In Journal of the ACM 28:1, pages 114–133. 1981. 31

[14] A.Church. **Applications of recursive arithmetic to the problem of circuit synthesis**. In JSummaries of the Summer Institute of Symbolic Logic, pages 3–50. 1957. 1

[15] I. De Crescenzo and S. La Torre. **Winning CaRet games with modular strategies**. In *CILC Workkshop proceeding*, **810** of *CEUR-WS.org*, pages 327–331. 2011. 6

[16] I. De Crescenzo and S. La Torre. **Modular Synthesis with Open Components**. In *RP*, **8169** of *Lecture Notes in Computer Science*, pages 96–108. 2013. 4, 6

[17] I. De Crescenzo, S. La Torre and Y.Velner. **Visibly Pushdown Modular Games**. In *GandALF*, **161** of *EPTCS*, pages 260–274. 2014. 6

[18] I. De Crescenzo and S. La Torre. **A General Modular Synthesis Problem for Pushdown Systems**. In *VMCAI*, **161** of *Lecture Notes in Computer Science*, pages 495–513. 2016. 6

[19] E.A. Emerson, C.S.Jutla and A.Sitsla. **Synthesis of hierarchical systems**. In *CAV* **697** of *Lecture Notes in Computer Science*, pages 385–396. 1993. 21

[20] W.R.Harris, S. Jha and T.W.Reps. **Secure programming via visibly pushdown**. In *CAV*, **7358** of *Lecture Notes in Computer Science*, pages 581–598. Springer, 2012. 6

[21] S.Jha, S.Gulwani, S.A.Seshia, A.Tiwari. **Oracle-guided Compent-based Program Synthesis**. In *ICSE*, **5504** of *ACM/IEEE International Conference on Software Engineering*, pages 215–224. 2010. 7

[22] B.Jobstmann, A. Griesmayer and R. Bloem. **Program repair as a game**. In *CAV*, **3576** of *Lecture Notes in Computer Science*, pages 226–238. Springer, 2005. 6, 132

[23] B.Jobstmann, S.Staber, A. Griesmayer and R. Bloem. **Finding and Fixing Faults**. In *J.Comput.Syst.Sci.*, **78** of *JCSS*, pages 441–460. Springer, 2012. 6, 132, 133

[24] T.A. Johnson and R. Eigenmann. **Context-sensitive domain-independent algorithm composition and selection**. In *PLDI*, , pages 181–192. Springer, 2006. 7

[25] S. La Torre, M.Napoli, M.Parente and G.Parlato. **Verification of scope-dependent hierarchical state machines**. In **206(9-10)** of *Inf.Comput*, pages 1161–1177. 2008. 136

[26] S. La Torre and G.Parlato. **On the complexity of LTL model-checking of recursive state machines**. In *ICALP* **4596** of *Lecture Notes in Computer Science*, pages 937–948. Springer, 2007. 129

[27] C.Löding, P. Madhusudan and O.Serre. **Visibly Pushdown Games**. In *FSTTCS*, **3328** of *Lecture Notes in Computer Science*, pages 408–420. Springer, 2004. 22, 34

REFERENCES

[28] Y.Lustig and M.Y.Vardi. **Synthesis from component libraries**. In de Alfaro L., editor, *FOSSACS*, **5504** of *Lecture Notes in Computer Science*, pages 395–409. Springer, 2009. 2, 4, 6, 7, 79, 80, 81, 82, 85, 130

[29] Y.Lustig and M.Y.Vardi. **Synthesis from recursive-components libraries**. In D'Agostino G., La Torre S., editors, *GandALF*, **54** of *EPTCS*, pages 1–16. 2011. 6, 79, 81, 83, 85, 86, 130, 131

[30] P.Madhusudan. **Synthesizing Reactive Programs**. In *CSL*, **12** of *LIPIcs*, pages 428–442. 2011. 1, 7

[31] D.Mandelin, L.Xu, R.Bodík and D.Kimelman. **Jungloid mining: Helping to navigate the API jungle**. In *PLDI*, pages 48–61. 2005. 7

[32] R.McNaughton. **Infinite games played on finite graphs**. In **65(2)** of *Ann.Pure Appl.Logic*, pages 149–184. 1993. 98

[33] D.E.Muller, P.E.Schupp. **Simulating Alternating tree automata by nondeterministic automata: New results and new proofs of the Theoremd of Rabin, McNaughton and Safra**. In *FOSSACS*, **141(1&2)** of *Theor.Comput.Sci.*, pages 69–107. 1995. 70

[34] A.Pnueli. **The Temporal Logic of Programs**. In *FOCS* pages 46–57 *IEEE Computer Society*. 1977. 71

[35] A.Pnueli and R.Rosner. **On synthesis of a reactive module**. In *POPL* pages 179–190. 1989. 21, 71

[36] G.L.Peterson and J.H.Reif. **Multiple-person alternation**. In *IEEE Foundations of Computer Science* pages 348–363. 1979. 44

[37] S.Salvati and I.Walukiewicz. **Evaluation is MSOL-compatible**. In *FSTTCS*, **24** of *LIPIcs*, pages 103–114. 2013. 1, 7

[38] S.Sardiña, F.Patrizi and G.De Giacomo. **Automatic synthesis of a global behavior from multiple distributed behaviors**. In *AAAI*, , pages 1063–1069. 2007. 7

[39] J.Sifakis. **A framework for Component-based Construction**. In *Software Engineering and Formal Methods, IEEE*, pages 293–300. 2005. 7

[40] W. Thomas. **Automata on Infinite Objects**. In *Formal Models and Semantics*, **B** of *Handbook of Theorical Computer Science*, pages 133–192. 1990. 14, 15, 116, 118, 119, 122, 130

[41] W. Thomas. **A short introduction to infinite automata**. In *DLT*, **2295** of *Lecture Notes of Computer Science*, pages 130–144. 2002. 16

[42] W. Thomas. **Infinite Games and Verification (extended abstract of a tutorial)**. In *CAV*, **2404** of *Lecture Notes of Computer Science*, pages 58–64. Springer, 2002. 20, 92

[43] W. Thomas. **Facets of Synthesis: Revisiting Church's Problem**. In *FOSSACS*, **5504** of *Lecture Notes of Computer Science*, pages 1–14. Springer, 2009. 1

[44] M.Vardi. **Reasoning about the past with two-way automata.**. In *Information and computation*, **115:1**, pages 1–37. 1994. 19, 20

[45] M.Vardi and P.Wolper. **Reasoning about infinite computations**. In *ICALP*, **1443** of *Lecture Notes of Computer Science*, pages 628–641. Springer,1998. 48

[46] I.Walukiewicz. **Pushdown Processes: Games and model-checking**. In **164(2)** of *Inf. Comput.*, pages 234–263. Springer, 2001. 6, 21, 30

[47] I.Walukiewicz. **A landascape with games in background**. In *Logic in Computer Sciencs*, of *IEEE Symposium*, pages 356–366. 2004.

[48] J.Zappe. **Modal μ-calculus and alternating tree automata**. In *Automata, logics and infinite games*, pages 171–184. Springer, 2002. 30