國立臺灣大學電機資訊學院電子工程學研究所

博士論文

Graduate Institute of Electronics Engineering

College of Electronical Engineering and Computer Science

National Taiwan University

Doctor Thesis

賽局上的合作時序邏輯的模型驗證以及密集錯誤回復力

Model Checking for Temporal Cooperation Logic and Resilience against Dense Errors on Game Graph

黃重豪

Chung-Hao Huang

指導教授：王凡博士

Advisor: Farn Wang, Ph.D.

中華民國 105 年 1 月

January, 2016

國立臺灣大學電子工程學研究所

博士論文

賽局上的合作時序邏輯的模型驗證以及密集錯誤回復力

黃重豪 撰

105
1

Abstract

This thesis is composed of 3 parts. In the first part, I introduce an algorithm to calculate the highest degree of fault tolerance a system can achieve with the control of a safety critical systems. Which can be reduced to solving a game between a malicious environment and a controller. During the game play, the environment tries to break the system through injecting failures while the controller tries to keep the system safe by making correct decisions. I found a new control objective which offers a better balance between complexity and precision for such systems: we seek systems that are k-resilient. A system is k-resilient means it is able to rapidly recover from a sequence of small number, up to k, of local faults infinitely many times if the blocks of up to k faults are separated by short recovery periods in which no fault occurs. k-resilience is a simple abstraction from the precise distribution of local faults, but I believe it is much more refined than the traditional objective to maximize the number of local faults. I will provide detail argument of why this is the right level of abstraction for safety critical systems when local faults are few and far between. I have proved, with respect to resilience, the computational complexity of constructing optimal control is low. And a demonstration of the feasibility through an implementation and experimental results will be in following chapters. The second part is to create an logic which can describe the different purposes of each player such as environment, controller, user, and etc in a system. I propose an extension to ATL (alternating-time logic), called BSIL(basic strategy-interaction logic), for the specification of strategies interaction of players in a system. BSIL is able to describe one system strategy that can cooperate with several strategies of the environment for different requirements. Such properties are important in practice and I show that such properties are not expressible in ATL*, GL (game logic), and AMC (alternating μ-calculus). Specifically, BSIL is more expressive than ATL but incomparable with ATL*, GL, and AMC in expressiveness. I show that, for fulfilling a specification in BSIL, a memoryful strategy is necessary. I also show that the model-checking complexity of BSIL is PSPACE-complete and is of lower complexity than those of ATL*, GL, AMC, and the general strategy logics. Which may imply that BSIL can be useful in closing the gap between large scale real-world projects and the time consuming game-theoretical results. I then show the feasibility of our techniques by implementation and experiment with our PSPACE model-checking algorithm for BSIL. The final part is an extension to BSIL called temporal cooperation logic(TCL). TCL allows successive definition of strategies for agents and agencies. Like BSIL the expressiveness of TCL is still incomparable with ATL*, GL and AMC. However, it can describe deterministic Nash equilibria while BSIL cannot. I prove that the model checking complexity of TCL is EXPTIME-complete. TCL enjoys this relatively cheap complexity by disallowing a too close entanglement between cooperation and competition while allowing such entanglement leads to a non-elementary complexity. I have implemented a model-checker for TCL and shown the feasibility of model checking in the experiment on some benchmarks.

Key words:

Contents

Abstract i

Contents iii

1 Introduction 1

2 Software Resilience against Dense Errors 11

2.1 Two-player concurrent game structures 11

2.2 Motivation 12

2.2.1 Background 12

2.2.2 Resilience in a Nutshell 15

2.3 Safety resilience games 19

2.4 Alternating-time μ-calculus with events 23

2.4.1 Syntax 25

2.4.2 semantics 26

2.5 Resilience level checking algorithm 28

2.5.1 High-level description of the algorithm 28

2.5.2 Realization with AMCE model-checking 29

2.5.3 Complexity 34

2.6 Tool implementation and experimental results 37

2.6.1 Benchmarks 38

2.6.2 Modeling of the fault-tolerant systems 39

2.6.3 Performance data 44

2.7 Related works 45
2.8 Conclusion 47

3 Basic Strategy-interaction Logic **48**

3.1 Existed Logics about Game and Strategy 48
3.1.1 Prior to Strategy Logics 48
3.1.2 With Strategy Logics 49
3.2 Running Example 49
3.2.1 Trying to Write Down a Correct Formal Specification 49
3.2.2 Resorting to General Strategy Logics for a Correct Specification . 51
3.2.3 BSIL: New Strategy Modalities Expressive Enough for the Specification 52
3.2.4 Symbolic Strategy Names and Path Obligations 52
3.2.5 Passing Down the Path Obligations While Observing the Restrictions Among S-Profiles 53
3.2.6 Finding Finite Satisfying Evidence for a Formula in a Computation Tree 55
3.3 Game Graphs 56
3.3.1 Concurrent Games 56
3.3.2 Turn-Based Games 58
3.4 BSIL 58
3.4.1 Syntax 58
3.4.2 semantics 59
3.4.3 ATL+ 60
3.4.4 Memory 61
3.5 Expressive Power of BSIL 61
3.5.1 Comparison with GL 62
3.5.2 Comparison with AMC 64
3.5.3 Comparison with ATL* 68
3.6 Algorithm and Complexity 68

3.6.1 Computing path obligations and passing them down the computation tree . . . 69
3.6.2 Procedures for checking BSIL properties . . . 75
3.6.3 Correctness proof of the algorithm . . . 77
3.6.4 Complexities of the algorithm . . . 79
3.6.5 Lower bound and completeness . . . 80
3.7 Automata for BSIL model-checking . . . 83
3.7.1 Alternating Automata (AA) . . . 84
3.7.2 Model Checking with AA's . . . 85
3.8 BSIL Satisfiability . . . 91
3.9 Experiment . . . 94
3.9.1 Implementation . . . 94
3.9.2 Benchmarks and their experiment report . . . 94
3.9.3 Discussion of the experiments . . . 96

4 Temporal Cooperation Logic **97**
4.1 TCL . . . 97
4.1.1 Syntax . . . 97
4.1.2 semantics . . . 99
4.2 Expressive Power of TCL . . . 102
4.3 Complexity of TCL . . . 104
4.4 Implementation and Experiment . . . 107

5 Conclusions **111**

Chapter 1

Introduction

There can be million lines of code in today's software system. On such a scale of complexity, defects in the source codes are unavoidable. Various empirical studies show that the defect density of commercial software system is around 1 to 20 defects in every 1000 lines of source code [**?**]. Therefore, developers of systems created many engineering techniques to contain the damage that could be caused by such defects. For example, a software system may have several measures to its disposal to avoid system failure, including resending the request, resetting the server, clearing the communication buffers, and etc when observing that a critical service request is not acknowledged. However, in general, since all the recovery cost time and money it is important to estimate how to organize the measures for the maximal resilience of the system against realistic errors. At the moment, an automated support which can suggest defence techniques to development teams is missing. I created a game theoretic approach to study this problem and carried out experiments to show how this approach can be helpful in synthesizing the most resilient defence of software systems against multiple errors.

The naive way to measure the safety level of a system is to find the number of errors that it can endure before running into failure state. But in second thought, no non-trivial system can handle unlimited errors without degrading to inevitable system failure. Thus, it would be meaningless to analyse the resilience level of the systems to software errors can proceed without creating a realistic error model in which practical control mechanism can be devised to defend the systems against errors. In this work, I am interested in defending the system against a more restricted error model, but still let the error model has a quantifiable level of power in order to simulate different error scenarios. Further more, I think a reasonable foundation need to take into consider that the life-time of a software system is much longer than the duration needed for a reasonably designed software system to recover from an error. Therefore, I propose to evaluate control mechanism of software systems on how many errors the control can endure before recovery to safe states. I then present an algorithm to find a control strategy that can handle the maximum number of such errors.

Let us standardize the basic terms before proceeding further. A design defect in software or hardware is called a *fault* in embedded systems. An *error* (sometimes called component failure in the literature) is the effect of a fault that causes a difference between the expected and the actual behavior of a software system, e.g., measurement errors, read/write errors, etc. An *error* does not always lead to a system failure, but may instead be repaired by, e.g., a defence mechanism in the software. That is, an *error* may be detected and fixed/neutralized before it creates any harm to the system or its users. A *failure* is the fact that users can observe the faulty behavior created by *errors*.

My specific goal is to develop a technique for finding a control mechanism of a software system which can against the maximal number of dense errors without degrading to failure. My inspiration is from methods for resilient avionic systems [**?**], where fault tolerance is designed to recover from a bounded number of errors. The number of errors a system needs to tolerate is calculated from the mean time between errors of individual components and the maximal duration of the system. I use the quality guarantees one obtains for an airplain(the system) as an example to demonstrate the difference between the objective to tolerate up to *k errors* and sequences of separated blocks of up to *k dense errors* in a short period. Assuming the operating time of the system is 20 hours, the mean time between exponentially distributed errors is 10 hours and the repair time is 3.6 seconds. The mean time between dense errors (consecutive errors before system recovery) is calculated in Table 1.1. The figures for k errors (component failures) are simply the values for the Poisson distribution with coefficient 2. To explain the figures for k dense errors,

k	0	1	2	3	4	5	6	...
k errors	0.865	0.594	0.333	0.143	0.053	0.017	0.005	...
k dense errors	0.865	$2 \cdot 10^{-4}$	$2 \cdot 10^{-9}$	$2 \cdot 10^{-14}$	$2 \cdot 10^{-19}$	$2 \cdot 10^{-24}$	$2 \cdot 10^{-29}$	...

Table 1.1: Probabilities of k dense errors

consider the density of 2 dense errors occurring in close succession. If an error occurs, the chance that the next error occurs within the repair time (3.6 seconds) is approximately $\frac{1}{10000}$. The goal to tolerate an arbitrary number of up to k-dense errors is, of course, much harder than the goal of tolerating up to k errors, but, as the example shows, the number k can be much smaller. Tolerating an arbitrary number of errors (with a distance of at least 3.6 seconds between them) creates the same likelihood to result in a system failure as tolerating up to 9 errors overall, and tolerating up to 15 errors still results in a 70% higher likelihood of a system failure than tolerating blocks of up to 2 errors in this example. Only errors for which this is the case could cause a system failure. The mean time between blocks of two dense errors is therefore not ten hours, but 100,000 hours. Likewise, it increases to 1,000,000,000 (one billion) hours for blocks of three dense errors, and so forth.

Maximizing the number of dense errors that are permitted before full recovery is therefore a natural design goal. After full recovery, the system is allowed again the same number of errors. Now, if the *mean time between errors* (*MTBE*) is huge compared to the time the system needs to fully recover, then the mean time between system failures (MTBF) grows immensely.

We view the problem of designing a resilient control mechanism towards dense errors as a two-player game, called *safety resilience game*, between the system (protagonist[1], 'he' for convenience) and a hostile agent (antagonist[2], 'she' for convenience) that injects errors into the system under execution. The protagonist wants to keep the system from failure in the presence of errors, while the antagonist wants to derail the system to failure. Specifically, system designers may model their system, defense mechanism, and error model as a finite game graph. The nodes in the graph represent system states. These system states

[1]In game theory, a protagonist sometimes is also called *player 1*.

[2]In game theory, an antagonist sometimes is also called *player 2*.

are partitioned into three classes: the safe states, the failure states, and the recovery states. Some transitions are labeled with errors while others are considered normal transitions. The game is played with respect to a resilience level k. If a play ever enters a failure state, then the antagonist wins in the play. Otherwise, the protagonist wins.

The protagonists plays by selecting a move, intuitively the ‘normal’ event that should happen next (unless an error is injected). The antagonist can then decide to trigger an error transition (injecting an error) with the intention to eventually deflect the system into a failure state. Our error model, however, restricts the antagonist to inject at most k errors before she allows for a long period of time that the system may use to recover to the safe states. (If the antagonist decides to use less than k errors, the protagonist does not know about this. It proves that this information is not required, as we will show that the protagonist can play memoryless.) After full recovery by the protagonist to the safe states, the antagonist is allowed again to inject the same number of errors, and so forth.

If the system can win this game, then the system is called *k-resilient*. For k-resilient systems, there exists a control strategy—even one that does not use memory—to make the system resilient in the presence of blocks of up to k dense errors. We argue that, if the component MTBF is huge compared to the time the system needs to fully recover, then the expected time for system breakdown grows immensely.

Besides formally defining safety resilience games, we also present algorithms for answering the following questions.

- Given an integer k, a set F of failure states, and a set S of safe states (disjoint from F), is there a recovery mechanism that can endure up to k dense errors, effectively avoid entering F, and quickly direct the system back to S. Sometimes, the system designers may have designated parts of the state space for the recovery mechanism. The answer to this question thus also implicitly tells whether the recovery mechanism is fully functional in the recovery process.

- Given an integer k and the set of failure states, what is the maximal set of safe states, for which the system has a strategy to maintain k-resilience? In game theory, this means that safety resilience games can be used for synthesizing safety regions for a given bound on consecutive errors before the system is fully recovered.

  The question can be extended to not only partition the states into safety, recovery, and failure states, but also for providing memoryless control on the safety and recovery states.

- Given a set of failure states, what is the maximal resilience level of the system that can be achieved with proper control? We argue that this maximal resilience level is a well-defined and plausible indicator of the defense strength of a control mechanism against a realistic error model.

With our technique, software engineers and system designers can focus on maximizing the number of dense errors that the system can tolerate infinitely often, providing that they are grouped into blocks that are separated by a short period of time, which is sufficient for recovery.

We investigate how to analyze the game with existing techniques. We present an extension to alternating-time μ-calculus (AMC) and propose to use the AMC model-checking algorithm on concurrent games to check resilience levels of embedded systems. We present reduction from safety resilience games to AMC formulas and concurrent game structures. Then we present a PTIME algorithm for answering whether the system can be controlled to tolerate up to a given number of dense errors. The algorithm can then be used to find the maximal resilience level that can be achieved of the system. The evaluation is constructive: it provides a control strategy for the protagonist, which can be used to control a system to meet this predefined resilience level.

In the second part of this thesis, I try to push the game strategy concept further. The idea is to create a temporal logic which include strategy quantifier in the syntax. At the moment, there are various logic that express such properties of strategic power of agents, including *ATL* (*alternating-time logic*), *ATL**, *AMC* (*alternating μ-calculus*), *GL* (*game logic*) [**?**], and *SL* (*strategy logics*) [**?**, **?**, **?**], for the specification of open systems. Each language also comes with a verification algorithm that helps to decide whether or not a winning strategy for the system exists. There is, however, a gap between the industrial need for efficient algorithms (and solvers) and the available technology offered from previous research. Frankly speaking, none of those languages represents a proper combination of expressiveness for close interaction among agent strategies and efficiency for specification verification. ATL, ATL*, AMC, and GL [**?**] allow us to specify that some players together have a strategy to satisfy some fully temporalised objective: strategy quantifiers mark the start of a state formula. As exemplified below, this is far from what the industry needs in specification.

Consider the example of a bank that need specify their information system embodied as a system security strategy and allowing a client to use a strategy to withdraw money, to use a strategy to deposit money, and to use a strategy to query for balance. Moreover, the same system strategy should forbid any illegal operation on the banking system. Specifically, the same system strategy must accommodate all strategies of the client for 'good behavior' (i.e, behavior in line with the specification), while blocking all strategies of the client that refer to undesired behavior, and thus preventing the client from damaging the system. We will show that *ATL ATL**, and *GL* [**?**] do not support such specifications. For example, it is not possible to specify in those languages that the system strategies used both in a withdrawal transaction and in a deposit transaction must be the same. Consequently, verification techniques for specifications in those languages cannot capture such real-world objectives for open systems.

To solve the expressiveness problem in the above example, strategy logics were proposed in [**?**, **?**, **?**] that allow for the flexible quantification of high-order strategy variables in logic formulas. However, their verification complexities are prohibitively high and hinder them from practical application. In retrospect, the strategy profiles in such strategy logics can be combined in unrestricted ways. For example, in [**?**], we can write down the following artificial and hypothetical property.

$$\langle\langle X\rangle\rangle[\![Y]\!]\langle\langle Z\rangle\rangle\Box((((1,X)\diamond p)\wedge\bigcirc(2,Y)\diamond q)\rightarrow(((2,Z)\diamond q)\wedge\neg(3,Z)\Box q))$$

Here $\langle\langle X\rangle\rangle$ and $\langle\langle Z\rangle\rangle$ declare the existence of strategies named X and Y respectively. $[\![Y]\!]$ is a universal quantification on a strategy named Y. Then operator $(1,X)$, $(2,Y)$, $(2,Z)$, and $(3,Z)$ respectively bind strategy X, Y, Z, and Z to agents $1,2,2$, and 3. As can be seen, the language is very free in style. According to the experiences in temporal logic development, usually proper restrictions in the modal operations can lead to a spectrum of sub-logics with different expressiveness and model-checking efficiency. The following are two examples:

- The validity problem of $\mathcal{L}$ (the first-order langauge of unary predicate symbols and the binary predicate symbol $\leq$) is non-elementary [**?**] while PTL, with the same expressiveness as $\mathcal{L}$, is only PSPACE-complete [**?**].
- Between CTL [**?**] and CTL* [**?**, **?**], there are many subclasses of CTL* with various balances between expressiveness and verification efficiency. Fair CTL [**?**], as a natural class between CTL and CTL*, is expressive enough for many practical specifications and still enjoys a polynomial time model-checking complexity. There are also other subclasses of CTL* with various balance considerations [**?**, **?**, **?**, **?**].

As can be seen, subclasses of temporal logics with proper balance between expressiveness and verification efficiency are not only theoretically interesting and can also be practically useful. Indeed, most specifications in real-world projects come in simple structures, for example, safety, liveness, etc. Thus it would be interesting to see what "natural" subclasses of strategy logics can be identified with a proper balance between expressiveness and model-checking efficiency. Moreover, it would be practical and appealing if the subclass can be characterized with elegant syntax. Indeed it is the purpose of this manuscript to propose new natural modal operators for strategy collaborations and extend ATL for a subclass of strategy logic.

In the following, we use the classical prisoner's dilemma to explain how we can design new modal operators for structured strategy collaboration to achieve a balance between expressiveness and model-checking efficiency.

Example 1. 1 **Prisoner's dilemma** Suppose the police is interrogating three suspects (prisoners). The police has very little evidence. A prisoner may cooperate (with his/her peers) and deny all charges made by the police. If all deny, they are all acquitted of all charges. However, each prisoner may choose to betray his/her peers and provide the police with evidence. If more than one prisoner choose to betray their peers, all will be sentenced and stay in jail. If only one chooses to betray, then the other prisoners will stay in jail, while (s)he will be a 'dirty witness' and all charges against him/her will be dropped.

We may want to specify that the three prisoners can cooperate with each other (by denying all charges), and will not be in jail. Let j_a be the proposition for prisoner a in jail. This can be expressed in Alur, Henzinger, and Kupferman's ATL*, GL, or AMC [**?**], respectively, as follows.[3]

$$\begin{array}{ll} \text{ATL}^*: & \langle 1,2,3\rangle \bigwedge_{a\in[1,3]} \Diamond\neg j_a \\ \text{GL:} & \exists\{1,2,3\}.\bigwedge_{a\in[1,3]} \forall\Diamond\neg j_a \\ \text{AMC:} & \mathbf{lfp} x.\langle 1,2,3\rangle \bigcirc \bigwedge_{a\in[1,3]}(x\vee\neg j_a) \end{array}$$

Here "$\langle 1,2,3\rangle$" and "$\exists\{1,2,3\}$" are both existential quantifiers on the collaborative strategy among prisoners $1,2$, and 3. Such a quantifier is called a *strategy quantifier* (*SQ*) for convenience. Operator '**lfp**' is the least fixpoint operator. Even though we can specify strategies employed by sets of prisoners and there is a natural relationship (containment) between sets with such logics, there is no way to relate strategies to each other. For example, if prisoners 1 and 2 are really loyal to prisoner 3, they can both deny the charges, make sure that prisoner 3 will not be in jail, and let prisoner 3 to decide whether they will be in jail. □

The research of strategies for related properties has a long tradition in game theory. It is easy to see the similarity and link between the specification problems for the prisoner's dilemma and the banking system. This observation suggests that finding a language with an appropriate and natural balance between the expressive power and the verification complexity of a specification language is a central challenge.

To meet this challenge, we propose an extension of ATL, called *BSIL* (*basic strategy-interaction logic*). In a first step, we extend ATL to ATL$^+$, where ATL$^+$ is the natural extension obtained by allowing for Boolean connectives of path quantifiers. (Cf. [**?**, **?**] for the similar extension of CTL to CTL$^+$.) We then introduce a new modal operator called *strategy interaction quantifier* (*SIQ*). In the following, we use several examples in the prisoner's dilemma to explain BSIL, starting with the following specification for the property discussed at the end of Example 1.

[3]Note that the three example formulas are not equivalent.

$$\langle 1,2\rangle((\langle +\rangle\Diamond\neg j_3)\wedge(\langle +3\rangle\Diamond\neg(j_1\vee j_2))\wedge\langle +3\rangle\Box(j_1\wedge j_2)) \qquad \text{(A)}$$

Here "$\langle +3\rangle$" is an existential SIQ that reasons over strategies of Prisoner 3 for collaborating with the strategies of Prisoners 1 and 2 introduced by the parent SQ "$\langle 1,2\rangle$". Similarly, "$\langle +\rangle$" means that no collaboration of any prisoner is needed. (For conciseness, we omit "$\langle +\rangle$" in the following.) We also call an SIQ an SQ. In BSIL formulas, we specifically require that no SIQ can appear as a topmost SQ in a path subformula.

As can be seen, SIQ imposes a hierarchical style of strategy collaboration which seems natural for practical specification. Consider another example. If Prisoner 1 really hates the others, (s)he can always betray other prisoners, making sure that Prisoners 2 and 3 will be in jail, and let them decide whether (s)he will be in jail, too. This property can be expressed in BSIL as follows.

$$\langle 1\rangle((\Box(j_2\wedge j_3))\wedge(\langle +2,3\rangle\Diamond\neg j_1)\wedge\langle +2,3\rangle\Box j_1) \qquad \text{(B)}$$

One restriction of BSIL is that no negation between an SIQ and its parent SIQ or SQ is allowed. This restriction then also forbids universal SIQ. While at first glance, it seems less than elegant in mathematics, it is both necessary for verification complexity and compatible with practical specification styles. When we take a closer look, in fact, there is an implicit universal SIQ at the end of every maximal syntax path from an SQ to its descendant SIQ. Thus, BSIL allows for the specification of the ways in combining system strategy profiles that can be computed statically to enforce the system policy against any hostile strategy profile of those agents not participating in the the system strategy profiles. If we explicitly allow for universal SIQs in BSIL, then the strategy profiles associated with existential SIQs in the scope of a universal SIQs may have to be computed dynamically. Thus the explicit universal SIQs will not only necessarily blow up the verification complexity, but also contradict the main-stream of game theory for statically calculable strategies. In contrast, some strategy logics [**?**] allow for the specification of strategy profiles that are not statically calculable.

In this work, we establish that BSIL is incomparable with ATL*, GL, and AMC in expressiveness. Although the strategy logics [**?**, **?**, **?**] are superclasses to BSIL with their flexible quantification of strategies and binding to strategy variables, their model-checking[4] complexity are all doubly exponential time hard. In contrast, BSIL enjoys a PSPACE-complete model-checking complexity for turn-based and concurrent game graphs. This may imply that BSIL could be a better balance between expressiveness and verification efficiency than ATL*, GL, AMC [**?**], and SL [**?**, **?**]. Further related work is the stochastic game logic (SGL) by Baier, Brázdil, Größer, and Kucera [**?**], which allows for expressing strategy interaction. However, for memoryful strategies, the model-checking problem of SGL is undecidable.

We also establish some other properties of BSIL. We show that the strategies for BSIL properties against turn-based games need to be memoryful. We prove that the BSIL model-checking problem is PSPACE-complete. However, the PSPACE model-checking algorithm need enumerate the labelings on computation trees and may suffer from high time complexity. We thus also present an alternative model-checking algorithm with time complexity quadratic in the size of a game graph and exponential only in the size of a BSIL specification. We also establish that the BSIL realisability problem is complete for doubly exponential time.

[4]A model-checking problem is to check whether a given model (game graphs in this work) satisfies a logic formulas (in ATL and its extensions in this work).

Chapter 2

Software Resilience against Dense Errors

2.1 Two-player concurrent game structures

To facilitate our explanation of resilience analysis in a game's perspective, we start by reviewing the game concepts related to our work. A concurrent game may involve several players, who make concurrent move decisions at the same time during transitions. The destination of a transition is jointly determined by the moves chosen by all players. Such a game model is very expressive and handy in describing interactions in a complex system. In this work, we adapt the finite concurrent games from [**?**] with event concepts on transitions. For the analysis of system resilience, we only have to consider two players in the game, the first is the system, and the second is the error model.

Definition 1. (2-player concurrent game structure): A *concurrent* game structure is a tuple $\mathcal{K} = \langle Q, r, P, \lambda, E_1, E_2, \delta \rangle$, where

- Q is a finite set of states.
- r is the initial state in Q.
- P is a finite set of atomic propositions.
- $\lambda : Q \mapsto 2^P$ is a proposition-labeling function of the states.
- E_1 and E_2 are finite sets of move symbols that the protagonist and the antagonist

can respectively choose in transitions. A pair in $E_1 \times E_2$ is called a *move vector*.

- δ is a function that maps from $Q \times E_1 \times E_2$ to Q. δ is called the transition function and conceptually specifies a successor state that results from a state and moves of the players.

Given a state $q \in Q$ and a vector $[e_1, e_2] \in E_1 \times E_2$, $\delta(q, e_1, e_2)$ is the successor state from q when each player $a \in \{1, 2\}$ chooses her respective move e_a. $\Box$

We prefer to represent the moves available to the players by symbols (rather than integers as in [**?**]), as move (or event) symbols can be used to reflect some physical meaning. For example, a move can correspond to the turning-off of a switch, the detection of an airplane, or the execution of an error handling routine. (Technically, representing moves as either integers or symbols does, of course, make no difference.)

2.2 Motivation

2.2.1 Background

Resilience to errors in computer systems is usually achieved through error recovery design as illustrated in Figure **??**. The system states can be partitioned into three regions: safe, recovery, and failure. The left part of the figure represents the safety region. The states in this safe region can be viewed as those for 'normal' operation. When an error occurs, the system goes through a recovery stage, where it follows some recovery mechanism. This is shown as the "recovering" area in Figure **??**. In this region, the system intuitively tries to repair the effects of an error and thus to recover to the safety region.

During the recovery (or: in the recovery region), however, errors may still happen. In general, fault-tolerant systems are built under the assumption that error detection and recovery is speedy and that there can only be a few errors during the process of recovery. If the recovery mechanism is not resilient enough, a few errors may drive the system into failure.

We illustrate this on the following examples.

Example 2. (Fault-tolerant computer architectures): In computer architectures, fault-tolerance is usually achieved via hardware duplication. Consider an example of a multi-processor system that includes n processor copies and m memory copies. The n processors each can follow the instructions of the original system, or be engaged in memory recovery. When a copy of the memory fails, a processor can be assigned to recover it. Majority check can be used to detect that a processor is faulty or that memory copy is faulty (often, both would happen at the same time). For recovery, we can set a free processor to recover some memory copy, or make a processor follow the code of the majority of processors.

The key to error resilience is to decide whether to make a processor follow the execution of the majority, or to assign it to recover faulty memory. If too many errors occur in a short while before the errors can be recovered from, then there may be no more processors left to carry out any more recovery. When such a critical situation arises, the system enters failure state when another error is induced.

The recovery mechanism described above is typical in the design of fault-tolerant systems [?]. As explained, a practical recovery mechanism usually does not rely on the detailed structure of the system. Instead, error-detection techniques such as parity checks, voting (for majority checks), etc., are usually employed. In fact, the number of duplicates is usually critical to the resilience of the system to errors. As long as the majority of the duplicate modules can be recovered in time (i.e., before the next wave of errors), resilience of the system can be achieved. □

Example 3. (Exception handling): At the operating system level, errors are usually signaled via interrupt lines and handled with routines called handlers. The first thing that needs to be done by a handler is to save the CPU state of the interrupted process. In some operating systems, a static memory space is used for this purpose for each handler. In such a scheme, if the same error happens again while executing the error handler, then the system can run into the risk that the CPU states of the interrupted handler can be overwritten and destroyed.

Another scheme is to use a stack to save the CPU states of the interrupted processes. Such a scheme seems resilient to errors that happen during the execution of error handlers.

Still, too many errors that happen during the execution of error handlers can deny critical functions of the system and incur failures, including missed timer updates and priority inversions. Thus, a proper assumption on the timely error recovery by the error handling routines is critical to the design of error resilience in such cases. □

Example 4. (Security attacks): Security in the Internet also relies on resilience to attacks of hackers, viruses, malware, etc. For example, one common technique of attacks to communication modules is to overflow the communication buffers. In such attacks, the sizes of the buffers and the ability of the security procedures to detect and recover from such overflowing attacks is crucial to the resilience design. □

These examples show that recovery is a crucial concept for designing systems that are resilient to errors. When system errors are detected in such a system, the system activates a recovery mechanism so as to remove the effect of the errors. When designing such systems, the system designers usually have in mind what errors and failures the systems can expect, according to the specification. To avoid failures in the occurrence of dense errors, the system designers usually incorporate many error recovery mechanism in the system, e.g., exception handlers and hardware/software redundancy. But, in general, it would be difficult for the designers to evaluate how effective their recovery mechanism is to dense errors. To overcome this difficulty, we believe that it is important to support them with automated analytical tools with a solid foundation.

Resilience has also been used in [**?**, **?**] with a similar goal. When synthesising code, one relies on assumptions of the behavior of the environment, and the formal specification would only ask for the provision of guarantees under the condition that the assumptions are satisfied. When assessing the quality of an implementation, the behavior in cases where the environment does not comply with the assumption matters. In [**?**, **?**], the resilience model we have introduced in the conference version [**?**] of this paper has been followed up upon, and proven to be well suited for reactive synthesis.

In this work, we use these observations to design a theoretical framework for synthesizing a control mechanism that provides the maximal resilience against software errors in a realistic error model.

2.2.2 Resilience in a Nutshell

From Example 2 to 4 in Subsection **??**, it is easy to see the common paradigm of error recovery in software systems.

> When errors are detected, a recovery mechanism will be activated to avoid failures and try to get back to normal execution.

Moreover, such a recovery mechanism usually needs to operate under the assumption that more errors may also happen during the recovery process. In practice, system designers have already implemented many defensive modules, e.g., exception handlers, which are certainly good candidates for the recovery segments. Thus, the recovery scheme we discuss is likely to have arisen in an ad-hoc fashion as a natural concept when software architects and programmers designed recovery mechanisms for critical software.

The vast state spaces of critical systems make an automated support for and a solid foundation of evaluating design alternatives particularly valuable.

In the following, we will use the examples from the previous subsection as a motivation for defining a new game, called *safety resilience game*, between the recovery mechanism (the protagonist) and the error-injecting agent (the antagonist). The game is specified with a set F of failure states, a set S of safe states (the safety region), the moves by the antagonist to inject errors, and the resilience level k that the designers want to achieve. The objective of the protagonist is to identify a control strategy so that the whole system can achieve the prescribed level (or the highest level) k of resilience for safety region S (a set of states) and failure state set F.

The game is played round by round. When the antagonist issues an error move, the play may be deflected into a recovery segment. If there are no more than $k-1$ errors in the recovery segment, then a k-resilient control mechanism must direct the recovery segment to end at a safe state. The above observation suggests that a safety region can be abstracted as a fixed point to the recovery procedure that transforms a safe state to another safe state via the recovery segment with at most $k-1$ errors. Conceptually, a fixed point to a procedure $f(x)$ is a set S of elements in the domain of x such that $S = \{f(x) \mid x \in S\}$.

To calculate the fixed point of the recovery procedure, we can use the greatest fixed point algorithm. The idea is to start from a superset of the recovery procedure fixed point. For convenience, we call a superset of the fixed point a *pseudo fixed point* (*PFP*). Then we iteratively check every state q in the PFP and eliminate q from the PFP if, after at most k errors from q, the recovery mechanism either cannot avoid failure or cannot direct the system back to the PFP. As the iterative checking and elimination goes on, the PFP will shrink and eventually stabilize. Note that its size is always finite, since the initial PFP must be no bigger than Q. The final PFP is then a greatest fixed point to the recovery mechansim for k-resilience and is the legitimate safety region.

This recovery procedure can be illustrated as in Figure **??** for resilience to 2 errors. In this figure, the states in set S' are computed as the precondition of states in S through those transitions in the figure. Each path from S' to S is a recovery segment. S and S' may overlap. The blue circles represent states in the recovery segments. If we calculate S' out of S, then, for each state $q' \in S'$, we can find a path from $q' \in S'$ via a path in the recovery segment to another state $q \in S$. The maximal number of errors in a recovery segment is 2. Thus the protagonist has a strategy to recover from errors in S' to S even when 2 errors happen in the corresponding recovery segment. When $S' = S$, then S is a fixed point to the precondition operator through the recovery segments in the figure.

Now we formally define the concept that we explained with Figure **??**.

Definition 2. (k-safety): Given a $k \in \mathbb{N}$, a state q is called k-safe with respect to a safety region $S \subseteq Q \smallsetminus F$ of non-failure states, denoted $q \in \mathsf{sfrch}_k(S)$, if there is a strategy for the protagonist to guarantee that we can reach back to S from q, provided that the overall count of errors is at most k. □

However, the definition can be subtle in its interpretation. Specifically, the ability to stand against one wave of k errors is not the same as that against repeated recovery from waves of k errors. If the recovery mechanism is not designed properly, the system may gradually lose a bit of control after each wave of k errors and eventually degrade to system-level failure.

Example 5. (Fault-tolerant computer architectures): Consider Example 2 with $2k+1$

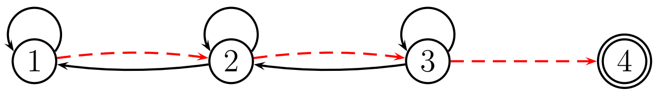


Figure 2.1: An example for calculating sfrch_k

processor copies, with the objective to maintain majority checks and to identify the bad processors. Indeed, according to the first, naïve solution, any safe state with a recovery strategy to $Q \smallsetminus F$ is good. After k processor copies fail, the majority checks are still capable to maintain the correctness of the combined behavior to follow the design of the original system. There seems to be nothing to do after k errors. Thus, naïvely, we can choose those states as the safety region if, at those states, majority checks still work.

However, there is no expectation that the system will be able to recover at any point in the future into a situation where it can bear another wave of k errors. It will fail and lose the function of majority checks just after one more error. In contrast, in this work, we aim to propose a dense error resilience criterion that given no more errors for enough time to allow recovery, the system will eventually recover to resilience to k dense errors again. $\square$

To look at this issue in more detail, please consider the transition system with four states, including a single failure state (state 4, marked by a double line) shown in Figure 2.1. The controlled transitions are depicted as black solid arrows, the error transitions are depicted as red dashed arrows. For $S = Q \smallsetminus F = \{1, 2, 3\}$, all states in S are in $\mathsf{sfrch}_0(S)$. For all $k \geq 1$, we have $\mathsf{sfrch}_k(S) = \{1, 2\}$: the protagonist can simply stay in $\{1, 2\}$ during the safety phase of the game, and once the antagonist plays an error transition, the game progresses into the recovery segment, where the protagonist's objective is satisfied immediately. This outlines the difference between k-sfrch-ty and the linear time property of being able to repeatedly tolerate waves of up to k errors, which would only be satisfied by states 1 and 2 for $k = 1$, and only for state 1 for $k = 2$.

This difference raises the question if the rules of our game are depriving the antagonist of some of the k errors that she should intuitively be allowed to insert in a wave. The answer is that this is not the case if we use any fixed point of sfrch_k as S. In this case, the protagonist would regain the capability to endure a wave of k errors when reaching a safe state after recovery. Instead of depriving the antagonist, one could say that we reset the number of errors in any recovery segment that the antagonist can inject to k. Thus such a fixed point of sfrch_k should consist of states, from which we can use a control mechanism to fend off repetitive waves of k dense errors in the recovery segments. For convenience, we call states in such a fixed point of sfrch_k the k-resilient states.

For a state to be in $\mathsf{sfrch}_k(S)$, the system (protagonist) has a strategy to recover to S, given that a long enough execution commenced without another round of k errors happening. We say that two successive errors are in the same *group of dense errors* if the sequence of states separating them was not long enough for recovery to the safety region. Vice versa, if two successive errors are far enough apart such that the protagonist can guarantee recovery in this separation, then they do not belong to the same group.

To check whether recovering to S by the protagonist (the fault-tolerance mechanism) is always possible, provided that at most k errors occurred during a recovery segment, observe that nesting sfrch_k once, i.e., $\mathsf{sfrch}_k(\mathsf{sfrch}_k(\cdot))$, corresponds to tolerating up to two rounds of up to k dense errors, and so forth. Thus, for S to be a target of recovery for k-resilience, S must be a fixed point of the operator sfrch_k from Definition 2, or, equivalently, $S = \mathsf{sfrch}_k(S)$ must hold. Moreover, if S is the greatest fixed point to k-resilience, then we we can apply $\mathsf{sfrch}_k()$ any number of times to S and still obtain S. Computationally, the greatest fixed point of sfrch_k can be constructed as by executing

$$\mathsf{sfrch}_k(\mathsf{sfrch}_k(\mathsf{sfrch}_k(\ldots\ \mathsf{sfrch}_k(S\Big)\ldots))),$$

using a sufficiently deep nesting that a fixed point is reached.

Note that this fixed point x to $x = \mathsf{sfrch}_k(x)$ is what we are really interested in, while $\mathsf{sfrch}_k(S)$ for a given S is an intermediate result that does not guarantee survival of the systems after waves of dense errors. If this greatest fixed point

$$R = \bigcup\{X \subseteq S \mid X = \mathsf{sfrch}_k(X)\}$$

is non-empty, the protagonist's strategy for the fixed point (guaranteeing eventual recovery to a state in the fixed point within no more than k errors, i.e., k-resilience) can be used to control the recovery mechanism, constraining its transitions to follow its winning strategy.

As explained in the introduction, there can be several natural control problems in our safety resilience game. First, the system designers may want to know whether the chosen safety region S can be supported by the recovery mechanism for resilience level k. Second, they may want to get design support for choosing the safety region for achieving resilience level k. Finally, they may want to know the maximal resilience level that they can achieve.

With the explanation in the above, in the rest of the manuscript, we will focus on the algorithm for constructing $\mathsf{sfrch}_k(\cdot)$ and evaluating k-resilient states.

2.3 Safety resilience games

A system is k-resilient if it can be controlled to tolerate infinitely many groups of up to k dense errors, provided that the system is given enough time to recover between these groups. As we have explained, in systems developed with defensive mechanism against errors, when errors are detected, recovery procedures should be activated. The major challenge is to decide given a set of failure states and a safety region, whether the recovery mechanism can support a resilience level required by the users. Our goal is to develop techniques with a solid foundation to assist the system designers in evaluating the resilience of their systems, to synthesize the controller strategy for the required resilience level, and to achieve the maximal resilience level.

We now formally define the safety resilience game played between a system (the protagonist) and an error-injector (the antagonist). Initially, the two players are given a 2-player concurrent game structure $\mathcal{K}$, a pebble in r, a set $F \subseteq Q$ of failure states, and a safety region $S \subseteq Q \setminus F$. Then the recovery region consists of states in $Q \setminus (F \cup S)$. The two players together make decisions and move the pebble from state to state. The antagonist tries to deflect a play into F by injecting sufficiently many errors, while the protagonist tries to avoid that the pebble reaches F. To achieve this, the protagonist can use the recovery region as the safety buffer and try to get back to S as soon as the play is deflected from S to the recovery region. If a system is resilient to k errors, then it means that the protagonist can handle up to $k-1$ errors while in the recovery region. Thus when checking whether a system is resilient to k errors, we only need to check those recovery segments with no more than $k - 1$ errors.

In the following, we formalize the concept.

Definition 3. (Safety resilience game structure): Such a structure is a pair $\langle \mathcal{K}, F \rangle$ with the following restrictions.

- $\mathcal{K}$ is a 2-player concurrent game structure $\langle Q, r, P, \lambda, E_1, E_2, \delta \rangle$. Conceptually, the first player represents the system / the protagonist, while the second player represents the error model / the antagonist.
- E_2 is partitioned into error and and non-error moves E_{error} and E_{noerr}, respectively. We require that only the 2nd player can issue *error* moves. Moreover, E_{noerr} must be non-empty.
- F is the set of failure states in Q with $r \notin F$.

The antagonist can choose if she wants to respond on a move of the protagonist with an error move. We allow for different non-error moves to reflect 'normal' nondeterministic behavior, e.g., caused by abstraction. We allow for different error moves to reflect different errors that can occur in the same step.

We sometimes refer to transitions with *error* moves by the antagonist as *error transitions* and to transitions with *noerr* moves by the antagonist as *controlled transitions*.

For a party $A \subseteq \{1, 2\}$, we refer with $\overline{A} = \{1, 2\} \setminus A$ to the players not in the party, and by E_A to the moves made by the players in A, that is, $E_{\{1,2\}} = E_1 \times E_2$, $E_{\{1\}} = E_1$, etc.

The antagonist can use both error and non-error moves to influence the game. In a simple setting, the antagonist may only have the choice to insert error-moves, while there is only a single controlled transition. In this simple case, the protagonist can choose the successor state alone unless the antagonist plays an error transition. Specifically, a safety resilience game structure is *simple* if E_2 contains only one error move. Considering simple safety resilience game structures leads to lower complexities, as it changes reductions from reachability in games (PTIME-complete [**?**]) to reachability in graphs (NL-complete [**?**]).
$\square$

Note that, in the game structure, only one system player and one error model player are allowed. This is purely for the simplicity of algorithm presentation. With proper reduction techniques, we can easily convert a game structure with more than one system player and more than one error model player to the structure in Definition 3. The standard technique would be using the transition rules of the product automata of the system players for the protagonist while using the transition rules of the product automata of the error model players for the antagonist. In fact, we indeed use this reduction technique in our experiment for analyzing the resilience levels of multi-agent systems.

From now on, we assume that we are in the context of a given safety resilience game structure $\mathcal{G} = \langle \mathcal{K}, F \rangle$.

Definition 4. (Recovery segements): We need to rigorously define *recovery segments*. A play prefix ρ is a *recovery segment* to safety region $S \subseteq Q \smallsetminus F$ if it satisfies the following constraints.

- $\rho(0) \in S$.
- If $|\rho| = \infty$, then all states in $\rho[1, \infty)$ are in $Q \smallsetminus (S \cup F)$. In this case, ρ is called a failed recovery segment.
- If $|\rho| \neq \infty$, then all states in $\rho[1, |\rho|-2]$ are in $Q \smallsetminus (S \cup F)$ and $\mathit{last}(\rho) = \rho(|\rho|-1)$ is either in F or S. If $\mathit{last}(\rho) \in F$, ρ is also a failed recovery segment; otherwise, it is a successful one.

We use $\mathit{level}(\rho, S)$ to denote the number of error moves between states in ρ with respect to the safety region S: $\mathit{level}(\rho, S) \overset{\text{def}}{=} \left|\{i \in [0, |\rho| - 1) \mid \rho_e(i) \models E_{\mathit{error}}\}\right|$. □

As stated in the introduction, we propose a game-theoretic foundation for resilience analysis of software systems. With this perspective, the protagonist acts as a maximizer, who wants to maximize the resilience levels along all plays. For this, the protagonist fixes a strategy that describe what he is going to do on each play prefix. The antagonist acts as a minimizer, who wants to minimize the resilience level. She can resolve nondeterminism and inject errors in order to achieve this, and (although this plays no major role in this setting) she knows the strategy the protagonist has fixed and can use this knowledge in principle.

The goal of the protagonist is therefore the same as the goal of the system designer: to obtain a strategy that offers a maximal level of resilience in a safety game. However, in order to avoid degenerate behavior where the protagonist benefits from being in the recovery phase and from the antagonist therefore being allowed less errors in the current wave of errors she may inject, we have to strengthen his obligation to eventually recover to the safe states when the environment chooses not to inject further errors. This way, the protagonist has no incentive to cycle in the recovery region. Consequently, he can recover to the safe region within $|Q|$ moves after the antagonist has inserted the last error of the current wave, irrespective of whether the antagonist would be allowed to insert further errors in this wave. This is the key reason why memoryless optimal control exists for this error model, why it is reasonable to assume swift recovery, and, consequently, why it is a posteriori justified to leave the separation time between two waves implicit: the time to traverse $|Q|$ states suffices.

Besides obtaining this from intuition, we can also consider the tree of successful recoveries for any protagonist strategy that can endure k error moves by the antagonist. The tree of recoveries from up to k errors is finite according to the definition of successful recovery segments. Then for any subtree t in this tree of recoveries with a node v in t such that v is labeled with the same state as the root of t with no error on the path, we can always replace t with the subtree rooted at v. After the replacement, we have a tree of recoveries with no greater depth than the original one. After repeating such replacements, this immediately provides a translation from such a strategy with unrestricted memory to one with memory of size k (the resilience level). The restriction to memoryless strategies follows from the construction we give in Section **??**, which does not depend on the memory and still yields a strategy, which is memoryless. Thus, in this work, we should define the resilience level of software systems based on *memoryless* protagonist strategies.

Based on the argument above, the gain of the protagonist in a play can be defined as follows.

Definition 5. (Gain): Given safety region $S \subseteq Q \setminus F$, the gain of a play ρ to S, in symbols *gain*(ρ, S), denotes the maximal integer $k \in \mathbb{N}$ such that, for all recovery segments ρ_r to

S in ρ, if *level*$(\rho_r, S) \leq k$, then ρ_r is a successful recovery segment to S. $\square$

The resilience level of a safety resilience game is defined as the maximum gain that the protagonist can guarantee in all plays with a memoryless strategy.

Definition 6. (Safety resilience game): Such a game is zero-sum and defined on a safety resilience game structure $\mathcal{G} = \langle \mathcal{K}, F \rangle$ and a safety region $S \subseteq Q \setminus F$. The gain of $\mathcal{G}$ to S, in symbols *gain*$(\mathcal{G}, S)$, is defined as the maximum gain that the protagonist can manage with memoryless strategies. Rigorously,

$$\textit{gain}(\mathcal{G}, S) \overset{\text{def}}{=} \max_{\sigma \in \Sigma^{(0)}} \min_{\sigma' \in \Sigma} \textit{gain}(\textit{play}(r, \sigma, \sigma'), S)$$

Please be recall that *play*(r, σ, σ') is the play from r according to strategies σ and σ' respectively of the two players. Moreover $\Sigma^{(0)}$ is the set of memoryless strategies.

We say that the resilience level of $\mathcal{G}$ to S is *gain*$(\mathcal{G}, S)$. A strategy ω for the protagonist is optimal to S if $\min_{\sigma' \in \Sigma} \textit{gain}(\textit{play}(r, \omega, \sigma'), S) = \max_{\sigma \in \Sigma^{(0)}} \min_{\sigma' \in \Sigma} \textit{gain}(\textit{play}(r, \sigma, \sigma'), S)$. When S is not given, we say that $\mathcal{G}$ is *k-resilient* if there exists a non-empty $S \subseteq Q \setminus F$ with *gain*$(\mathcal{G}, S) \geq k$. $\square$

Remark. While the option of using memoryless strategies plays a minor role in the technical argument, it plays a paramount role in the usefulness of the resulting control strategy: choosing memoryless strategies implies that all recovery segments are short. In particular, all sub-paths (recovery segments) between two waves of dense errors injected by the antagonist are shorter—and usually significantly shorter—than the size of $\mathcal{G}$. In consequence, any time span long enough for traversing the recovery segment will lead to a full recovery. It is therefore sufficient for a temporal distance we have to assume between two waves of dense errors.

2.4 Alternating-time μ-calculus with events

We propose to solve our resilience game problems with an existing technology, i.e., model-checking of alternating-time μ-calculus (*AMC*) formulas. AMC is a propositional temporal logic with fixed point operators. For example, the following formula

$$\mu X.(\mathit{safe} \vee \langle 1 \rangle \bigcirc X) \tag{A}$$

uses least fixed point operator μ to declare a fixed point variable X for a set of states. Subformula $\langle 1 \rangle \bigcirc \phi$ existentially quantifies over the protagonist strategies that can direct the plays to a successor state satisfying ϕ. Together, the formula specifies a set X of states that can inductively reach a safe state with the control of the protagonist. Specifically, the formula says that a state is in X if either it is *safe* or the protagonist can direct to a successor state known to be in X. For our game structures, we only need strategy quantification of up to two players.

However, we need extend AMC with some simple syntax sugar. There are two extensions. The first is for Boolean combinations of path modalities in the scope of strategy quantification. For example, the following AMCE formula

$$\langle 1 \rangle ((\mathit{smoke} \Rightarrow \bigcirc \mathit{alarmOn}) \vee \bigcirc \mathit{windowClosed}) \tag{B}$$

says that the protagonist can enforce either of the following two path properties with the same strategy.

- If there is smoke, then the alarm will be turned on in the next state.
- The window will always be closed in the next state.

Such a formula is not in ATL and AMC [**?**].

The second extension is for restricting transitions that may participate in the evaluation of path formulas. The restriction is via constraints on moves on transitions and can, in our extension to AMC, be specified with a move symbol set to the next-state modal operators. For example, the following AMCE formula

$$\langle 1 \rangle ((\bigcirc^{2:\mathit{error}} \mathit{alarmOn}) \wedge (\bigcirc^{\neg 2:\mathit{error}} \neg \mathit{alarmOn})) \tag{C}$$

says that the protagonist can

- turn on the alarm when an error occurs; and
- keep the alarm silent when no error occurs.

Before we formally present AMCE, we need define expressions for constraints on moves of players in transitions. We adapt an idea from [**?**]. Specifically, a *move expression* η is of the following syntax.

$$\eta ::= a : e \mid \eta_1 \vee \eta_2 \mid \neg\eta_1$$

Here, a is a player index in $\{1, 2\}$ and e is a move symbol in $E_1 \cup E_2$. $\vee$ and $\neg$ are standard disjunction and negation. Typical shorthands of Boolean operations can also be defined out of $\vee$ and $\neg$. A total move vector can be expressed as $[e_1, e_2]$ where for all $a \in \{1, 2\}$, $e_a \in E_a$ is the move by player a specified in the vector. We say $[e_1, e_2]$ satisfies η, in symbols $[e_1, e_2] \models \eta$, if and only if the following constraints are satisfied.

- $[e_1, e_2] \models a : e$ if, and only if, e_a is e.
- $[e_1, e_2] \models \eta_1 \vee \eta_2$ if, and only if, $[e_1, e_2] \models \eta_1$ or $[e_1, e_2] \models \eta_2$.
- $[e_1, e_2] \models \neg\eta_1$ if, and only if, $[e_1, e_2] \not\models \eta_1$.

2.4.1 Syntax

A formula ϕ in AMCE has the following syntax.

$$\begin{array}{rcl} \phi & ::= & p \mid X \mid \phi_1 \vee \phi_2 \mid \neg\phi_1 \mid \mu X.\phi_1 \mid \langle A \rangle \psi \\ \psi & ::= & \mid \psi_1 \vee \psi_2 \mid \neg\psi_1 \mid \bigcirc^{\eta}\phi_1 \end{array}$$

Here, ϕ is a state formula, ψ is a path formula, p is an atomic proposition symbol in P (atomic proposition set, as in Definition 1), and X is a set variable for subsets of Q. The Boolean connectors are the common ones: $\vee$ for disjunction and $\neg$ for negation. Note that we allow for Boolean combinations of the next operators $\bigcirc$ under strategy quantification $\langle A \rangle$. This is one major difference of AMCE from AMC.

Formula $\mu X.\phi_1$ is the usual least fixed point operation to ϕ_1. According to the tradition in [**?**], we require that all free occurrences of X in ϕ_1 must occur within an even number of scopes of negations. This is because sentences with a negative occurrence, like $\mu X.\neg X$, have no natural semantics. A set variable X is *bound* in a formula ϕ if it is inside a declaration scope of X. If it is not bound, then it is *free*. An AMCE sentence is an AMCE state formula without free set variables. In most cases, we are interested in specifications given as AMCE sentences.

The A in $\langle A \rangle$ is a finite set of player indices in $[1, 2]$. Conceptually, $\langle A \rangle\psi$ means that players in A can collaborate to make ψ true. For example, $\langle \{1, 2\} \rangle \bigcirc p$ means that players 1 and 2 can collaborate to make p true in the next state. We follow the notations in [**?**] and omit the parentheses in formulas like $\langle A \rangle \psi$. For example, $\langle \{2\} \rangle \bigcirc p$ and $\langle \{1, 2\} \rangle \bigcirc p$ will be abbreviated as $\langle 2 \rangle \bigcirc p$ and $\langle 1, 2 \rangle \bigcirc p$ respectively.

We allow event restrictions as superscripts in $\bigcirc^{\eta} \phi_1$ with a move expression η. The operator is important in supporting the evaluation of safety resilience levels with traditional model-checking technology. Note that since AMC [?] only allows for the next-state temporal modality, only the choice of moves to the next states of a strategy matters. Formula $\bigcirc^{\eta} \phi_1$ is thus evaluated at states with respect to move vectors satisfying constraint η. The formula is true of a move vector $[e_1, e_2]$ if and only if $[e_1, e_2] \models \eta$ implies the satisfaction of ϕ at state $\delta(q, e_1, e_2)$. Also $\bigcirc^{1:E_1} \phi_1$ can be written as $\bigcirc \phi_1$ in AMC [?] and the superscript to $\bigcirc$ can be omitted.

We also adopt shorthands in the below. The β refers to state or path formulas.

$$
\begin{aligned}
\textit{true} &\stackrel{\text{def}}{=} p \vee \neg p \\
\textit{false} &\stackrel{\text{def}}{=} \neg p \wedge p \\
\beta_1 \wedge \beta_2 &\stackrel{\text{def}}{=} \neg((\neg \beta_1) \vee (\neg \beta_2)) \\
\beta_1 \Rightarrow \beta_2 &\stackrel{\text{def}}{=} (\neg \beta_1) \vee \beta_2 \\
\nu X.\phi &\stackrel{\text{def}}{=} \neg \mu X.\neg \phi \\
[A]\psi &\stackrel{\text{def}}{=} \neg \langle A \rangle \neg \psi
\end{aligned}
$$

2.4.2 semantics

In the following, we adapt the presentation style of [?] to define the semantics of AMCE inductively over the structure of the subformulas. The value of a state formula at a state is determined by the interpretation of the set variables. Such an interpretation I maps set variables to subsets of Q. In comparison, the value of a path formula at a state is determined by both the interpretation of the set variables and the move vector chosen by the players. For convenience and conciseness of presentation, we extend the definition of interpretation of [?] also to record the chosen move vector by some players. Specifically, we use an auxiliary variable "move" for the present chosen move vector in the evaluation of path formulas. Given an interpretation I, $I(\text{move})$ records the chosen move vector of all players in I. For example, $I(\text{move}) = [\texttt{setAlarm}, \perp]$ means the chosen move vector that player 1 sets on an alarm while player 2 does nothing under interpretation I.

We need the following concept for collaborative choices of moves to the next states by some players. An *enforced move vector set* by $A \subseteq [1, 2]$ is a maximal set of move vectors that agree on the choices of moves by players with indices in A. Specifically, given an enforced move vector set C by A, we require that, for every $[e_1, e_2] \in C$, $[e'_1, e'_2] \in C$, and $a \in A$, $e_a = e'_a$. For convenience, we let Γ^A denote the set of all enforced move sets by A.

Following the semantics style of [**?**], we can extend I to be an interpretation of all state and path formulas. Intuitively, given a state or path formula β, $I(\beta)$ is the set of states that satisfy β according to the assumption on values of set variable values and auxiliary variable "move." More precisely, $I(\beta)$ is a subset of Q that satisfies the following inductive rules.

- $I(p) = \{q \mid p \in \lambda(q)\}$.
- $I(\beta_1 \vee \beta_2) = I(\beta_1) \cup I(\beta_2)$.
- $I(\neg\beta_1) = Q - I(\beta_1)$.
- $I(\mu X.\phi_1)$ is the smallest set $Y \subseteq Q$ with $Y = I[X \mapsto Y](\phi_1)$, where $I[X \mapsto Y]$ is a new interpretation identical to I except that X is interpreted as Y.
- $I(\langle A\rangle\psi)$ is the set of states such that there is an enforced move vector set C by A such that, for all move vectors $\epsilon \in C$, $I[\mathsf{move} \mapsto \epsilon](\psi)$ holds:

$$I(\langle A\rangle\psi) = \bigcup_{C\in\Gamma^A} \bigcap_{\epsilon\in C} I[\mathsf{move} \mapsto \epsilon](\psi)$$

- Given $I(\mathsf{move}) = [\mathsf{e}_1, \mathsf{e}_2]$, if $[e_1, e_2] \models \eta$, then $I(\bigcirc^\eta\phi_1) = \{q \in Q \mid \delta(q, e_1, e_2) \in I(\phi_1)\}$; otherwise $I(\bigcirc^\eta\phi_1) = Q$.

A concurrent game structure is a model of an AMCE sentence ϕ, if its initial state r is in the interpretation of ϕ ($r \in I(\phi)$) for any interpretation I.

Note that, strictly speaking, AMCE does not add much to the expressiveness of AMC. In the literature, propositions have often been used to record events. Intuitively, we would need one atomic proposition for each event to mark that it has just occurred. This event marker would be true exactly at states right after the event happened. (One would possibly have to create multiple copies of states to reflect this.)

As discussed in [**?**], such a modeling technique leads to an unnecessary blow up of the state space, which could be exponential in the number of players in general concurrent games. By properly selecting the transitions with respect to operators like $\bigcirc^{\eta}$, such auxiliary propositions are not necessary when encoding the state space. Thus, AMCE can also be of interest to practitioners for the efficient analysis and verification of general concurrent games.

2.5 Resilience level checking algorithm

In Subsection **??**, we have proposed the idea of the $\mathsf{sfrch}_k(\cdot)$ operator and proposed to use its greatest fixed point for the evaluation of k-resilience. In the following, we first establish some properties of k-safety and then use AMC model-checking technology to solve the safety resilience games.

2.5.1 High-level description of the algorithm

The following lemma shows the sufficiency of k-safety as a building block for solving safety resilience games.

Lemma 6. For a safety resilience game $\mathcal{G}$, $\mathsf{sfrch}_k(\cdot)$ has a greatest fixed point.

Proof : The lemma follows from the facts that the function sfrch_k is monotonic ($S \subseteq S'$ implies $\mathsf{sfrch}_k(S) \subseteq \mathsf{sfrch}_k(S')$ because a winning strategy for the protagonist for S is also a winning strategy for S' for all states in $\mathsf{sfrch}_k(S)$) and operates on a finite domain. $\square$

For the example in Figure 2.1, considering $S = \{1\}$ ($\{1\} = \mathsf{sfrch}_2(\{1, 2, 3\})$), the only state in S, state 1, is 2-resilient: it can recover with the recovery strategy to always go to the left.

The set of k-resilient states of $\mathcal{G}$, can be calculated as the greatest solution to $S = \mathsf{sfrch}_k(S)$ with $S \subseteq Q \setminus F$. Technically we can start the inductive calculation of the greatest fixed point from base case $S_0 = Q \setminus F$, and successively calculate $S_{i+1} = \mathsf{sfrch}_k(S_i)$, for each $i \geq 0$. The set of k-resilient states is then the limit S_∞. As soon as we have $S_{i+1} = S_i$, a fixed point is reached. We then have $S_i = S_\infty$ and can stop the inductive construction. Since S_0 is finite and $S_{i+1} \subseteq S_i$ holds for all $i \geq 0$, we will eventually reach a j with $S_{j+1} = S_j = S_\infty$.

2.5.2 Realization with AMCE model-checking

We need formally define the interaction among strategies of players. We borrow the notation of function composition. Given two partial functions β_1 and β_2, we use $\beta_1 \circ \beta_2$ to represent their composition. Specifically, we have the following definition.

$$\beta_1 \circ \beta_2(a) = \begin{cases} \beta_1(a) & \text{if } \beta_2(a) \text{ is undefined.} \\ \beta_2(a) & \text{otherwise} \end{cases}$$

For our purpose, a partial strategy vector is a mapping from $\{1, 2\}$ to Σ and can be undefined for some players in $\{1, 2\}$. It is for a party $A \subseteq \{1, 2\}$ if it is defined only for players in A and represents a collaborative strategy of the players with a defined strategy in A. It is total if it is defined for all players.

For convenience, we also define partial move vectors as mappings from $\{1, 2\}$ to E. A partial move vector is for a party $A \subseteq \{1, 2\}$ if it is defined only for players in A. It is total if it is defined for all players in $\{1, 2\}$. Given two partial move vectors γ_1 and γ_2, we define $\gamma_1 \circ \gamma_2$ to represent the composition of the two vectors.

Given an S, we propose to construct $\mathsf{sfrch}_k(S)$ in an induction on k. We need the following preliminary concepts for the presentation.

Definition 7. (Traps) For $A \subseteq \{1, 2\}$, a *trap* for A is a subset $Q' \subseteq Q$ that party $\{1, 2\} \setminus A$ has a strategy vector β to keep all plays from leaving Q'. Formally, we require that, for every $q \in Q'$ and partial move vector γ for A, there exists a partial move vector γ' for $\{1, 2\} \setminus A$ such that $\delta(q, \gamma \circ \gamma'(1), \ldots, \gamma \circ \gamma'(m)) \in Q'$. $\square$

Base case, sfrch0(S)

In the base case, $\mathsf{sfrch}_0(S)$ characterizes those states, from which the protagonist can direct the plays to S and stay there via a protagonist strategy when there is no error injected by the antagonist. Thus $\mathsf{sfrch}_0(S)$ is the greatest trap for the antagonist to S when no error happens and the greatest solution to the following equation.

$$X = \left\{ q \;\middle|\; \begin{array}{l} q \in X \cap S, e \in E_1, \\ \forall e' \in E_2 (e' \neq \textit{noerr} \Rightarrow \delta(q, e, e') \in X) \end{array} \right\}.$$

In AMCE, we can alternatively define $\mathsf{sfrch}_0(S)$ as follows.

$$\mathsf{sfrch}_0(S) \overset{\text{def}}{=} \nu X.(S \wedge \langle 1 \rangle \bigcirc^{\neg 2:\textit{error}} X).$$

This is the usual safety kernel of S, which consists of those states, from which any controlled transition is safe. It can be computed by the usual greatest fixed point construction.

Lemma 7. $\mathsf{sfrch}_0(S)$ can be constructed, together with a suitable memoriless control strategy, in time linear to the size of $\mathcal{G}$.

Proof : A state $q \in S$ can stay in $\mathsf{sfrch}_0(S)$ if there is a choice $e \in E_1$ such that for all $f \in E_2$, $\delta(q, e, f) \in \mathsf{sfrch}_0(S)$. Basically, we can use the typical approach of iterative elimination to calculate $\mathsf{sfrch}_0(S)$. That is, we first let $K_0 = Q - S$. Then we a sequence of mutually disjoint sets $K_1, K_2, \ldots, K_i, \ldots$ such that for all $i \geq 1$, states in K_{i+1} can be shown to be not in $\mathsf{sfrch}_0(S)$ by evidences of states in $K_i \cup \ldots \cup K_0$. Linear time can be achieved with careful book-keeping of the choices of moves at all states in S. We need a counter c_q for each $q \in S$ initialized to $|E_1|$ for the initial number of candidate choices of moves. Then for each $[q, e] \in S \times E_1$, we need a Boolean flag $b_{[q,e]}$ initialized to *true* to represent that $\{[e, f] \mid f \in E_2\}$ is still a valid choice of moves at q to satisfy $\mathsf{sfrch}_0(S)$. For each state q, we also need to maintain a list of transition source states. That is, for each $\delta(q', e, f) = q$, we need record $[q', e, f]$ in list L_q. Then the iterative elimination proceeds as the algorithm in table 2.1. The algorithm is linear time since each transition $\delta(q, e, f)$ is checked exactly once. $\square$

Inductive cases, sfrchk(S)

Now we explain how to define the inductive cases of $\mathsf{sfrch}_k(S)$. The condition is for those states from which plays can be directed to S via a recovery segment in $Q \setminus (S \cup F)$ with k or less errors injected by the antagonist. An intermediate step for the construction of k-sfrch states is the construction of an attractor that controls, through controlled moves,

Table 2.1: Algorithm for $\mathsf{sfrch}_0(S)$ by iterative elimination

```
sfrch_0(S)
 1: for $q \in S$ do $c_q = |E_1|$ end for
 2: for $q \in S, e \in E_1$ do $b_{[q,e]} = \textit{true}$ end for
 3: Let $i = 0$ and $K_0 = Q - S$.
 4: while $K_i \neq \emptyset$ do
 5:    Let $K_{i+1} = \emptyset$.
 6:    for $q \in K_i$ and $[q', e, f] \in L_q$ do
 7:       if $b_{[q',e]}$ is true then
 8:          Let $c_{q'} = c_{q'} - 1$.
 9:          if $c_{q'}$ is 0 then add $q'$ to $K_{i+1}$. end if
10:       end if
11:       Set $b_{[q',e]}$ to false.
12:    end for
13:    Increment $i$ by 1.
14: end while
15: return $S - (K_0 \cup \ldots \cup K_i)$.
```

the play prefixes to stay in a subset $L \subseteq Q \setminus F$ of non-failure states. As only controlled (non-error) moves are allowed, this is merely a backward reachability cone.

The *controlled limited attractor set* of a set X for a limited region $L \subseteq Q$, denoted $\mathsf{cone}_L(X)$ is the set from which there is a protagonist strategy to move to X without leaving L and errors injected by the antagonist. Technically, $\mathsf{cone}_L(X)$ is the least solution to equation:

$$Y = X \cup \left\{ q \;\middle|\; \begin{array}{l} q \in L, e \in E_1, \\ \forall e' \in E_2 \setminus \{\textit{error}\}(\delta(q,e,e') \in Y) \end{array} \right\}.$$

The controlled limited attractor set $\mathsf{cone}_L(X)$ can be constructed using simple backward reachability for X of controlled transitions through states of L. In AMCE, this can be constructed as follows.

$$\mathsf{cone}_L(X) \overset{\text{def}}{=} \mu Y.(X \vee (L \wedge \langle 1 \rangle \bigcirc^{\neg 2:\textit{error}} Y))$$

Note that the protagonist must use the same move irrespective of the move of the antagonist to both stay in L and approach X, provided that the antagonist does not inject an error.

The controlled limited attractor set $\mathsf{cone}_L(X)$ is used in the construction of $\mathsf{sfrch}_k(S)$. We further construct a descending chain $V_0 \supseteq V_1 \supseteq \ldots \supseteq V_{k-1}$ of limited attractors V_i.

From V_i we have an attractor strategy towards S for the protagonist, which can tolerate up to i further errors. The respective V_i are attractors that avoid failure states. Moreover, from a state in V_i with $i > 1$, any error transition leads to V_{i-1}.

A state $q \in Q$ is *fragile* for a set $B \subseteq Q$ if, for all moves of the protagonist, at least one of its successors is outside of B. (The intuition is that this is an error move, and for simple safety resilience game structures, we can restrict the definition to failure states.) The set of fragile states for B is

$$\mathsf{frag}(B) \stackrel{\text{def}}{=} \{q \mid \forall e \in E_1 \exists e' \in E_2(\delta(q, e, e') \notin B)\}.$$

In AMCE, we have the following formulation of $\mathsf{frag}(B)$.

$$\mathsf{frag}(B) \stackrel{\text{def}}{=} [1] \bigcirc \neg B.$$

Technically, it is, however, easier to construct its dual

$$Q \setminus \mathsf{frag}(B) = \langle 1 \rangle \bigcirc B.$$

This dual can be constructed using a controlled backward reachability to B with any strategy of the protagonist.

The limited regions L_i of states allowed when approaching S also form a descending chain $L_0 \supseteq L_1 \supseteq \ldots \supseteq L_k$. Using these building blocks, we can compute the k-sfrch states as follows. The states in L_{i+1} are the non-failure states from which all error transitions lead to a state in V_i. The sets V_i contain the states from which there is a controlled path to S that progresses through L_i; all error transitions originating from any state of this path lead to V_{i-1}. V_0 is therefore just the set of states from which there is a controlled path to S.

From all states in V_{k-1}, the protagonist therefore has an optimal strategy in the recovery segment of the game described earlier: if the antagonist can play at most $k-1$ errors, then the protagonist can make sure that S is reached.

Starting with $L_0 \stackrel{\text{def}}{=} Q \setminus F$ that characterizes cones on the way to S without any errors, we define the V_k's and L_k's inductively by

$$L_k \stackrel{\text{def}}{=} L_0 \setminus \mathsf{frag}(Q \setminus \mathsf{cone}_{L_{k-1}}(S)),$$

$$\begin{array}{rcl}
L_0 & \overset{\text{def}}{=} & \neg F \\
L_k & \overset{\text{def}}{=} & \neg F \wedge \langle 1 \rangle \bigcirc \mu y.S \vee (L_{k-1} \wedge \langle 1 \rangle \bigcirc^{\mathit{error}} y \wedge \bigcirc L_{k-1}) \\
\mathsf{sfrch}_0(S) & \overset{\text{def}}{=} & \nu x.(S \wedge \langle 1 \rangle \bigcirc^{\mathit{error}} x) \\
\mathsf{sfrch}_k(S) & \overset{\text{def}}{=} & \mathsf{sfrch}_0(S \wedge L_k) \\
\mathsf{res}_k(\mathcal{G}) & \overset{\text{def}}{=} & \nu S.((Q \smallsetminus F) \wedge \mathsf{sfrch}_k(S)) : \text{the set of } k\text{-resilient states}
\end{array}$$

Table 2.2: Algorithm for k-resilient states

In AMCE, this can be defined inductively as follows.

$$L_0 \overset{\text{def}}{=} \neg F$$
$$L_k \overset{\text{def}}{=} L_0 \wedge \langle 1 \rangle \bigcirc \mathsf{cone}_{L_{k-1}}(S).$$

Finally, we choose $\mathsf{sfrch}_k(S) \overset{\text{def}}{=} \mathsf{sfrch}_0(S \cap L_k)$. In AMCE, this can be expressed as follows.

$$\mathsf{sfrch}_k(S) \overset{\text{def}}{=} \mathsf{sfrch}_0(S \wedge L_k).$$

Algorithm for the set of k-resilient states

Finding a control strategy for k-sfrch control within $\mathsf{sfrch}_k(S)$ is simple: as long as we remain in $\mathsf{sfrch}_k(S) = \mathsf{sfrch}_0(S \cap L_k)$, we can choose any control move that does not leave $\mathsf{sfrch}_k(S)$. Once $\mathsf{sfrch}_k(S)$ is left through an error transition to $V_{k-1}, V_{k-2}, ...,$ we determine the maximal i for which it holds that we are in V_i and follow the attractor strategy of $\mathsf{cone}_{L_i}(S)$ towards S.

In summary, we present our algorithms for the set of k-resilient states in Table 2.2. In fact, we have presented two algorithms. The first constructs $\mathsf{sfrch}_k(S)$, which can be used for checking whether the safety region S provided by the users is indeed a good one. The way to do it is to simply check whether S is a solution to $\mathsf{sfrch}_k(x) = x$.

Then our second algorithm calculates $\mathsf{res}_k(\mathcal{G})$ as the greatest fixed point S of $\mathsf{sfrch}_k(.)$ as the recommendation for the safety region:

$$\mathsf{res}_k(\mathcal{G}) = \bigcup \{S \subseteq Q \mid S = \mathsf{sfrch}_k(S) \text{ and } S \cup F = \emptyset\}.$$

In this way, the users do not have to calculate and provide the safety region, which would be error prone. According to the argument and lemmas from above, we get the following theorem.

Theorem 8. $\mathcal{G}$ is k-resilient if, and only if, $r \in \mathsf{res}_k(\mathcal{G})$. □

2.5.3 Complexity

A rough complexity of our resilience level checking algorithm straightforwardly follows the complexity of AMC model-checking. Specifically, the following lemma explains the maximal resilience level that we need consider. For convenience, let $k_{\max}$ be the maximal resilience level of $\mathcal{G}$.

Lemma 9. $k_{\max}$ is either infinite or no greater than $|Q \smallsetminus F|$.

Proof : We assume that $k_{\max}$ is greater than $|Q \smallsetminus F|$ but not infinite. This means that there exists a failed recovery segment ρ with $k+1$ errors injected by the antagonist. Since the protagonist can only use memoryless strategies, there must be two position indices $i < j < |\rho| - 1$ with $\rho(i) = \rho(j)$ in the recovery segment such that at $\rho(i)$ and $\rho(j)$, the protagonist makes the same move while the antagonist makes different moves. This implies the existence of a shorter failed recovery segment $\rho[0, i]\rho[j+1, |\rho|-1]$. By repeating the above argument, we can eventually identify a failed recovery segment of length $\leq |Q \smallsetminus F|$ that contradicts the assumption and establishes the lemma. □

With Lemma 9, we can use the complexity of AMC model-checking problem [**?**] to straightforwardly establish the $O(k_{\max}|E|)^2 = O(|Q \smallsetminus F| \cdot |E|)^2$ complexity of $\mathsf{res}_k(\mathcal{G})$ when k is $k_{\max}$. In the following, we present a more detailed analysis of the complexity of our resilience level checking algorithm. All individual steps in the construction (intersection, difference, predecessor, and attractor) are linear in the size of the safety resilience game, and there are $O(k)$ of these operations in the construction. This provides a bi-linear (linear in k and $|\mathcal{G}|$) algorithm for the construction of sfrch_k and a strategy for the protagonist.

Lemma 10. A memoryless control strategy for the states in $\mathsf{sfrch}_k(S)$ can be constructed in time linear in both k and the size $|\mathcal{G}|$ of the safety resilience game $\mathcal{G}$. □

The construction of $\mathsf{res}_k(\mathcal{G})$ uses the repeated execution of $(Q \smallsetminus F) \wedge \mathsf{sfrch}_k(\cdot)$. The execution of $\mathsf{sfrch}_k(\cdot)$ needs to be repeated at most $|Q \smallsetminus F|$ times until a fixed point is reached, and each execution requires at most $O(k \cdot |\mathcal{G}|)$ steps by Lemma 10.

For the control strategy of the protagonist, we can simply use the control strategy from $\mathsf{sfrch}_k(S_\infty)$ from the fixed point S_∞. This control strategy is memoryless (cf. Lemma 10).

Lemma 11. $\mathsf{res}_k(\mathcal{G})$ and a memoryless k-resilient control strategy for $\mathsf{res}_k(\mathcal{G})$ can be constructed in $O(k \cdot |Q \smallsetminus F| \cdot |\mathcal{G}|)$ time. $\square$

Finding the resilience level $k_{\max}$ for the initial state r requires at most $O(\log k_{\max})$ many constructions of $\mathsf{res}_i(\mathcal{G})$. We start with $i = 1$, double the parameter until $k_{\max}$ is exceeded, and then use logarithmic search to find $k_{\max}$.

Corollary 12. For the initial state r, we can determine the resilience level $k_{\max} = \max\{i \in \mathbb{N} \mid r \in \mathsf{res}_i(Q \smallsetminus F)\}$ of r, $\mathsf{res}_{k_{\max}}(Q \smallsetminus F)$, and a memoryless $k_{\max}$-resilient control strategy for $\mathsf{res}_{k_{\max}}(Q \smallsetminus F)$ in $O(|Q \smallsetminus F| \cdot |\mathcal{G}| \cdot k_{\max} \log k_{\max})$ time. $\square$

Simple safety resilience game structures. For simple safety resilience game structures, checing if a state is in $\mathsf{sfrch}_0(S)$ is NL-complete.

Lemma 13. Testing if a state is in $\mathsf{sfrch}_0(S)$ is NL-complete.

Proof. NL completeness can be shown by reduction to and from the repeated ST-reachability [**?**] (the question whether there is a path from a state S to a state T and from T to itself in a directed graph). $\square$

Likewise, the controlled limited attractor set $\mathsf{cone}_L(S)$ can be constructed using simple backwards reachability for G of controlled transition through states of L. For $A = \mathsf{cone}_L(S)$, determining whether a state is in A is NL-complete (see [**?**]).

The complexity of determining whether or not a state q is in $\mathsf{sfrch}_k(S)$ thus depends on whether or not we consider k to be a fixed parameter. Considering k to be bounded (or fixed) is natural in our context, because k is bounded by the redundancy.

Lemma 14. For a fixed parameter k, testing if a state s of a simple safety resilience game structures is in $\mathsf{sfrch}_k(S)$ is NL-complete.

Proof. Testing if a state is in L_0 is in NL. By an inductive argument, we can show that

- provided that testing if a state is in L_i is in NL, we can test if a state is in $A_i = \mathsf{cone}_{L_i}(S)$ by using the nondeterministic power to guess a path towards S, while verifying that we are in L_i in every state we pass before S is reached; and
- if we can check if a state is in A_i in NL, then we can check if it is in $Q \smallsetminus A_i$ [**?**], in $\mathsf{frag}(Q \smallsetminus A_i)$ (with one nondeterministic transition), and in $L_{i+1} = L_0 \smallsetminus \mathsf{frag}(S \smallsetminus A_i)$ [**?**] in NL.

Testing that a state is in $S \cap L_k$ is therefore in NL and testing if it is in $\mathsf{sfrch}_0(S \cap L_k)$ reduces to guessing a state t in $\mathsf{sfrch}_k(G)$ and an ST path (a path from s to t followed by a loop from t to t), verifying for all states on the path that they are in $S \cap L_k$.

For hardness, note that the last step of the construction alone is NL-complete (Lemma 7).
□

If k is considered an input, then reachability in AND-OR graphs can easily be encoded in LOGSPACE: It suffices to use the nodes of an AND-OR graph as the states, the outgoing edges of OR nodes as the result of the choice of the protagonist only (while the move of the antagonist has no influence on the outcome, no matter whether or not she induces an error), and to model the AND nodes as a state, where the no-error move of the antagonist will lead in cycling in the state, while the antagonist can choose the successor from the graph when inducing an error. Choosing k to be the number of nodes of the AND-OR graph and F to be the target nodes of the AND-OR graph, the target nodes of the AND-OR graph are not reachable from a state s iff $s \in \mathsf{sfrch}_k(Q \smallsetminus F)$.

Given that reachability in AND-OR graphs is PTIME-complete [**?**], this provides:

Lemma 15. If k is considered an input parameter, then testing if a state s of a simple safety resilience game structures is in $\mathsf{sfrch}_k(S)$ is PTIME-complete. □

The complexity of $\mathsf{res}_k(S)$ is (almost) independent of the parameter k:

Theorem 16. The problem of checking whether or not a state s is k-resilient for a set S is PTIME-complete for all $k > 0$ and NL-complete for $k = 0$.

Proof. We have shown inclusion in PTIME in Lemma 11. For hardness in the $k > 0$ case, we can use the same reduction from the reachability problem in AND-OR graphs as for $\mathsf{sfrch}_k(S)$.

For $k = 0$, $\mathsf{sfrch}_0(G) = \mathsf{sfrch}_0\Big(\mathsf{sfrch}_0(G)\Big)$ implies $\mathsf{res}_0(G) = \mathsf{sfrch}_0(G)$. The problem of checking if a state is in $\mathsf{res}_0(G)$ is therefore NL-complete by Lemma 13. □

Hardness for general safety resilience game structures. For general resilience game structures, we can again use a LOGSPACE reduction from the reachability in AND-OR graphs: We again use the nodes of an AND-OR graph as the states, and the outgoing edges of OR nodes are selected based on the choice of protagonist only. For the AND nodes, we leave the choice to the antagonist only, whithout the need to invoke an error. (That is, errors play no role in this reduction. The antagonist may be allowed to insert one, but she can always obtain the same transition without doing so.)

Marking F as the target nodes, we get $\mathsf{res}_k(Q \setminus F) = \mathsf{sfrch}_l(Q \setminus F)$ for all non-negative integers k, l, and $s \in \mathsf{sfrch}_0(Q \setminus F)$ iff the target nodes of the AND-OR graph are not reachable from s. With Lemmas 10 and 11, we get the following theorem.

Theorem 17. For all $k \geq 0$, the problems of checking whether or not a state s is in $\mathsf{res}_k(Q \setminus F)$ and $\mathsf{res}_k(Q \setminus F)$, respectively, are PTIME-complete for general safety resilience game structures.

2.6 Tool implementation and experimental results

In the following, we report our implementation and experiment with our constructions. Our implementation is based on symbolic on-the-fly model-checking techniques and built on the simulation/model-checking library of REDLIB in `https://github.com/yyergg/Resil` for fast implementation. Our implementation and benchmarks can also be found in the same page.

We adopt *CEFSM* (*communicating extended finite-state machine*) [**?**] as a convenient language for the description of abstract models of our concurrent game structures. A CEFSM consists of several finite-state machines extended with shared variables for the modeling of shared memory and with synchronizations for the modeling of message-passing in distributed systems. This is justifiable since the fault-tolerant algorithms may themselves be subject to restrictions in concurrent or distributed computation. Indeed, we found CEFSM very expressive in modeling the benchmarks from the literature [?, ?].

The translation from our CEFSMs to state transition systems, such as finite Kripke structures, is standard in the literature. All state spaces, conditions, preconditions, post-conditions, fixed points, etc., are represented as logic formulas. The logic formulas are then implemented with multi-value decision diagrams (MDD) [?].

We then took advantage of the support of REDLIB for writing down template automatas for constructing complex models. We specified a template automata with REDLIB to describe the moves of the players. Conceptually, the player automatas are constructed as an instance of the template automata. Then the whole game structure is constructed as the product of all player automatas. Finally, we use the API of REDLIB to do on-th-fly construction of the game structure which can be advantageous since unreachable states will never be generated.

2.6.1 Benchmarks

We use the following five parameterized benchmarks to check the performance of our techniques. Each benchmark has parameters for the number of participating modules in the model. Such parameterized models come in handy for the evaluation of the scalability of our techniques with respect to concurrency and model sizes.

1. We use the example of a fault-tolerant computer architecture (Example 2) as our first benchmark. An important feature of this benchmark is that there is an assumed mechanism for detecting errors of the modules. Once an error is detected, a processor can be assigned to recover the module, albeit to the cost of a reduced redundancy in the executions.

2. Voting is a common technique for fault tolerance through replication when there is no mechanism to detect errors of the modules [?]. In its simplest form, a system can

guarantee correctness, provided less than half of its modules are faulty. This benchmark implements this simple voting mechanism. Every time a voting is requested, the modules submit their ballots individually. Then we check how many module failures the system can endure and recover.

3. This is a simplified version of the previous voting benchmark, where we assume that there is a blackboard for the client to check the voting result.

4. *Practical Byzantine fault-tolerance* (*PBFT*) algorithm: We use an abstract model of the famous algorithm by Castro and Liskov [**?**]. It does not assume the availability of an error-detection mechanism but uses voting techniques to guarantee the correctness of computations when less than one third of the voters are faulty. This algorithm has impact on the design of many protocols [**?**, **?**, **?**, **?**, **?**] and is used in Bitcoin [**?**], a peer-to-peer digital currency system.

5. *Fault-tolerant clock synchronization algorithm*: Clock synchronization is a central issue in distributed computing. In [**?**], Ramanathan, Shin, and Butler presented several fault-tolerance clock synchronization algorithms in the presence of Byzantine faults with high probability. We use a nondeterministic abstract model of the convergence averaging algorithm from their paper. The algorithm is proven correct when no more than one third of the local clocks can drift to eight time units from the median of all clock readings.

2.6.2 Modeling of the fault-tolerant systems

Appropriate modeling of the benchmarks is always important for the efficient verification of real-world target systems. Many unnecessary details can burden the verification algorithm and blow up the computation, while sketchy models can then give too many false alarms and miss correct benchmarks. We have found that there is an interesting aspect in the modeling of the above benchmarks. Replication and voting are commonly adopted techniques for achieving fault-tolerance and resilience. Such fault-tolerant algorithms usually consist of several identical modules that use the same behavior templates.

This observation implies that the identity of individual modules can be unimportant for some benchmarks. For such benchmarks, we can use counter abstraction [**?**, **?**] in their models. Specifically, with counter abstraction, we can model all system players with one player that keeps a counter $c(l)$ for each control location l in the template automatas. Then at a state of the whole game graph, $c(l)$ records the number of system players at location l. With this technique, a system with $m-1$ system player and one error model player is then reduced to two players: one counter-abstraction player for all the system players and one remaining error model player. If a system player enters a location l in a global transition, then in the model, $c(l)$ is incremented by one in the abstract global transition. If a system player leaves l in the global transition, then $c(l)$ is decremented by one in the abstract global transition. But the succession of location movements of a particular player is omitted from the abstraction.

We found that we can use counter abstraction to prove the correctness of benchmarks 1, 2, and 3. In contrast, the PBFT and the clock synchronization algorithms use counters for each module to model the responses received from its peer modules. As a result, we decided not to use counter abstraction to model these two algorithms in this work.

In the following, we explain how to apply our techniques to analyze the resilience levels of the avionic systems in Example 2. The application is achieved in three steps. We first model the system under analysis either as a plain CEFSM or with counter abstraction (if our analysis tool cannot handle the complexity of the plain CEFSM). We then build the product automaton of the CEFSM as the resilience game structure except for the move vectors. Finally, we convert the labels on the transitions of the product automaton to move vectors of the two players. Note that the moves may not correspond to the transition labels of the CEFSM.

Step 1: the construction of the CEFSM

We first present the CEFSM model template of Example 2 in Figure **??**. The CEFSM model has n processors and m memory modules. Figures **??**(a) and (b) are for the abstraction of processors and memory copies, respectively. The ovals represent local states of a processor or a memory module, while the arrows represent transitions. The transitions of a CEFSM are labeled with '*error*', 'C' (for Control), or 'R' (for recovery).

We also use synchronizers to bind process transitions. For example, when a memory module moves into a faulty state, an idle processor may issue an `fd` (error-detected) event and try to repair the module by copying memory contents from normal memory modules. Such error-detection is usually achieved with standard hardware. Note that the benchmarks are models that reflect the recovery mechanism, abstracting away the details of the original systems. A central issue in the design of this recovery mechanism is then the resilience level of the controlled systems. We need three synchronizers: *fd* for error detection by a processor, *rs* for recovery success, and *rf* for recovery failure. The three synchronizers are used to bind a transition from a processor and another from a memory module into a synchronized transition. For example, a processor at state *pidle* and a memory module at state *mfaulty* may simultaneously enter their *pcopy* and *mcopy* states respectively through synchronizers $!fd$ (for sending the synchronizer) and $?fd$ (for receiving). We also conveniently use a variable q in this synchronized transition to capture the identifier of the memory module receiving the synchronizer. A transition without synchronization labels is considered a trivial synchronized transition. The transition system of the CEFSM operates with interleaving semantics at the abstraction level of the synchronized transitions.

For counter abstraction, we need four global variables *crp*, *cfp*, *crm*, and *cfm* respectively to keep track of the numbers of running processors, faulty processors, running memory modules, and faulty memory modules. We also need a local variable *idm* for each processor to record the faulty memory module identifier that the processor is responsible for recovery. We label the controllable, error, and recovery transitions respectively with 'C', '*error*', and 'R'. We also label each transition with synchronizers and actions. At any moment, the processors and the memory modules may enter their running states, execute a task, and generate the outcome. A processor starts its execution from state *prun* while a memory module starts from state *mrun*.

Step 2: building the product automata

The product automata is a Kripke structure whose states are of is a vector $[p_1, \ldots, p_n, i_1, \ldots, i_n, s_1, \ldots, s_m$ of $2n + m$ elements. For all k, p_k and i_k respectively represent the current location and the current *idm* value of processor k while s_k represents the current location of memory module k. Then interleaving semantics that each time only a global transition (a single local process transition without synchronizers or two local process transition bound by a synchronizer) is executed is adopted to determine the transition relation from one state to another. Such techniques are standard in model construction. REDLIB can help in this regard by constructing the Kripke structure in an on-the-fly style to avoid the construction of those states not reachable from the initial state.

Step 3: the labeling of the move vectors

After the second step, we have the game structure ready except for the move vectors on the transitions. We use $E_1 = \{C, R, \mathit{nop}\}$, where *nop* represents "no operation," and $E_2 = \{\mathit{noerr}, \mathit{error}\}$. Then we use the following three rules to label move vectors.

- Every global transition with one component local process transition labeled with *error* is labed with move vector $[\mathit{nop}, \mathit{error}\}$.
- Every global transition with a component local process transition labeled with R is labeled with move vector $[R, \mathit{noerr}]$.
- All other global transitions are labeled with move vector $[C, \mathit{noerr}]$.

Counter abstraction of the example

We also use the CEFSM in figure **??** to explain counter abstraction. We need eight counter variables: *pr*, *pi*, *pc*, *pf*, *mr*, *mi*, *mc*, and *mf* to respectively record the number of processes in location prun, pidle, pcopy, pfaulty, mrun, midle, mcopy, and mfaulty in a state. Then the counter abstraction of the CEFSM is in Figure **??**. The initial state are specified with constraint: $\mathit{pr} = n \wedge \mathit{pi} = 0 \wedge \mathit{pc} = 0 \wedge \mathit{pf} = 0 \wedge \mathit{mr} = m \wedge \mathit{mi} = 0 \wedge \mathit{mc} = 0 \wedge \mathit{mf} = 0$ on the counters. The state in the product automata must satisfy the following constraints: $pr + pi + pc + pf = n \wedge mr + mi + mc + mf = m$. As can be seen, we do not care which processor is in the idle mode, in the running mode, etc., in this abstraction. Similarly, we do not care which memory module is in the idle mode, in the running mode, and etc. The local state transition only keeps tracks of the number of processors in each mode and the number of memory modules in each mode. We also do not care which processor is in charge of the recovery of which memory module. Such an abstraction can be done automatically.

The labeling of the move vectors on the transitions in the Kripke structure (product automaton) follows the same rules for the product automaton from the CEFSM in Figure **??**.

Analysis of the game structure

The majority outcome of the processors and memory copies is used as the outcome of the system. A processor may enter the faulty state. A memory module may also enter the faulty state. Processors may control to recover themselves or a faulty memory module by copying the contents of a functioning memory module to the faulty one. At any moment, we want to make sure that we can always recover to a global condition with the following two restrictions.

- There are at least two more processors in the running mode than the processors in the faulty mode.
- There are at least two more memory copies in the running mode than memory copies in the faulty mode.

Together, the failure condition is $crp - cfp < 2 \vee crm - cfm < 2$. That is, all states in the transition system satisfying $crp - cfp < 2 \vee crm - cfm < 2$ are in set F.

Tool implementation and the benchmarks used in the experiment can all be found in our Sourceforge REDLIB project at `https://github.com/yyergg/Resil`.

Table 2.3: Performance data for resilience calculation s: seconds; M: megabytes

benchmarks	concurrency	k	game sizes		sfrch_k		
			#nodes	#edges	time	memory	time
avionics	2 processors & 2 memory modules	2	118	750	0.62s	114M	0.85s
	2 processors & 3 memory modules	2	414	3252	0.94s	139M	1.10s
	3 processors & 3 memory modules	3	1540	15090	4.67s	225M	8.38s
	3 processors & 4 memory mdules	3	5601	63889	42.86s	815M	155s
avionics (counter abstraction)	6 processors & 6 memory modules	2	1372	6594	2.89s	129M	3.54s
	7 processors & 7 memory modules	3	2304	11396	10.7s	216M	23.4s
	8 processors & 8 memory modules	3	3645	18432	43.8s	1009M	135s
voting	1 client & 20 replicas	9	9922	23551	7.01s	260M	36.7s
	1 client & 26 replicas	12	20776	49882	19.9s	474M	79.6s
simple voting	1 client & 150 replicas	74	458	1056	0.71s	159M	31.7s
	1 client & 200 replicas	99	608	1406	1.06s	161M	162s
	1 client & 250 replicas	124	758	1756	1.36s	163M	307s
PBFT	1 client & 6 replicas	2	577	897	0.34s	72M	1.05s
	1 client & 9 replicas	4	2817	4609	13.3s	564M	58.5s
clock sync	1 client & 15 servers	7	16384	229376	45.1s	3075M	62.4s
	1 client & 17 severs	8	65536	1070421	870s	14725M	915s

2.6.3 Performance data

We report the performance data in Table 4.1 for the resilience algorithms described in Section **??** against the parameterized benchmarks in the above with various parameters. The second column shows the concurrency sizes. The third column shows the values of k for the rows. The fourth and fifth columns show the sizes of the concurrent game structures. The sixth and seventh columns show the time and spaces used to calculate $\mathrm{sfrch}_k()$. Similarly, the eighth and ninth columns show the time and spaces for calculating the $\mathrm{res}_k()$.

The benchmark in Figure **??** does not have nodes in $\mathsf{sfrch}_2(G)$ and $\mathsf{res}_2(G)$. So we changed the benchmark to see how we check our implementation with $k > 1$. The change is that the recovery transition from state *pcopy* to *pidle* of processors are relabeled as controllable. This change significantly limits the ability of the system errors to derail the system. For the avionics system, the resilience level k is set to one less than half the number of processors. For the voting and simple voting benchmarks, the value of k is set to one less than half the number of replicas (voters). For the PBFT and clock synchronization algorithm, we choose k to be one less than one third of the number of replicas.

The performance data has been collected with a Virtual Machine (VM) running opensuse 11.4 x86 on Intel i7 2600k 3.8GHz CPU with 4 cores and 8G memory. The VM only uses one core and 4G memory.

The time and space used to calculate resilience is a little bit more than that to check for sfrch. The reason is that sfrch_k is a pre-requisite for calculating res_k. In our experiment, sfrch_k is usually very close to res_k and does not require much extra time in calculating res_k out of sfrch_k.

The experiments show that our techniques scale to realistic levels of redundancy. For fault-tolerant hardware, usually the numbers of replicas are small, for example, less than 10 replicas. Thus our techniques seem very promising for the verification and synthesis of hardware fault-tolerance.

On the other hand, nowadays, software fault-tolerance through networked computers can create huge numbers of replicas. Our experiment shows that counter abstraction can be a useful techniques for the modeling and verification of software resilience. Specifically, for the avionics benchmark, we can verify models of much higher concurrency and complexity with counter abstraction than without.

2.7 Related works

We have applied game-based technqiues [?, ?, ?] for synthesizing a control mechanism with maximal resilience to software errors. The synthesis of control strategies is essential in solving games with temporal and ω-regular objectives. For these more complex objective, synthesis goes back to Church's solvability problem [?] and inspired Rabin's work on finite automata over infinite structures [?] and Büchi and Landweber's works on finite games of infinite duration [?, ?]. A righ body of literature on synthesis has since been developed [?, ?, ?, ?, ?, ?, ?].

Traditionally, fault tolerance refers to various basic fault models [?], such as a limited number of errors [?]. These traditional fault models are subsumed by more general synthesis or control objectives [?, ?, ?]; as simple objectives with practical relevance, they have triggered the development of specialized tools [**?**, **?**].

Dijkstra's self-stabilization criterion [**?**, **?**] suggests to build systems that eventually recover to a 'good state', from where the program commences normally. Instead of *constructing* a system to satisfy such a goal, one might want to apply control theory to *restrict* the execution of an existing system to achieve an additional goal. Our control objective is a recovery mechanism for up to k errors. After recovery, the system has to tolerate up to k errors again, and so forth. In this work, we suggest a mechanism to synthesize a recovery mechanism for a given fault model and recovery primitives.

In [**?**], an interesting notion of robustness based on Hamming and Lewenstein distance related to the number of past states is defined. It establishes a connection between these distances with a notion of synchronization that characterizes the ability of the system to reset for combinatorial systems. In [**?**], 'ratio games' are discussed, where the objective is to minimize the ratio between failures induced by the environment and system errors caused by them.

Besides using our simple game model that neither refer directly to time, nor to probabilities, one can also consider models that make these aspects explicit. Their analysis is far more complex (with [**?**] offering the best complexity bounds), and so are the resulting strategies. If we, for example, return to the example of airplanes with an operation time of 20 hours referred to in Table 1.1, then an optimal timed model would take the remaining operation time into account. When the remaining time is two minutes, the balance between being resilient against waves of two errors and being resilient against 5 errors looks very different, and the optimal control would change over time rather than being static. Another implication of more complex models would be that the error model would have to be more detailed. Even if one assumes that a simple concept like safe states persists, it depends on the fineties of such a model if a two step path back to it where an error after step one leads to system failure is preferable over a much longer path, say through 10,000 intermediate states, where one error can be tolerated during recovery.

We believe that the independence from such details is an advantage of our technique, partly because it is simpler and cheaper, and partly because the further advantages one can obtain from more detailed error models rely heavily on very knowledge of (or, realistically, on very detailde assumptions on) how errors are distributed.

In [**?**, **?**] the resilience model we have introduced [**?**] has been applied for synthesising robust control in an assume-guaranee setting to produce robustness against occasional noncompliance of the environment with the assumptions of its behavior.

2.8 Conclusion

We have introduced an approach for the development of a control of safety critical systems that maximizes the number of *dense* errors the system can tolerate. Our techniques are inspired by the problem of controlling systems with redundancy: in order to deflect the effect of individual errors, safety critical systems are often equipped with multiple copies of various components. If one or more components fail, such systems can still work properly as long as the correct behavior can be identified.

This has inspired the two-phase formulation of the safety resilience problems in this article. In the first phase, we identify a k-resilient region, while we develop a control strategy for recovery in the second phase. After an error, the controller can recover to the k-resilient region without encountering a system failure, unless the error is part of a group of more than k errors that happen in close succession. Such a recovering strategy is memoryless. Being memoryless on a small abstraction in particular implies that the recovery is fast.

The system can, once recovered, tolerate and recover from k further dense errors, and so forth. Consequently, our control strategy allows for recovery from an arbitrary number of errors, provided that the number of dense errors is restricted. This is the best guarantee we can hope for: our technique guarantees to find the optimal parameter k. This parameter is bound to be small (smaller than the number of redundant components). Optimizing it is computationally inexpensive, but provides strong guarantees: the likelihood of having more than k errors appear in short succession after an error occurred are, for independent errors, exponential in k. As errors are few and far between, each level of resilience gained reduces the likelihood of system-level failures significantly.

Chapter 3

Basic Strategy-interaction Logic

3.1 Existed Logics about Game and Strategy

3.1.1 Prior to Strategy Logics

Kupferman, Vardi, and Wolper proposed Module checking, a famous framework for checking whether open systems can satisfy temporal logic properties [?]. In the framework, the open systems are modeled as a 2-agent turn-based games, one agent is the system and the other is the environment. The specification properties can be in CTL, CTL*, and LTL. Since there are only two agents, non-trivial strategy collaboration does not exist and the techniques in our model-checking algorithms cannot contribute to the improvement of their algorithms.

Alur, Henzinger, and Kupferman presented ATL (alternating-time temporal logic), ATL*, AMC (alternating-time μ-calculus), and GL (game logic) with strategy quantifier $\langle\langle M\rangle\rangle$ [?].

Brihaye, Lopes, Laroussinie, and Markey introduced a very expressive extension to ATL* with other players' stratregies and memory constraints [?]. They also showed that the model-checking problem of this extension is decidable.

Pinchinat introduced a way to specify expressive constraints on strategies in concurrent games by extending μ-calculus with decision modalities [?]. Laroussinie and Markey reported decidability of satisfiability problems in this direction with new context constraints, e.g., the number of moves by agents are bound [?].

3.1.2 With Strategy Logics

Chatterjee, Henzinger, and Piterman introduced a strategy logic allowing for first-order quantification over strategies [?]. The decision procedure is however non-elementary.

Mogavero, Murano, Perelli, Sauro, Vardi et al also identified fragments of strategy logics with enjoys a doubly exponential time complexity model-checking algorithm [?, ?, ?]. For example, in [?], conjunction and disjunction cannot happen in the same scope of strategy quantification. BSIL is also a fragment of strategy logic with restriction on the hierarchy of SIQs and sometimes can be handy in expressing Boolean relation among strategies. Moreover, the restriction on BSIL results in a much lower model-checking complexity of PSPACE.

3.2 Running Example

We use the banking example in the introduction to explain the idea of this work. Suppose that a bank want to provide the following services.

- Depositing to an account by a client from the root screen.
- Transferring money from an account in the bank to an account in another bank, also from the root screen.

As we can see, the deposit service involves the interaction between a client and the bank while the transfer service involves at least three parties: a client, the bank, and a partner bank. Also in the meantime, the bank wants to forbid any client and partner from checking sensitive information of other clients, .e.g., checking the password of another client.

3.2.1 Trying to Write Down a Correct Formal Specification

To develop the services, the bank manager in charge of the project need specify the services and make sure that the specification is not erroneous. If they turn to the literature, the bank manager may choose ATL or its extensions, e.g., fair ATL, ATL*, AMC, GL, etc., to specify the services. The choice is plausible at first glance since ATL and fair ATL both have a polynomial time model-checking algorithm and could support the verification of the services. So the manager could easily specify the services with the following formula.

$$\langle 1\rangle \left(\begin{array}{ll} & \Box\neg \mathit{checkOthersPassword} \\ \wedge & \langle 2\rangle\Diamond \mathit{depositDone} \\ \wedge & \langle 2,3\rangle\Diamond \mathit{transferDone} \end{array} \right) \tag{C}$$

Here we use agent index 1 for the bank, 2 for the client, and 3 for the partner bank. But on a second look, we can see that that this specification is too restrictive. Subformula $\langle 2\rangle\Diamond \mathit{depositDone}$ says that the client can force a deposit transaction no matter how the banking system responds. In practice, there are many factors beyond the control of the client for the success of a transaction. For example, when the banking system is in maintenance or out of order, then the transaction may fail.

After realizing the problem with formula (C), the manager decides to rewrite the specification as follows.

$$\begin{array}{ll} & \langle 1\rangle\Box\neg \mathit{checkOthersPassword} \\ \wedge & \langle 1,2\rangle\Diamond \mathit{depositDone} \\ \wedge & \langle 1,2,3\rangle\Diamond \mathit{transferDone} \end{array} \tag{D}$$

This specification fixes the issue observed in formula (C) since now $\langle 1,2\rangle\Diamond \mathit{depositDone}$ says with the collaboration of the banking system, the client can finish a deposit transaction. However, there is a subtle issue which nullifies the formula. In the collaboration between the banking system and the client for property $\langle 1,2\rangle\Diamond \mathit{depositDone}$, the banking system is no longer required to maintain property $\Box\neg \mathit{checkOthersPassword}$ since according to the semantics of ATL, the specification does not require the banking system to use the same strategy to enforce both $\langle 1\rangle\Box\neg \mathit{checkOthersPassword}$ and $\langle 1,2\rangle\Diamond \mathit{depositDone}$. The same issue also appears in $\langle 1,2,3\rangle\Diamond \mathit{transferDone}$. That is, the banking system's strategy in enforcing $\langle 1,2,3\rangle\Diamond \mathit{transferDone}$ is not required to be the same one in enforcing $\langle 1\rangle\Box\neg \mathit{checkOthersPassword}$. Thus, if the bank cannot sue their contractors for developing the service system if the systems leaks clients' passwords in transferring funds.

3.2.2 Resorting to General Strategy Logics for a Correct Specification

After several trials, the manager eventually finds out that the specification cannot be expressed in ATL, ATL*, GL, and AMC [?]. Then the manager turns to strategy logics [?, ?] in the literature and finds out that the specification can be expressed as follows.

$$\langle x\rangle\langle y\rangle\langle z\rangle\langle w\rangle \left(\begin{array}{ll} & (1,x)\Box\neg \mathit{checkOthersPassword} \\ \wedge & (1,x)(2,y)\Diamond \mathit{depositDone} \\ \wedge & (1,x)(2,z)(3,w)\Diamond \mathit{transferDone} \end{array}\right) \tag{E}$$

This formula declares four existentially quantified strategies: x, y, z, and w, and says the following.

- On using strategy x, the banking system can enforce $\Box\neg$*checkOthersPassword*.
- On using strategy x and y respectively, the banking system and a client can finish the deposit transaction.
- On using strategy x, z, and w respectively, the banking system, the client, and a partner bank can also finish a fund transfer transaction.

The manager is happy with the elegant specification and wants to verify the specification on the model of the development delivery. Then he finds out that there is no tractable algorithm and working tools for checking this specification or synthesize the strategies. In fact, the complexity reported in the literature is at best doubly exponential [?, ?]. So the manager finds himself in the dilemma between expressiveness and verification efficiency.

But when the manager bangs his head to the theoretical challenges, he soon realizes that strategy logics offer more expressiveness than he expects. For example, he could write down the following formula.

$$\langle x\rangle\langle y\rangle[z]\langle w\rangle \left(\begin{array}{ll} & (1,x)\Box\neg \mathit{checkOthersPassword} \\ \wedge & (1,x)\Box(2,y)\Diamond \mathit{depositDone} \\ \wedge & (1,x)\Box(2,z)(3,w)\Diamond \mathit{transferDone} \end{array}\right) \tag{F}$$

Here $[z]$ universally quantifies a strategy named z. Thus subformula $(1,x)(2,z)(3,w)\Diamond$*transferDone* in fact says that the banking system has a strategy x such that for all strategies z of the client, the partner bank have a strategy w to help enforcing $\Diamond$*transferDone*. Considering that a strategy is an action decision function on the history, this in fact means that strategy w is clairvoyant and contradicts the implementation assumption of all practical banking systems. Indeed the huge verification complexity of strategy logics can be attributed to such free-style binding operations of strategy names to agents. Thus, the manager asks himself whether there is a subclass of strategy logics that allows for elegant expressions of natural and practical specifications while supporting verification with less complexity. Similar examples in fact can appear in different projects with services that rely on collaboration of multiple agents.

3.2.3 BSIL: New Strategy Modalities Expressive Enough for the Specification

In fact, in conceivable applications, the developers cannot implement and are not interested at clairvoyant strategies. In other words, for practical specifications, it is likely that existential SIQs are expressive enough. In BSIL, formula (E) can be rewritten as follows.

$$\langle 1 \rangle \left(\begin{array}{ll} & \Box\neg \mathit{checkOthersPassword} \\ \wedge & \langle +2 \rangle \Diamond \mathit{depositDone} \\ \wedge & \langle +2, 3 \rangle \Diamond \mathit{transferDone} \end{array} \right) \tag{G}$$

This formula says that the banking system has a strategy that at any instant, the system can ensure that no password is leaked, the system allows a client to finish a deposit transaction, and the system allows a client to transfer funds from or to a partner bank.

3.2.4 Symbolic Strategy Names and Path Obligations

But to verify such a BSIL property, we need new techniques to check that along plays, the same decision is made for the same strategy. Consider the model of the banking system in figure **??**. We need to find strategies of the agents to fulfill $\Box\neg$*checkOthersPassword*, $\Diamond$*depositDone*, and $\Diamond$*transferDone*. To assist in our explanation of our verification algorithms, we use symbolic strategy names x, y, z, w and the binding notations in formula (F).

In fact, formula (G) is exactly the same as formula (E). Thus by interpreting all symbolic strategy names as existential quantified, formula (G) can be rewritten as follows.

$$((1,x)\Box\neg \textit{checkOthersPassword}) \wedge ((1,x)(2,y)\Diamond \textit{depositDone}) \wedge ((1,x)(2,z)(3,w)\Diamond \textit{transferDone}) \quad \text{(H)}$$

Note that since we only allow explicit existential strategy quantification, the number of strategy bindings are exactly determined by the number and sizes of SQs and SIQs in the formula.

If we label $(1,x)\Box\neg$*checkOthersPassword*, $(1,x)(2,y)\Diamond$*depositDone*, and $(1,x)(2,z)(3,w)\Diamond$*transferDo* on nodes of a computation tree in a labeling procedure, then the symbolic names precisely reflect the the constraints on the decisions along plays. For example, suppose that both $(1,x)\Box\neg$*checkOthersPassword* and $(1,x)(2,y)\Diamond$*depositDone* are labeled on a state of location *dep*. This state has three successor states: for convenience C1 of location *idle* via $[\text{fail}, \text{commit}, \bot]$, C2 of location *idle* via $[\text{succ}, \text{cancel}, \bot]$, and C3 of location *depDone* via $[\text{succ}, \text{commit}, \bot]$. If strategy x chooses action succ, then obligation $(1,x)\Box\neg$*checkOthersPassword* will be passed down to C2 and C3. In this situation, whether obligation $(1,x)(2,y)\Diamond$*depositDone* is labeled on C2 or C3 relies on the action decision of strategy y. But the constraint on strategy decision is that if $(1,x)(2,y)\Diamond$*depositDone* is labeled on C2, then $(1,x)\Box\neg$*checkOthersPassword* must also be labeled on C2; and if $(1,x)(2,y)\Diamond$*depositDone* is labeled on C3, then $(1,x)\Box\neg$*checkOthersPassword* must also be labeled on C3. The reason is that the path obligations are enforced by the same strategy of the bank.

3.2.5 Passing Down the Path Obligations While Observing the Restrictions Among S-Profiles

The above observation points out how to judge whether a passing-down scheme of path obligations from a parent state to its child states are consistent with the strategy quantifications. For example, suppose that we have the following characteristics of the strategy profile $(1,x)(2,y)(2,z)(3,w)$.

- Strategy x of the banking system never issues action fail.
- Strategy y and z never cancel transactions for its client.
- Strategy w never shows busy message for the partner bank.

Then the verification problem of strategy interaction can be visualized as finding strategies x, y, z, and w that selecting paths in a computation tree to satisfy three path obligations: $\Box\neg$*checkOthersPassword*, $\Diamond$*depositDone*, and $\Diamond$*transferDone*. Figure **??** shows part of a computation tree when strategy x for the banking system is fixed with the following restriction.

- At location *idle*, *depDone*, and *xferDone*, always issue action $\bot$.
- At location *dep* and *xferCfm*, only issue action succ.

The leaves of the tree all end at location *idle* at which the same tree structure can be replicated. We can start synthesizing strategy y, z, and w in the context of this strategy x which, for convenience of explanation, is memoryless (or positional).

As we have suggested, we can label the path obligations on the nodes of the computation tree and examine whether a strategy profile can fulfill all the path obligations. In the literature, we can name the nodes by their branching paths. For example, the root is named ε which is the empty string. Then the node at location *xferDone* is named '100' and the rightmost *idle* node is named '12.' We begin by label node 'ε with the three path obligations: $(1, x)\Box\neg$*checkOthersPassword*, $(1, x)(2, y)\Diamond$*depositDone*, and $(1, x)(2, z)(3, w)\Diamond$*transferDone*. Then we can pass the path obligations to the children in the following way.

- We must pass $(1, x)\Box\neg$*checkOthersPassword* to both child '0' and '1' of the root since the two children are both selected by action $\bot$ of the banking system.
- We can pass $(1, x)(2, y)\Diamond$*depositDone* and $(1, x)(2, z)(3, w)\Diamond$*transferDone* respectively to child '0' and '1' of the root since the two children are selected with the same strategy x of the banking system and different strategies (y and z) of the client.

Note that we can check whether the passing down of the path obligation is consistent with the strategy quantifications in the input formula (G) just by checking the labeling of path obligations with strategy name bindings since all consistency information are maintained there.

Similarly, from node ‘0’, strategy x and y can together choose to pass $(1, x)(2, y)\Diamond$*depositDone* to node ‘00’ while strategy x must unilaterally pass $(1, x)\Box\neg$*checkOthersPassword* to both ‘00’ and ‘01.’ Then at node ‘00,’ $(1, x)(2, y)\Diamond$*depositDone* is fulfilled eventually and no longer need be passed down. The passing down of path obligations from node ‘1’ and fulfillment of $(1, x)(2, z)(3, w)\Diamond$*transferDone* can then be reasoned similarly.

3.2.6 Finding Finite Satisfying Evidence for a Formula in a Computation Tree

As we can see, the existence of the tree top in figure **??** is sufficient for synthsizing strategy profile $(1, x)(2, y)(2, z)(3, w)$ to satisfy formula (G). From this example, it is easy to see that the requirement for such a tree top. First, all eventual-formulas (or until-formulas) from the root SQ to the next level of SQs have to be fulfilled by the strategy profile. Second, the leaves of the tree top do not contain any eventual-formula (or until-formula) labels. Thus our PSPACE algorithm for model-checking BSIL formulas is actually a nondeterministic one that guess and check the consistency of the tree top and strategic actions at the nodes in the tree top.

Then the final question regarding our algorithms is the terminating condition for searching the tree tops and strategy profiles. Let us consider a tree node labeled with path obligations and how to (nondeterministically) explore for a tree top and a strategy profile. Note that the path obligations labeled on a child will be no more than those of its parent. In fact, the path obligations can stop being passed down in a path when it is fulfilled. Path obligations are simply passed down and not generated along the paths. As a result, along a path longer than the size of the game graph, either the number of path obligations decreases or two nodes, say v and v' (for convenience, we assume v is an ancestor of v'), with identical location and labels must appear. If the tree top is an evidence of existence of a strategy profile and the latter happens, then we can replace the the subtree rooted at v with the one rooted at v' and the result tree top is still an evidence. This observation implies that we can focus on tree top such that along any path, no duplication of nodes with the same location and the same path obligation labels. This implies that we only have to explore tree tops of depth no greater than the size of the game graph times the number of temporal modalities in the given BSIL formula.

3.3 Game Graphs

3.3.1 Concurrent Games

A *concurrent game* is played by many agents that make their moves concurrently. Such a game can be formalized with the following definition.

Definition 8. A concurrent game graph is a tuple $A = \langle m, Q, r, P, \lambda, R, \Delta, \delta \rangle$ with the following restrictions.

- m is the number of agents in the game.
- Q is a finite set of states.
- $r \in Q$ is the *initial state* of A.
- P is a finite set of atomic propositions.
- Function $\lambda : Q \mapsto 2^P$ labels each state in Q with a set of atomic propositions.
- $R \subseteq Q \times Q$ is the set of transitions.
- Δ is a set of tokens that can be issued by the agents during transitions.
- $\delta : (R \times [1, m]) \mapsto \Delta$ is a function that specifies the token (move symbol) issued by each agent in a transition.

During a transition, each agent selects a token. If there is no transition matching all the tokens selected by the agents, then there is no transition. Otherwise, the matching transition takes place. $\square$

In Figure **??**, we have the graphical representation of a concurrent game graph. The ovals represent states while the arcs represent state transitions. We also put down the λ values inside the corresponding states. On each edge, we label the tokens issued by the agents. Specifically, the label on arrow (q, q') is $[\delta((q, q'), 1), \ldots, \delta((q, q'), m)]$. For example, in Figure **??**, on edge (v, u), we have label $[a, b]$ meaning that to make the transition, agent 1 has to choose token a while agent 2 has to choose b.

For convenience, in the remaining part of the manuscript, we assume that we are always in the context of a given game graph $\mathcal{G} = \langle m, Q, r, P, \lambda, R, \Delta, \delta \rangle$. Thus, when we write $Q, r, P, \lambda, R, \Delta$, and δ we respectively refer to the corresponding components of this $\mathcal{G}$.

A *state predicate* of P is a Boolean combination of elements in P. The satisfaction of a state predicate η at a state q, in symbols $q \models \eta$, is defined in the standard way.

A *play* is an infinite path in a game graph. A play is *initial* if it begins with the initial state. Given a play $\rho = \bar{q}_0\bar{q}_1 \ldots$, for every $k \geq 0$, we let $\rho(k) = \bar{q}_k$. Also, given $h \leq k$, we let $\rho[h, k]$ denote $\rho(h) \ldots \rho(k)$ and $\rho[h, \infty)$ denote the infinite tail of ρ from $\rho(h)$. A *play prefix* is a finite segment of a play from the beginning of the play. Given a play prefix $\rho = \bar{q}_0\bar{q}_1 \ldots \bar{q}_n$, we use $|\rho| = n + 1$ for the *length* of ρ. For convenience, we use *last*(ρ) to denote the last state in ρ, i.e., $\rho(|\rho| - 1)$.

Let Q^* be the set of finite sequences of states in Q. For an agent $a \in [1, m]$, a *strategy* σ for a is a function from Q^* to Δ. An *agency* A of $[1, m]$ is an integer subset of $[1, m]$. For example, "$\{1, 3, 4\}$" represents the agency that consists of agents 1, 3, and 4. A *strategy profile* (or *S-profile*) Σ of an agency $A \subseteq [1, m]$ is a partial function from $[1, m]$ to the set of strategies such that, for every $a \in [1, m]$, $a \in A$ iff $\Sigma(a)$ is defined. The composition of two S-profiles Σ, Π, in symbols $\Sigma \circ \Pi$, is defined with the following restrictions for every $a \in [1, m]$.

- If $\Pi(a)$ is defined, then $\Sigma \circ \Pi(a) = \Pi(a)$.
- If $\Sigma(a)$ is defined and $\Pi(a)$ is undefined, then $\Sigma \circ \Pi(a) = \Sigma(a)$.
- If $\Sigma(a)$ and $\Pi(a)$ are both undefined, then $\Sigma \circ \Pi(a)$ is also undefined.

We will use composition of S-profiles to model inheritance of strategy bindings from ancestor formulas.

A play ρ is compatible with a strategy σ of an agent $a \in [1, m]$ iff for every $k \in [0, \infty)$, $\delta((\rho(k), \rho(k + 1)), a) = \sigma(\rho[0, k])$. The play is compatible with an S-profile Σ of agency A iff for every $a \in A$, the play is compatible with $\Sigma(a)$ of agent a.

3.3.2 Turn-Based Games

Another popular game structure is *turn-based game* in which at each state, at most one agent gets to decide the next state. For example, in figure **??**, we have the graphical representation of a turn-based game graph with initial state v. The ovals and squares represent states respectively of agent 1 and agent 2. The arcs represent state transitions.

In fact, every turn-based game can be represented as a special case of concurrent games. Specifically, a turn-based game $\mathcal{G} = \langle m, Q, r, P, \lambda, E, \Delta, \delta \rangle$ can be viewed as a concurrent game with the following restrictions.

- $\Delta = Q \cup \{\perp\}$ where $\perp$ denotes a dummy move not in Q.
- For every $(q, q') \in R$, if q belongs to agent a, then $\delta((q, q'), a) = q'$ and for every $a' \neq a$, $\delta((q, q'), a') = \perp$.
- For every $(q_1, q_2), (q_3, q_4) \in R$ and agent a, $\delta((q_1, q_2), a) = \perp$ if and only if $\delta((q_3, q_4), a) = \perp$.

  This restriction says that every state can be owned by only one agent.

For convenience, for a turn-based game, the owner of a state q, $\omega(q)$ in symbols, is defined as agent a with $\forall (q, q') \in E(\delta((q, q'), a) = q'$. For ease of notation, we denote with $Q_a = \{q \in Q \mid \omega(q) = a\}$ the states owned by an agent a.

In the investigation of many research issues, turn-based game graphs are easier to handle than concurrent game graphs. So in latter sections, we sometimes use turn-based game graphs in examples and explanation of the theory as we see fit.

3.4 BSIL

3.4.1 Syntax

For concurrent game graph $\mathcal{G}$ of m agents, we have three types of formulas: *state formulas*, *tree formulas*, and *path formulas*. State formulas describe properties of states. Tree formulas describe interaction of strategies. Path formulas describe properties of plays. BSIL formulas are constructed with the following three syntax rules, respectively, for state formulas ϕ, tree formulas τ, and path formulas θ.

$$\begin{aligned}
\phi &::= p \mid \neg\phi_1 \mid \phi_1 \vee \phi_2 \mid \langle A\rangle\tau \mid \langle A\rangle\theta \\
\tau &::= \tau_1 \vee \tau_2 \mid \tau_1 \wedge \tau_2 \mid \langle +A\rangle\tau_1 \mid \langle +A\rangle\theta \\
\theta &::= \neg\theta_1 \mid \theta_1 \vee \theta_2 \mid \bigcirc\phi_1 \mid \phi_1 \mathrm{U} \phi_2
\end{aligned}$$

Here, p is an atomic proposition in P and A is an agency of $[1,m]$. $\langle A\rangle$ is a *strategy quantifier* (*SQ*) and $\langle +A\rangle$ is a *strategy interaction quantifier* (*SIQ*). $\langle A\rangle\psi$ means that there exists an S-profile for the agency A that makes all plays satisfy ψ. Formulas of the form $\langle +B\rangle\psi_1$ must happen within an SQ. Intuitively, they mean that there exists an S-profile of B that collaborates with the strategies declared with ancestor formulas to make ψ_1 true. For convenience, we view SQs as special cases of SIQs. Also, for conciseness, we omit null SIQs $\langle +\rangle$.

State formulas ϕ are called *BSIL formulas*. From now on, we assume that we are always in the context of a given BSIL formula χ. Note that we strictly require that strategy interaction cannot cross path modal operators. This restriction is important and allows us to analyze the interaction of strategies locally in a state and then enforce the interaction along all paths from this state.

For convenience, we also use the common shorthands.

$$\mathit{true} \equiv p \vee (\neg p) \qquad \mathit{false} \equiv \neg\mathit{true} \qquad \psi_1 \Rightarrow \psi_2 \equiv (\neg\psi_1) \vee \psi_2$$

$$\Diamond\phi_1 \equiv \mathit{true}\ \mathrm{U}\phi_1 \qquad \Box\phi_1 \equiv \neg\Diamond\neg\phi_1$$

The SQs and SIQs introduced above are all existential in that they are satisfied with one S-profile. Note that there is no universal SQs and SIQs in BSIL. This is purely for the complexity of the model-checking algorithm (and problem) that we are going to present later. Thus BSIL can be used to specify the different ways of combining S-profiles to enforce system policy.

For ease of notations, we may abbreviate $\langle\{a_1,\ldots,a_n\}\rangle$ and $\langle +\{a_1,\ldots,a_n\}\rangle$ as $\langle a_1,\ldots,a_n\rangle$ and $\langle +a_1,\ldots,a_n\rangle$, respectively.

3.4.2 semantics

BSIL subformulas are interpreted with respect to S-profiles. A state or a tree formula ϕ is satisfied at a state q with S-profile Σ, denoted $\mathcal{G}, q \models_\Sigma \phi$, if, and only if, the following inductive constraints are satisfied.

- $\mathcal{G}, q \models_\Sigma p$ iff $p \in \lambda(q)$.
- For state formula ϕ_1, $\mathcal{G}, q \models_\Sigma \neg\phi_1$ iff $\mathcal{G}, q \models_\Sigma \phi_1$ is false.
- For state or tree formulas ψ_1 and ψ_2, $\mathcal{G}, q \models_\Sigma \psi_1 \wedge \psi_2$ iff $\mathcal{G}, q \models_\Sigma \psi_1$ and $\mathcal{G}, q \models_\Sigma \psi_2$.
- For state or tree formulas ψ_1 and ψ_2, $\mathcal{G}, q \models_\Sigma \psi_1 \vee \psi_2$ iff either $\mathcal{G}, q \models_\Sigma \psi_1$ or $\mathcal{G}, q \models_\Sigma \psi_2$.
- $\mathcal{G}, q \models_\Sigma \langle A \rangle \tau$ iff there exists an S-profile Π of A with $\mathcal{G}, q \models_\Pi \tau$.
- $\mathcal{G}, q \models_\Sigma \langle +A \rangle \tau$ iff there exists an S-profile Π of A with $\mathcal{G}, q \models_{\Sigma \circ \Pi} \tau$. Here, the composition $\Sigma \circ \Pi$ of the S-profiles Σ and Π models the inheritance of strategy bindings, Σ, from ancestor formulas.
- $\mathcal{G}, q \models_\Sigma \langle A \rangle \theta$ iff there exists an S-profile Π of A such that, for all plays ρ from q compatible with Π, $\rho \models_\Pi \theta$ holds. Intuitively, this means that ρ satisfies path formula θ with S-profile Π.
- $\mathcal{G}, q \models_\Sigma \langle +A \rangle \theta$ iff there exists an S-profile Π of A such that, for all plays ρ from q compatible with $\Sigma \circ \Pi$, $\rho \models_{\Sigma \circ \Pi} \theta$ holds.

A play ρ satisfies a path formula θ with S-profile Σ, in symbols $\rho \models_\Sigma \theta$, iff the following restrictions hold.

- For a path formula θ_1, $\rho \models_\Sigma \neg\theta_1$ iff it is not the case that $\rho \models_\Sigma \theta_1$.
- For path formulas θ_1 and θ_2, $\rho \models_\Sigma \theta_1 \vee \theta_2$ iff either $\rho \models_\Sigma \theta_1$ or $\rho \models_\Sigma \theta_2$.
- $\rho \models_\Sigma \bigcirc \psi_1$ iff $\mathcal{G}, \rho(1) \models_\Sigma \psi_1$.
- $\rho \models_\Sigma \psi_1 \mathrm{U} \psi_2$ iff there exists an $h \geq 0$ with $\mathcal{G}, \rho(h) \models_\Sigma \psi_2$ and for all $j \in [0, h)$, $\mathcal{G}, \rho(j) \models_\Sigma \psi_1$.

For convenience, we let $\perp$ be a null S-profile, i.e., a function that is undefined on everything. If ϕ_1 is a BSIL (state) formula and $\mathcal{G}, q \models_\perp \phi_1$, then we may simply write $\mathcal{G}, q \models \phi_1$. If $\mathcal{G}, r \models \phi_1$, then we also write $\mathcal{G} \models \phi_1$.

3.4.3 ATL+

ATL$^+$ is the syntactic fragment of BSIL given by the following grammar

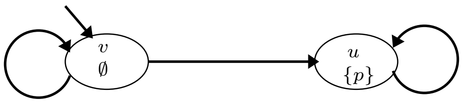


Figure 3.1: A simple turn-based game that demands memoryful strategies

$$\begin{aligned} \phi &::= p \mid \neg\phi_1 \mid \phi_1 \vee \phi_2 \mid \langle A \rangle \theta \\ \theta &::= \neg\theta_1 \mid \theta_1 \vee \theta_2 \mid \bigcirc \phi_1 \mid \phi_1 \mathrm{U} \phi_2 \end{aligned}$$

ATL$^+$ can also be viewed as an extension of ATL [**?**] that, similar to ...'s extension of CTL to CTL$^+$ [**?**, **?**], allows for the Boolean combination of path formulas. As such, all complexities of ATL$^+$ must reside between those of BSIL and ATL as well as between those of BSIL and CTL$^+$, which we will use to establish the lower bounds for ATL$^+$ and BSIL.

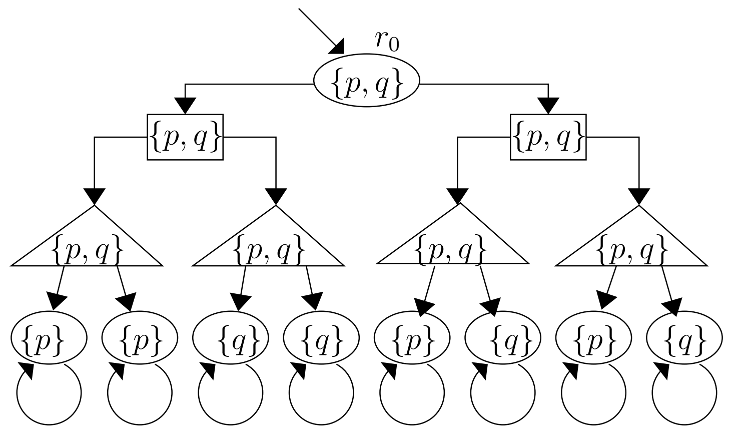
3.4.4 Memory

In this subsection, we show a simple example in which the agents need memory to achieve their objective for ATL$^+$ specifications. This is exemplified by the simplest possible case: the turn based game in Figure 3.1 with one agent, two states, one atomic proposition, and two memoryless strategies that do not count the unreached states in the histories. For the ATL$^+$ sentence $\langle 1 \rangle ((\neg \bigcirc p) \wedge \Diamond p)$, apparently agent 1 needs memory to enforce it.

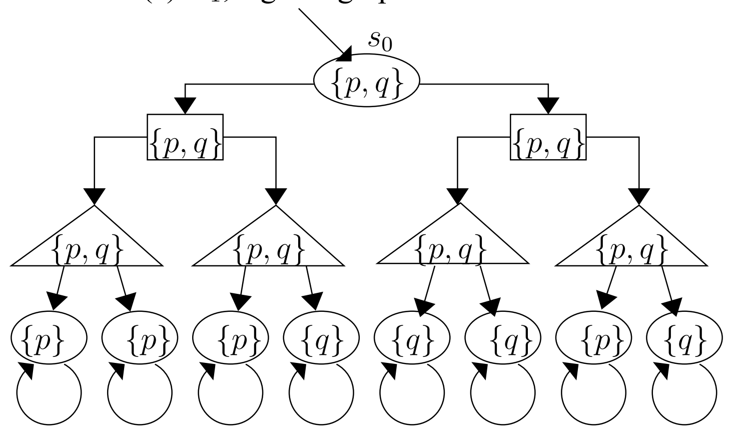
Lemma 18. The strategies of the agents in ATL$^+$ (and thus in BSIL) specifications need memory. This even holds for the single agent case. □

3.5 Expressive Power of BSIL

In this section, we establish that BSIL is incomparable with ATL*, AMC, and GL [**?**] in expressiveness. In fact, we shall first establish the incomparability between BSIL with GL and AMC. Then the imcomparability with BSIL follows since GL and AMC are superclasses of ATL* [**?**].



(a) G_1, a game graph for base case.



(b) H_1, another game graph for base case.
○ belongs to Agent 1; □ belongs to Agent 2; and △ belongs to Agent 3.

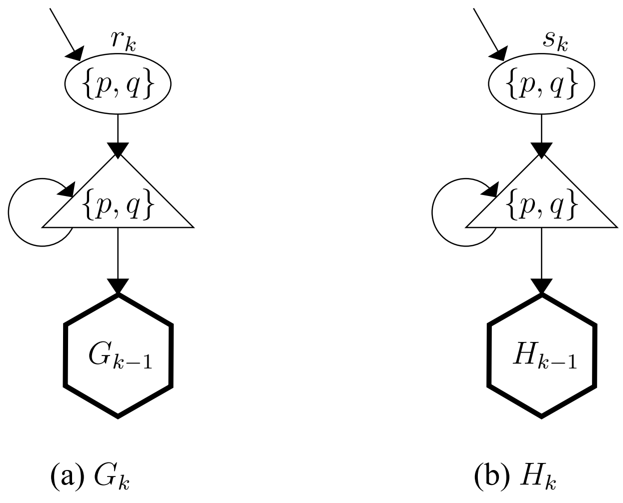
Figure 3.2: Base cases for the expressiveness of BSIL over ATL*

3.5.1 Comparison with GL

GL separates strategy quantifications from path quantifications. In comparison, ATL* and BSIL combines these two quantifications into SQs and SIQs. Thus with GL, we can specify that, for all S-profiles of A, there exists a play saitsfying ψ_1. The (existential) strategy quantification for agency A of GL is of the form $\exists A.\psi_1$. The path quantifications are just ordinary CTL modalities: $\forall\Box, \forall\mathrm{U}, \exists\Box$, and $\exists\mathrm{U}$.

The following two lemmas show the relation between GL and BSIL. Lemma 19 uses two inductive families of game graphs to show that GL is not as expressive as BSIL. The base cases, G_1 and H_1, are in figure 3.2 for three agents. The inductive cases, G_{k+1} and H_{k+1}, are respectively constructed out of G_k and H_k as in figure 3.3.

Lemma 19. Every GL formula ϕ with k ($k > 0$) SQs cannot distinguish G_k and H_k while



◯ belongs to Agent 1; □ belongs to Agent 2; and △ belongs to Agent 3.

Figure 3.3: Inductive cases for the expressiveness of BSIL over ATL*

$\langle 1\rangle((\langle +2\rangle\Box p) \wedge \langle +2\rangle\Box q)$ can.

Proof : It is clear that $\langle 1\rangle((\langle +2\rangle\Box p) \wedge (\langle +2\rangle\Box q))$ can G_k and H_k no matter what value k is. The proof continues by an induction k, the numbrer of SQs in ϕ.

Base case: Assume that ϕ has only one modal operator. Then there are the following case analysis of GL formulas.

- **Case 1:** *ϕ is $\exists A.\phi_1$ where ϕ_1 is a Boolean combination of formulas of the form $\exists\psi$ or $\forall\psi$ where ψ is a path formula.* Note that ϕ_1 characterizes a set of states that either start a play satisfying ψ or start only plays satisfying ψ. The following case analysis shows that, for every trace subset S and every agency A, there exists a strategy of A to characterize S in figure 3.2(a) iff there exists a strategy for A to characterize S in figure 3.2(b).
  - **Case 1a:** *ϕ is $\exists\emptyset.\phi_1$.* In this case, there is no strategy and the sets of traces imposed by no strategy in the two state graphs are both $\{\Box p, \Box q\}$. Thus, $\exists\emptyset.\phi_1$ cannot distinguish the trace sets of the two state graphs.
  - **Case 1b:** *ϕ is $\exists\{1\}.\phi_1$.* From figure 3.2, no matter what strategies agency $\{1\}$ may choose, the trace sets for the two game graphs are both $\{\Box p, \Box q\}$. Thus, $\exists\{1\}.\phi_1$ cannot distinguish the trace sets of the two state graphs.
  - **Case 1c:** *ϕ is $\exists\{2\}.\phi_1$.* This case is similar to case 1b.

- **Case 1d:** *ϕ is $\exists\{3\}.\phi_1$.* This case is similar to case 1b.
- **Case 1e:** *ϕ is $\exists\{1,2\}.\phi_1$.* Agency $\{1,2\}$ can cooperate to force three trace sets: $\{\Box p\}, \{\Box q\}, \{\Box p, \Box q\}$ in both of the game graphs in figure 3.2. Thus, $\exists\{1,2\}.\phi_1$ cannot distinguish the trace sets of the two state graphs.
- **Case 1f:** *ϕ is $\exists\{1,3\}.\phi_1$.* This case is similar to case 1e.
- **Case 1g:** *ϕ is $\exists\{2,3\}.\phi_1$.* This case is similar to case 1e.
- **Case 1h:** *ϕ is $\exists\{1,2,3\}.\phi_1$.* Agency $\{1,2,3\}$ can cooperate to force three trace sets: $\{\Box p\}, \{\Box q\}, \{\Box p, \Box q\}$ in both of the game graphs in figure 3.2. Thus, $\exists\{1,2,3\}.\phi_1$ cannot distinguish the trace sets of the two state graphs.

- **Case 2:** *ϕ is a Boolean combination of formulas in case 1.* Since case 1 does not distinguish the two game graphs, this case can neither do it.

Thus the base case is proven.

Induction step: If ϕ has k modal operators, to tell the difference between G_k and H_k, we need a modal subformula of ϕ that can tell the difference of G_{k-1} and H_{k-1}. But according to the inductive hypothesis, this is impossible. Thus the lemma is proven. $\Box$

Lemma 20. GL formula $\exists\{1\}.((\exists\Box p) \wedge \exists\Box q)$ is not equivalent to any BSIL formula.

Proof : The proof basically follows the same argument in [**?**] that $\exists\{1\}.((\exists\Box p) \wedge \exists\Box q)$ is not equivalent to any ATL* formula. $\Box$

Lemmas 19 and 20 together show that GL and BSIL are not comparable in expressiveness.

3.5.2 Comparison with AMC

AMC is an extension from μ-calculus and allows for multiple fixpoints interleaved together [**?**]. An AMC formula contains fixpoint operators on state set variables. The only modality of AMC is of the form $\langle A \rangle \bigcirc \psi$ and least fixpoint **lfp**$x.\psi_1(x)$ where $\psi_1(x)$ is a Boolean function of atomic propositions and state set variables (including x). It is required that every occurrence of x in ψ_1 is under even number of negations. The duality of the least fixpoint operator is the greatest fixpoint operator **gfp**. Formula **gfp**$x.\psi_1$ is defined as $\neg$**lfp**$x.\neg\psi_1(x)$.

To establish that AMC is not as expressive as BSIL, we basically follow the proof style for Lemma 26 and use the same two families of game graphs. The statement of the lemma requires notations for state set variables and other details in AMC. We need to define the domain of values for the free state set variables in AMC formulas. Let X be the set of state set variables. Without loss of generality, we assume that no two subformulas of the form **lfp**$x.\phi$ in a given AMC formula share the same quantified name of x. Given a subformula **lfp**$x_i.\phi$ with free variables $x_1, \ldots, x_n$ in ϕ and no modal operator $\langle \ldots \rangle \bigcirc$ in ϕ, we define ϕ as a *base template* for x_i. Then we can define the *base formula domain* of x_i, denoted $F_0(x_i)$, as the smallest set with the following restrictions. We let $\phi[x_1 \mapsto \eta_1, \ldots, x_n \mapsto \eta_n]$ be the AMC formula identical to ϕ except that every occurrence of x_i in ϕ are respectively replaced with η_i.

- For each $x_i \in X$ with base template ϕ, $\phi[x_1 \mapsto \mathit{false}, \ldots, x_n \mapsto \mathit{false}] \in F_0(x_i)$.
- For each $x_i \in X$ with base template ϕ and $\phi_1 \in F_0(x_1), \ldots, \phi_n \in F_0(x_n)$, $\phi[x_1 \mapsto \phi_1, \ldots, x_n \mapsto \phi_n] \in F_0(x_i)$.

Note that there is neither variables nor '**lfp**' operators in $F_0(x)$ for every x. Thus we can define the characterization κ of a formula ϕ_1 in $F_0(x)$ for $\mathcal{G}$, in symbols $\kappa(\mathcal{G}, \phi_1)$, as follows.

- For each atomic proposition $p \in P$, $\kappa(\mathcal{G}, p) = \{q \mid p \in \lambda(q)\}$.
- For each $\phi \in F_0(x)$, $\kappa(\mathcal{G}, \neg\phi) = Q - \kappa(\mathcal{G}, \phi)$.
- For each $\phi_1, \phi_2 \in F_0(x)$, $\kappa(\mathcal{G}, \phi_1 \vee \phi_2) = \kappa(\mathcal{G}, \phi_1) \cup \kappa(\mathcal{G}, \phi_2)$.

A *valuation* ν of variables in X for $\mathcal{G}$ is a mapping from X such that, for each $x \in X$, there exists a base domain formula ϕ of x such that $\nu(x) = \kappa(\mathcal{G}, \phi)$.

The expressiveness comparison between AMC and BSIL relies on the following lemma.

Lemma 21. For every AMC formula ϕ without modal operator of the form $\langle \ldots \rangle \bigcirc$, state set variable x, and a formula ϕ_1 in $F_0(x)$, $r_0 \in \kappa(G_0, \phi_1)$ iff $s_0 \in \kappa(H_0, \phi_1)$ in figure 3.2.

Proof : We can prove this with an structural induction on ϕ_1. The base case is straightforward. The inductive step follows since Boolean combinations of subformulas that cannot distinguish G_0 and H_0 can neither distinguish the two game graphs. $\square$

Now we want to classify AMC formulas according to the nesting depths of operators of the form $\langle \ldots \rangle \bigcirc$ in a formula. Specifically, we let AMC$^{(k)}$ be the set of AMC formulas with exactly nesting depth k of operator $\langle \ldots \rangle \bigcirc$. For example, **lfp**$x.\langle 1 \rangle \bigcirc (p \to$ **lfp**$y.\langle 2 \rangle (x \vee y \wedge \bigcirc q))$ is in AMC$^{(2)}$. Then, AMC$^{(0)}$ is the smallest set with the following restrictions.

- **Case 1a:** For each atomic proposition $p \in P$, $p \in$ AMC$^{(0)}$.
- **Case 1b:** For each proposition variable $x \in X$, $x \in$ AMC$^{(0)}$.
- **Case 1c:** For each $\phi \in$ AMC$^{(0)}$, $\neg\phi \in$ AMC$^{(0)}$.
- **Case 1d:** For each $\phi_1, \phi_2 \in$ AMC$^{(0)}$, $\phi_1 \vee \phi_2 \in$ AMC$^{(0)}$.
- **Case 1e:** For each $\phi \in$ AMC$^{(0)}$ and proposition variable $x \in X$, **lfp**$x.\phi \in$ AMC$^{(0)}$.

Then AMC$^{(k)}$, $k > 0$, is the smallest set with the following restrictions.

- **Case 2a:** For each $A \subseteq [1, m]$ and $\phi \in$ AMC$^{(k-1)}$, $\langle A \rangle \bigcirc \phi \in$ AMC$^{(k)}$,
- **Case 2b:** For each $\phi \in$ AMC$^{(k)}$, $\neg\phi \in$ AMC$^{(k)}$.
- **Case 2c:** For each $\phi_1 \in$ AMC$^{(k)}$ and $\phi_2 \in \bigcup_{h \leq k}$ AMC$^{(h)}$, $\phi_1 \vee \phi_2 \in$ AMC$^{(k)}$.
- **Case 2d:** For each $\phi_1 \in \bigcup_{h \leq k}$ AMC$^{(h)}$ and $\phi_2 \in$ AMC$^{(k)}$, $\phi_1 \vee \phi_2 \in$ AMC$^{(k)}$.
- **Case 2e:** For each $\phi \in$ AMC$^{(k)}$ and proposition variable $x \in X$, **lfp**$x.\phi \in$ AMC$^{(k)}$.

Note that there could be free variables in the formulas classified in the above. The evaluation of such formulas for a game graph depends on the valuation of the free variables.

Given two game graphs G, H and an AMC formulas ϕ, we say that two valuations of ν and ν' respectively of G and H are *consistent* if for every $x \in X$, there exists a $\phi_1 \in F_0(x)$ such that $\nu(x) = \kappa(G, \phi_1)$ and $\nu'(x) = \kappa(H, \phi_1)$. In the following, we adopt the AMC semantic notations in [**?**]. Given a game graph G, an AMC formula ϕ, and a valuation ν of state set variables in X, $(\phi)^G(\nu)$ denotes the set of states of G that satisfy ϕ with valuation ν.

Lemma 22. Assume that G_k and H_k are defined in figures 3.2 and 3.3. For every k, AMC formula $\phi \in$ AMC$^{(k)}$, and two consistent valuations ν and ν' respectively of G_k and H_k, $r_k \in (\phi)^{G_k}(\nu)$ iff $s_k \in (\phi)^{H_k}(\nu')$.

Proof : We use an induction on k to prove the lemma.

Base case: When ϕ is in case 1a through 1d, the lemma follows straightforwardly. In case 1e, (**lfp**$x.\psi)^{G_1}(\nu)$ can be expanded as follows. We let

- $\psi^{G_1,\nu,(0)}$ be $(\psi[x \mapsto \mathit{false}])^{G_1}(\nu)$ and
- for each $h > 0$, $\psi^{G_1,\nu,(h)}$ be $(\phi[x \mapsto \psi_1^{G_1,(h-1)}])(\nu)$.

Then $(\mathbf{lfp}x.\phi)^{G_1}(\nu) = \bigcup_{h\geq 0} \psi^{G_1,\nu,(h)}$ according to the semantics of AMC. Similarly, $(\mathbf{lfp}x.\phi)^{H_1}(\nu') = \bigcup_{h\geq 0} \psi^{H_1,\nu',(h)}$. According to the same argument for cases 1a through 1d, for each $h \geq 0, r_0 \in \psi^{G_1,\nu,(h)}$ iff $s_0 \in \psi^{H_1,\nu',(h)}$. Thus it is clear that $r_0 \in (\mathbf{lfp}x.\phi)^{G_1}(\nu)$ iff $s_0 \in (\mathbf{lfp}x.\phi)^{H_1}(\nu')$. Thus the lemma is proven in this case.

Induction step: To tell the difference between G_k and H_k, we need a formula with the following structure.

- At least one nesting of operators like $\langle\ldots\rangle\bigcirc$ in a least fixpoint operation to infer the reachability of G_{k-1} and H_{k-1}.
- A modal subformula nested inside a $\langle\ldots\rangle\bigcirc$ modal operator of ϕ that can tell the difference of G_{k-1} and H_{k-1}. But according to the inductive hypothesis, this is impossible.

Thus the lemma is proven. $\square$

Then with Lemma 22, we conclude the proof for Lemma 23 in the following.

Lemma 23. For every AMC formula ϕ, there are two game graphs that ϕ cannot distinguish while $\langle 1\rangle((\langle +2\rangle\Box p) \wedge \langle +2\rangle\Box q)$ can.

Proof : In the proof for Lemma 22, it is apparent that ϕ with $\phi \in \text{AMC}^{(k)}$ cannot tell G_k and H_k. $\square$

By the same argument in [**?**], for one-agent game, BSIL coincides with CTL and is not as expressive as AMC.

Lemma 24. For game graphs of one agent, AMC is strictly more expressive than BSIL.

Proof : For one-agent games, AMC is equivalent to μ-calculus and BSIL is equivalent to CTL which is strictly less expressive than μ-calculus. $\square$

A comment on Lemmas 19 and 23 is that the path modal formulas in the lemmas can be changed independently to $\Diamond\neg p$ and $\Diamond\neg q$ without affecting the validity of the lemma. This can be used to show that the example properties in the introduction are indeed inexpressible in ATL*, GL, and AMC.

3.5.3 Comparison with ATL*

It is easy to see that BSIL is a super-class of ATL. Thus we have the following lemma.

Lemma 25. BSIL is at least as expressive as ATL. $\square$

Then Lemmas 19 and 23 lead to the fact that there are some BSIL properties that ATL* cannot express since GL and AMC are both super-class of ATL* [**?**].

Lemma 26. For every ATL* formula ϕ, there are two game graphs that ϕ cannot distinguish while $\langle 1\rangle((\langle +2\rangle\square p) \wedge \langle +2\rangle\square q)$ can. $\square$

Lemmas 25 and 26 together establish that ATL is strictly less expressive than BSIL. Then the following lemma shows the reverse direction.

Lemma 27. ATL* formula $\langle 1\rangle\square\Diamond p$ is not equivalent to any BSIL formula.

Proof : The proof is similar to the proof for the inexpressibility of $\langle 1\rangle\square\Diamond p$ with ATL [**?**]. $\square$

Lemmas 26 and 27 together establish that ATL* and BSIL are not comparable in expressiveness.

3.6 Algorithm and Complexity

The model-checking problem of BSIL is contained in PSPACE mainly due to the restriction that disallows negation in tree formulas. As in the model-checking algorithms of ATL [**?**], we can evaluate the proper state subformulas independently and then treat them as auxiliary propositions. Moreover, as in the evaluation of $\Diamond$-formulas in ATL model-checking, if a $\Diamond$-formula can be enforced with an S-profile, it can be enforced in a finite number of steps along every play compatible with the strategy in a computation tree. Once a bound b for this finite number of steps is determined, we can enumerate all strategies embedded in the computation tree up to depth b and try to find one that enforces a BSIL formula.

As explained in section **??**, our algorithm also labels subformulas and their symbolic S-profiles on the nodes in a computation tree. The formula is satisfied if and only if we can find a finite tree top with labels consistent with the SQs and SIQs in the input formula and can be extended to an infinite computation tree. In Subsection 3.6.1, we derive the labels, i.e., subformulas and their symbolic S-profiles, that are sufficient for our model-checking algorithm and present two procedures:

- `localEval`() that checks whether the satisfaction of formulas at a state can be decided locally.
- `sucSet`() that nondeterministically chooses a scheme to pass down the subformulas and their symbolic S-profiles to the child nodes without violating the restrictions on the S-profiles declared with the SQs and SIQs.

There are, however, severe differences between exploring a computation tree for BSIL and exploring one for ATL [**?**]. For BSIL, we have to take the interaction of strategies into account. For example, we may have to enforce a subformula $\langle 1\rangle((\langle +2\rangle\Diamond p)\wedge\langle +2\rangle(\Box q\vee\Diamond r))$. Then, when exploring the computation tree, we may follow two strategies of Agent 2, one to enforce $\Diamond p$ and the other to enforce $\Box q$ or $\Diamond r$. There are the following situation for the interaction between these two strategies. The two strategies may make the same decision all the way until we reach a tree node v. (For turn-based games, v has to be owned by Agent 2.) This can be conceptualized as passing the obligations of $\Diamond p$ and $\Box q\vee\Diamond r$ along the path from the root to v. Then, at node v, the two strategies may differ in their decisions and pass down the two obligations to different branches.

Then, in Subsection 3.6.2, we present our algorithm in two parts, one for model-checking BSIL state formulas and the other for model-checking BSIL tree formulas. In Subsection 3.6.3, we prove the correctness of the algorithm. In Subsection 3.6.4, we show that our algorithm is in PSPACE. Together with Lemma 35 in Subsection 3.6.5, we then establish the PSPACE-completeness of the BSIL and ATL$^+$ model-checking problems.

3.6.1 Computing path obligations and passing them down the computation tree

We use $\{a_1\mapsto s_1,\ldots,a_n\mapsto s_n\}$ to denote a partial function that maps a_i to s_i for each $i\in[1,n]$. Given a partial function f, we denote the domain of f by $\mathit{def}(f)$. Inheriting the notations in [?], we may also represent the the mapping as $(a_1, s_1)(a_2, s_2)\dots(a_n, s_n)$.

We need some special techniques in checking tree formulas. We adopt the concept of strategy variables from [?,?]. A *strategy variable binding* (*SV-binding* for short) is a partial function from $[1, m]$ to strategy variables. Given an SV-binding Λ, $\Lambda \circ (a_1, s_1)\dots(a_n, s_n)$ is the SV-binding that is identical to Λ except that Agent a_i is bound to s_i for every $i \in [1, n]$.

Suppose that we are given SV-bindings $\Lambda_1, \dots, \Lambda_n$ and S-profiles $\Sigma_1, \dots, \Sigma_n$. We say that $\Lambda_1, \dots, \Lambda_n$ *matches* $\Sigma_1, \dots, \Sigma_n$ if, and only if, for every $a \in [1, m]$ and $i, j \in [1, n]$ with $a \in \mathit{def}(\Sigma_i) \cap \mathit{def}(\Sigma_j)$, $\Sigma_i(a) = \Sigma_j(a)$ if, and only if, $\Lambda_i(a) = \Lambda_j(a)$.

Given an SV-binding Λ and a state, tree, or path formula ψ, $\Lambda\psi$ is called a *bound formula*. $\Lambda\psi$ is a *bound path obligation* (*BP-constraint*) if ψ is a Boolean combination of path formulas. A Boolean combination of BP-obligations is called a *Boolean bound formula* (*BB-formula*). The strategy variables in BB-formula are only used to tell whether or not two path properties are to be enforced with the same strategy. For example, the property $\langle 1\rangle((\langle +2\rangle\Diamond p) \wedge \langle +2\rangle(\Box q \vee \Diamond r))$ can be rewritten as BB-formula $((1, s_1)(2, s_2)\Diamond p) \wedge (1, s_1)(2, s_3)\Box q \vee \Diamond r$, which says that Agent 1 must use the same strategy to fulfill both $\Diamond p$ and $\Box q \vee \Diamond r$, while Agent 2 may use different strategies to fulfill these two path properties.

Suppose we are given a function π that maps symbolic strategy names to strategies. Similar to the semantics of strategy logics [?] with strategy variables, we can also define the satisfaction of BB-formulas $\Lambda\psi$ at a state q with π, in symbols $\mathcal{G}, q \models^{\pi} \Lambda\psi$, as follows.

- $\mathcal{G}, q \models^{\pi} \Lambda_1\psi_1 \vee \Lambda_2\psi_2$ iff $\mathcal{G}, q \models^{\pi} \Lambda_1\phi_1$ or $\mathcal{G}, q \models^{\pi} \Lambda_2\phi_2$ holds.
- $\mathcal{G}, q \models^{\pi} \Lambda_1\phi_1 \wedge \Lambda_2\phi_2$ iff both $\mathcal{G}, q \models^{\pi} \Lambda_1\phi_1$ and $\mathcal{G}, q \models^{\pi} \Lambda_2\phi_2$ hold.
- Given an SV-binding Λ and a path formula ψ_1 with an S-profile $\Sigma = \{a \mapsto \pi(\Lambda(a)) \mid a \in \mathit{def}(\Lambda)\}$, $\mathcal{G}, q \models^{\pi} \Lambda\psi_1$ iff, for all plays ρ compatible with Σ from q, $\rho \models_{\Sigma} \psi_1$ holds.

In Table 3.1, we present equivalence rules to rewrite state, tree, and path formulas to BB-formulas using the procedure *bf*(). For convenience, we use a procedure *newVar*() that returns a strategy variable that has not been used before. In general, the semantics of

Table 3.1: Rewriting rules for BB-formulas

$\mathit{bf}(\Lambda\neg\neg\phi)$	$\equiv$	$\mathit{bf}(\Lambda\phi)$
$\mathit{bf}(\Lambda(\tau_1 \vee \tau_2))$	$\equiv$	$\mathit{bf}(\Lambda\tau_1) \vee \mathit{bf}(\Lambda\tau_2)$
$\mathit{bf}(\Lambda(\tau_1 \wedge \tau_2))$	$\equiv$	$\mathit{bf}(\Lambda\tau_1) \wedge \mathit{bf}(\Lambda\tau_2)$
$\mathit{bf}(\Lambda\langle a_1, \ldots, a_n\rangle\psi)$	$\equiv$	$\mathit{bf}(\{a_1 \mapsto \mathit{newVar}(), \ldots, a_n \mapsto \mathit{newVar}()\}\psi)$
$\mathit{bf}(\Lambda\langle +a_1, \ldots, a_n\rangle\psi)$	$\equiv$	$\mathit{bf}(\Lambda \circ \{a_1 \mapsto \mathit{newVar}(), \ldots, a_n \mapsto \mathit{newVar}()\}\psi)$
$\mathit{bf}(\Lambda \bigcirc \phi_1)$	$\equiv$	$\Lambda \bigcirc \mathit{bf}(\emptyset\phi_1)$
$\mathit{bf}(\Lambda\neg \bigcirc \phi_1)$	$\equiv$	$\Lambda \bigcirc \mathit{bf}(\emptyset\neg\phi_1)$
$\mathit{bf}(\Lambda\phi_1 \mathrm{U} \phi_2)$	$\equiv$	$\Lambda \mathit{bf}(\emptyset\phi_1)\mathrm{U}\mathit{bf}(\emptyset\phi_2)$
$\mathit{bf}(\Lambda\neg\phi_1 \mathrm{U} \phi_2)$	$\equiv$	$\Lambda((\mathit{bf}(\emptyset\phi_1)\mathrm{U}\mathit{bf}(\neg\emptyset(\phi_1 \vee \phi_2))) \vee \Box\mathit{bf}(\emptyset\neg\phi_2))$
$\mathit{bf}(\Lambda p) \equiv p$	;	$\mathit{bf}(\Lambda\neg p) \equiv \neg p$
$\mathit{bf}(\Lambda \mathit{true}) \equiv \mathit{true}$	;	$\mathit{bf}(\Lambda\neg \mathit{true}) \equiv \mathit{false}$
$\mathit{bf}(\Lambda \mathit{false}) \equiv \mathit{false}$	;	$\mathit{bf}(\Lambda\neg \mathit{false}) \equiv \mathit{true}$

ϕ_1, ϕ_2: state or path formulas. τ_1, τ_2: tree formulas. ψ_1, ψ_2: tree or path formulas.

BSIL deals with the satisfaction of a set of subformulas bound to different S-profiles. The following two lemmas relate the rules in Table 3.1 with the semantics of BSIL formulas.

Lemma 28. Suppose we are given a state q, a BSIL subformula ψ, and a S-profile Σ such that $\mathcal{G}, q \models_\Sigma \psi$. There exist an SV-binding Λ and a function π such that $\mathcal{G}, q \models^\pi \mathit{bf}(\Lambda\psi)$.

Proof : We construct Λ and π as follows. Without loss of generality, we assume ψ is unique in the input formula. For every $a \in \mathit{def}(\Sigma)$, we let $\Lambda(a) = s_a^\psi$ and $\pi(s_a^\psi) = \Sigma(a)$. It is clear that the functional composition of Λ and π is actually Σ. Then, according to the semantics of $\mathcal{G}, q \models^\pi \mathit{bf}(\Lambda\psi)$ presented in the above, $\mathcal{G}, q \models^\pi \mathit{bf}(\Lambda\psi)$ since for all play ρ compatible with Σ from q, $\rho \models_\Sigma \psi$. Thus the lemma is proven. $\Box$

Lemma 29. Suppose we are given a state q, a BSIL subformula ψ, a SV-bindings Λ, and a function π such that $\mathcal{G}, q \models^\pi \mathit{bf}(\Lambda\psi)$. Then there exist an S-profile Σ such that $\mathcal{G}, q \models_\Sigma \psi$.

Proof : We can construct Σ by defining, for all $a \in \mathit{def}(\Lambda)$, $\Sigma(a) = \pi(\Lambda(a))$. Thus, Σ is the functional composition of Λ and π. Then according to the semantics of $\mathcal{G}, q \models^\pi \mathit{bf}(\Lambda\psi)$ presented in the above, $\mathcal{G}, q \models^\pi \mathit{bf}(\Lambda\psi)$ implies $\mathcal{G}, q \models_\Sigma \psi$. Thus the lemma is proven. $\Box$

To ease the presentation of our algorithms, we also assume that there is a procedure that rewrites a BB-formula to an equivalent BB-formula in disjunctive normal form. Specifically, a *disjunctive normal BB-formula* (*DNBB-formula*) is the disjunction of conjunctions of BP-obligations. The rewriting of a BB-formula ϕ to a DNBB-formula can be done by repeatedly applying the distribution law of conjunctions of disjunctions until a DNBB-formula is obtained.

Example 30. DNBB-formula rewriting: We have the following rewriting process for a BSIL formula for five agents.

$$\begin{aligned}
& \mathit{bf}(\emptyset\langle 1,2\rangle\,(\langle +3\rangle(\Box p \vee \Diamond q) \wedge \langle +3\rangle(\langle +2\rangle\Diamond r \vee \langle +4\rangle\Box q))) \\
\equiv\ & \mathit{bf}((1,s_1)(2,s_2)\,(\langle +3\rangle(\Box p \vee \Diamond q) \wedge \langle +3\rangle(\langle +2\rangle\Diamond r \vee \langle +4\rangle\Box q))) \\
\equiv\ & \mathit{bf}((1,s_1)(2,s_2)\langle +3\rangle(\Box p \vee \Diamond q)) \wedge \mathit{bf}((1,s_1)(2,s_2)\langle +3\rangle(\langle +2\rangle\Diamond r \vee \langle +4\rangle\Box q)) \\
\equiv\ & \mathit{bf}((1,s_1)(2,s_2)(3,s_3)(\Box p \vee \Diamond q)) \wedge \mathit{bf}((1,s_1)(2,s_2)(3,s_4)(\langle +2\rangle\Diamond r \vee \langle +4\rangle\Box q)) \\
\equiv\ & (1,s_1)(2,s_2)(3,s_3)(\Box p \vee \Diamond q) \\
& \wedge ((1,s_1)(2,s_5)(3,s_4)\Diamond r \vee (1,s_1)(2,s_2)(3,s_4)(4,s_6)\Box q) \\
\equiv\ & ((1,s_1)(2,s_2)(3,s_3)(\Box p \vee \Diamond q) \wedge (1,s_1)(2,s_5)(3,s_4)\Diamond r) \\
& \vee ((1,s_1)(2,s_2)(3,s_3)(\Box p \vee \Diamond q) \wedge (1,s_1)(2,s_2)(3,s_4)(4,s_6)\Box q)
\end{aligned}$$

This DNBB-formula sheds some light on the analysis of BSIL formulas. As can be seen, the formula is satisfied iff one of the two outermost disjuncts is satisfied. Without loss of generality, we examine the first disjunct:

$$\eta_1 \equiv (1,s_1)(2,s_2)(3,s_3)(\Box p \vee \Diamond q) \wedge (1,s_1)(2,s_5)(3,s_4)\Diamond r$$

There are the following two S-profiles involved in the satisfaction of the formula.

- Σ_1 for $(1,s_1)(2,s_2)(3,s_3)$ of $\{1,2,3\}$ used to satisfy $\Box p \vee \Diamond q$.
- Σ_2 for $(1,s_1)(2,s_5)(3,s_4)$ of $\{1,2,3\}$ used to satisfy $\Diamond r$.

This disjunct imposes the restrictions that Σ_1 and Σ_2 must agree in their moves by agent 1. (Or for turn-based games, they must agree in their choices at nodes owned by Agent 1.) Similarly, we can examine

$$\eta_2 \equiv (1,s_1)(2,s_2)(3,s_3)(\Box p \vee \Diamond q) \wedge (1,s_1)(2,s_2)(3,s_4)(4,s_6)\Box q$$

There is a new S-profile introduced.

- Σ_3 for $(1,s_1)(2,s_2)(3,s_4)(4,s_6)$ of $\{1,2,3,4\}$ used to satisfy $\Box q$.

This disjunct imposes the restrictions that Σ_1 and Σ_3 must agree in their moves by Agents 1 and 2. In the following, we use the observation in this example to construct structures from DNBB-formulas for the model-checking of conjunctive DNBB-formulas. $\Box$

For the ease of notation, we represent a conjunctive DNBB-formula η as a set of BP-obligations in our algorithms. Our goal is to design a computation tree exploration procedure that given a set C of BP-obligations, labels each node in the tree with a subset of C for the set of path formulas that some S-profiles have to enforce without violating the restrictions of strategy interaction imposed in C through the strategy variables. In the design of the procedure, one central component is how to label the children of a node with appropriate sets of BP-obligations as inherited path obligations from C. We need two basic procedures for this purpose. The first is to evaluate the truth values of path literals in BP-obligations with a proposition interpretation when possible. Specifically, when we can deduce the truth values of U-formulas and $\Box$-formulas from the truth values of propositions (or state subformulas) at a state, the procedure changes the respective U-formula and $\Box$-formula to their respective truth values. The procedure is as follows.

```
localEval($W, \theta$) // $\lambda()$ has been extended with satisfied state subformulas at each state.
1:  **switch** ($\theta$)
2:  **case** *true* or *false*: **return** $\theta$
3:  **case** $p$: **if** $p \in W$ **then return** *true* **else return** *false* **end if**
4:  **case** $\neg p$: **if** $p \in W$ **then return** *false* **else return** *true* **end if**
5:  **case** $\theta_1 \vee \theta_2$:
6:      Let $\theta_1$ be localEval($W, \theta_1$) and $\theta_2$ be localEval($W, \theta_2$).
7:      **if** $\theta_1$ is *true* or $\theta_2$ is *true* **then return** *true*.
8:      **else if** $\theta_1$ is *false* **then return** $\theta_2$. **else if** $\theta_2$ is *false* **then return** $\theta_1$. **else return**
        $\theta_1 \vee \theta_2$.
9:      **end if**
10: **case** $\theta_1 \wedge \theta_2$:
11:     Let $\theta_1$ be localEval($W, \theta_1$) and $\theta_2$ be localEval($W, \theta_2$).
12:     **if** $\theta_1$ is *false* or $\theta_2$ is *false* **then return** *false*.
13:     **else if** $\theta_1$ is *true* **then return** $\theta_2$. **else if** $\theta_2$ is *true* **then return** $\theta_1$. **else return**
        $\theta_1 \wedge \theta_2$.
```

```
14:     end if
15:  case $\bigcirc\phi_1$:  return $\theta$
16:  case $\Box\phi_1$:  if $\phi_1 \notin W$ then  return false else return $\theta$ end if
17:  case $\phi_1 \mathrm{U} \phi_2$:  if $\phi_2 \in W$ then  return true else if $\phi_1 \notin P$ then return false else
     return $\theta$ end if
18:  end switch
```

Statements 17 checks if $\Lambda\theta$ is fulfilled. When it is fulfilled, θ is changed to *true*. Statements 17 and 16 also check if $\Lambda\theta$ is violated. When a violation happens, θ is changed to *false*.

Then we need a procedure, `next()`, that calculates the BP-obligations passed down from a previous state. This is simply done by replacing every $\bigcirc\phi$ by ϕ in the BP-obligations.

With the two basic procedures defined above, we now present a procedure that nondeterministically calculates sets of BP-obligations passed down to the successor states. This is accomplished with the procedure `sucSet`(q, C) in the following. Given a node q in the computation tree and a set C, the procedure nondeterministically returns an assignment of BP-obligations to children of q to enforce the BP-obligations in C without violating the strategy interaction of BP-obligations.

```
sucSet$(q, C)$ // $\lambda()$ has been extended with satisfied state subformulas at each state.

1:  Convert $C$ to $\{\Lambda\mathtt{localEval}(\lambda(q), \theta) \mid \Lambda\theta \in C\}$.
2:  if $\Lambda$false $\in C$ then  return $\emptyset$ end if
3:  Let $S$ be the set of all symbolic strategy variables in $C$. That is, $S = \{s \mid a \mapsto s \in
    \Lambda, \Lambda\theta \in C\}$.
4:  Nondeterministically pick an $\alpha_s \in \Delta$ for each $s \in S$.
5:  Let $K$ be $\{(q', \emptyset) \mid (q, q') \in R\}$.
6:  for each $\Lambda\theta \in C$ do
7:      if for all $(q, q')$, there is an $a \in def(\Lambda)$ with $\delta((q, q'), a) \neq s_{\Lambda(a)}$ then  return $\emptyset$ end
        if
```

```
8:     for $(q', C') \in \Delta$ with $\forall a \in \mathit{def}(\Lambda)(\delta((q, q'), a) = s_{\Lambda(a)})$ do
9:         Replace $(q', C')$ with $(q', C' \cup \{\Lambda\texttt{next}(\theta)\})$ in $K$.
10:    end for
11: end for
12: return $K$.
```

The nondeterministic choices at statement 4 make sure that one symbolic strategy variable is mapped to exactly one move. The loop at statement 6 iterates through all the path obligations at the current node and passes them down to the children if necessary. The if-statement at line 7 checks whether all obligations can be passed down to some children. If some obligations are not passed due to mismatch between moves of the strategies and the labels on the transitions, then the we return with failure. Otherwise, statement 8 passes the obligations to all children with matching transition labels. The obligations to children are recorded in K which is returned with success at statement 12.

3.6.2 Procedures for checking BSIL properties

The procedure in the following checks a BSIL state property ϕ at a state q of A.

```
checkBSIL($q, \phi$)
1: if $\phi$ is $p$ then if $\phi \in \lambda(q)$ then return true. else return false. end if
2: else if $\phi$ is $\phi_1 \vee \phi_2$ then return checkBSIL($q, \phi_1$) $\vee$ checkBSIL($q, \phi_2$)
3: else if $\phi$ is $\neg\phi_1$ then return $\neg$checkBSIL($q, \phi_1$)
4: else if $\phi$ is $\langle A \rangle \tau$ for a tree or path formula $\tau$ then return checkTree($q, \langle A \rangle \tau$)
5: end if
```

The procedure is straightforward and works inductively on the structure of the input formula. For convenience, we need procedure checkSetOfBSIL($Q'\phi_1$) in the following that checks a BSIL property ϕ_1 at each state in Q'.

```
checkSetOfBSIL($Q', \phi_1$)
1: if $\phi_1 \notin P \cup \{\mathit{true}, \mathit{false}\}$ then
2:     for each $q' \in Q'$ do
3:         if checkBSIL($q', \phi_1$) then Let $\lambda(q')$ be $(\lambda(q') \cup \{\phi_1\}) - \{\neg\phi_1\}$.
```

```
4:         else Let $\lambda(q')$ be $(\lambda(q') - \{\phi_1\}) \cup \{\neg\phi_1\}$. end if
5:     end for
6: end if
```

Then we use procedure `checkTree`$(q, \langle A\rangle\tau)$ in the following to check if a state q satisfies $\langle A\rangle\tau$.

```
checkTree$(q, \langle A\rangle\tau)$
1: Rewrite bf$(\emptyset\langle A\rangle\tau)$ to DNBB-formula $\eta_1 \vee \ldots \vee \eta_n$.
2: for $i \in [1, n]$ do
3:     Represent $\eta_i$ as a set $C$ of BP-obligations.
4:     for each $\Lambda\theta$ in $C$. do
5:         if $\theta$ is $\bigcirc\phi_1$ then checkSetOfBSIL$(\{q' \mid (q, q') \in \mathcal{R}\}, \phi_1)$.
6:         else if $\theta$ is $\phi_1 \mathrm{U} \phi_2$ then checkSetOfBSIL$(Q, \phi_1)$; checkSetOfBSIL$(Q, \phi_2)$;
           end if
7:     end for
8:     if recTree$(q, C)$ then return true. end if
9: end for
10: return false.
```

We first rewrite $\langle A\rangle\tau$ to its DNBB-formula at statement 1 by calling *bf*$(\emptyset\langle A\rangle\tau)$ and using the distribution law of conjunctions over disjunctions. (In practice, to contain the complexity in PSPACE, we only need to enumerate the disjuncts of the DNBB-formula in PSPACE.) We then iteratively check with the loop starting from statement 2 if $\langle A\rangle\tau$ is satisfied due to one of its conjunctive DNBB-formula components of $\langle A\rangle\tau$. At statement 3, we construct the set C of BP-obligations of the component. We evaluate the subformulas with the inner loop starting at statement 4. Finally at statement 8, we explore the computation tree, with procedure `recTree`(q, C) in the following, and pass down the path obligations to the children according to the restrictions of the SV-binding in C.

```
recTree$(q, C)$
1: if $(q, C)$ coincides with an ancestor in the exploration then
2:     if there is no $\Lambda\phi_1 \mathrm{U} \phi_2$ in $C$ then return true; else return false. end if
```

```
3: end if
4: if sucSet($q, C$) is empty then return false end if
5: for each $(q', C') \in$ sucSet($q, C$) with $C' \neq \emptyset$ do
6:     if recTree($q', C'$) is false then return false. end if
7: end for
8: return true.
```

Note that procedure `recTree`(q, C) is nondeterministic since it employs `sucSet`(q, C) to nondeterministically calculate an assignment Δ of path obligations to the children of q.

3.6.3 Correctness proof of the algorithm

In order to prove the correctness of this algorithm, we define *obligation distribution trees* (*OD-trees*) in the following. An OD-tree for a set C of BP-obligations and game graph $\mathcal{G}$ from a state $q_0 \in Q$ is a labeled computation tree $\langle V, \bar{r}, \alpha, E, \beta \rangle$ with the following restrictions.

- V is the set of nodes in the tree.
- $\bar{r} \in V$ is the root of the tree.
- $\alpha : V \mapsto Q$ labels each tree node with a state. Also $\alpha(\bar{r}) = q_0$.
- $E \subseteq V \times V$ is the set of arcs of the tree such that, for each $(q, q') \in R$, there exists an $(v, v') \in E$ with $\alpha(v) = q$ and $\alpha(v') = q'$.
- $\beta : V \mapsto 2^C$ labels each node with a subset of C for path formulas in χ that need to be fulfilled at a node. Moreover, we have the following restrictions on β.
  - $C = \beta(r)$.
  - For every $v \in V$, there exists a $\Delta =$ `sucSet`$(\alpha(v), \beta(v))$ such that, for every $(q', C') \in \Delta$, there exists a $(v, v') \in E$ with $\alpha(v') = q'$ and $\beta(v') = C'$.

The OD-tree is *fulfilled* iff, for every path $v_0 v_1 \dots v_k \dots$ along the tree from the root, there exists an $h \geq 0$ such that, for every $j \geq h$, there is no $\Lambda\phi_1 \mathrm{U} \phi_2 \in \beta(v_j)$. We have the following connection between an OD-tree and an execution of procedure `recTree`(q, C) from the root of an OD-tree.

Lemma 31. For a set C of BP-obligations, $\texttt{recTree}(q, C)$ returns true iff there exists a fulfilled OD-tree for C and $\mathcal{G}$ from q.

Proof : In order to prove the lemma, we show both directions.

$(\Rightarrow)$: It is straightforward to see that $\texttt{recTree}(q, C)$ returns true only if a finite tree has been constructed with leafs duplicating their ancestors. According to statement 2 of $\texttt{recTree}(q, C)$, it is clear that along the path from that ancestor to a leaf, no node is labeled with a BP-obligation of the form $\Lambda\phi_1 \mathrm{U} \phi_2$ by β. Thus we can extend the leaves by duplicating the subtree rooted at their duplicating ancestors. In this way, we can extend the finite tree to a fulfilled OD-tree.

$(\Leftarrow)$: Suppose there exists a fulfilled OD-tree for C and $\mathcal{G}$ from q. Since all infinite paths from the root stabilize to suffices without index labels by β for until-formulas (as the tree is finitely branching, it would otherwise contain an infinite path with standing untility by Köngs lemma), we can repeatedly replace every subtree T with a subtree T' of T such that the root of T and T' have the same α and β labels. We can repeat this replacement until no node labeled with an until-formula has the same α and β labels as one of its descendants. The existence of such an OD-tree after the replacements implies that $\texttt{recTree}(q, C)$ eventually explores such a tree, finds the termination condition at all leafs, and returns *true*. $\square$

Lemma 32. Given a conjunctive DNBB-formula η represented as a set C of BP-obligations, there exists a function π on strategy variables in η with $\mathcal{G}, q \models^{\pi} \eta$ iff there exists a fulfilled OD-tree for $\mathcal{G}$ and C from q.

Proof : The lemma can also be proven in two directions. In the forward direction, we can use π to construct S-profiles to enforce η. The S-profiles can then be used to construct a fulfilled OD-tree for $\mathcal{G}$ and C from q.

In the backward direction, we can follow the paths and obligations that are passed-down in the OD-tree and construct S-profiles that enforce η. Then, from these S-profiles, due to the one-to-one correspondence between the strategy variables and the strategies in the S-profiles, we can define a π with $\mathcal{G}, q \models^{\pi} \eta$. $\square$

The correctness of procedure $\texttt{recTree}(q, C)$ then directly follows from Lemmas 31 and 32. Then the correctness of procedure checkBSIL(q, ϕ) follows by a structural induction on a given BSIL formula and the correctness of procedure recTree(q, C).

Lemma 33. Given a state q in $\mathcal{G}$, checkBSIL(q, χ) iff $\mathcal{G}, q \models_{\perp} \chi$. $\square$

3.6.4 Complexities of the algorithm

The algorithm that we presented in Subsections 3.6.1 and 3.6.2 can run in PSPACE mainly because we can enumerate the conjunts in a DNF in PSPACE and can implement procedure recTree(q, C) with a stack of polynomial height. To see this, please recall that we use the procedure sucSet(q, C) to calculate the assignment of BP-obligations to the children to q in the computation tree. Specifically, procedure sucSet(q, C) nondeterministically returns a set Δ with elements of the form (q', C') such that $(q, q') \in R$ and $C' \subseteq C$ since in procedure sucSet(q, C), a path obligation $\Lambda\theta$ is passed down to a child and recorded in the corresponding C' only when it matches the for loop condition at statement 8. Thus, along any path in the OD-tree, the sets of literal bounds never increase. Moreover, when there is a node in the exploration of OD-tree that coincides with an ancestor, we backtrack in the exploration. This implies that, along any path segment longer than $|Q|$, one of the following two conditions hold.

- A backtracking happens at the end of the segment.
- The sets of BP-obligations along the segment must decrease in size at least once.

These conditions lead to the observation that, with procedure sucSet(q, C), the recursive exploration of a path can grow no longer than $1 + |C| \cdot |Q|$. This leads to the following lemma.

Lemma 34. The BSIL model-checking algorithm in Subsections 3.6.1 and 3.6.2 is in PSPACE.

Proof : For convenience, we let $\#(\chi)$ be the number of modal formulas in χ. Following the argument from above, it is straightforward to check that to explore an OD-tree, we only need a stack of at most $1 + \#(\chi) \cdot |Q|$ frames. In each frame, we only need to record a state in Q, a subset of $[1, \#(\chi)]$ for the path obligations, and a Δ returned from procedure sucSet(q, C). Procedure sucSet(q, C) can be nondeterministically computed by randomly assigning the obligations in C to the successors of q and check if the assignment satisfies strategy interaction restriction of the BP-obligations. Procedure $\texttt{checkBSIL}(q, \phi)$ can then be executed in space cubic in the size of ϕ for the Δ's at nodes along the path. Thus, we conclude that the algorithm is a PSPACE algorithm. $\square$

A rough analysis of the time complexity of our algorithm follows. Let $|\chi|$ be the length of a BSIL formula χ. At each call to $\texttt{sucSet}()$, the size of C is at most $|\chi|$. The number of root-to-leaf paths in an OD-tree is at most $|\chi|$ since we only have to pass down $|\chi|$ BP-obligations. We can use the positions of the common ancestors of the leaves of such paths to analyze the number of the configurations of such OD-trees. The common ancestors can happen anywhere along the root-to-leaf paths. Thus, there are $(1 + |\chi| \cdot |Q|)^{|\chi|}$ ways to arrange the positions of the common ancestors since the length of paths are at most $1 + |\chi| \cdot |Q|$. The number of ways that the BP-obligations can be assigned to the leaves is at most $|\chi|^{|\chi|}$. The number of state labeling of the nodes on the paths is at most $|Q|^{|\chi|\cdot(1+|\chi|\cdot|Q|)}$. Thus, given a C, the total number of different OD-trees is in $O(|Q|^{|\chi|\cdot(1+|\chi|\cdot|Q|)}|\chi|^{|\chi|}(1 + |\chi| \cdot |Q|)^{|\chi|}) = O(|Q|^{|\chi|\cdot(2+|\chi|\cdot|Q|)}|\chi|^{2|\chi|})$. There are $O(2^{|\chi|})$ different possible values of C. There are at most $|\chi|$ OD-trees to construct for the model-checking task. Thus, the total time complexity of our algorithm is in $O(|\chi| 2^{|\chi|}|Q|^{|\chi|\cdot(2+|\chi|\cdot|Q|)}|\chi|^{2|\chi|})$.

3.6.5 Lower bound and completeness

We close by establishing the PSPACE lower bound for ATL$^+$ model-checking, and hence the PSPACE-completeness of ATL$^+$ and BSIL model-checking. This is done by reduction from the prenex QBF (quantified Boolean formula) satisfiability problem [**?**] to an ATL$^+$ model-checking problem for a 2-agent game graph, where both the game graph and the ATL$^+$ specification are linear in the prenex QBF problem we reduce from. We assume a QBF property $\eta \equiv \bigtriangledown_1 p_1 \ldots \bigtriangledown_l p_l (C_1 \wedge C_2 \wedge \ldots \wedge C_n)$ with a set $P = \{p_1, \ldots, p_l\}$ of atomic propositions and the following restrictions.

- For each $k \in [1, l]$, $\bigtriangledown_k$ is either $\exists$ or $\forall$.
- For each $k \in [1, n]$, C_k is a clause $l_{k,1} \vee \ldots \vee l_{k,h_k}$, where, for each $j \in [1, h_k]$, $l_{k,j}$ is a *literal*, i.e., either an atomic proposition or a negated atomic proposition.

Intuitively, the reduction is to translate the QBF formula to a game graph and an ATL$^+$ formula for a traversal requirement on the game graph. The atomic propositions are then encoded as path constraints on the game graph. The interesting thing about the reduction is that the ATL$^+$ formula (naturally) contains no SIQs at all. This reduction may be interpreted as that we get the SIQ in the expressiveness without paying extra computation complexity.

Suppose that Γ_p represents the subgraphs for the truth of an atomic proposition p. The rest of the game graph is partitioned into subgraphs Ω_p responsible for the interpretation of atomic proposition p for all $p \in P$. Then the prenex QBF formula actually can be interpreted as a requirement for covering those Γ_p's with the decisions in those Ω_p's. For example, the following formula $\eta \equiv \exists p \forall q \exists r((p \vee q \vee r) \wedge (\neg p \vee \neg r))$ can be read as *there exists a decision in Ω_p such that for every decision in Ω_q, there exists a decision in Ω_r such that*

- *one of Γ_p, Γ_q, and Γ_r is covered; and*
- *either Γ_p or Γ_r is not covered.*

The details of constructing those Γ_p's and Ω_p's can be found in the proof of the following lemma that establishes the PSPACE complexity lower-bound.

Lemma 35. The ATL$^+$ model-checking problem for turn-based game graphs is PSPACE-hard.

Proof : Suppose we are given a prenex QBF η with propositions in P. We assume without loss of generality that all propositions are bound variables. (Note that, for the satisfiability problem, we can simply bind all free propositions by leading existential quantifiers.) We also use P for the atomic proposition set of $\mathcal{G}$. The idea is to use, for each $p \in P$, "$\Diamond p$" to encode that p is true, while "$\Box \neg p$" is used to encode that p is false. (Note that $\Box \neg p \equiv \neg \Diamond p$.) Then we construct a two-agent turn-based game graph G_η as shown in Figure 3.4, which reflects the structure of η: there is a sequence of (true) decisions (the states u_i and the state v_{n+1} have only one outgoing transition, and no true decision is taken there), which refer to the truth of the individual $\Box p_i$. These decisions are taken in the order given by the prenex quantifiers of η, and the existential decisions are taken by Agent 1 while the universal decisions are taken by Agent 2.

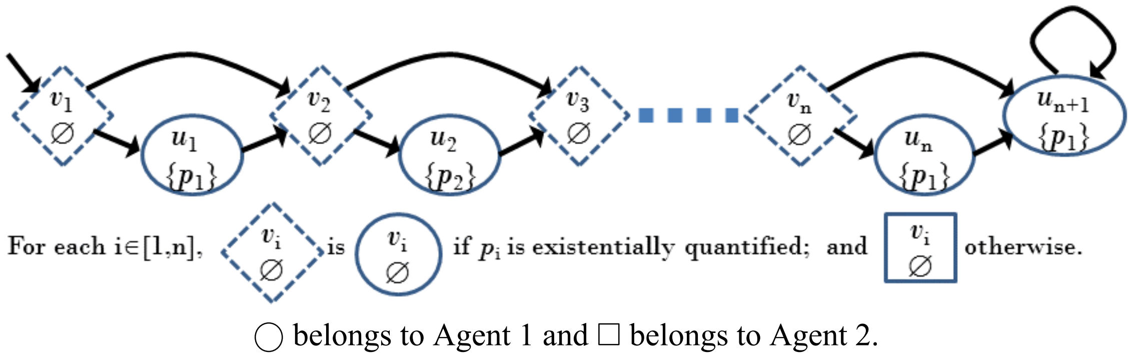

v_1
$\varnothing$
u_1
$\{p_1\}$
v_2
$\varnothing$
u_2
$\{p_2\}$
v_3
$\varnothing$
v_n
$\varnothing$
u_n
$\{p_1\}$
u_{n+1}
$\{p_1\}$
For each i∈[1,n], v_i $\varnothing$ is v_i $\varnothing$ if p_i is existentially quantified; and v_i $\varnothing$ otherwise.
◯ belongs to Agent 1 and □ belongs to Agent 2.

Figure 3.4: Game graph for the PSPACE-hardness proof of a Boolean formula with n propositions

In the graph, we use oval nodes for states owned by Agent 1 and square nodes for states owned by Agent 2. In each state, we put down its name and the set of atomic propositions that are true at the node. For each atomic proposition $p_i \in P$, we have a corresponding subgraph consisting of nodes v_i and u_i. The subgraph of u_i corresponds to Γ_{p_i} and and that of v_i, u_i together corresponds to Ω_{p_i}.

The design of the graph allows only at most one visit to states $v_1, \ldots, v_n$. For all $i \in [1, n]$, state v_i is owned by Agent 2 if p_i is universally quantified; and owned by Agent 1 otherwise. If p_i is owned by Agent 1, then Agent 1 can choose either (v_i, u_i) or (v_i, v_{i+1}). If p_i is owned by Agent 2, then both choices of Agent 2 at node v_i must yield satisfaction of η.

We construct an ATL$^+$ formula ϕ_η as $\langle 1 \rangle \bigwedge_{1 \leq i \leq k} \bigvee_{1 \leq j \leq h_k} \tau(l_{i,j})$ where $\tau(p) \overset{\text{def}}{=} \Diamond p$ and $\tau(\neg p) \overset{\text{def}}{=} \Box \neg p$. I.e., ϕ_η is obtained from η by replacing the leading quantifiers in η by $\langle 1 \rangle$.

We now show that $\mathcal{G}_\eta \models \phi_\eta$ if, and only if, η is satisfied (i.e., if η is a tautology). The latter is the case if there is a winning strategy for a 'satisfier' in the following game between a 'satisfier' and a 'refuter': following the order of the bound variables, the 'satisfier' and 'refuter' choose the truth value of the existentially and universally bound variables, respectively. When all variables are assigned truth values, the 'satisfier' wins if the CNF formula is satisfied with these values. Otherwise, the refuter wins.

Taking a winning strategy of the satisfier in this game obviously provides a winning strategy for Agent 1 and, vice versa, a winning strategy of Agent 1 in the model-checking game can be used as a winning strategy for the 'satisfier' in the satisfaction game. $\square$

We have an example for the reduction in the proof of Lemma 35.

Example 36. : Given $\eta \equiv \exists p \forall q \exists r ((p \vee q \vee r) \wedge (\neg p \vee \neg r))$, according to the construction in the proof of Lemma 35, we have the following ATL$^+$ formula: $\phi_\eta \overset{\text{def}}{=} \langle 1 \rangle (((\Diamond p) \vee (\Diamond q) \vee (\Diamond r)) \wedge ((\Box \neg p) \vee (\Box \neg r)))$. $\square$

Following Lemmas 35 and 34, we obtain the complexity of our model-checking problems.

Lemma 37. The BSIL and ATL$^+$ model-checking problems are PSPACE-complete. $\square$

3.7 Automata for BSIL model-checking

In this section, we discuss a simple encoding of BSIL model-checking in *alternting automatas* on infinite trees [**?**, **?**]. This naturally raises the question why we should study a second approach to BSIL model-checking. The answer is twofold. First, for model-checking itself, it will allow us to establish that the model complexity of BSIL model-checking is polynomial time complete: the problem to decide for a fixed BSIL formula ϕ whether or not a game graph $\mathcal{G}$ is a model of ϕ is P complete. As it is widely believed that models are usually large while specifications are small, a polynomial time bound in the size of the model might be considered more attractive than a PSPACE bound on the complete input. Second, it provides us with the full access of automata based analysis tools, which will allow us to establish a doubly exponential upper bound on the decision problem of whether or not a BSIL formula ϕ is satisfiable.

We start this section by introducing alternating automata, and then continue to encode the model-checking algorithm from the previous section. These automata constructions are then used to establish the inclusion of the model-checking algorithm in PTIME for fixed formulas, while hardness is shown by reducing reachability in AND/OR graphs [**?**] to model-checking the BSIL formula $\langle 1 \rangle \Diamond p$. Beyond establishing PTIME inclusion, we actually show that the problem is fixed parameter tractable: the problem is, for a fixed formula, only quadratic in the size of the model.

3.7.1 Alternating Automata (AA)

Alternating automata (*AA*) are used to recognized ω-regular tree languages over labeled trees. Let $\mathbb{N}$ denote the set of non-negative integers. Let $\mathbb{D} = [1, d]$ be an interval of $\mathbb{N}$. A $\mathbb{D}$-tree T is a non-empty prefix-closed subset of $\mathbb{D}^*$ (and hence $T \subseteq \mathbb{D}^*$). In T, $\mathbb{D}$ can be interpreted as *directions* from each node in T. Such a T is an ordered tree in the sense that the children of a node in T are naturally ordered according to the directions in $\mathbb{D}$. For example, when $d = 5$, the children of 1212 can be 12122, 12123, and 12125 in order.

An X-labeled tree of Υ is a pair $\langle T, \xi \rangle$, where $\xi : T \to X$ is a function from T to X. We use $\mathbb{B}^+(P)$ to denote the set of positive[1] Boolean combinations of elements in P. A satisfying assignment to a formula in $\mathbb{B}^+(P)$ is a subset of P such that the formula is true if all elements in the set is interpreted true.

For the convenience of the readers, we briefly review the definition of AA.

Definition 9. An alternating automaton (*AA*) A is a tuple $\langle X, U, u_0, \delta, \gamma \rangle$, such that

- X is the set of labels of the analysed trees,
- U is a finite set of states,
- $u_0 \in U$ is an initial state,
- $\delta : (U \times X) \mapsto \mathbb{B}^+(\mathbb{D} \times U)$ is a function that maps each pair of a state in U and an input letter in X to a positive Boolean combination of pairs of directions and states, and
- $\gamma : U \mapsto \mathbb{N}$ is a valuation function that labels each state with a non-negative integer, called its *priority*.

Alternating automata are interpreted over X-labeled trees. A *run* of A on an X-labeled tree $\Upsilon = \langle T, \xi \rangle$ is a tree $\langle T', \xi' \rangle$ with $\xi' : T' \mapsto (T \times U)$ with the following inductive restrictions.

- $\xi'(\varepsilon) = (\varepsilon, u_0)$. ε is the null sequence.
- For every $u \in U$ and sequence $\zeta \in T$ and $\zeta' \in T'$ with $\xi'(\zeta') = (\zeta, u)$ and $\xi(\zeta) = x$, there is a satisfying assignment S of $\delta(u, x)$ such that for every $(i, u') \in S$,

[1]A Boolean formula is positive if there is no negation in the formula.

there exists a $k \in \mathbb{N}$ with $\zeta' k \in T'$ and $\xi'(\zeta' k) = (\zeta i, u')$.

As can be seen, the out-degree of a node in T' is at most $d \cdot |U|$, while the out-degree of T is at most d. $\square$

Intuitively, we want to construct AAs with *states* that represent path obligations. The transitions then define, in which way these path obligations are sent down the tree. An element (i, u') in the transition formula simply means that the path obligation from u is sent to the child state u' at direction i of the input X-labeled tree.

The priority of a node in a run tree is the priority of the state in its label. The priority of an infinite path is the highest priority taken by infinitely many nodes on the path. A run tree is accepting if the priority of all infinite paths are even, and a tree is accepted if it has an accepting run. The set of trees accepted by such an automaton is called its language and an automaton is called empty if its language is empty.

An AA is called *nondeterministic* if all functions in the image of δ can be written as a disjunction over conjuncts that contain at most one pair per direction. If they contain only one such disjunct, it is called *deterministic*.

An AA is called a *safety* AA if all its states have the same even priority. For safety AA, the priority function is therefore omitted. An AA is called *weak* if for all its states u and all input letters x, the function $\delta(u, x)$ refers only to states with priorities greater or equal to $\gamma(u)$. It is called a *Büchi* AA if only prioritiy values 1 and 2 are used, i.e., the image of γ is contained in $\{1, 2\}$. Note that weak AA can be rewritten as language equivalent Büchi AA by changing only the priority function from γ to γ' where γ' maps a state u to 2 if $\gamma(u)$ is even, and to 1 otherwise.

3.7.2 Model Checking with AA's

In this subsection, we relate the algorithms used for model-checking from Section **??** to alternating automata. In order to approach this translation, we start with the simple case, where we model-check a tree against a BSIL formula of the form $\langle A \rangle \tau$, where τ does not contain an SQ. This is plausible since we can evaluate those SQs and replace them with auxiliary propositions. In addition, we assume that all strategy decisions are already made. This is also reasonable since we only allow existential SIQs and we forbid negations directly applied to SIQs. Thus we can check the model by checking the existence of S-profiles that fulfill the SQs and SIQs. For convenience, we let $\mathbb{S}$ be the set of strategies that fulfills the SQs and SIQs, if existent.

To keep the presentation simple, we study the algorithm for turn-based games and then turn to the general case of concurrent games.

Turn-based Games

Like in Section **??**, we start with the case that the DNBB formula has only one disjunct. For this case, we model-check a tree that has the form of an unravelling of $\mathcal{G}$. To relate this to the automata we have introduced, we assume for the moment that the successors are ordered: a tree node with k successors has a successor in direction 1 through k. Each node is labelled with a quadruple (a, k, S, P), containing the following information:

- $a \in [1, m]$ shows the owner of the node,
- $k \in \mathbb{D}$ shows the number of successors, and
- $S \subseteq \mathbb{S}$ is the set of strategies, for which we can reach the node. The strategies in S are used just like atomic propositions.
- P is the set of atomic propositions valid in the node.

For such a tree, we check two things for the satisfaction of a BSIL property:

(A1): The labelling of states as reachable is consistent; that is, there are strategies in S such that, for each strategy, exactly the nodes labelled by it are reachable.

(A2): For the respective set of paths described by this labelling, the single disjunct of the DNBB is satisfied.

Note that, by these, we do not check that the form of the tree complies with an unravelling of $\mathcal{G}$. However, the following observation obviously holds.

Lemma 38. $\mathcal{G}$ is a model of ϕ if, and only if, we can extend the unravelling of $\mathcal{G}$ such that (A1) and (A2) are satisfied. $\square$

To test this, we will first show how to construct two AAs that check (A1) and (A2) individually, and then construct a nondeterministic Büchi AA that checks (A1) and (A2) together. Using a nondeterministic AA is attractive because it allows for projecting away the strategies and interpreting $\mathcal{G}$ directly with the resulting AA.

Lemma 39. We can construct a nondeterministic safety AA $\mathcal{A}_s = \langle X, U_s, u_0^s, \delta^s \rangle$ with $2^{|\mathbb{S}|}$ states that recognises the trees that satisfy (A1).

Proof : We can simply use $X = [1, m] \times \mathbb{D} \times 2^{\mathbb{S}} \times 2^P$, $U_s = 2^{\mathbb{S}}$, and $u_0^s = \mathbb{S}$. For all $k \leq d$, we have a transition function δ_k^s for the k successor case (of the input labeled tree node) with the following restrictions.

- For an Agent a owning a state and a set of strategies $S \subseteq \mathbb{S}$, we let $S_a \subseteq S$ be the strategy names of strategies for Agent a. We let Π_a^k be the set of k tuples of disjoint sets $(S_a^1, S_a^2, S_a^3, \ldots, S_a^k)$ that cover[2] S_a and $S_a^{\neg} = S - S_a$ be the set of strategies not owned by a.
- We let $\delta_k^s(a, S) = \bigvee_{(S_a^1, \ldots, S_a^k) \in \Pi_a^k} \bigwedge_{i \in [1,k]} (i, S_a^i \cup S_a^{\neg})$. Then we let $\delta_s\big(S, (a, k, S, P)\big) = \delta_k^s(a, S)$ and $\delta_s\big(S, (a, k, S', P)\big) =$ *false* if $S \neq S'$.

First, the automaton is nondeterministic, and it is easy to see that a run tree must have the same form and the states in its nodes must comply with the strategies in the label of the input tree.

With this observation, the correctness of the construction can be shown by a simple inductive argument. For the induction basis, the initial node is reachable under all strategies, and the run tree is labeled with all strategies. For the induction step, it is easy to show by induction that a node of a (minimal) run tree (and, similarly, the input tree) must have the following property.

- If a state is not labelled with a strategy s_i, then no successor is labeled with s_i.
- If a state is labelled with a strategy s_i owned by a different agent than the node, then all successor states must be labeled with s_i.
- If a state is labelled with a strategy s_i owned by the same agent as the node, then

[2]Note that disjoint cover does not mean patition, as we allow for empty sets.

exactly one successor state is labelled with s_i.

Also, any combination of the labels according to the last rule is possible. This exactly characterize the labelling of reachability for the different strategies. $\square$

Now we want to construct a weak deterministic AA for property (A2). Let $\mathbb{C}$ denote the set of BP-obligations from property (A2).

Lemma 40. We can construct a weak deterministic automaton $\mathcal{A}_\wedge = \langle X, U_\wedge, u_0^\wedge, \delta_\wedge, \gamma_\wedge \rangle$ with $2^{|\mathbb{C}|}$ states that recognises the run trees that satisfy (A2).

Proof : We can simply use $X = [1, m] \times \mathbb{D} \times 2^{\mathbb{S}} \times 2^P$, $U_\wedge = 2^{\mathbb{C}}$, and $u_0^\wedge = \mathbb{C}$. For the transition formulas, we have the following restrictions.

- $\delta_\wedge\big(C, (a, k, S, P)\big) =$ *false* if there exists a $\Lambda\theta \in C$ with $\Lambda \not\subseteq S$. That is, we require that all transitions can only use strategies that have been passed down from the ancestors.
- $\delta_\wedge\big(C, (a, k, S, P)\big) =$ *false* if there exists a $\Lambda\theta \in C$ with $\texttt{localEval}(P, \theta) =$ *false*. Please recall that $\texttt{localEval}(P, \theta)$ converts a θ that is locally violated by P to *false*.
- Otherwise, $\delta_\wedge\big(C, (a, k, S, P)\big) = \bigwedge_{i \in [1,k]} (i, C')$ with $C' = \{\Lambda\texttt{next}(\texttt{localEval}(P, \theta)) \mid \Lambda\theta \in C\}$.

Note that $\mathcal{A}_\wedge$ is made deterministic by encoding the choices of strategy passing-down in the input symbol (a, k, S, P). Then we define $\gamma_\wedge$ such that $\gamma_\wedge(C)$ is odd iff C contains an until formula $\phi_1 \mathrm{U} \phi_2$. The correctness of the construction is straightforward. $\square$

Intersection of $\mathcal{A}_s$ and $\mathcal{A}_\wedge$ can, as usual, be done on the state level.

Corollary 41. We can construct a nondeterministic weak AA $\mathcal{A}' = \langle [1, m] \times \mathbb{D} \times 2^{\mathbb{S}} \times 2^P, U, u_0, \delta', \gamma \rangle$ with $2^{|\mathbb{S}|+|\mathbb{C}|}$ states that recognises trees that satisfy (A1) and (A2).

Proof : The new state set U are simply $U_s \times U_\wedge$ and the initial state is $(u_0^s, u_0^\wedge)$. The transition function returns false if either δ_s or $\delta_\wedge$ return false, and are applied independently for the two projections otherwise. Finally, $\gamma(u_s, u_\wedge) = \gamma_\wedge(u_\wedge)$. $\square$

Corollary 42. We can construct a nondeterministic weak automaton $\mathcal{A} = \langle [1, m] \times \mathbb{D} \times 2^{\mathbb{S}} \times 2^P, U, u_0, \delta, \gamma \rangle$ with $2^{|\mathbb{S}|+|\mathbb{C}|}$ states that recognises the trees with labelling functions that are projections from the trees that satisfy (A1) and (A2).

Proof : As usual, this is achieved by choosing $\delta\big(u,(a,k,P)\big) = \bigvee_{S\subseteq\mathbb{S}} \delta'\big(u,(a,k,S,P)\big)$.
□

The above lemmas and corrolaries make it easy to proof our major claim in the section.

Theorem 43. Model-checking BSIL formulas can be done in time exponential in the BSIL formula and bilinear in the number of states and transitions of the model.

Proof : Corollary 42 establishes this for PSIL formulas of the form $\phi = \langle A\rangle\tau$, where τ does not contain any SQ. We can extend this to general BSIL state formulas with the following steps.

(1) First, this extends to the case of several disjuncts simply by checking the claim for each disjunct individually.

(2) Second, we can model-check this for any state (treating it as the initial state) and subsequently store the result by introducing a fresh atomic proposition p_ϕ, and replace the sub-formula ϕ in the specification by p_ϕ.

(3) Repeating the above two steps until we have reduced the model-checking problem to model-checking a Boolean formula.

This procedure requires the model-checking procedure to be repeated for every SQ as many times as $\mathcal{G}$ has states. (In principle, it would suffice for the topmost SQ to model-check it for only the initial state.)

The model-checking algorithm for each state and SQ requires to run up to the number of disjuncts many times the respective construction algorithm of $\mathcal{A}$. We can assume without loss of generality that the turn-based game structure is properly labeled, as it contains a label for the ownership as well as a label for the atomic propositions, so we only have to add a label for the number of successors and number the directions.

The naïve approach of playing the acceptance game by offering an acceptance player the choice of a disjunct and a rejection player the choice of a direction is obviously too expensive. But note that we can split the decision as follows.

- Let the acceptance player choose the set S from $\delta\big(u,(a,k,P)\big) = \bigvee_{S\subseteq\mathbb{S}} \delta'\big(u,(a,k,S,P)\big)$,
- Let the acceptance and rejection player choose $\delta_s^k(a,S)$ successively by, starting with $I_0 = \mathbb{D}$ and $S_0 = S_a$. Then we use binary search style to iteratively narrow

down $I_0, I_1, \ldots$ by repeating the following steps for $i = 0, 1, 2, \ldots$:

(1) Cut $I_i = [l, u]$ with $m = \left\lfloor \frac{l+u}{2} \right\rfloor$ into $I_i^l = [l, m]$ and $I_i^u = [m+1, u]$.

(2) Let the acceptance player choose disjoint sets L_i and U_i whose union is S_i.

(3) Let the rejection player choose to continue with either $I_{i+1} = I_i^l$ and $S_{i+1} = L_i$, or with $I_{i+1} = I_i^u$ and $S_{i+1} = U_i$.

The repetition goes on until $I_i = \{j\}$ is singleton and then execute $\big(j, S_i \cup (S - S_a)\big)$. That is, by approaching the chosen transition in a logarithmic search we can avoid the overhead. Solving this game is linear in its state space, and this is linear in the number of transition of $\mathcal{G}$. $\square$

For PTIME hardness, we reduce the reachability checking in an AND/OR graphs [**?**] to model-checking for the simple BSIL formula $\langle 1 \rangle \Diamond p$. For the reduction, it suffices to turn AND nodes to nodes owned by Agent 1 and OR nodes to nodes owned by Agent 2.

Theorem 44. Model-checking BSIL formulas is PTIME complete in the size of the model. $\square$

Concurrent Games

The differences that occur when studying the general case of concurrent game graphs rather than turn-based games structures are moderate.

Note that the weak deterministic AA for property (A2) from Lemma 40 is not affected by this change. The automaton that checks for property (A1), on the other hand, is affected.

To account for this change, we first re-visit how the transition function of the automaton from the nondeterministic safety automaton in Lemma 39 works. The basic mechanism is a disjunction over the possible choices in Π_a^k, which represents the possible decisions made by the agent a who owns the current position for his different strategies in S_a.

Having a concurrent game graph $A = \langle m, Q, r, P, \lambda, R, \Delta, \tau \rangle$, all agents need to make their respective decision. The strategies of each agent decide which token this agent selects.

adjust the construction of the automaton that checks for property (A1) by using a not to name the agent, but to name the state (that is $a \in Q$), and assuming $Q = \{1, \ldots, k\}$, we redefine δ_s^k to

$$\delta_s^k(a, S) = \bigvee_{f: S \to \Delta} \bigwedge_{q \in Q} (q, S_q^f),$$

where $S_q^f \subseteq S$ is the subset of strategy names such that $s \in S_q^f$ if, and only if,

- $s \in S$ and
- $\tau\big((a, q), b\big) = f(s)$ holds for the Agent b for whom s is a strategy of Agent b.

f is simply a function that captures the decisions of the strategy.

The remainder of the constructions is not affected: the arguments used to establish Lemma 38, Corollaries 42 and 41, and Theorem 43 are not affected by this change.

Note that some information about the CGG is now incorporated in the automaton that checks (A1). The only minor precaution to be taken is that the number of decisions might be reduced for some agents in some states.

It is also possible to encode the transition function as part of the input letter instead. Note that, when model-checking, this information is available throught he CGG under consideration. The main theorem is therefore unaffected.

Theorem 45. Model-checking BSIL formulas for CGGs is PTIME complete in the size of the model. □

3.8 BSIL Satisfiability

In this section, we show that the satisfiability problem of BSIL is 2EXPTIME complete. The upper bound can be inferred by the simulation theorem [**?**] (or the automata constructions behind them [**?**, **?**, **?**]), while hardness is a consequence of the inclusion of CTL$^+$ [**?**].

In the first step, we provide an AA for checking the consistency of a labelling as it is constructed in the model-checking approach in the previous section. This labelling contains explicit information about whether or not a BSIL SQ formula is satisfiable. In the following we assume without loss of generality that every state in the turn based or concurrent game is reachable from the initial state without mentioning this explicitly. (Note that unreachable states have no impact on the correctness and can simply be removed.)

Lemma 46. We can build an AA $\mathcal{B}$ which is exponential in the size of a BSIL formula χ and accepts a fully labelled concurrent game graph iff the labelling is consistent and the concurrent game structure is a model of χ.

Proof : For each SQ subformula ϕ of χ, we can build a weak nondeterministic AA $\mathcal{A}_\phi$ that consists of the nondeterministic AA $\mathcal{A}_\eta$ for each disjunct η in the DNBB formula of ϕ (where we assume, without loss of generality, that their states are disjoint and $\mathcal{A}_\eta$ has initial state u_η) plus a fresh initial state u_0 with priority 0. The transition function for u_0 is simply $\delta(u_0, x) = \bigvee_\eta \delta(u_\eta, x)$, that is, in the first step one arbitrary individual automata is entered and never left again.

Likewise, we can build a weak AA $\mathcal{D}_\phi$ that accepts the complement language of $\mathcal{A}_\phi$ by dualising it. Let K denote the set of subformulas of χ that start with an SQ. Assume, without loss of generality, that the states of the individual $\mathcal{A}_\phi$ and $\mathcal{D}_\phi$ are disjoint and that the initial states of $\mathcal{A}_\phi$ and $\mathcal{D}_\phi$ are u_ϕ and $\overline{u}_\phi$, respectively. Then we can build $\mathcal{B}$ by adding two fresh states, a state u and the initial state u_0 (both with priority 0) and define the transition formula as follows.

- $\delta\big(u_0, (a, k, P)\big) =$ *false* if $P \not\models \chi$ and
- $\delta\big(u_0, (a, k, P)\big) = \delta\big(u, (a, k, P)\big)$ if $P \models \chi$,
- $\delta\big(u, (a, k, P)\big) = \bigwedge\limits_{i\in[1,k]} (i, u) \wedge \bigwedge_{\phi\in K, p_\phi\in P} \delta\big(u_\eta, (a, k, P)\big) \wedge \bigwedge_{\phi\in K, p_\phi\notin P} \delta\big(\overline{u}_\eta, (a, k, P)\big)$.

For the states of the individual $\mathcal{A}_\phi$ and $\mathcal{D}_\phi$, their transition and priority function is used.

□

The argument only uses the automata from the previous section in the last line. Consequently, we can argue completely analogously for the concurrent game graphs.

Lemma 47. We can build an AA $\mathcal{B}$ which is exponential in the size of a BSIL formula χ and accepts a fully labelled turn-based game iff the labelling is consistent and the turn-based game is a model of χ. □

Theorem 48. The satisfiability of a BSIL formula ϕ can be checked in time doubly exponential in the size of φ. If φ is satisfiable by a turn-based game, then a model can be constructed in time doubly exponential in φ.

Proof : The main difference to the model-checking case is that we cannot infer a sufficient branching degree from the model. We therefore proceed in two steps: we assume that we know a sufficient branching degree, and discuss a synthesis algorithm for it.

The first observation is that, for states in a $\mathcal{D}_\phi$, we can use any subtree as this automaton shows that something holds for all strategies. The reduction to a subtree makes the property easier to satisfy. The second observation is that a winning strategy for the acceptance player can be assumed to be memoryless. Thus, for each state of an AA $\mathcal{A}_\phi$ and each state of a tree accepted, strategies from the respective S_a (from the proof for Lemma 39) are sent to at most $|S_a| \leq |\mathbb{S}|$ directions. But successors of nodes to whom from no state a non-empty subset of S_a is sent can be pruned, provided at least one successor remains. This gives a bound on the number of directions needed, which is bilinear in the number of states of all $\mathcal{A}_\phi$ put together and the maximal number of strategies. (The latter can be estimated by the number of SIQs bound by any SQ plus one times the number of agents.) Thus, the number of directions can be restricted to a number exponential in χ.

Having established this bound of directions, we can build a language equivalent nondeterministic Büchi AA in time exponential in $\mathcal{B}$ (for the given k-bounded branching degree), whose emptiness can be checked in polynomial time. $\square$

Similarly, the particularities of a turn-based game are only used when determining the number of directions needed. The argument directly extends to the number of *decisions* needed in a concurrent game graph: the set Δ can be bounded accordingly, using the same argument. For m agents, we would then have $|\Delta|^m$ directions.

Theorem 49. The satisfiability of a BSIL formula ϕ can be checked in time doubly exponential in the size of φ. If φ is satisfiable, then a model can be constructed in time doubly exponential in φ.

3.9 Experiment

3.9.1 Implementation

We implemented a semi-symbolic model-checker of BSIL with **REDLIB** [?, ?, ?, ?, ?] which is a free library for symbolic model-checking based on decision diagrams. **REDLIB** supports symbolic pre-condition and post-condition calculation of discrete transitions. Our model-checker starts from a symbolic representataion of the initial condition. Then it repeatedly applies the post-condition procedure of **REDLIB** to explore the symbolic state representations in the computation tree. Our model-checker use the result in section **??** to bound the exploration depth of tree.

3.9.2 Benchmarks and their experiment report

Then we experimented with two parameterized benchmarks. The first is the prisoners' dilemma described in Example 1 with the number of prisoners as a parameter. Then we applied the model-checker to check whether the model satisfies formula (A) and (B) in the introduction. The time and memory usage of the model-checker is reported in Table 3.2.

Table 3.2: Experiment data for the prisoners' dilemma model

#prisoners	Formula (A)		Formula (B)	
	Time	Mem	Time	Mem
2	0.50s	72M	0.51s	71M
3	0.92s	75M	0.88s	75M
4	1.23s	83M	1.14s	80M
5	3.19s	146M	2.90s	140M
6	6.71s	388M	6.52s	361M
7	25.10s	1043M	23.11s	979M

s: seconds (computation time);
M: megabytes (memory usage);

The experiment is conducted on a PC with Intel i7-2600k 3.4GHZ CPU and 8G RAM running Ubuntu 12.04.

The second benchmark is for campaign strategies of 2 political parties, Party 1 and Party 2, for seats in the congress. There are several parameters in our model, the number of seats in the congress, the amount of budget of each party in a round, and the number of candidates of each party in a round. The game is played by the chiefs of the parties, Chief 1 (a gentleman) and Chief 2 (a lady), who decide the amount of support that a candidate receive in a round. Candidates with more budget in a round will be elected in the round. If two or more candidates get the same amount of support, the winner will be decided randomly.

In our model, in each round, chief 1 first decides his budget allotment, then chief 2 decides her budget allotment, and then the election result is determined. For example, with 3 dollars in his budget for two candidates, the strategy of a Chief can be written as $(x, y) \in \{(3,0),(2,1),(1,2),(0,3)\}$ where x and y respectively denote the support (in dollars) alloted to the first and the second candidates in his party. If Chief 1 uses strategy $(2,1)$ and Chief 2 uses strategy $(0,2)$ (Assume she has less budget.), the result of the round is that the two parties both win one seat in the round. On the other hand, if Chief 2 chooses strategy $(1,1)$ instead, then there is no strategy for Chief 1 to win more seats for his party.

We use the following propositions and predicates to write specifications. For each $i \in [1,2]$ and $j \in [1,c]$ where c is the number of candidates, $\mathit{elected}_{i,j}$ means the j'th candidate of party i is elected now. Then we can use propositions $\mathit{elected}_{i,j}$ to construct a predicate win_i that means party i wins more seats than party $2-i$ in the round. Assume that the first candidate in each party is the chief. Then the following BSIL formula specifies that Chief 1 has a strategy to make sure candidate 1 to be elected and in the same time allow the opponent party to win more seats in every round.

$$\langle 1\rangle((\langle +\rangle\Diamond \mathit{elected}_{1,1}) \wedge (\langle +2\rangle\Box \mathit{win}_2)) \tag{C}$$

Similarly, the following formula specifies that the chief of party 1 has a strategy to assure that his party can win more seats than his opponent party while allowing party two to assure the winning of a candidate.

$$\langle 1\rangle((\langle +\rangle\Diamond \mathit{win}_1) \wedge (\langle +2\rangle\Diamond \mathit{elected}_{2,1})) \tag{D}$$

With different values of budget and candidate count, we may use our model-checker to analyze the strategy for achieving various goals. The time and memory usage data for checking formula (C) and (D) is in Table 3.3.

Table 3.3: Experiment Data Election

Parameters					Formula (C)			Formula (D)		
c_1	b_1	c_2	b_2	s	Result	Time	Mem	Result	Time	Mem
2	2	2	4	1	UNSAT	0.52s	61M	UNSAT	0.53s	61M
2	4	2	4	2	UNSAT	1.20s	75M	UNSAT	1.15s	75M
2	4	2	4	3	SAT	0.73s	75M	UNSAT	1.14s	75M
3	6	3	6	3	SAT	1.51s	211M	UNSAT	0.93s	211M
3	9	3	6	3	UNSAT	9.14s	837M	SAT	5.22s	681M
3	6	3	9	3	SAT	8.74s	502M	UNSAT	8.49s	502M
3	9	3	9	4	SAT	158s	6631M	UNSAT	93.17s	5082M

c_1: # candidates of party 1
b_1: total budget in a round of party 1
c_1: # candidates of party 2
b_1: total budget in a round of party 2
s: # of seats in the congress

s: seconds (computation time)
M: megabytes (memory usage)

The experiment is conducted on a PC with Intel i7-2600k 3.4GHZ CPU and 8G RAM running Ubuntu 12.04.

3.9.3 Discussion of the experiments

The experiment shows that some possibilities of using our tool to flexibly support the analysis and synthesis of collaborating strategies among several agents. In the experiment with the prisoners' dilemma, the prisoners are modeled as individual agents. In the experiment with the election game, the chairpersons of the parties are modeled as individual agents and their candidates and respective allocated budgets are modeled as numbers. It would be interesting to see how we can explore the techniques in modeling and specifying concurrent games with our tool.

On the verification side, we can see that the CPU time and memory usage exhibits typical combinatorial explosion for tools for solving difficult problems. As a preliminary tool, our experiment and implementation do shed light for performance enhancement research in the future. Specifically, our search for the OD-trees does take the shape of the game graphs and the BSIL formulas into account. It is possible to design heuristics that utilizes the syntax information of the graphs and the formulas to construct OD-trees.

Chapter 4

Temporal Cooperation Logic

4.1 TCL

4.1.1 Syntax

A TCL formula ϕ is constructed with the following three syntax rules.

$$\begin{array}{rcl}
\phi & ::= & p \mid \neg\phi_1 \mid \phi_1 \vee \phi_2 \mid \langle A \rangle \psi \\
\psi & ::= & \phi \mid \eta \mid \psi_1 \vee \psi_2 \mid \psi_1 \wedge \psi_2 \mid \langle +A \rangle \psi_1 \mid \langle +A \rangle \bigcirc \psi_1 \mid \langle +A \rangle \eta_1 \mathrm{U} \psi_1 \mid \langle +A \rangle \psi_1 \mathrm{R} \eta_1 \\
 & \mid & \langle -A \rangle \psi_1 \mid \langle -A \rangle \bigcirc \psi_1 \mid \langle -A \rangle \eta_1 \mathrm{U} \psi_1 \mid \langle -A \rangle \psi_1 \mathrm{R} \eta_1 \\
\eta & ::= & \phi \mid \eta_1 \vee \eta_2 \mid \eta_1 \wedge \eta_2 \mid \langle + \rangle \bigcirc \eta_1 \mid \langle + \rangle \eta_1 \mathrm{U} \eta_2 \mid \langle + \rangle \eta_1 \mathrm{R} \eta_2 \\
 & \mid & \langle -A \rangle \bigcirc \eta_1 \mid \langle -A \rangle \eta_1 \mathrm{U} \eta_2 \mid \langle -A \rangle \eta_1 \mathrm{R} \eta_2
\end{array}$$

Here, p is an atomic proposition in $\mathcal{P}$ and $A \subseteq \{1, \dots, m\}$ is an agency. Property $\langle A \rangle \psi_1$ is an (existential) strategy quantification (SQ) specifying that there exist strategies of the agents in A that make all plays consistent with these strategies satisfy ψ_1. Property $\langle +A \rangle \psi_1$ is an (existential) strategy interaction quantification (SIQ) and can only occur bound by an SQ. Intuitively, $\langle +A \rangle \psi_1$ means that there exist strategies of the agents in A that work with the strategies introduced by the ancestor formulas. Likewise, $\langle -A \rangle$ indicates a revocation of the strategy binding for the agents in A. $\langle + \rangle$ is an abbreviation for $\langle +\emptyset \rangle$ or, equivalently $\langle -\emptyset \rangle$. Thus, it neither binds nor revokes the binding of the strategy of any agent. Yet, it provides a temporalisation in that it provides a tree formula that can be interpreted at a particular point.

'U' is the *until* operator. The property $\psi_1 \mathrm{U} \psi_2$ specifies a play along which ψ_1 is true until ψ_2 becomes true. Moreover, along the play, ψ_2 must eventually be fulfilled. 'R' is the *release* operator. Property $\psi_1 \mathrm{R} \psi_2$ specifies a play along which either ψ_2 is always true or $\psi_2 \mathrm{U}(\psi_1 \wedge \psi_2)$ is satisfied. (Release is dual to until: $\neg(\phi_1 \mathrm{U} \phi_2) \Leftrightarrow \neg\phi_2 \mathrm{R} \neg\phi_1$.)

In the following we may use $\langle ?A \rangle \psi$ to conveniently denote an SQ or SIQ formula with '?' is empty, '+', or '-'. An SIQ $\langle \pm A \rangle \psi$ is called non-trivial if A is not empty, and trivial otherwise.

Formulas ϕ are called *TCL formulas*, *sentences*, or *state formulas*. Formulas ψ and η are called *tree formulas*. Note that we strictly require that non-trivial strategy interaction cannot cross path modal operators. This restriction is important because it offers a sufficient level of locality to efficiently model-check a system against a TCL property. To illustrate this and to provide a simple extension that offers more expressive power to the cost of a much higher complexity, we informally discuss a small extension, *extended TCL* (ETCL), where the production rule of ψ also contains $\neg\psi$ and show that it can be used to encode ATL*, and the realisability problem of prenex QPTL can be reduced to ETCL model-checking.

For convenience, we also have the following shorthand notations.

$$\begin{array}{rclrcl}
\mathit{true} & \equiv & p \vee (\neg p) & \mathit{false} & \equiv & \neg \mathit{true} \\
\phi_1 \wedge \phi_2 & \equiv & \neg((\neg\phi_1) \vee (\neg\phi_2)) & \phi_1 \Rightarrow \phi_2 & \equiv & (\neg\phi_1) \vee \phi_2 \\
\Diamond\phi_1 & \equiv & \mathit{true}\,\mathrm{U}\phi_1 & \Box\phi_1 & \equiv & \mathit{false}\,\mathrm{R}\phi_1 \\
\neg \bigcirc \phi_1 & \equiv & \bigcirc\neg\phi_1 & \langle A \rangle \bigcirc \psi_1 & \equiv & \langle A \rangle \langle + \rangle \bigcirc \psi_1 \\
\langle A \rangle \psi_1 \mathrm{U} \psi_2 & \equiv & \langle A \rangle \langle + \rangle \psi_1 \mathrm{U} \psi_2 & \langle A \rangle \psi_1 \mathrm{R} \psi_2 & \equiv & \langle A \rangle \langle + \rangle \psi_1 \mathrm{R} \psi_2
\end{array}$$

In general, it would also be nice to have the universal SQs and SIQs as duals of existential SQs and SIQs, respectively. Couldn't we add, or encode by pushing negations to state formulas, a property of the form $[+A]\psi_1$, meaning that, for all strategies of agency A, ψ_1 will be fulfilled? In principle, this is indeed no problem, and extending the semantics would be simple. This logic would be equivalent to allowing for negations in the production rule of ψ. The problem with this logic is that it is too succinct. We will briefly discuss in the following section that model-checking becomes non-elementary if we allow for such negations.

From now on, we assume that we are always in the context of a given TCL sentence.

4.1.2 semantics

In order to prepare the definition of a semantics for TCL formulas, we start with the definition of a semantics for sentences of the form $\langle A\rangle\psi$, where ψ does not contain any SQs. We call these formulas *primitive* TCL formulas.

Due to the design of TCL, strategy bindings can only effectively happen at non-trivial SQs $\langle A\rangle$ and when a non-trivial SIQ $\langle +B\rangle$ is interpreted. To ease referring to these strategies, we first define the *bound agency* of a subformulas ϕ of a TCL sentence χ, denoted *bnd*(ϕ), as follows.

- For state formulas ϕ, *bnd*$(\phi) = \emptyset$.
- For state formulas $\langle A\rangle\psi$, *bnd*$(\psi) = A$ (unless ψ is a state formula).
- For tree formulas $\psi_1 = \langle +A\rangle\psi_2$, *bnd*$(\psi_2) =$ *bnd*$(\psi_1) \cup A$.
- For tree formulas $\psi_1 = \langle -A\rangle\psi_2$, *bnd*$(\psi_2) =$ *bnd*$(\psi_1) \smallsetminus A$.
- For all other tree formulas ψ_1 or ψ_2 with $\psi = \psi_1 \mathsf{OP} \psi_2$, with $\mathsf{OP} \in \{\wedge, \vee, \mathcal{U}, \mathcal{R}\}$, we have *bnd*$(\psi_1) =$ *bnd*(ψ) or *bnd*$(\psi_2) =$ *bnd*(ψ), respectively.

bnd shows, which agents have strategies assigned to them by an SIQ or SQ. Note that this leaves the *bnd* undefined for all state formulas not in the scope of an SQ formulas. For completeness, we could define *bnd* as empty in these cases, but a definition will not be required in the definition of the semantics.

As the introduction of additional strategies through non-trivial SIQ $\langle +B\rangle$ is governed by a *positive* Boolean combination, all strategy selections can be performed concurrently. Such a design leads us to the concept of strategy schemes.

A *strategy scheme* σ is the set of strategies introduced by any non-trivial SQ $\langle A\rangle$ or SIQ $\langle +A\rangle$. By abuse of notation, we use $\sigma[\phi, a]$ to identify such a strategy. Read in this way, σ can be viewed as a partial function from subformulas and their bound agencies to strategies. Thus, $\sigma[\phi, a]$ is defined if $a \in$ *bnd*(ϕ) is in the bound agency of ϕ.

For example, given a strategy scheme σ for a TCL sentence $\langle 1\rangle\Diamond((\langle +2\rangle\bigcirc p)\wedge\langle 2\rangle\Box q)$, the strategy used in σ by Agent 1 to enforce the whole formula can be referred to by

$$\sigma[\langle 1\rangle\Diamond((\langle +2\rangle \bigcirc p) \wedge \langle 2\rangle\Box q), 1],$$

but also by $\sigma[\langle +2\rangle \bigcirc p, 1]$, while $\sigma[\langle 2\rangle\Box q, 1]$ is undefined.

We use a simple tree semantics for TCL formulas. A (computation) tree T_r is obtained by unravelling $\mathcal{G}$ from r and expand the ownership and labelling functions from $\mathcal{G}$ to T_r in the natural way. Technically, we have the following definition.

Definition: Computation Tree. A *computation tree* for a turn based game $\mathcal{G}$ from a state q, denoted T_q, is the smallest set of play prefixes that contains q and, for all $\pi \in T$ and $(\mathit{last}(\pi), q') \in \mathcal{E}$, $\pi q' \in T$. $\Box$

The *strategy-pruned* tree for a tree node π, a strategy scheme σ, and a subformula ψ_1 of χ from a state q, in symbols $T_q\langle \pi, \sigma, \psi_1\rangle$, is the smallest subset of T_q such that:

- $\pi \in T_q\langle \pi, \sigma, \psi_1\rangle$;
- for all $\pi' \in T_q\langle \pi, \sigma, \psi_1\rangle$ with $\omega\Big((\mathit{last}(\pi')\Big) \notin \mathit{bnd}(\psi_1)$ and $(\mathit{last}(\pi'), q') \in \mathcal{E}$, $\pi' q' \in T_q\langle \pi, \sigma, \psi_1\rangle$;
- for all $\pi' \in T_q\langle \pi, \sigma, \psi_1\rangle$, $a = \omega\Big((\mathit{last}(\pi')\Big)$, and $q' = \sigma[\psi_1, a](\pi')$ with $a \in \mathit{bnd}(\psi_1)$, $\pi' q' \in T_q\langle \pi, \sigma, \psi_1\rangle$.

Given a computation tree or a strategy-pruned tree T and a node $\pi \in T$, for every $\pi q \in T$, we say that πq is a successor of π in T. A play ρ is a *limit of* T (or an infinite path in T), in symbols $\rho \overset{\infty}{\in} T$, if there are infinitely many prefixes of ρ in T.

We now define the semantics of subformulas of primitive TCL formulas inductively as follows. Given the computation tree T_q of $\mathcal{G}$, a tree node $\pi \in T_q$, and a strategy scheme σ, we write $T_q, \pi, \sigma \models \psi_1$ to denote that T_q satisfies ψ_1 at node π with strategy scheme σ.

While the notation might seem heavy on first glance, note that the truth for state formulas merely depends on the state $\mathit{last}(\pi)$ in which they are interpreted, and the tree formulas are simply interpreted on a strategy pruned tree rooted in π and defined by the strategy scheme.

- For state formulas ϕ other than SQ formulas, we use the state formula semantics:

$T_q, \pi, \sigma \models \phi$ iff $\mathcal{G}, \mathit{last}(\pi) \models \phi$, with the usual definition.

  - $\mathcal{G}, q \models p$ if, and only if, $p \in \lambda(q)$,
  - $\mathcal{G}, q \models \neg\phi$ if, and only if, $\mathcal{G}, q \not\models \phi$,
  - $\mathcal{G}, q \models \phi_1 \vee \phi_2$ if, and only if, $\mathcal{G}, q \models \phi_1$ or $\mathcal{G}, q \models \phi_2$, and
  - $\mathcal{G}, q \models \phi_1 \wedge \phi_2$ if, and only if, $\mathcal{G}, q \models \phi_1$ and $\mathcal{G}, q \models \phi_2$.

  (Note that this allows for using negation for state formulas.)

- $T_q, \pi, \sigma \models \psi_1 \vee \psi_2$ iff $T_q, \pi, \sigma \models \psi_1$ or $T_q, \pi, \sigma \models \psi_2$. (The ψ_i are no state formulas.)
- $T_q, \pi, \sigma \models \psi_1 \wedge \psi_2$ iff $T_q, \pi, \sigma \models \psi_1$ and $T_q, \pi, \sigma \models \psi_2$ hold.
- $T_q, \pi, \sigma \models \langle \pm A \rangle \bigcirc \psi$ iff, for all successors $\pi q'$ of π in $T_q\langle \pi, \sigma, \langle \pm A \rangle \bigcirc \psi_1 \rangle$, $T_q, \pi q', \sigma \models \psi$ holds.
- $T_q, \pi, \sigma \models \langle \pm A \rangle \psi_1 \mathrm{U} \psi_2$ iff, for all limits $\rho \overset{\infty}{\in} T_q\langle \pi, \sigma, \langle \pm A \rangle \psi_1 \mathrm{U} \psi_2 \rangle$, there is a $k \geq |\pi| - 1$ such that $T_q, \rho[0,k], \sigma \models \psi_2$ and, for all $h \in [|\pi|-1, k-1]$, $T_q, \rho[0,h], \sigma \models \psi_1$ hold.
- $T_q, \pi, \sigma \models \langle \pm A \rangle \psi_1 \mathrm{R} \psi_2$ iff, for all limits $\rho \overset{\infty}{\in} T_q\langle \pi, \sigma, \langle \pm A \rangle \psi_1 \mathrm{R} \psi_2 \rangle$, one of the following two restrictions are satisfied.
  - For all $k \geq |\pi| - 1$, $T_q, \rho[0,k], \sigma \models \psi_2$.
  - There is a $k \geq |\pi| - 1$ such that $T_q, \rho[0,k], \sigma \models \psi_1 \wedge \psi_2$, and, for all $h \in [|\pi| - 1, k]$, $T_q, \rho[0,h], \sigma \models \psi_2$.
- $T_q, \pi, \sigma \models \langle \pm A \rangle \psi_1$ iff $T_q, \pi, \sigma \models \psi_1$.
- $\mathcal{G}, q \models \langle A \rangle \psi_1$ iff there is a strategy scheme σ such that $T_q, q, \sigma \models \psi_1$.

If ϕ_1 is a TCL sentence then we write $\mathcal{G} \models \phi_1$ for $\mathcal{G}, r \models \phi_1$.

Note that, while asking for the existence of a strategy scheme refers to all strategies introduced by some SQ or SIQ in the TCL sentence, only the strategies introduced by the respective SQ and the SIQs in its scope are relevant.

The simplicity of the semantics is owed to the fact that it suffices to introduce new strategies at the points where eventualities become true for the first time. Thus, they do not really depend on the position in which they are invoked and we can guess them up-front. (Or, similarly, together with the points on the unravelling where they are invoked.) This is possible, simply because the validity of state formulas (and hence of TCL sentences) cannot depend on the validity of the left hand side of an until (or the right hand side of a release) *after* the first time it has been satisfied.

4.2 Expressive Power of TCL

Note that TCL is not a superclass of BSIL since BSIL allows for negation in front of SIQs while TCL does not. However, by examining the proofs in [**?**] for the inexpressibility of BSIL properties by ATL*, GL, and AMC, we find that the BSIL sentence used in the proofs is also a TCL sentence. This leads to the conclusion that there are properties expressible in TCL but cannot be expressed in ATL*, GL, and AMC.

Lemma 50. There are TCL sentences that cannot be expressed in any of ATL*, GL, or AMC. □

TCL is, in fact, not only a powerful logic, but also contains important logics either as syntactical fragments or can embed them in a straight forward way. ATL and CTL can be viewed as syntactic fragments of TCL.

But it is also simple to embed LTL and even CTL*. We start with ∃LTL, the less used variant where one is content if one path satisfies the formula. We then translate an LTL formula, which we assume w.l.o.g. to be in negative normal form (negations only in front of atomic propositions). Then "there is a path that satisfies ϕ" is equivalent to $\langle 1, \ldots, m\rangle\widehat{\phi}$, where $\widehat{\phi}$ is derived from ϕ by replacing every occurrence of $\bigcirc$, U, and R by $\langle + \rangle\bigcirc$, $\langle + \rangle$U, and $\langle + \rangle$R, respectively. The simple translation is possible because the formula $\widehat{\psi}$ is de-facto interpreted over a path, the path formed by the joint strategy of the agency $[1, m]$. The $\langle + \rangle$ operators we have added have no effect on the semantics in such a case, just as a CTL formula can be interpreted as the LTL formula obtained by deleting all path quantifiers when interpreted over a word.

Consequently, we have the expected semantics for $\forall LTL$: "all paths satisfy ϕ" is equivalent to $\neg\langle A\rangle\widehat{\neg\phi}$, where $\neg\phi$ is assumed to be re-written in negative normal form. The encoding of ∃LTL and ∀LTL can easily be extended to the encoding of CTL*.

Lemma 51. TCL is more expressive than CTL* and LTL. □

This encoding does not extend to ATL*. $\langle 1\rangle((\Box p) \vee \Box q)$ is an ATL* property that cannot be expressed with TCL.

This is different from the ATL property $(\langle 1\rangle\Box p)\vee\langle 1\rangle\Box q$ or the TCL property $\langle 1\rangle((\langle +\rangle\Box p) \vee \langle +\rangle\Box q)$. In fact, the proofs and examples in [?] can also be applied in this work to show that there are properties of ATL* (or GL, or AMC) that cannot be expressed with TCL. This leads to the following lemma.

Lemma 52. TCL is incomparable in expressiveness with ATL*, GL, and AMC. □

Note, however, that allowing for a negation in the definition of ψ would change the situation. Then an ATL* formula $\langle A\rangle\psi$ (assuming for the sake of simplicity that ψ is an LTL formula), would become $\langle A\rangle\neg\langle +[1,m] \setminus A\rangle\widehat{\neg\psi}$ in the extended version of TCL. The translation extends to full ATL*, but this example also demonstrates why negation is banned: even without nesting, we can, by encoding ATL*, encode a 2EXPTIME complete model-checking problem, losing the appealing tractability of our logic.

In fact, it is easy to reduce the realisability problem of prenex QPTL, and hence a non-elementary problem, to the model-checking problem of extended TCL. Using the game structure from Figure 4.1, we can encode the realisability of a prenex QPTL formula with $n-1$ variables, for simplicity of the form $\forall p_2\exists p_3\forall p_4\ldots\exists p_n\phi$, where $p_2,\ldots,p_n$ are all propositions occurring in ϕ. We reduce this to model-checking the formula

$$\phi' = \langle 1\rangle\neg\langle +2\rangle\neg\langle +3\rangle\neg\langle +4\rangle\neg\ldots\neg\langle +n\rangle(\psi_\phi \wedge \langle +\rangle\Box p_1),$$

where ψ_ϕ can be obtained from $\widehat{\phi}$ by replacing

- every literal p_i by $\langle -1\rangle\langle +1\rangle \bigcirc (p_i \wedge \langle +\rangle \bigcirc p_i)$, and
- every literal $\neg p_i$ by $\langle -1\rangle\langle +1\rangle \bigcirc (p_i \wedge \langle +\rangle \bigcirc \neg p_i)$.

These formulas are technically not extended TCL formulas as $\langle +i\rangle\psi_1$ is not part of the production rule of ψ, but $\langle +i\rangle\psi_1$ can be used as an abbreviation for $\langle +i\rangle \mathit{false}\mathrm{U}\psi_1$.

Checking satisfiability of ϕ is is equivalent to model-checking ϕ' on the game shown in Figure 4.1. The game has $n+1$ nodes, agents, and atomic propositions. The nodes in Figure 4.1 are labeled with the agent that owned the nodes, and the atomic proposition p_i

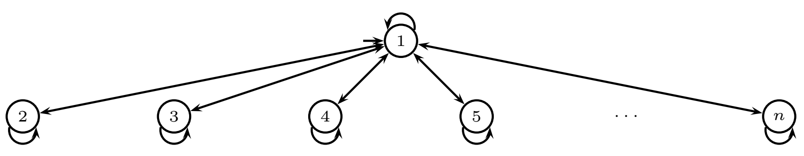


Figure 4.1: The turn-based game graph from the non-elementary hardness proof of extended TCL.

is true exactly in node i. From his state, Agent 1 can move to any other state, while all other agents can either stay in their state or return to the state owned by Agent 1.

The game starts in the node owned by Agent 1, and in order to comply with the specification, the outermost strategy profile chosen by Agent 1 must be to stay in the initial state for ever. ψ_ϕ is chosen to align the truth of p_i at position $j \in \mathbb{N}$ with the decision that Agent i makes on the history $1^j i$: *true* corresponds to staying in i and *false* with returning to 1.

It is not hard to establish a matching upper bound for model-checking extended TCL.

4.3 Complexity of TCL

In this section, we show that model-checking TCL formulas is EXPTIME-complete in the formula and P-complete in the model (and for fixed formulas), while the satisfiability problem is 2EXPTIME-complete. As the proof of inclusion of the satisfiability problem in 2EXPTIME builds on the proof of the inclusion of model-checking in EXPTIME, we start with an outline of the EXPTIME hardness argument for the TCL model-checking problem and then continue with describing EXPTIME and 2EXPTIME decision procedures for the TCL model and satisfiability checking problem, respectively. 2EXPTIME hardness for TCL satisfiability is implied by the inclusion of CTL* as a de-facto sub-language [**?**].

We show EXPTIME hardness by a reduction from the PEEK-G_6 [**?**] game. An instance of PEEK-G_6 consists of two disjoint sets of boolean variables, $P_1 = \{p_1, \ldots, p_h\}$ (owned by a safety agent) and $P_2 = \{p_{h+1}, \ldots, p_{h+k}\}$ (owned by a reachability agent), a subset $I \subseteq P_1 \cup P_2$ of them that are initially *true*, and a boolean formula γ in CNF over $P_1 \cup P_2$ that the reachability agent wants to become *true* eventually. The game is played in turns between the safety and the reachability agent (say, with the safety agent moving first), and

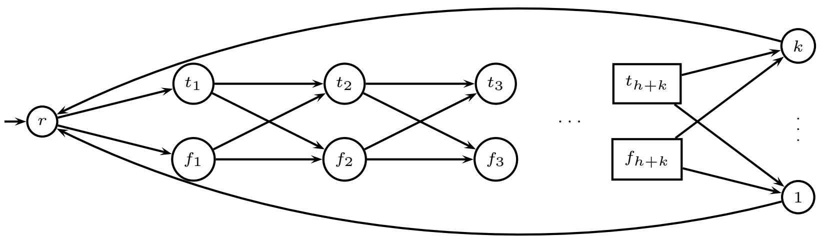


Figure 4.2: The turn-based game graph from the EXPTIME hardness proof.

each player can change the truth value of one of his or her variables in his/her turn.

Lemma 53. TCL model-checking is EXPTIME hard for primitive TCL formulas.

Proof. To reduce determining the winner of an instance of a PEEK-G_6 game to TCL model-checking, we introduce a 2-agent game $\mathcal{G} = \langle 2, \mathcal{Q}, r, \omega, \mathcal{P}, \lambda, \mathcal{E} \rangle$ as shown in Figure 4.2, where Agent 1 (he, for convenience) represents the safety agent while Agent 2 (she, for convenience) represents the reachability agent. t_{h+k} and f_{h+k} are the only states owned by Agent 2.

The game is played in rounds, and a round starts each time the game is at state r. If the game goes through t_i this is identified with the variable p_i to be true. Likewise, going through f_i is identified with the variable being false.

It is simple to write a TCL specification that forces the safety player to toggle the value of exactly one of his variables in each round, and to toggle the value of the variable p_{h+i} of the reachability player defined by the state i she has previously moved to, while maintaining all other variable values. Requiring additionally that the safety agent can guarantee that the boolean formula is never satisfied provides the reduction. □ □

The details of the construction are available in the full version. It is interesting that a game with only two agents suffices for the proof. Two agents are also sufficient to show P hardness for fixed formulas, as solving a reachability problem for AND-OR graphs [**?**] naturally reduces to showing $\langle 1 \rangle \Diamond p$.

Lemma 54. TCL model-checking for fixed formulas is P hard for primitive TCL formulas. □

In order to establish inclusion in EXPTIME and P, respectively, we use an automata based argument.

Theorem 55. The model-checking problem of TCL formulas against turn-based game graphs is EXPTIME-complete, and P-complete for fixed formulas.

Proof. We first show the claim for primitive TCL formulas $\phi = \langle A \rangle \psi$.

To keep the proof simple, we first consider a tree automaton $\mathcal{U}$ that checks the acceptance of ψ for a given strategy scheme σ. That is, $\mathcal{U}$ checks if ${T_q}^+, q, \sigma \models \psi$ under the assumption that both σ and the truth values for the subformulas starting with a $\langle \pm B \rangle$ are encoded in the nodes of ${T_q}^+$.

Such an automaton would merely have to run simple consistency checks, and it is simple to construct a suitable universal weak tree automaton $\mathcal{U}$, which is polynomial in the size of ϕ. From there it is simple to infer a deterministic Büchi tree automaton $\mathcal{D}$, which is exponential in the weak universal tree automaton [**?**].

It is then a trivial step (projection) to *guess* σ and the truth annotation of the subformulas on the fly, turning the deterministic Büchi tree automaton $\mathcal{D}$ that requires a correct annotation into a nondeterministic Büchi automaton $\mathcal{N}$ of the same size that checks $\mathcal{G}, q \models \phi$. Acceptance can be checked in time quadratic in the size of the product of $\mathcal{N}$ and $\mathcal{G}$ [**?**].

To take the step to full TCL, we can model-check the truth of primitive TCL formulas and then use the result of this model-checking instead of the respective subformula.

Hardness is inherited from Lemmata 53 and 54. □ □

This argument shows more: the complexity of TCL model-checking for fixed formulas does not depend on the formula. It suffices to solve a number of Büchi games, where both the size of the game and the number of games to be played is linear in $\mathcal{G}$.

Corollary 56. Viewing the size of a TCL sentence as a parameter, TCL model-checking is fixed parameter tractable.

The automata construction from the proof of Theorem 55 extends to a construction for satisfiability checking.

Theorem 57. The TCL satisfiability problem is 2EXPTIME-complete.

Proof. As usual, it is convenient to construct an enriched model that contains the truth of all subformulas for a TCL sentence ϕ that start with an SQ.

In a first step, we construct an alternating tree automaton $\mathcal{A}$ that recognises the enriched models of a specification. This is quite simple: $\mathcal{A}$ merely has to check that the boolean combination of SQ formulas that forms the TCL sentence ϕ is satisfied and that the truth assignment of each SQ is consistent. But this is simple, as we can use the tree automaton $\mathcal{N}_{\phi'}$ from the proof for Theorem 55 to validate the claim that a subformula ϕ' of ϕ that starts with an SQ is true, and its dual to validate that it is false. Hence, such an automaton has only two states more than the sum of the states of the individual $\mathcal{N}_{\phi'}$. In particular, it is exponential in ϕ.

For the resulting alternating automaton, we can again invoke the simulation theorem [?] to construct an equivalent nondeterministic parity automaton, which has doubly exponentially many states in ϕ (and whose transition table is doubly exponential in ϕ) and whose colours are exponential in ψ. Solving the emptiness game of this automaton reduces to solving a parity game, which can be done in time doubly exponential in ψ, e.g., using [?].

Hardness is inherited from CTL* satisfiability checking [?]. □ □

4.4 Implementation and Experiment

As a proof of concept, we have implemented a model-checker, `tcl`, in C++. `tcl` accepts models composed of extended automata that communicate with synchronisers and shared variables, with an explicit shared variable `turn` that specifies the turn of agents at a state. A turn-based game graph is then constructed as the product of the extended automata. Such an input format facilitates modular description of the interaction among the agents.

The implementation builds on a prototype for a PSPACE logic [?]. The extension is possible because we can reduce the complexity of TCL to PSPACE by simply restricting the number of operators in the η production rules in the scope of any SQ to be logarithmic in the size of the TCL sentence. We show this for primitive TCL sentences.

Lemma 58. Model-checking can be done in space bilinear in the size of the turn based game structure and the state and tree formulas that are produced using the ψ production rules and exponentially only in the number of η produced tree formulas.

Proof. We have seen that, for a primitive TCL sentence ϕ, we can use a single strategy scheme and only have to refer to the *first* position that the right hand side of an until or the left hand side of a release operator is true. Moreover, it suffices to guess just a minimal set of positions where tree formulas are true. In particular, the left hand side of a release, the right hand side of an until, and a next formula are then marked true exactly once, and the respective release and until formulas never need to be marked as true after such an event.

We can therefore use an alternating algorithm that guesses such minimal truth claims. The algorithm alternates between a verifier who guesses a truth assignment and the current decisions of the strategy scheme, and a falsifier, who guesses the direction into which to expand the path.

It is now easy to see that they will produce an infinite path in this way, and on this path each obligation that refers to a tree subformula from a ψ production rule can appear only on a continuous interval. The points where these obligations change is therefore linear in the size of ϕ. However, it also needs to track the truth value of tree formulas produced by the η production rule. (If there are multiple untilities introduced by η production rules, this also includes a marker that distinguishes a leading until, which is changed in a round robin fashion when the leading untility is fulfilled.)

The number of possible assignments is then exponential in the number of tree subformulas from η production rules. Note that $\Box$ formulas can be exempt from this rule: they are monotonous and hence incur a small impact similar to the formulas introduced using the ψ production rule.

Hence, if $|\mathcal{G}|$ denotes the size of the turn based game and k the number of temporal operators (different to $\Box$) introduced by η production rules, we end up in a cycle if there is no change in the truth assignment temporal operators that are introduced by ψ production rules or $\Box$ operators we reach a cycle within $|\mathcal{G}| \cdot k \cdot 2^k$ steps. Hence, we reach a cycle in a number of steps that is linear in $|G|$ and the size of ϕ, and exponential only in the size of

η-produced temporal operators (different to $\Box$).

Upon reaching a cycle, is suffices to check if the cycle is accepting. (No standing obligation by an until.) $\Box$ $\Box$

The model-checker uses a stack to explicitly enumerate all paths of all tree tops with depth prescribed by Lemma 58. The tool can be downloaded from Sourceforge at project REDLIB at: `http://sourceforge.net/projects/redlib/`.

We use the parametrised models of the iterated prisoners' dilemma as our benchmarks to check the performance of our implementation. A brief explanation of the models can be found in the introduction. The unique parameter to the models are the number of prisoners m. There is also a policeman in the models. We build a turn-based game graph for each value of m in the experiments. The parametrisation helps us to observe how our algorithm and implementation scale to model and formula sizes. To simplify the construction of the state-space representation, we assume that, in each iteration, the prisoners make their decisions in a fixed order. After all prisoners have made their decisions, the policeman makes his decision. Subsequently, the whole game moves to the next iteration. We use seven benchmark formulas on these models in our experiments. The first five benchmarks are taken from the examples (A) through (E) from the introduction. Benchmarks (F) and (G) are the following two properties, taken from [**?**].

- Property (F) specifies that all prisoners except Prisoner 1 can collaborate to release Prisoner 1 and let Prisoner 1 decide their fate.

  $$\langle 2,\ldots,m\rangle\Big((\langle +\rangle\Diamond\neg\mathtt{jail}_1)\wedge\textstyle\bigwedge_{i\in\{2,\ldots m\}}(\langle +1\rangle\Diamond\neg\mathtt{jail}_i)\wedge(\langle +1\rangle\Box\mathtt{jail}_i\Big) \tag{F}$$

- Property (G) specifies that Prisoner 1 has a strategy to put all other prisoners in jail while leaving her fate to them.

  $$\langle 1\rangle\Big((\textstyle\bigwedge_{i\in\{2,\ldots m\}}\langle +\rangle\Box\mathtt{jail}_i)\wedge(\langle 2,\ldots,m\rangle\Diamond\neg\mathtt{jail}_1)\wedge\langle 2,\ldots,m\rangle\Box\mathtt{jail}_1\Big) \tag{G}$$

For these benchmarks, we have collected the performance data for various parameter values in Table 4.1. For small models, the memory usage is dominated by the normal overhead, such as the representation of variable tables, state-transition tables, formula structures, etc. The data shows that our prototype can handle the various benchmarks, and scales well on five of the seven benchmarks. Ignoring the overhead, it also shows the

Table 4.1: Performance data of model-checking the TCL fragment

properties \ m	2	3	4	5	6	7	8	9	10
(A)	0.71s	0.94s	5.41s	66.3s	945s	>1000s			
	163M	165M	185M	350M	1307M				
(B)	0.50s	0.52s	0.61s	0.71s	1.11s	1.62s	5.77s	20.9s	68.1s
	163M	163M	164M	165M	168M	176M	214M	270M	376M
(C)	0.51s	0.51s	0.6s	0.82s	1.01s	1.81s	5.54s	18.2s	48.3s
	163M	163M	164M	165M	168M	176M	200M	241M	318M
(D)	0.5s	0.51s	0.57s	0.74s	1.01s	1.79s	7.41s	33.8s	141s
	163M	163M	164M	165M	168M	175M	232M	312M	430M
(E)	0.51s	0.66s	19.1s	>1000s					
	163M	164M	194M						
(F)	0.51s	0.53s	0.61s	0.71s	1.01s	1.70s	5.38s	15.2s	53.7s
	163M	163M	163M	165M	168M	175M	202M	243M	295M
(G)	0.52s	0.52s	0.65s	0.72s	1.03s	1.85s	4.86s	16.1s	93.5s
	163M	163M	164M	165M	169M	177M	189M	208M	235M

s: seconds; M: megabytes.

The models are with 1 policeman and m prisoners. The experiment was carried out on an Intel i5 2.4G notebook with 2 cores and 4G memory, running ubuntu Linux version 11.10.

exponential growth. The models, however, are growing exponentially, too. We assume that this growth i the main cause of the exponential growth of the response time.

Chapter 5

Conclusions