WARSAW UNIVERSITY OF TECHNOLOGY

DISCIPLINE OF SCIENCE – INFORMATION AND COMMUNICATIONS TECHNOLOGY

FIELD OF SCIENCE – ENGINEERING AND TECHNOLOGY

Ph.D. Thesis

Teofil Sidoruk, M.Sc.

State Space Reductions for Multi-agent Systems

Supervisor

Prof. Wojciech Penczek, Ph.D., D.Sc.

WARSAW 2023

Acknowledgements

First and foremost, I express my profound gratitude to Prof. Wojciech Penczek, who previously supervised my M.Sc. thesis and convinced me to take this next step, for giving me the opportunity to do so at the Institute of Computer Science PAS, for his invaluable guidance and support throughout my Ph.D. studies, and, above all, for always being incredibly kind, generous, and helpful.

For the past five years, I have have had the privilege of being part of the Theory of Distributed and Computing Systems group at ICS PAS. I would like to take this opportunity to thank everyone I met there, starting with Wojtek Jamroga, whose knowledge, the ability to pass it on, and work ethic remain an unmatched reference point for me. It has been an equal pleasure to work with my other co-authors: Damian Kurpiewski (who also provided feedback on this thesis), Michał Knapik, Łukasz Mikulski, and Łukasz Maśko.

I am very grateful to Laure Petrucci, Jaime Arias, and Carlos Olarte from Université Sorbonne Paris Nord, for the ongoing scientific collaboration, one that I really appreciate and hope to continue in the future.

For their invaluable help with many aspects of my first scientific publications, I also extend my gratitude to Artur Niewiadomski and Piotr Świtalski from the Siedlce University of Natural Sciences and Humanities.

Finally, last but certainly not least, I am truly blessed to have my family's unquestionable support, for which I will be forever grateful.

Abstract

The Ph.D. thesis "State Space Reductions for Multi-agent Systems", written under the supervision of Prof. Wojciech Penczek, discusses several approaches to mitigating the problem of state- and transition-space explosion, which is a major obstacle in the formal verification of multi-agent systems. Asynchronous multi-agent systems exacerbate this issue, requiring to account for all possible sequences of agents' actions, in any order. At the same time, they are often preferable to synchronous formalisms for modelling real-world systems, processes, and protocols. This emphasises the need for efficient reduction techniques, especially when dealing with the verification of *strategic abilities* of autonomous agents, whose computational complexity is significantly higher compared to the case of properties concerning just the temporal evolution of a system.

Chapter 1 recalls the historical background of formal verification techniques, including model checking, and discusses their relevance and necessity in practical applications. The associated issue of state explosion is introduced, as well as the idea of model reductions.

Chapter 2 introduces the necessary theoretical background, in particular the definitions of Asynchronous Multi-agent Systems (AMAS) and their execution semantics, the syntax and semantics of Alternating-time Temporal Logic (**ATL***), and key notions related to the strategic ability of agents.

Chapter 3 recalls the technique of partial order reduction (POR) as previously defined for Linear Temporal Logic (**LTL**) and subsequently demonstrates how it can be adapted to **sATL***, a subset of **ATL*** that remains significantly more expressive than **LTL**. The correctness of reductions is proven in various semantical settings, including both memoryless and perfect recall strategies, subjective strategic ability, and an epistemic extension of **sATL***. The obtained results are notable both in theory and practice, as they provide significant model reductions for the new purpose of verifying strategic ability properties at no additional computational cost, and potentially using existing tools and algorithms designed for **LTL**.

Chapter 4 defines two specialised techniques, called pattern-based reduction and layer-based reduction. While applicable to a much smaller class of models than POR, they are able to exploit specific characteristics of these models, thus yielding additional gains in model reduction. Attack-defence trees (ADTrees) are introduced and translated to a multi-agent setting (Guarded Update Systems, $\mathcal{GUS}$) in order to provide an example of models exhibiting the desired characteristics, namely a tree synchronisation topology between components.

Chapter 5 revisits the setting of ADTrees, but looks at model reductions from a different perspective, aiming to minimise the number of agents in a multi-agent system. To that end, the formalism of AMAS is extended to represent ADTrees, superseding the previous formulation of $\mathcal{GUS}$ which did not include agents. Representing security scenarios from ADTrees as extended AMAS allows for studying a new aspect of these models, now considered in the context of two opposing agent coalitions of specific size and assignment to particular tasks. Therefore, an algorithm synthesising an optimal assignment using a minimal number of agents is proposed, proved to be correct, and evaluated experimentally.

Chapter 6 concludes the thesis, summarising it and identifying several potential avenues of further research.

Prior works

The material in this thesis is based on the following papers to which the author of this thesis has contributed:

[1] W. Jamroga, W. Penczek, T. Sidoruk, P. Dembinski, and A. Mazurkiewicz, "Towards Partial Order Reductions for Strategic Ability," *Journal of Artificial Intelligence Research*, vol. 68, pp. 817–850, 2020

[2] W. Jamroga, W. Penczek, and T. Sidoruk, "Strategic Abilities of Asynchronous Agents: Semantic Side Effects and how to tame them," in *Proceedings of KR 2021*, 2021, pp. 368–378

[3] D. Kurpiewski, W. Jamroga, Ł. Maśko, Ł, Mikulski, W. Pazderski, W. Penczek, and T. Sidoruk, "Verification of Multi-Agent Properties in Electronic Voting: A Case Study," in *Proceedings of AiML 2022*. College Publications, 2022, pp. 531–556

[4] J. Arias, C. Budde, W. Penczek, L. Petrucci, T. Sidoruk, and M. Stoelinga, "Hackers vs. Security: Attack-Defence Trees as Asynchronous Multi-agent Systems," in *Proceedings of ICFEM 2020*. Springer, 2020, pp. 3–19

[5] L. Petrucci, M. Knapik, W. Penczek, and T. Sidoruk, "Squeezing State Spaces of (Attack-Defence) Trees," in *Proceedings of ICECCS 2019*. IEEE, 2019, pp. 71–80

[6] J. Arias, L. Petrucci, Ł. Maśko, W. Penczek, and T. Sidoruk, "Minimal Schedule with Minimal Number of Agents in Attack-Defence Trees," in *Proceedings of ICECCS 2022*. IEEE, 2022, pp. 1–10

For the details regarding the author's involvement in specific aspects of the above works, we refer the reader to the beginning of each chapter of the thesis.

Keywords

multi-agent systems, asynchronous execution, state explosion, partial order reduction, temporal logic, strategic ability, attack-defence trees, scheduling

Streszczenie

Niniejsza rozprawa doktorska, zatytułowana "State Space Reductions for Multi-agent Systems" i napisana pod kierunkiem prof. dr hab. inż. Wojciecha Penczka, przedstawia kilka technik ograniczających poważny problem w formalnej weryfikacji systemów wieloagentowych, jakim jest eksplozja przestrzeni stanów i tranzycji. Problem ten dotyczy w szczególności systemów asynchronicznych, w których konieczne jest uwzględnienie wszystkich możliwych ciągów akcji agentów w dowolnej kolejności. Jednak właśnie taka semantyka wykonania często jest formalizmem lepiej odpowiadającym potrzebom modelowania i weryfikacji rzeczywistych systemów, procesów i protokołów, które są zwykle, przynajmniej na pewnym poziomie abstrakcji, asynchroniczne. Tym bardziej świadczy to, że efektywne metody redukcji modeli są w praktyce niezbędne, zwłaszcza gdy mamy do czynienia z weryfikacją *zdolności strategicznej* autonomicznych agentów, charakteryzującą się znacząco wyższą złożonością obliczeniową niż w przypadku własności czysto temporalnych.

Rozdział 1 pokrótce przedstawia rys historyczny technik formalnej weryfikacji, w tym weryfikacji modelowej (ang. *model checking*) i uzasadnia potrzebę ich praktycznego stosowania. Ponadto omawia problem eksplozji przestrzeni stanów i tranzycji i wprowadza zagadnienie redukcji modeli.

Rozdział 2 zawiera podstawy teoretyczne niezbędne do późniejszego zdefiniowania metod redukcji, w szczególności definicje asynchronicznych systemów wieloagentowych (AMAS) i ich semantyki wykonania, składnię i semantykę logiki temporalnej czasu alternującego (**ATL***), oraz kluczowe zagadnienia dotyczące zdolności strategicznej agentów, takie jak pojęcia strategii i jej wyniku.

Rozdział 3 zaczyna się od przywołania istniejącej formalizacji redukcji częściowoporządkowych (POR) dla logiki temporalnej czasu liniowego (**LTL**), a następnie przedstawia adaptację tej techniki do **sATL***, podzbioru **ATL*** o nadal znacząco większej wyrażalności niż **LTL**. Poprawność uzyskanych redukcji jest udowodniona w różnych wariantach semantycznych, uwzględniających zarówno strategie z pełną historią stanów jak i bezpamięciowe, subiektywną definicję zdolności strategicznej, a także rozszerzenie **sATL*** o operator epistemiczny. Uzyskane wyniki są istotne zarówno z teoretycznego, jak i praktycznego punktu widzenia. Pozwalają bowiem na uzyskanie znaczącej redukcji modeli w nowym zastosowaniu, tj. weryfikacji własności dotyczących zdolności strategicznej agentów, nie zwiększając przy tym złożoności obliczeniowej algorytmu redukcji w stosunku do **LTL**. Co więcej, możliwe jest wykorzystanie już istniejących narzędzi implementujących algorytmy dla tej ostatniej logiki.

Rozdział 4 definiuje dwie wyspecjalizowane techniki redukcji w oparciu o wzorce (ang. *pattern-based reduction*) oraz o warstwy (ang. *layer-based reduction*). W odróżnieniu od redukcji częściowoporządkowych, można je zastosować dla dużo mniejszej klasy modeli. Pozwala to jednak na wykorzystanie ich charakterystycznych własności do uzyskania bardziej efektywnej redukcji niż umożliwiana przez POR. Jako przykład modeli o pożądanych własnościach, przede wszystkim z topologią synchronizacji pomiędzy komponentami będącą drzewem, zostają wykorzystane scenariusze bezpieczeństwa reprezentowane jako w postaci drzew ataku/obrony (ang. *attack-defence trees*, ADTree) po translacji do formalizmu automatowego (ang. *guarded update systems*, $\mathcal{GUS}$).

Rozdział 5 powraca do formalizmu ADTree, ale rozpatruje zagadnienie redukcji modeli z innej per-

spektywy, a mianowicie dążąc do zmimimalizowania liczby agentów w systemie wieloagentowym. W tym celu formalizm AMAS zostaje rozszerzony, aby umożliwić reprezentowanie scenariuszy bezpieczeństwa z ADTree. W ten sposób zostaje uogólniona automatowa reprezentacja w postaci $\mathcal{GUS}$, rozważana w poprzednim rozdziale, która dotychczas nie brała pod uwagę agentów. Translacja z ADTree do AMAS pozwala na rozważanie scenariuszy ataku/obrony w nowym kontekście, uwzględniającym liczbę agentów w przeciwnych koalicjach oraz ich konkretny przydział do poszczególnych zadań. Zostaje zaproponowany algorytm optymalnego planowania zadań z wykorzystaniem minimalnej liczby agentów, a jego poprawność udowodniona i oceniona eksperymentalnie.

Rozdział 6 stanowi podsumowanie rozprawy doktorskiej oraz wytycza potencjalne kierunki dalszych badań naukowych.

Istniejące prace

W rozprawie zostały wykorzystane definicje, przykłady, wyniki teoretyczne i eksperymentalne z następujących prac naukowych z udziałem jej autora:

[1] W. Jamroga, W. Penczek, T. Sidoruk, P. Dembinski, and A. Mazurkiewicz, "Towards Partial Order Reductions for Strategic Ability," *Journal of Artificial Intelligence Research*, vol. 68, pp. 817–850, 2020

[2] W. Jamroga, W. Penczek, and T. Sidoruk, "Strategic Abilities of Asynchronous Agents: Semantic Side Effects and how to tame them," in *Proceedings of KR 2021*, 2021, pp. 368–378

[3] D. Kurpiewski, W. Jamroga, Ł. Maśko, Ł, Mikulski, W. Pazderski, W. Penczek, and T. Sidoruk, "Verification of Multi-Agent Properties in Electronic Voting: A Case Study," in *Proceedings of AiML 2022*. College Publications, 2022, pp. 531–556

[4] J. Arias, C. Budde, W. Penczek, L. Petrucci, T. Sidoruk, and M. Stoelinga, "Hackers vs. Security: Attack-Defence Trees as Asynchronous Multi-agent Systems," in *Proceedings of ICFEM 2020*. Springer, 2020, pp. 3–19

[5] L. Petrucci, M. Knapik, W. Penczek, and T. Sidoruk, "Squeezing State Spaces of (Attack-Defence) Trees," in *Proceedings of ICECCS 2019*. IEEE, 2019, pp. 71–80

[6] J. Arias, L. Petrucci, Ł. Maśko, W. Penczek, and T. Sidoruk, "Minimal Schedule with Minimal Number of Agents in Attack-Defence Trees," in *Proceedings of ICECCS 2022*. IEEE, 2022, pp. 1–10

Szczegółowe informacje dotyczące konkretnego wkładu autora w powyższe artykuły znajdują się na początku każdego rozdziału niniejszej rozprawy.

Słowa kluczowe

systemy wieloagentowe, asynchroniczna semantyka wykonania, eksplozja przestrzeni stanów, redukcje częściowoporządkowe, logika temporalna, zdolność strategiczna, drzewa ataku/obrony, planowanie

Contents

Chapter 1

Introduction

We live in a time where computer systems, both software and hardware, have become virtually ubiquitous in nearly all aspects of our daily existence. There is now an entire generation of people who grew up in a world where pervasive internet access is often taken for granted, and common tasks, from navigation to financial operations, have been made infinitely more convenient. One curious aspect of this gradual, yet dramatic, change that has occurred over the last few decades is the way we view instances of incorrect operation of computer systems and their underlying algorithms, often stemming from undiscovered issues in their design. It is safe to say, at least in the context of software applications targeting a large consumer base, that such errors are not only commonplace, but also not at all unexpected by the users. Despite extensive testing performed by dedicated internal and external quality assurance teams, it seems that no piece of software, from video games to office suites to operating systems, is immune to "bugs", as these issues are often colloquially referred to.

On the other hand, it does not take long to come up with examples of areas and applications where this state of things is not just undesirable, but outright unacceptable. These include critical systems involved in e.g. the operation of medical devices, plane navigation, rocket propulsion, spacecraft control, nuclear power plants, to name a few. When failure means huge financial losses at best, and human injuries or deaths at worst, it is clear that the usual approach to software quality does not suffice. In particular, while testing does have its role in the process, and often allows for easily encountering errors if they are present, it cannot tell anything about their absence. Therefore, for these "mission critical" systems a different approach is necessary, one that does not aim at finding errors, but rather at proving or disproving – with all the mathematical rigour it entails – that the system satisfies formally specified properties. This approach is referred to as *formal verification*.

1.1 Formal verification

To better motivate the necessity of formal verification in the design of computer systems, it is perhaps worth recalling a few historical examples where it likely would have prevented significant losses.

In 1996, the Ariane 5 rocket exploded shortly after take-off from the Kourou cosmodrome in French Guiana, leading to the destruction of $500 million worth of equipment for the European Space Agency. The cause was eventually identified as an error in the rocket's internal software, where an exception occurred due to the conversion of 64-bit floating point numbers to 16-bit signed integers, exceeding the maximal value of the latter type. Although damaging both financially and in terms of publicity and perceived reliability for the agency (especially since this was to be the rocket's maiden flight), the failed launch nonetheless had a good effect, since it brought the issue to the public attention, emphasising the need for formal methods in mission-critical software verification, and leading to increased funding for new

techniques to achieve that goal.

Today, formal verification encompasses many different techniques, with varying degrees of scalability and automation. The manual approach involves an expert (or, more likely, a large group of experts) in the fields of mathematics and logics meticulously creating a mathematical model of the system, and then constructing a formal proof that the expected properties indeed hold. This is a less than ideal solution due to its practical feasibility: today's systems are often several orders of magnitude larger than what would be possible to tackle this way. Consider, for example, modern CPUs and GPUs, whose transistor counts are in the billions, or even tens of billions in case of the latter. Even if somehow a proof could be constructed manually for such large systems, by the time their design was confirmed to be correct, the products would be long obsolete in the market, defeating the purpose of verification.

Theorem provers allow for automating the process of constructing proofs of correctness to a certain extent. However, as in the fully manual approach, it still requires the involvement of experts, typically from the outside. Therefore, while proficient in verification, they may be lacking some essential information about the system's architecture, possibly as a result of miscoordination or misunderstanding with the engineers and programmers responsible for actual implementation.

Finally, *model checking*, on which we will focus in this thesis in the context of reduction techniques, offers a fully automated way of verifying models. In this approach, the manually constructed proof is replaced with an automated verifier, or *model checker*, which takes as input the *model* of the verified system. Thus, rather than on mathematicians and logicians constructing a proof, i.e. outside experts, the emphasis is now on creating an *abstraction* of the system. Naturally, the people who designed it in the first place, and as such have intimate knowledge about its architecture, are the ideal candidates for handling this task. This familiarity with the system in turn allows them to choose the components to include in the representation, and those to abstract away from. For instance, one may wish to ignore the cryptographic aspect of a data exchange protocol, e.g. if the underlying encryption is a commonly used standard that has already been formally verified before, thereby reducing the complexity of the problem and freeing the capacity to model more relevant components to a higher degree of granularity.

1.2 Model checking

The foundations of model checking have been laid in the 1980s, with the seminal works of Clarke and Emerson [7] and Queille and Sifakis [8]. However, it was Pnueli [9] in the late 1970s who first proposed the use of temporal logic in computer science, as a means of expressing important properties of systems, such as invariance, eventuality, or fairness. Previously, temporal reasoning had been squarely in the domain of philosophy, where it is essential to mention the early work of Łoś, built upon by Prior [10], whose *tense logic* can be seen as a direct predecessor to the logics used in formal verification today. Furthermore, Kamp [11] provided an important contribution with the theorem connecting temporal logic to first order logic, as well as the introduction of the "until" operator that remains a staple of linear and branching temporal logics and their various extensions.

Over the next decades, the two main flavours of temporal logic, Linear Temporal Logic (**LTL**) and Computation Tree Logic (**CTL**), have been used in practice to define specifications of real-world software and hardware systems, successfully verified through model checking algorithms. These include, among others, integrated circuits [12], communication protocols [13], security systems [14], device drivers [15], and spacecraft control software [16, 17, 18].

Major subsequent developments on the model checking front include *symbolic model checking* [19]. By adopting a symbolic representation, based on ordered binary decision diagrams (OBDDs), it allowed for handling sets of states and transitions, rather than individual, explicitly represented ones. This dramatically increased, by many orders of magnitude, the size of models that could be realistically

tackled with model checking procedures. Furthermore, *bounded model checking*, proposed by Biere [20], leverages SAT-solvers to check the satisfiability of a Boolean formula that encodes the negation of the intended property of the system, plus relevant constrains on the initial states and some number k of transition steps. In other words, the satisfiability of the formula implies that there is a counterexample of length k for the original property, otherwise the bound k may be increased if desired. This approach may be further augmented by using SMT (Satisfiability Modulo Theories) solvers, whose specification language allows for first-order logic, avoiding the conversion of high-level constraints to propositional formulas, and thus leading to much more natural and compact encodings.

As for the logical formalisms, numerous extensions of the linear and branching logics **LTL** and **CTL** have been proposed and studied. These include adaptations to timed systems, both using the discrete as well as continuous semantics of time, where temporal operators are additionally restricted with intervals in which they are evaluated. In this thesis, however, we will focus on extensions of temporal logics with additional *modalities*: primarily the *strategic*, but also the *epistemic* modality, which add an entirely new layer of reasoning on top of the existing temporal formalisms. In this setting, components of the system can now be considered as autonomous *agents* with partial or full *knowledge* about the state of the system and possibly also memory of previous actions. The notion of *strategic ability*, i.e. whether an agent (or a group of several agents) have a conditional plan, called a *strategy*, to enforce some temporal goal, is of particular interest. The most popular temporal-strategic logic is Alternating-time Temporal Logic (**ATL***), proposed by Alur, Henzinger, and Kupferman [21].

Model checking procedures for **ATL*** allow for extending the applicability of formal verification to many new areas. With the increasing proliferation of computer systems, algorithms, and protocols in many aspects of our lives where they have been previously absent, it has become more important than ever to take into account the human side of computing. The advent of social engineering, popularised by Kevin Mitnick's hacking exploits in the 1990s, put the focus on the "weakest link" in the chain of system security, i.e. the human users. In multi-agent systems, the autonomous components may represent people and hardware or software components alike, making this setting a perfect formalism for reasoning about systems where the technical and social aspects intersect. Electronic voting protocols serve as an excellent example and material for case studies here. In an election, it is equally important to ensure that the encryption of data and the used exchange protocols meet all required criteria, as it is to preserve the integrity of the process against potential social-based threats, from the national level down to potential coercion of individual voters. Furthermore, it may be desirable to provide some means of vote verifiability, in line with the growing trend to focus on transparency and trustworthiness in computing. **ATL*** model checking allows for the verification of relevant properties that combine these two aspects.

1.3 State explosion

However, the practical verification of strategic ability in multi-agent systems remains a major challenge. The computational complexity of **ATL*** model checking can be considered manageable compared to other formalisms combining the strategic modality with temporal reasoning, such as Strategy Logic [22]. Nonetheless, the problem is significantly harder than the verification of purely temporal **LTL** and **CTL** properties. In fact, under certain strategy semantics (namely, for agents with imperfect information about the global state of the system, but with full recall of their previous actions), it becomes undecidable.

Secondly, **ATL*** extends temporal logics with the strategic modality, and as such inherits from the former many aspects of model checking procedures and associated issues. In particular, the state explosion problem remains a major obstacle, especially when dealing with asynchronous systems, as is the case in this thesis. Assuming the execution semantics where each component, i.e. agent, can proceed without synchronising with all others, is often better suited than the synchronous alternative for modelling and

verifying real-world systems, from concurrent programs with potential race conditions to man-in-the-middle attacks in security scenarios, to elections at the level of individual voters, who may cast their ballots at any time and in any order, possibly having been coerced to pick a particular candidate or recast. The downside is that asynchronous models include all possible interleavings of transitions, even those intuitively seen as equivalent and leading to the same global state, just in a different order. As a result, the state- and transition-space is significantly larger than in comparable synchronous formalisms. This further exacerbates the issue of practical verification, which in itself is already more complex than for temporal logics. Consequently, reduction techniques aiming at pruning the models while preserving all verified properties, become essential.

1.4 Model reductions

For linear and branching time temporal logics, *partial order reduction* (POR) is the most popular approach to containing state explosion. It was first defined for **LTL** and **CTL** over three decades ago and has since been implemented in verifiers such as Spin. Besides the potential gains it offers, i.e. exponential reduction in best case scenarios, from the practical standpoint, the main strength of this method is that the full model (potentially too large to be generated, e.g. due to memory requirements) is never created. Instead, the reduction occurs while generating the global model from its local components. Ideally, this can be combined with *on-the-fly* model checking.

The focus on this thesis will be on analogous model reduction techniques for the verification of strategic abilities of agents, i.e. formulas of **ATL*** (or, more precisely, a subset of **ATL*** that lends itself well to reduction algorithms while remaining expressive enough to specify virtually all practically relevant properties), in asynchronous systems. First and foremost, we will demonstrate that the existing partial order reduction scheme for linear time logic **LTL** not only can be adapted to that new setting including the strategic modality, but does not incur extra computational costs. This is a notable theoretical result with major practical ramifications, as it effectively allows for reusing existing implementations of POR, thus far available for purely temporal properties, in the verification of a more expressive logic, where autonomous agents can employ strategies to influence the temporal evolution of the system.

Furthermore, we will propose other reduction techniques for asynchronous multi-agent systems, trading the universal applicability of POR for potentially higher efficiency of obtained reduction, albeit in a smaller class of models that exhibit certain characteristics. To that end, we will investigate multi-agent systems translated from attack-defence trees, a popular formalism primarily used for expressing and studying security scenarios. In this setting, we will propose two specialised reduction techniques that exploit the particular topology of synchronisation between local components in such models.

Finally, we note that representing attack-defence trees as multi-agent systems in this manner allows for reasoning about these security scenarios on an entirely new level, i.e. considering agent coalitions of a particular size and their specific assignments to particular tasks. This leads to looking at model reductions from a different perspective, namely by minimising the number of agents, and their corresponding local automata, in the system. We propose and implement an algorithm that tackles this optimisation problem, synthesising the shortest schedule using the lowest possible number of agents, thereby obtaining a different type of model reduction in a multi-agent system.

1.5 Thesis statement

We study model reduction techniques for asynchronous multi-agent systems, primarily within the context of verifying strategic ability properties, but also discuss other approaches that are only applicable to

models that exhibit particular characteristics. Therefore, the main thesis can be stated as follows:

Partial order reduction for linear temporal logic can be adapted to the verification of strategic ability in asynchronous multi-agent systems, and specialised techniques can yield additional gains, albeit for a smaller class of models.

Chapter 2

Preliminaries

The material in Chapter 2 is based on the following papers to which the author of this thesis has contributed:

[1] W. Jamroga, W. Penczek, T. Sidoruk, P. Dembinski, and A. Mazurkiewicz, "Towards Partial Order Reductions for Strategic Ability," *Journal of Artificial Intelligence Research*, vol. 68, pp. 817–850, 2020

[2] W. Jamroga, W. Penczek, and T. Sidoruk, "Strategic Abilities of Asynchronous Agents: Semantic Side Effects and how to tame them," in *Proceedings of KR 2021*, 2021, pp. 368–378

The author's involvement in these works includes many definitions referenced in this chapter, and especially modifications to the formalism of asynchronous multi-agent systems that add auxiliary ϵ-transitions, redefine protocol functions, and frame the notion of reactive opponents as a fairness-style optional condition. Note that the main contribution is actually related to partial order reduction for strategic ability, which is the subject of Chapter 3 and thus will be discussed at the beginning of that chapter.

2.1 Asynchronous semantics for strategic ability

Introduced by Alur, Henzinger and Kupferman, *alternating-time temporal logic* [23, 21] has become one of the main ways of reasoning about the interactions in a multi-agent setting. Thus far, reasoning about the strategic ability of agents has been typically done in synchronous formalisms, such as that of Concurrent Game Systems (CGS) [24]. However, this is not always the preferable approach. Many systems are either inherently asynchronous, or at least certain aspects of the interaction between agents are better modelled that way.

An asynchronous semantics for strategic ability was introduced in [1]. In this section, we recall that formal setting, starting with the definition of asynchronous multi-agent systems.

2.1.1 Asynchronous Multi-agent System (AMAS)

An asynchronous multi-agent system is essentially a network of automata, where each agent corresponds to a single automaton.

Definition 2.1.1 (Asynchronous Multi-agent Systems [25, 1])**.** *An* asynchronous multi-agent system (AMAS) *consists of* n agents $\mathcal{A} = \{1, \ldots, n\}$*, each associated with a tuple* $A_i = (L_i, \iota_i, Evt_i, P_i, T_i)$*, including:*

- *a set of* local states $L_i = \{l_i^1, l_i^2, \ldots, l_i^{m_i}\}$*;*
- *an* initial state $\iota_i \in L_i$*;*
- *a set of* events $Evt_i = \{e_i^1, e_i^2, \ldots, e_i^{m_i}\}$*;*
- *a* local protocol $P_i \colon L_i \to 2^{Evt_i}$*, which selects the events available at each local state;*
- *a (partial)* local transition function $T_i \subseteq L_i \times Evt_i \times L_i$*, such that* $(l_i, e, l_i') \in T_i$ *for some* $l_i' \in L_i$ *iff* $e \in P_i(l_i)$*.*

Sets Evt_i do not need to be disjoint, that is, an event may be shared by two or more agents. By $Evt = \bigcup_{i \in \mathcal{A}} Evt_i$ and $L = \bigcup_{i \in \mathcal{A}} L_i$ we denote, respectively, the set of all events and the set of all local states. Furthermore, for each event $e \in Evt$, the set $Agt(e) = \{i \in \mathcal{A} \mid e \in Evt_i\}$ has all agents that can perform event e.

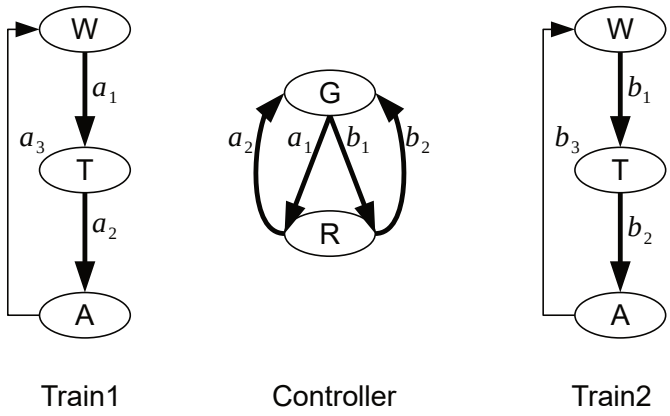


Figure 2.1: Example AMAS: Train-Gate-Controller.

Example 2.1.2 (TGC)**.** *Let TGC_n be an AMAS consisting of n trains $(t_1, \ldots, t_n)$ and the controller c. The trains run on separate circular tracks that jointly pass through a narrow tunnel. Each train can be waiting for the permission to enter (state W), riding inside the tunnel (T), or riding somewhere away of the tunnel (A). The controller switches between green light (state G) and red light (R). Initially, both trains are waiting and the controller displays Green. Figure 2.1 presents this AMAS for $n = 2$ trains (TGC_2).*

2.1.2 Interleaved Interpreted Systems (IIS)

The execution semantics for AMAS is provided by Interleaved Interpreted Systems. Local (private) events are asynchronously interleaved, and agents synchronise on shared events, which are jointly executed by all agents who have them in their protocols.

Definition 2.1.3 (Interleaved Interpreted System [25, 1])**.** *Let $\mathcal{PV}$ be a set of propositional variables. An* interleaved interpreted system (IIS)*, or a* model*, is an AMAS extended with the following elements:*

- *a set of* global states $St \subseteq \prod_{i=1}^{n} L_i$*;*
- *an* initial global state $\iota \in St$*;*
- *a* (partial) global transition function $T \colon St \times Evt \to St$*, such that*
  $T(g_1, e) = g_2$ *iff* $\begin{cases} T_i(g_1^i, e) = g_2^i & \forall i \in Agt(e) \\ g_1^i = g_2^i & \forall i \in \mathcal{A} \backslash Agt(e) \end{cases}$,
  where g_1^i is the i-th local state of g_1;

- *a* valuation function $V\colon St \to 2^{\mathcal{PV}}$.

For state $g = (l_1, \ldots, l_n)$, we denote the local component of agent i by $g^i = l_i$. Also, we will sometimes write $g_1 \xrightarrow{e} g_2$ instead of $T(g_1, e) = g_2$.

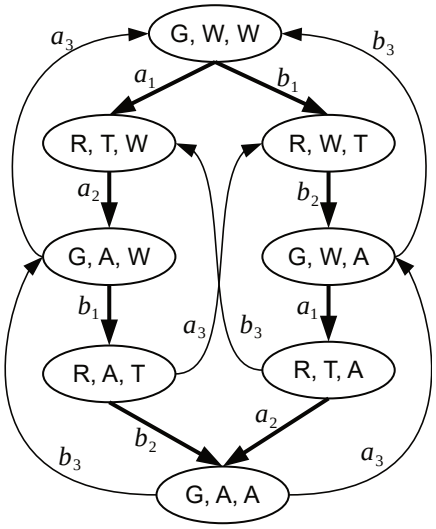


Figure 2.2: IIS for the Train-Gate-Controller AMAS from Example 2.1.2.

Example 2.1.4. *Recall the AMAS from example 2.1.2. Let* $\mathcal{PV} = \{\mathsf{in}_1, \ldots, \mathsf{in}_n\}$ *with* $\mathsf{in}_i \in V(g)$ *iff* $g^i = T$. *That is, proposition* in_i *denotes that train* t_i *is currently in the tunnel. The state/transition structure of the IIS (model) for* TGC_2 *is depicted in Figure 2.2.*

The basic notion in asynchronous execution is that of a *path*, which is a sequence of interleaved global states and events that transition between them. For a number of reasons, we consider only infinite paths. This choice follows a standard approach inherited from *distributed systems*, and will be further elaborated in Section 2.3.

Definition 2.1.5 (Interleaved path)**.** *Let* M *be a model. An* interleaved path*, hereinafter called simply a* path*, is any infinite sequence* $\pi = g_0 e_0 g_1 e_1 g_2 \ldots$ *of interleaving states and events in* M*, such that* $g_i \xrightarrow{e_i} g_{i+1}$ *for every* $i \geqslant 0$.

For path π, we denote its sequence of events by $Evt(\pi) = e_0 e_1 e_2 \ldots$, and refer to its i-th global state as $\pi[i] = g_i$. $\Pi_M(g)$ denotes the set of all paths starting at global state g in M.

2.2 Alternating-time Temporal Logic (ATL*)

Alternating-time Temporal Logic **ATL***, proposed by Alur, Henzinger and Kupferman [23, 21], adds a *strategic modality* on top of the existing formal machinery from purely temporal logics. In that sense, it can be seen as a generalisation of **CTL***, where instead of the two path quantifiers "there exists a path" and "for all paths", we now restrict the temporal reasoning to any subset of paths consistent with a particular strategy of an agent or group of agents.

Definition 2.2.1 (Syntax of **ATL***)**.** *Let* $\mathcal{PV}$ *be a set of propositional variables and let* $\mathcal{A}$ *be the set of all agents. The language of* **ATL*** *is defined by the following grammar (where* $\mathsf{p} \in \mathcal{PV}$ *and* $A \subseteq \mathcal{A}$*):*

$$\varphi ::= \mathsf{p} \mid \neg\varphi \mid \varphi \wedge \varphi \mid \langle\langle A \rangle\rangle \gamma,$$
$$\gamma ::= \varphi \mid \neg\gamma \mid \gamma \wedge \gamma \mid \mathrm{X}\,\gamma \mid \gamma\,\mathrm{U}\,\gamma,$$

where $\langle\!\langle A \rangle\!\rangle \gamma$ stands for "coalition A has a joint strategy to enforce property γ", X for "next", and U for "strong until". The other temporal operators can be obtained as follows: "release" (dual of U) is defined as $\gamma_1 \,\mathrm{R}\, \gamma_2 \equiv \neg((\neg\gamma_1)\,\mathrm{U}\,(\neg\gamma_2))$, "sometime" as $\mathrm{F}\,\gamma \equiv \mathit{true}\,\mathrm{U}\,\gamma$, and "always" as $\mathrm{G}\,\gamma \equiv \mathit{false}\,\mathrm{R}\,\gamma$. Moreover, the **CTL*** operator "for all paths" can be defined as $\mathsf{A}\gamma \equiv \langle\!\langle \varnothing \rangle\!\rangle \gamma$. Boolean connectives, *true* and *false* are defined as usual.

Example 2.2.2. *The following formulas of* **ATL*** *specify interesting properties of the TGC_2 model depicted in Example 2.1.4:*

$\langle\!\langle c \rangle\!\rangle \mathrm{F}\ \mathsf{in}_1$ *(the controller can let train t_1 in),*

$\langle\!\langle c \rangle\!\rangle \mathrm{G}\ \neg\mathsf{in}_1$ *(the controller can keep t_1 out forever),*

$\langle\!\langle c \rangle\!\rangle \mathrm{F}\,(\mathsf{in}_1 \wedge \mathrm{F}\,\neg\mathsf{in}_1)$ *(the controller can let t_1 through),*

$\neg\langle\!\langle t_1, t_2 \rangle\!\rangle \mathrm{F}\,(\mathsf{in}_1 \vee \mathsf{in}_2)$ *(neither train can get in without the help of the controller, even if it collaborates with the other train).*

2.2.1 Strategic ability of agents

Intuitively, the *strategic ability* of an agent i to achieve a given goal (specified by some temporal formula ϕ) is the ability of i to enforce property ϕ in the model no matter what the other agents do. The conditional plan that specifies choices to be taken by i in all possible situations is called a *strategy*.

2.2.2 Taxonomy of strategies

Multiple types of strategies can be defined, depending on factors such as the agents' memory of actions and their knowledge about current state of the model. The taxonomy proposed by Schobbens [26] differentiates between agents with perfect and imperfect *information* about the global state, as well as between *memoryless* agents and those with *perfect recall* of their actions. Thus, it defines four "canonical" strategy types.

Definition 2.2.3 (Types of strategies in AMAS [25, 1])**.** *By $Y \in \{\mathrm{ir}, \mathrm{Ir}, \mathrm{iR}, \mathrm{IR}\}$ we denote the types of strategies in AMAS, which are defined as follows:*

- ir *(imperfect information, imperfect recall). Formally, an* ir*-strategy for agent i is a function $\sigma_i \colon L_i \to Evt_i$ such that $\sigma_i(l) \in P_i(l)$ for each local state $l \in L_i$.*
- Ir *(perfect information, imperfect recall). Formally, an* Ir*-strategy for agent i is a function $\sigma_i \colon St \to Evt_i$ such that $\sigma_i(g) \in P_i(g^i)$ for each global state $g \in St$.*
- iR *(imperfect information, perfect recall). Formally, an* iR*-strategy for agent i is a function $\sigma_i \colon L_i^+ \to Evt_i$ such that $\sigma_i(h^i) \in P_i(last(h^i))$, where $last(h^i)$ denotes the last state of the history h^i.*
- IR *(perfect information, perfect recall). Formally, an* IR*-strategy for agent i is a function $\sigma_i \colon St^+ \to Evt_i$ such that $\sigma_i(h) \in P_i(last(h))$, where $last(h^i)$ denotes the last state of the history h^i.*

We will sometimes use Σ_A^Y to denote the set of all strategies of type Y. The notion of strategic ability is naturally generalised to groups of agents working together to achieve a common goal. Thus, a *joint Y-strategy* of a coalition $A \subset \mathcal{A}$ is simply a tuple of Y-strategies σ_i, one for each agent $i \in A$.

In the following, we will be considering strategies of types ir and iR. The reason for this restriction to imperfect information is simple: the focus of this thesis is on state space reduction methods, and in particular on partial order reduction. As will be shown in Chapter 3, that approach is not applicable in the perfect information setting.

2.2.3 Outcome sets

Each strategy σ considered in some model M is associated with its *outcome set*, that is, the set of all paths in M that are consistent with the strategy. In other words, the outcome set contains all paths that may occur when coalition agents follow their strategy, while the others freely choose events from their protocols.

We first give an auxiliary definition of an *enabled event*.

Definition 2.2.4 (Enabled event [25, 1]). *Let $A = (1, \ldots, m)$, $g \in St$, $e, e' \in Evt$, and let $\vec{e}_A = (e_1, \ldots, e_m)$ be a tuple of events such that every $e_i \in P_i(g^i)$. That is, every e_i can be selected by its respective agent i at state g.*

Event $e \in Evt$ is enabled *at $g \in St$ if $g \xrightarrow{e} g'$ for some $g' \in St$, i.e., $T(g, e) = g'$.*

Event $e' \in Evt$ is enabled by $\vec{e}_A$ *at $g \in St$ iff*

- *for every $i \in Agt(e') \cap A$, we have that $e' = e_i$, and*
- *for every $i \in Agt(e') \backslash A$, we have that $e' \in P_i(g^i)$.*

Thus, e' is enabled by $\vec{e}_A$ if all the agents that "own" e' can choose e' for execution, even when $\vec{e}_A$ has been selected by the coalition A. We denote the set of events enabled at global state g (respectively, enabled at g by a tuple $\vec{e}_A$) by $enabled(g)$ (respectively, $enabled(g, \vec{e}_A)$). Clearly, we have that $enabled(g, \vec{e}_A) \subseteq enabled(g)$.

Now, we can formally define the notion of an outcome of a strategy as follows.

Definition 2.2.5 ((Standard) outcome [2]). *Let $A \subseteq \mathcal{A}$ and $Y \in \{\mathrm{ir}, \mathrm{iR}\}$. The* (standard) outcome *of strategy $\sigma_A \in \Sigma_A^Y$ in state g of model M is the set $out_M^{\mathrm{Std}}(g, \sigma_A) \subseteq \Pi_M(g)$ such that $\pi = g_0 e_0 g_1 e_1 \cdots \in out_M^{\mathrm{Std}}(g, \sigma_A)$ iff $g_0 = g$, and for each $m \geqslant 0$ we have that $e_m \in enabled_M(g_m, \sigma_A(g_m))$.*

Note that the set defined above is referred to as a *standard outcome*, as opposed to its subsets containing only paths that meet certain additional criteria. These restrictions include *concurrency fairness* and related conditions, such as *opponent reactivity*, which will be discussed in subsequent sections.

2.2.4 Concurrency fairness

Wherever concurrency is considered, it is often desirable to ensure that the execution of the system is a *fair* one. For example, the scheduler of an operating system typically does not continuously grant I/O requests to a single process or thread ahead of others, as it would lead to starvation. Similarly, in the context of strategic ability in AMAS, one may wish to exclude paths where some agent's choices (i.e., their enabled events) are consistently ignored in favor of another's.

In order to give the definitions concurrency-fair paths and strategy outcomes, one must first formalise the notions of invisibility and independence of events.

Definition 2.2.6 (Invisible events [1]). *Let M be a model, $A \subseteq \mathcal{A}$ a subset of agents, and $PV \subseteq \mathcal{PV}$ a subset of propositions. An event $e \in Evt$ is* invisible *w.r.t. agents A and propositions PV if $Agent(e) \cap A = \varnothing$ and for each two global states $g, g' \in St$ we have that $g \xrightarrow{e} g'$ implies $V(g) \cap PV = V(g') \cap PV$.*

The set of all invisible events for A, PV is denoted by $Invis_{A,PV}$, and its closure, i.e., the set of visible events, by $Vis_{A,PV} = Evt \backslash Invis_{A,PV}$.

Definition 2.2.7 (Independent events [1]). *Let $e, e' \in Evt$. Events e, e' are* weakly independent *if they satisfy the relation $WI \subseteq Evt \times Evt$, defined as: $WI = \{(e, e') \in Evt \times Evt \mid Agent(e) \cap Agent(e') = \varnothing\}$. Events e, e' are* strongly independent*, or simply* independent*, if they satisfy the relation $I_{A,PV} \subseteq Evt \times Evt$, defined as: $I_{A,PV} = WI \setminus (Vis_{A,PV} \times Vis_{A,PV})$.*

Events e, e' are called *dependent* if $(e, e') \notin I_{A,PV}$. Note that this applies to visible events regardless of whether they are weakly independent.

Definitions 2.2.6 and 2.2.7 will be recalled in Chapter 3, as the concepts of invisible and independent events are essential in obtaining model reductions by determining which states and transitions can be safely pruned from the model. Now, they allow for defining concurrency fairness in the context of paths and outcomes.

Definition 2.2.8 (Concurrency-fair path [1]). *Let M be a model. A path $\pi \in \Pi_M(g)$ satisfies the* concurrency-fairness condition *(**CF**) if there is no event enabled in all states of π from $\pi[i]$ on, and at the same time weakly independent from all the events actually executed in $\pi[i], \pi[i+1], \pi[i+2], \ldots$.*

We denote the set of all concurrency-fair paths starting at global state g of M by $\Pi_M^{\mathrm{Fair}}(g)$.

The *concurrency-fair outcome* of a strategy is then defined as a subset of the standard outcome, restricted only to concurrency-fair paths.

Definition 2.2.9 (Concurrency-fair outcome [1]). *Let $A \subseteq \mathcal{A}$ and $Y \in \{\mathrm{ir}, \mathrm{iR}\}$. The* concurrency-fair outcome *(**CF**-outcome) of strategy $\sigma_A \in \Sigma_A^Y$ is defined as $out_M^{\mathrm{Fair}}(g, \sigma_A) = out_M^{\mathrm{Std}}(g, \sigma_A) \cap \Pi_M^{\mathrm{Fair}}(g)$.*

2.2.5 Semantics

With the above definition of outcome, asynchronous semantics for strategic ability in AMAS can be formally established. We parameterise the satisfaction relation $\models$ with subscript Y denoting the strategy semantics, and the superscript Z, denoting the type of outcome sets. Furthermore, we will be using the parameters Y and Z also with logical formalisms whenever a particular semantics of strategic ability and specific types of outcome sets are considered. For instance, $\mathbf{ATL}^{*\,\mathrm{Std}}_{\mathrm{ir}}$ denotes the logic **ATL*** with memoryless, imperfect information strategies and standard outcome sets, and $\models^{\mathrm{Std}}_{\mathrm{ir}}$ is the satisfaction relation in this semantical setting. Note that Y and Z may be omitted if a statement is universally applicable to all considered semantics.

Definition 2.2.10 (Asynchronous semantics of **ATL*** [1]). *Let $Y \in \{\mathrm{ir}, \mathrm{iR}\}$ and $Z \in \{\mathrm{Std}, \mathrm{Fair}\}$. The asynchronous semantics of* **ATL***, *for strategies of type Y, is defined by the following clauses.*

$M, g \models^Z_Y \mathsf{p}$ *iff* $\mathsf{p} \in V(g)$, *for* $\mathsf{p} \in \mathcal{PV}$;

$M, g \models^Z_Y \neg\varphi$ *iff* $M, g \not\models^Z_Y \varphi$;

$M, g \models^Z_Y \varphi_1 \wedge \varphi_2$ *iff* $M, g \models^Z_Y \varphi_1$ *and* $M, g \models^Z_Y \varphi_2$;

$M, g \models^Z_Y \langle\!\langle A \rangle\!\rangle \gamma$ *iff there is a strategy* $\sigma_A \in \Sigma_A^Y$ *such that* $out_M(g, \sigma_A) \neq \varnothing$ *and, for each path* $\pi \in out_M^Z(g, \sigma_A)$, *we have* $M, \pi \models^Z_Y \gamma$;

$M, \pi \models^Z_Y \varphi$ *iff* $M, \pi[0] \models^Z_Y \varphi$;

$M, \pi \models^Z_Y \neg\gamma$ *iff* $M, \pi \not\models^Z_Y \gamma$;

$M, \pi \models^Z_Y \gamma_1 \wedge \gamma_2$ *iff* $M, \pi \models^Z_Y \gamma_1$ *and* $M, \pi \models^Z_Y \gamma_2$;

$M, \pi \models^Z_Y \mathrm{X}\,\gamma$ *iff* $M, \pi[1, \infty] \models^Z_Y \gamma$;

$M, \pi \models^Z_Y \gamma_1 \,\mathrm{U}\, \gamma_2$ *iff* $M, \pi[i, \infty] \models^Z_Y \gamma_2$ *for some* $i \geqslant 0$ *and* $M, \pi[j, \infty] \models^Z_Y \gamma_1$ *for all* $0 \leqslant j < i$.

The clause $\langle\!\langle A \rangle\!\rangle \gamma$ represents the notion of *objective strategic ability*, in the sense that one, "objective" starting point g is assumed, and it suffices for the formula γ to be satisfied on all outcome paths from that initial state g. However, this is not the only way of defining strategic ability, and in fact, another

variant is often adopted instead. Intuitively, *subjective strategic ability* requires the strategy of coalition A to succeed (that is, γ to be satisfied) not just on all outcome paths starting from g, but also from all states that A might consider as possible "subjective" starting points [27]. Formally:

Definition 2.2.11 (Subjective semantics for strategic ability in **ATL*** [3])**.** *Let $Y \in \{\mathrm{ir}, \mathrm{iR}\}$ and $Z \in \{\mathrm{Std}, \mathrm{Fair}\}$. The subjective semantics of the strategic modality in* **ATL***, *for strategies of type Y, and Z-outcomes, is defined by the following clause.*

$M, g \overset{\mathbf{S}}{\models}{}^{Z}_{Y} \langle\langle A \rangle\rangle \gamma$ *iff there is a strategy $\sigma_A \in \Sigma^Y_A$ such that, for each path $\pi \in \bigcup_{i \in A} \bigcup_{g' \sim_i g} out^Z_M(g', \sigma_A)$, we have $M, \pi \overset{S}{\models}{}^{Z}_{Y} \gamma$.*

We refer the interested reader to [27] for an in-depth discussion and comparison of different variants of strategic ability.

2.3 Handling semantic side effects

As noted before, execution semantics for AMAS provided by IIS only allows infinite paths. This is consistent with the typical approaches in the theory of distributed systems, dating back to asynchronous, parallel automata nets (APA nets) [28] in the early 1980s, as well as models based on process algebras [29, 30, 31, 32]. Furthermore, in keeping with this legacy, the clause for strategic modality in Definition 2.2.10 disregards strategies with empty outcome sets.

However, there are some caveats. At a high level, they ultimately stem from the addition of strategic reasoning on top of existing representation inherited from distributed systems. Strategic ability is, in some aspects, fundamentally different from purely temporal properties. As opposed to concurrent processes that lack *agency*, agents in AMAS are proactive in the sense that they are, in principle, able to select any available strategy. In particular, this includes events that deliberately miscoordinate with other agents, leading to a deadlock, and thus block the execution of the system altogether. While consistent with the notion of agency in strategic play, it is clearly at odds with the aforementioned approaches to concurrency.

On the other hand, modelling many real-world scenarios requires also reactive agents that should always, or at least in some instances, defer to the choice of another. For example, in the model of an electronic voting protocol, such as Selene [3], voters are clearly proactive and able to pick their preferred candidate, while the ballot machine simply accepts their choices. In its current form, AMAS semantics does not address this duality, which can lead to counterintuitive, if not downright paradoxical results, such as the ballot machine having a strategy to force a vote for a particular candidate simply by virtue of refusing to accept any other ballot.

These issues motivate several changes to the AMAS execution semantics, discussed in the rest of this section.

2.3.1 Deadlocks and finite paths

First, we consider the issue of finite paths. In order to retain deadlocked executions while keeping existing formalisms defined for infinite paths only, we introduce special "silent" transitions, denoted by ϵ. More precisely, we define an *undeadlocked IIS*, where the set of all events Evt is augmented with ϵ. This additional event, which does not belong to any particular agent, is added as a loop to each global state where there exists a combination of agents' choices that blocks the system. Formally:

Definition 2.3.1 (Undeadlocked IIS [2])**.** *Let S be an AMAS, and assume that no agent in S has ϵ in its alphabet of events. The* undeadlocked model of S*, denoted $M^\epsilon = IIS^\epsilon(S)$, extends the model $M = IIS(S)$ as follows:*

- $Evt_{M^\epsilon} = Evt_M \cup \{\epsilon\}$, *where* $Agt(\epsilon) = \varnothing$;
- *For each* $g \in St$, *we add a loop* $g \xrightarrow{\epsilon} g$ *iff there is a selection of agents' choices* $\vec{e}_A = (e_1, \ldots, e_k)$, $e_i \in P_i(g)$, *such that* $enabled_M(g, \vec{e}_A) = \varnothing$. *In that case, we also fix* $enabled_{M^\epsilon}(g, \vec{e}_A) = \{\epsilon\}$.

Paths are defined as previously in Definition 2.1.5. The following property is trivially obtained from the definition of undeadlocked IIS.

Proposition 2.3.2. *For any AMAS* S, *any global state* $g \in IIS^\epsilon(S)$, *and any strategy* σ_A, *we have that* $enabled_{IIS^\epsilon(S)}(g, \sigma_A(g)) \neq \varnothing$.

2.3.2 Opponent reactivity

Clearly, coalition agents should always follow their joint strategy, which – recall from Section 2.2.2 – is simply a tuple consisting of one strategy per agent in the coalition, with no further restrictions. Thus, in particular, the joint strategy in question may be a miscoordinated one, where two or more agents disagree on how to proceed in some global state. Adding ϵ loops in undeadlocked IIS addresses the issue of deadlocks and resulting finite paths not being considered in outcome sets of strategies. However, this raises another relevant dilemma, namely, whether or not the opponents (i.e., agents outside the coalition) should also be able to specifically pick events that lead to deadlocks.

At first glance, preventing opposing agents from making certain choices might appear as an artificial, unnecessary restriction. However, giving them complete freedom of choice has a major downside. In many cases, it precludes the verification of relevant strategic formulas, in particular reachability goals, due to the fact there is usually some combination of opponents' actions that blocks further execution, often early on. Practically, this limits the specification to simple safety properties.

On the other hand, there may be situations in which it is desirable to model the behavior of opposing agents in this manner. For instance, one can imagine some attack-defence scenario in which a blocking the system from further execution is effectively a success for the defenders. However, clearly this does not apply to all AMAS, going against our intuition of the modelled reality in many cases.

Proposed in [2], the notion of *opponent reactivity* (**RO**) allows the modeller to explicitly make an assumption about how opposing agents should behave in the system. In that sense, it is clearly an analogous condition to concurrency fairness (cf. Section 2.2.4). Adopting **RO** means restricting outcome sets to paths where opponents are "reactive", i.e. they do not actively choose events that lead to a deadlock. In other words, rather than block the execution, they have to proceed whenever they have the option to do so. Rejecting **RO**, on the other hand, means "proactive" agents that can freely choose events in accordance with their protocols. The notions is formalised as follows:

Definition 2.3.3 (Opponent-reactive path [2])**.** *Let* M^ϵ *be an undeadlocked model, and* $\pi = g_0 e_0 g_1 e_1 \ldots$ *a path in* M^ϵ. π *is* opponent-reactive for strategy σ_A *iff we have that for all* $n \geqslant 0$, $e_n = \epsilon$ *implies* $enabled(g_n, \sigma_A(g_n)) = \{\epsilon\}$. *We denote the set of all such paths starting at global state* g *of* M^ϵ *by* $\Pi^{\mathrm{React}}_{M^\epsilon}(g)$.

Analogously to the case for **CF**, the *opponent-reactive outcome* is a restriction of the standard one to its opponent-reactive paths.

Definition 2.3.4 (Opponent-reactive outcome [2])**.** *Let* M^ϵ *be an undeadlocked model,* $A \subseteq \mathcal{A}$ *and* $Y \in \{\mathrm{ir}, \mathrm{iR}\}$. *The* opponent-reactive outcome (**RO**-outcome) *of strategy* $\sigma_A \in \Sigma_A^Y$ *in global state* g *of* M^ϵ, *is defined as* $out^{\mathrm{React}}_{M^\epsilon}(g, \sigma_A) = out^{\mathrm{Std}}_{M^\epsilon}(g, \sigma_A) \cap \Pi^{\mathrm{React}}_{M^\epsilon}(g)$.

The satisfaction relations $\models^Z_Y$ and ${}^{\mathrm{S}}\models^Z_Y$ for the objective and subjective semantics of strategic ability, respectively, are denoted by fixing $Z = \mathrm{React}$ when assuming the opponent reactivity condition, analogously to the case of standard and concurrency-fair outcomes (see Definitions 2.2.10 and 2.2.11).

2.3.3 Modelling the extent of agents' choice

The opponent reactivity condition offers a convenient way of adjusting how agents outside the coalition should act, depending on a particular model and in accordance with our intuition behind it. However, there remains one outstanding issue it does not address, also related to this choice between proactive and reactive behavior. Clearly, regardless of the considered coalition and the specified properties to verify, in a particular model some agents should always be proactive (e.g. a voter casting a ballot for a candidate of their choice), while some should always be reactive (e.g. the ballot machine acknowledging and counting the vote, but not being able to affect the choice). In order to allow for modelling this duality, another change to the AMAS execution semantics was proposed in [2].

Definition 2.3.5 (AMAS with explicit control [2])**.** *Let S be an AMAS defined as in Definition 2.1.1. To allow for explicit specification of proactive and reactive agents in S, we redefine their protocol functions as $P_i : L_i \to 2^{2^{Evt_i}\setminus\{\varnothing\}}\setminus\{\varnothing\}$. Accordingly, we assume that $T_i(l,e)$ is defined iff $e \in \bigcup P_i(l)$.*[1]

That is, rather than individual events, $P_i(l)$ now lists (nonempty) *subsets of events* $X_1, X_2, \cdots \subseteq Evt_i$, each capturing a choice available for agent i at its local state l. If the agent chooses $X_j = \{e_1, e_2, \ldots\}$, then only an event in that subset can be executed within that local component of the AMAS. However, the agent has no firmer control over which one will be fired.

The previous formulation of AMAS in Definition 2.1.1 (i.e., without explicit control) can now be considered a special case of the above definition, where $P_i(l)$ is always a list of singletons. Thus, whenever we refer to an AMAS in the remainder of this thesis, Definition 2.3.5 is assumed unless specifically stated. In particular, note that adopting AMAS with explicit control does not require any changes in the definitions of IIS and undeadlocked IIS, because the protocols are not actually used to obtain the global states and transitions of the generated model.

Strategies still assign choices to local states. Hence, compared to Definition 2.2.3 the range of functions σ_i is changed from Evt_i to $2^{Evt_i}\setminus\{\varnothing\}$.

Definition 2.3.6 (Strategies in AMAS with explicit control [2])**.** *To match the redefined protocols in AMAS with explicit control, the type of agent i's strategies is changed as follows:*

- (ir) $\sigma_i\colon L_i \to 2^{Evt_i}\setminus\{\varnothing\}$*, such that* $\sigma_i(l) \in P_i(l)$ *for each* $l \in L_i$*.*
- (Ir) $\sigma_i\colon St \to 2^{Evt_i}\setminus\{\varnothing\}$*, such that* $\sigma_i(g) \in P_i(g^i)$ *for each* $g \in St$*.*
- (iR) $\sigma_i\colon L_i^+ \to 2^{Evt_i}\setminus\{\varnothing\}$*, such that* $\sigma_i(h^i) \in P_i(last(h^i))$*.*
- (IR) $\sigma_i\colon St^+ \to 2^{Evt_i}\setminus\{\varnothing\}$*, such that* $\sigma_i(h) \in P_i(last(h))$*.*

Accordingly, we lift the set of events enabled by $\vec{e}_A = (e_1, \ldots, e_m)$ at g to match the new types of protocols and strategies.

Definition 2.3.7 (Enabled events in AMAS with explicit control [2])**.** *Event $e' \in Evt$ is* enabled by $\vec{e}_A$ at $g \in St$ *iff*

- *for every $i \in Agt(e') \cap A$, we have that $e' \in e_i$, and*
- *for every $i \in Agt(e')\setminus A$, we have that $e' \in \bigcup P_i(g^i)$.*

With the above changes in the definitions of strategies and enabled events, the outcome, **CF**-outcome, and **RO**-outcome of σ_A in M, g of an AMAS with explicit control are given as in Definitions 2.2.5, 2.2.9, and 2.3.4, respectively.

[1] For a set of sets X, we use $\bigcup X$ to denote its "flattening" $\bigcup_{x\in X} x$.

2.4 Extending ATL* to epistemic properties

A number of extensions have been proposed and investigated for **ATL***, including a multi-valued variant [33], as well as formalisms augmented with representations of time [34] and knowledge [35, 3], to name a few. In this section, we focus on the latter, i.e., an epistemic extension of **ATL***, denoted by **ATLK***.

In order to introduce knowledge operators $\mathrm{K_i}$ and define semantics for the epistemic modality, we first need to define the notion of global states *indistinguishable* for an agent or a group of agents. It is captured by the following relations:

Definition 2.4.1 (Indistinguishable states [35, 3]). *Let S be an AMAS. For each $i \in \mathcal{A}$, the relation $\sim_i = \{(g, g') \in St \times St \mid g^i = g'^i\}$ denotes that states g, g' are indistinguishable for agent i.*

The relation $\sim_J = \bigcap_{j \in J} \sim_j$ extends this notion to a group of agents $J \subseteq \mathcal{A}$.

By **ATLK***, we denote the extension of **ATL*** with knowledge operators $\mathrm{K_i}$, defined below.

Definition 2.4.2 (Syntax of **ATLK***). *Let $\mathcal{PV}$ be a set of propositional variables and let $\mathcal{A}$ be the set of all agents. The language of* **ATLK*** *is defined by the following grammar (where $\mathsf{p} \in \mathcal{PV}$ and $A \subseteq \mathcal{A}$):*

$$\begin{aligned} \varphi &::= \mathsf{p} \mid \neg\varphi \mid \varphi \wedge \varphi \mid \langle\langle A \rangle\rangle \gamma, \\ \gamma &::= \varphi \mid \neg\gamma \mid \gamma \wedge \gamma \mid \mathrm{X}\,\gamma \mid \gamma\,\mathrm{U}\,\gamma \mid \mathrm{K_i}\,\psi, \\ \psi &::= \mathsf{p} \mid \neg\psi \mid \psi \wedge \psi \mid \mathrm{K_i}\,\psi, \end{aligned}$$

where $\mathrm{K_i}\,\psi$ stands for "agent i knows that ψ", and the other operators are defined as previously for **ATL*** in Definition 2.2.1. Note that with the above syntax, temporal and strategic operators cannot be nested inside ψ, i.e., within the epistemic modality.

The semantics of $\mathrm{K_i}\,\psi$ is formally defined as follows:

Definition 2.4.3 (Semantics of the epistemic modality in **ATLK*** [36, 37]). *The semantics of the epistemic modality in* **ATLK*** *is defined by the following clause.*

$M, g \models_Y \mathrm{K_i}\,\psi$ *iff* $M, g' \models_Y \psi$ *for every* $g' \in St$ *such that* $g' \sim_i g$.

The remaining operators retain the same semantics as in the case for **ATL***, *see Definition 2.2.10.*

Typically, the clause for the epistemic modality refers to every state that is *reachable* from the initial global state of the model (or from any initial state, if multiple ones are defined). Here, this requirement is redundant, as by definition of the transition function in IIS, only reachable states can appear in the model.

2.5 Relevant subsets of ATL* and ATLK*

In the rest of this thesis, rather than on full **ATL***, we will focus on its subset with two important restrictions. Firstly, we do not consider formulas with nested strategic modalities. Furthermore, in order to apply partial order reduction (discussed in the next chapter), we also do not use the next step operator X. This subset is denoted by **sATL***, which stands for "simple **ATL***". Analogously, **sATLK*** is the corresponding subset of **ATLK***. While coming at a cost of reduced expressive power of the language, these restrictions still allow for specifying the vast majority of practically relevant properties. In particular, the ability of an agent to endow others with another ability, or deprive them of it, i.e., the types of properties that require nested strategic modalities to express, are rarely of practical interest [1]. Instead, one typically wants to verify whether an agent or coalition have a strategy to reach a winning state (e.g. $\langle\langle A \rangle\rangle \mathrm{F}\,\mathsf{win}$), or to always avoid a losing one (e.g. $\langle\langle A \rangle\rangle \mathrm{G}\,\neg\mathsf{lose}$), both of which are expressible in **sATL***.

2.6 Summary

In this chapter, we have presented Asynchronous Multi-agent Systems (AMAS), the main formalism that will be considered throughout the rest of this thesis in the context of various techniques for state space reduction. Its execution semantics is provided by Interleaved Interpreted Systems (IIS), with interleaving of private events of local automata, and synchronisation on ones shared by two or more agents. Furthermore, we have introduced the syntax and semantics of strategic-temporal logic **ATL*** (and its subset **sATL***), including their epistemic extensions, which are used to specify relevant properties of AMAS models to be verified by model checking. Because the strategic modality introduces certain nuances not present when considering purely temporal properties, we have also discussed slight modifications to the AMAS formalism, which achieve two goals. Firstly, with minimal changes to the existing formulation, they allow for properly conveying the inherent difference between two types of autonomous agents: proactive ones, who have the initiative and authority to make choices in particular situations, and reactive ones, who wait for the choices of others and defer to them. Secondly, this updated semantics for AMAS remains compatible with the partial order reduction technique, which will be discussed in Chapter 3.

2.6.1 Related work

Asynchronous Multi-agent Systems have been introduced in [25, 1], with semantical issues involving the strategic modality and proposed changes to avoid them extensively discussed in [2]. An application of AMAS to the modelling of a real world voting protocol Selene [3] provides an excellent practical example.

Of course, it should be noted that the asynchronous approach is certainly not new in the theory of concurrent systems. It dates back at least to the early 1980s, with AMAS having been inspired by model such as Asynchronous Parallel Automata Nets (APA Nets), proposed by Priese [28]. However, there is a major difference in the strategic aspect between their approaches. Unlike AMAS, where autonomous agents able to freely choose their strategies are considered, APA Nets were never meant to model proactive agents and their strategic ability, but rather a set of reactive components that eventually converge on some joint behaviour. This also applies to formalisms based on process algebras, including Communicating Sequential Processes (CSP) [29], Calculus of Communicating Systems (CCS) [30], Algebra of Communicating Processes (ACP) [31], and π-calculus [32].

Chapter 3

Partial Order Reduction

The material in Chapter 3 is based on the following papers to which the author of this thesis has contributed:

[1] W. Jamroga, W. Penczek, T. Sidoruk, P. Dembinski, and A. Mazurkiewicz, "Towards Partial Order Reductions for Strategic Ability," *Journal of Artificial Intelligence Research*, vol. 68, pp. 817–850, 2020

[2] W. Jamroga, W. Penczek, and T. Sidoruk, "Strategic Abilities of Asynchronous Agents: Semantic Side Effects and how to tame them," in *Proceedings of KR 2021*, 2021, pp. 368–378

[3] D. Kurpiewski, W. Jamroga, Ł. Maśko, Ł, Mikulski, W. Pazderski, W. Penczek, and T. Sidoruk, "Verification of Multi-Agent Properties in Electronic Voting: A Case Study," in *Proceedings of AiML 2022*. College Publications, 2022, pp. 531–556

The author's involvement in these works includes the formal results regarding partial order reduction (POR) recalled in this chapter, in particular the proof of correctness of POR for **sATL*** in [1], the proof that POR remains applicable under modified AMAS execution semantics with ϵ-transitions in [2], and the proofs that POR can be applied to the epistemic extension of **sATL*** and the subjective semantics of strategic ability in [3]. The thesis adds new, previously unpublished results for POR with perfect recall strategies; only the memoryless semantics was considered in [1, 2, 3]. Additionally, for the paper [1] the author implemented all benchmarks in the Promela modelling language and conducted the experimental evaluation of reduction efficiency using the Spin verifier. Here, the selection of models from [1] has been extended with Faulty TGC.

3.1 Introduction

The explosion of state and transition space is a major challenge in model checking. Asynchronous formalisms like AMAS particularly exacerbate this issue, due to the fact models have to include all possible interleavings of events executed by agents. As a result, the number of states and transitions in the IIS increases exponentially with the number of agents in the AMAS, quickly reaching into the millions and even billions for non-trivial, yet still relatively small examples.

Clearly, the inclusion of all orderings of events is problematic; furthermore, in many models, one can often intuitively realise that it should be sufficient to include just some of them while still preserving all relevant properties of interest and without any loss of generality. For instance, consider the case of an electronic voting protocol like Selene [3]. Suppose there are n voter agents, who can choose one of k candidates. Assuming that each agent's participation in the election is represented as a private event

of its local automaton in the AMAS, the resulting IIS will include multiple paths where the end result is exactly the same (i.e., all candidates received exactly the same number of votes by exactly the same voters), except the ballots have been cast in a different order. From the perspective of verifying an actual voting protocol and its relevant properties, like coercion resistance [38], voter verifiability [39] or receipt freeness [39], it certainly does not matter what the voting order is. More precisely, the formulas specifying these properties do not include the next step operator X , essentially rendering all paths that differ only in the order of votes cast equivalent when verifying these properties.

In this section, we formally define this notion of equivalence and introduce the partial order reductions (POR) algorithm based upon it. We begin with the definition of a reduced model, also called a submodel.

Definition 3.1.1 (Submodel)**.** *Let M, M' be models. M' is a* reduced model *(or* submodel*) of M, denoted $M' \subseteq M$, if M and M' extend the same AMAS, and we have that $St' \subseteq St$, $\iota \in St'$, T is an extension of T', and $V' = V|_{St'}$.*

3.2 Preserving equivalences in reduced models

Historically, partial order reductions for branching time temporal logic **CTL*** (as well as its subsets and epistemic variants, such as **CTLK**) have been defined using *stuttering bisimulation* [40, 35]. For linear time logic **LTL**, the construction was instead based upon *stuttering trace equivalence* [41], and, under the concurrency-fair semantics (cf. Section 2.2.4), on *Mazurkiewicz traces* [42, 43]. Recall that for all of these logics, as already indicated for **sATL*** in the previous section, the next step operator X has to be omitted from the syntax in order to obtain a valid reduction. Otherwise, any formula containing X would be able to distinguish between equivalent paths.

The aforementioned notions of equivalence are not equally discriminative, which directly impacts the practical effectiveness of obtained reduction, i.e., how many paths can be eliminated from the reduced model while preserving satisfaction for all formulas of a given logic. In particular, since **CTL*** is a superset of **LTL**, its higher expressivity requires equivalences to be more discriminative than those for **LTL**. This produces more equivalence classes and in turn preserves more representative paths in the reduced model, resulting in smaller reduction.

Clearly, this means that partial order reductions for the full language of **ATL***, which generalises **CTL***, would require even stricter equivalences and thus yield yet smaller practical gains. This is the main reason that **sATL*** rather than full **ATL*** was investigated in [25]. On the one hand, it remains more expressive than **LTL**, with the strategic modality adding an entirely new class of properties to reason about, and non-nested strategic modalities being sufficient for stating most properties of practical interest and relevance. On the other hand, its distinguishing power is not significantly larger than that of **LTL**: note that **sATL*** merely augments **LTL** with a single strategic modality at the beginning of the formula, and also allows Boolean combinations of such properties.

This suggests the same notions of equivalence as for **LTL**, namely stuttering trace equivalence (without concurrency fairness) and preserving representatives of Mazurkiewicz traces (under the **CF** semantics) could be adapted to **sATL***. This is indeed the case. Furthermore, as we will show in the remainder of this section, it is a relatively direct adaptation. Existing procedures and their implementations developed over the years for **LTL** can be reused in this new setting in a rare case of a "free lunch". Of course, nothing is ever truly free in the world of computational complexity, and in this case it can be conjectured that the equivalences upon which **LTL** reductions are built upon are actually slightly more discriminative than required by the language of **LTL**, and sufficiently enough to also differentiate between formulas of **sATL***.

We now recall the formal definitions of stuttering trace equivalence and Mazurkiewicz traces.

3.2.1 Stuttering trace equivalence

Intuitively, two paths are stuttering equivalent if they can be divided into corresponding finite segments, such that exactly the same propositions are satisfied in each segment. Two models are called stuttering path equivalent if each path in one model has a corresponding stuttering equivalent path in the other.

Definition 3.2.1 (Stuttering equivalence)**.** *Paths $\pi, \pi' \in \Pi_M(g)$ are* stuttering equivalent*, denoted $\pi \equiv_s \pi'$, if there exists a partition $B_0 = (\pi[0], \ldots, \pi[i_1 - 1]),\ B_1 = (\pi[i_1], \ldots, \pi[i_2 - 1]),\ \ldots$ of the states of π, and an analogous partition $B'_0, B'_1, \ldots$ of the states of π', such that for each $j \geqslant 0 : B_j$ and B'_j are nonempty and finite, and $V(g) \cap \mathcal{PV} = V(g') \cap \mathcal{PV}$ for every $g \in B_j$ and $g' \in B'_j$.*

States g and g' are stuttering path equivalent*, denoted $g \equiv_s g'$, iff for every path π starting from g, there is a path π' starting from g' such that $\pi' \equiv_s \pi$, and for every path π' starting from g', there is a path π starting from g such that $\pi \equiv_s \pi'$.*

Models M and $M' \subseteq M$ are stuttering path equivalent*, denoted $M \equiv_s M'$, iff for each path $\pi \in \Pi_M(\iota)$, there is a path $\pi' \in \Pi_{M'}(\iota)$ such that $\pi \equiv_s \pi'$.*

The following theorem connects stuttering equivalence with **LTL**, stating that two stuttering equivalent models satisfy exactly the same formulas of **LTL** without the next step operator X .

Theorem 3.2.2 ([41])**.** *If $M \equiv_s M'$, then, for any* **LTL** *formula φ (without the operator* X *) over $\mathcal{PV}$, we have $M, \iota \models \varphi$ iff $M', \iota' \models \varphi$.*

3.2.2 Mazurkiewicz traces

Although traces were already studied in the late 1960s within a combinatorial context [44], trace theory was first formulated by Mazurkiewicz in the subsequent decade [45, 46, 42]. Since then, it has made a major impact on a number of quite diverse areas, especially those of formal languages and of concurrent systems. In this thesis, we are interested in the latter, where trace theory provides the formal, mathematical framework for reasoning about concurrent processes. Here, we recall the key definitions of finite and infinite traces.

Definition 3.2.3 (Finite traces)**.** *Let Evt^* be the set of finite sequences of events, and let $w, w' \in Evt^*$. We say that $w \sim_I w'$ iff $w = w_1 e e' w_2$ and $w' = w_1 e' e w_2$, for some $w_1, w_2 \in Evt^*$ and $(e, e') \in I_\varnothing$. Let $\equiv_I$ be the reflexive and transitive closure of $\sim_I$. Finite traces are the equivalence classes of the relation $\equiv_I$, denoted by $[w]_{\equiv_I}$, and formally defined as $[w]_{\equiv_I} = \{w' \in Evt^* \mid w' \equiv_I w\}$.*

To define infinite traces, we need additional concepts.

Definition 3.2.4 (Infinite traces)**.** *Let Evt^ω be the set of infinite sequences of events, and let $v, v' \in Evt^\omega$. The relation $\leqslant_I$ is defined as follows:*

$$v \leqslant_I v' \text{ iff } \forall u \in Pref(v) \exists \hat{u} \in Pref(v) \exists u' \in Pref(v')(u \in Pref(\hat{u}) \wedge \hat{u} \equiv_I u'),$$

where $Pref(v)$ denotes the set of the finite prefixes of v.[1] *Moreover, let $v \equiv_I^\omega v'$ iff $v \leqslant_I v'$ and $v' \leqslant_I v$. Infinite traces are the equivalence classes of the relation $\equiv_I^\omega$, denoted by $[v]_{\equiv_I^\omega}$, and formally defined as $[w]_{\equiv_I^\omega} = \{v' \in Evt^\omega \mid v' \equiv_I^\omega v\}$.*

[1] That is, each finite prefix of v can be extended to a permutation (under commuting adjacent independent events) of some prefix of v'.

The following theorem connects equivalences induced by traces with paths in AMAS models. Note that while it originally refers to traces in a *finite state program* in the cited work of Peled [43], we consider sequences of events indiscriminately of agents and strategies. Hence, the theorem remains applicable to IIS as defined in Definition 2.1.3 and to their undeadlocked variant of Definition 2.3.1 (as well as to the extension of IIS that will be introduced in Chapter 5).

Theorem 3.2.5 ([43]). *Let M be a model. If $\pi, \pi' \in \Pi_M(\iota)$ such that $Evt(\pi) \equiv_I^\omega Evt(\pi')$, then $\pi \equiv_s \pi'$.*

Thus, $\mathbf{LTL}_{-\mathrm{X}}$ cannot distinguish between paths over representatives of the same (infinite) trace.

3.3 The POR algorithm

It must be emphasised that a model, reduced or not, is not of interest in itself from a practical standpoint. Rather, we are interested in the results of a *model checking* procedure, i.e. whether some property holds in the model or not. The key aspect of partial order reductions is that the reduction is applied while generating the model from its representation, in this case the network of asynchronous automata comprising an AMAS. In other words, the full model containing all interleavings, which may in fact be too large to be stored in available memory, is never created. Furthermore, it is possible to go one step further and perform *on-the-fly model checking* at the same time as generating the reduced model for maximum efficiency. While this is indeed how partial order reductions are applied in practice, the focus of this thesis is on reductions themselves, so model checking procedures will not be discussed here. However, we refer the interested reader to e.g. [41, 47, 48].

Technically, the model is constructed by systematically exploring the space of global states, starting from the intial one ι, via depth-first search (DFS). Successor global states are obtained by taking all enabled events (see Definition 2.2.4) and following the global transition function accordingly. The reduction consists in skipping some enabled events when generating the model, thereby eliminating their associated transitions and successor states. This subset of enabled events is referred to in the seminal papers on partial order reduction as a *stubborn set*, a *persistent set*, or an *ample set*. In this thesis, we will use the latter term. Clearly, ample sets should be as small as possible in order to maximise model reduction, but at the same time provably sufficient to preserve all properties of the considered logic. Furthermore, the computational complexity of obtaining ample sets is no less important in any practical applications. As it turns out, the exact computation of a minimal ample set is an NP-hard problem [49], therefore calculating it (for each global state!) is practically not feasible. Instead, much more efficient heuristics have been proposed that yield overapproximations. They will be discussed in the next subsection.

In the algorithm, the stack represents a path $\pi = g_0 e_0 g_1 e_1 \cdots g_n$ that is currently being visited. For the top element of the stack g_n, the following four operations are computed in a loop:

1. Identify the set $\mathit{enabled}(g_n) \subseteq Evt$ of enabled events.
2. Heuristically select a subset $E(g_n) \subseteq \mathit{enabled}(g_n)$ of possible events (see Section section 3.3.1).
3. For any event $e \in E(g_n)$, compute the successor state g' such that $g_n \xrightarrow{e} g'$, and add g' to the stack thereby generating the path $\pi' = g_0 e_0 g_1 e_1 \cdots g_n e g'$. Recursively proceed to explore the submodel originating at g' by means of the present algorithm, beginning at step 1.
4. Remove g_n from the stack.

The algorithm begins with the stack comprising of the initial state of the model M of an AMAS, and terminates when the stack is empty. Note that the model generated by the algorithm must be a submodel

of M. Moreover, it is generated directly from the AMAS, without ever generating the full model M. Finally, the size of the reduced model crucially depends on the ratio $E(g)/enabled(g)$. The choice of $E(g)$ is discussed in the next subsection.

3.3.1 Heuristics

It turns out that the heuristics previously defined for **LTL** partial order reductions [43, 41], remain applicable in the new context of verifying properties specified with **sATL***, involving strategic ability of agents. The following three conditions, originally inspired by [41] and used for **sATL*** reductions in [1], are sufficient for selecting ample sets $E(g_n) \subseteq enabled(g_n)$ that yield reduced models preserving all formulas of **sATL***.

C1 Along each path π in M that starts at g, each event that is dependent on an event in $E(g)$ cannot be executed in π unless an event in $E(g)$ is executed first in π. Formally, $\forall \pi \in \Pi_M(g)$ such that $\pi = g_0 e_0 g_1 e_1 \ldots$ with $g_0 = g$, and $\forall e' \in Evt$ such that $(e', e'') \notin I_A$ for some $e'' \in E(g)$, if $e_i = e'$ for some $i \geqslant 0$, then $e_j \in E(g)$ for some $j < i$.

C2 If $E(g) \neq enabled(g)$, then $E(g) \subseteq Invis_A$.

C3 For every cycle in M' there is at least one node g in the cycle for which all the successors of g are expanded, i.e., $E(g) = enabled(g)$.

In case there are multiple subsets of $enabled(g)$ satisfying conditions **C1**-**C3**, any of them can be selected as the ample set.

3.4 Adapting POR for strategic ability

The previous two sections have established the theoretical basis of partial order reductions (Section 3.2) and recalled the POR algorithm for **LTL** (Section 3.3), in particular the heuristic conditions for the choice of ample sets. This section covers the main technical result of this chapter, namely, the adaptation of these reductions from purely temporal **LTL** properties to temporal-strategic formulas of **sATL**$^{*Z}_{Y}$, and even further, to temporal-strategic-epistemic properties specified using **sATLK**$^{*Z}_{Y}$. Correctness of reductions is established in number of semantical settings: assuming the objective or subjective notion of strategic ability, proactive or reactive opponents ($Z \in \{\mathrm{Std}, \mathrm{React}\}$), and memoryless or perfect recall strategies with imperfect information ($Y \in \{\mathrm{ir}, \mathrm{iR}\}$). We also demonstrate why this approach cannot be extended also to agents with perfect information.

3.4.1 sATL* with imperfect information

We state two auxiliary lemmas that will be useful throughout the remainder of this section. The first one concerns the relation between outcome sets of imperfect information (ir and iR) strategies in the full model and its submodel.

Lemma 3.4.1. *Let M' be a submodel of M, and $Y \in \{\mathrm{ir}, \mathrm{iR}\}$. For each Y-joint strategy σ_A we have $out_{M'}(\iota, \sigma_A) = out_M(\iota, \sigma_A) \cap \Pi_{M'}(\iota)$ and $out^{\mathrm{Fair}}_{M'}(\iota, \sigma_A) = out^{\mathrm{Fair}}_{M}(\iota, \sigma_A) \cap \Pi^{\mathrm{Fair}}_{M'}(\iota)$.*

Proof. Note that each Y-strategy in M is also a well defined Y-strategy in M', since ir- as well as iR-strategies are defined on the local states of the AMAS, extended by both M and M' (cf. Definition 3.1.1). Thus, the lemma follows directly from Definitions 2.2.5 and 2.2.9, together with the fact that $\Pi_{M'}(\iota) \subseteq \Pi_M(\iota)$. □

The second lemma says that paths which follow the same sequence of events from agent i's perspective cannot be distinguished by any strategy of i.

Lemma 3.4.2. *Let M be a model, $\pi, \pi' \in \Pi_M(\iota)$, $Y \in \{\mathrm{ir}, \mathrm{iR}\}$, and for some $i \in \mathcal{A}$: $Evt(\pi) \mid_{Evt_i} = Evt(\pi') \mid_{Evt_i}$. Then, for each Y-strategy σ_i, we have that $\pi \in out_M(\iota, \sigma_i)$ iff $\pi' \in out_M(\iota, \sigma_i)$.*

Proof. Let $\pi = g_0 e_0 g_1 e_1 g_2 e_2 \ldots$ and $Evt(\pi) \mid_{Evt_i} = e_{i_0} e_{i_1} e_{i_2} \ldots$ be the sequence of events of agent i in π. This sequence can be either finite or infinite. If it is empty, then the thesis trivially holds. So, assume that $Evt(\pi) \mid_{Evt_i}$ is not empty. Let L be equal to the length of $Evt(\pi) \mid_{Evt_i}$ if $Evt(\pi) \mid_{Evt_i}$ is finite or be equal to ∞ otherwise. For each e_{i_j} let $\pi[e_{i_j}] = \pi[i_j] = g_{i_j}$ denote the global state from which e_{i_j} is executed in π, where $0 \leqslant j < L$.

By induction we can show that for each $0 \leqslant j < L$ we have $\pi[e_{i_j}]^i = \pi'[e_{i_j}]^i$. For $j = 0$ it is easy to note that $\pi[e_{i_0}]^i = \pi'[e_{i_0}]^i = \iota^i$, which follows from the fact that the paths π and π' start at the same global state ι and $Evt(\pi) \mid_{Evt_i} = Evt(\pi') \mid_{Evt_i}$.

Assume that the thesis holds for $j = k$. The induction step follows from the fact the local evolution T_i is a function, so if $\pi[e_{i_k}]^i = \pi'[e_{i_k}]^i = l$ for some local state $l \in L_i$, then $\pi[e_{i_{k+1}}]^i = \pi'[e_{i_{k+1}}]^i = T_i(l, e_{i_k})$.

So, the events of Evt_i are executed from the same local states in π and π', which means that for each ir-strategy σ_i, we have that $e_{i_j} \in \sigma_i(\pi[i_j]^i)$ iff $e_{i_j} \in \sigma_i(\pi'[i_j]^i)$ for $0 \leqslant j < L$.

Furthermore, it means that for each $j \geqslant 0$ such that e_{i_j} is defined, we have $\pi[e_{i_0}]^i .. \pi[e_{i_j}]^i = \pi'[e_{i_0}]^i .. \pi'[e_{i_k}]^i$, so the local histories $h_i, h_i' \in L_i^+$ are also the same in π and π' at each global state from which an event of agent i is executed.

Thus, for each ir-strategy σ_i, we have that $e_{i_j} \in \sigma_i(\pi[i_j]^i)$ iff $e_{i_j} \in \sigma_i(\pi'[i_j]^i)$ for $0 \leqslant j < L$. Analogously, for each iR-strategy σ_i, we have that $e_{i_j} \in \sigma_i(h_i)$ iff $e_{i_j} \in \sigma_i(h_i')$ for $0 \leqslant j < L$.

Consequently, for each Y-strategy σ_i, we have $\pi \in out_M(\iota, \sigma_i)$ iff $\pi' \in out_M(\iota, \sigma_i)$, which concludes the proof. □

The lemma can be easily generalised to joint strategies $\sigma_A \in \Sigma_A^Y$, $Y \in \{\mathrm{ir}, \mathrm{iR}\}$. However, the same property does not hold for perfect information strategies, since they are defined on the global states of the model rather than those of agents' local components. Hence, the state can be changed by any agent's execution of any event.[2]

Definition 3.4.3 (Structural condition $\mathbf{AE}(A)$)**.** *Consider model M and its submodel M'. Let $A \subseteq \mathcal{A}$, $Y \in \{\mathrm{ir}, \mathrm{iR}\}$, and $Z \in \{\mathrm{Std}, \mathrm{React}\}$. By $\boldsymbol{AE}(A)_Y^Z$ we denote the property:*

$$\forall \sigma_A \in \Sigma_A^Y \quad \forall \pi \in out_M^Z(\iota, \sigma_A) \quad \exists \pi' \in out_{M'}^Z(\iota, \sigma_A) : \quad \pi \equiv_s \pi'.$$

In order to prove that model reductions based on stuttering equivalence are applicable to **sATL***, we need to show that firstly, the reduced models obtained using the POR algorithm discussed in Section 3.3 satisfy $\mathbf{AE}(A)$, and secondly, that models satisfying this structural property preserve satisfaction of **sATL*** formulas. We start with the former.

Theorem 3.4.4. *Let $A \subseteq \mathcal{A}$, $Y \in \{\mathrm{ir}, \mathrm{iR}\}$, $Z \in \{\mathrm{Std}, \mathrm{React}\}$, M be a model, and $M' \subseteq M$ be the reduced model generated by DFS with the choice of $E(g')$ for $g' \in St'$ given by conditions* **C1, C2, C3** *and the independence relation I_A. Then, M' satisfies $\boldsymbol{AE}(A)_Y^Z$.*

Proof. Notice that the reduction of M under the conditions **C1, C2, C3** above is equivalent to the reduction of M without the ϵ-loops under the conditions **C1, C2, C3** of [47], and then adding the ϵ-loops to each state of the reduced model where there exists a miscoordinating combination of agents' choices (cf. Section 2.3). Although the setting is slightly different, it can be shown similarly to [41,

[2] For the same reason, Lemma 3.4.2 it is not applicable to any multi-agent formalism with synchronous execution semantics (e.g., iCGS).

Theorem 12] that the conditions **C1, C2, C3** guarantee that the models: (i) M without ϵ-loops and (ii) M' without ϵ-loops are stuttering path equivalent. More precisely, for each path $\pi = g_0 e_0 g_1 e_1 \cdots$ with $g_0 = \iota$ (without ϵ-transitions) in M there is a stuttering equivalent path $\pi' = g'_0 e'_0 g'_1 e'_1 \cdots$ with $g'_0 = \iota$ (without ϵ-transitions) in M' such that $Evt(\pi)|_{Vis_A} = Evt(\pi')|_{Vis_A}$, i.e., π and π' have the same maximal sequence of visible events for A. **(*)**

We will now prove that this implies $M \equiv_s M'$. Removing the ϵ-loops from M eliminates two kinds of paths: (a) paths with infinitely many "proper" events, and (b) paths ending with an infinite sequence of ϵ-transitions. Consider a path π of type (a) from M. Notice that the path π_1, obtained by removing the ϵ-transitions from π, is stuttering-equivalent to π. Moreover, by **(*)**, there exists a path π_2 in M' without ϵ-transitions, which is stuttering-equivalent to π_1. By transitivity of the stuttering equivalence, we have that π_2 is stuttering equivalent to π. Since π_2 must also be a path in M', this concludes this part of the proof.

Consider a path π of type (b) from M, i.e., π ends with an infinite sequence of ϵ-transitions. Let π_1 be the sequence obtained from π after removing ϵ-transitions, and π_2 be any infinite path without ϵ-transitions such that π_1 is its prefix. Then, it follows from **(*)** that there is a stuttering equivalent path $\pi'_2 = g'_0 e'_0 g'_1 e'_1 \cdots$ with $g'_0 = \iota$ in M' such that $Evt(\pi_2)|_{Vis_A} = Evt(\pi'_2)|_{Vis_A}$. Consider the minimal finite prefix π'_1 of π'_2 such that $Evt(\pi'_1)|_{Vis_A} = Evt(\pi_1)|_{Vis_A}$. Clearly, π'_1 is a sequence in M' and can be extended with an infinite number of ϵ-transitions to the path π' in M'. It is easy to see that π and π' are stuttering equivalent.

So far, we have shown that our reduction under the conditions **C1, C2, C3** guarantees that the models M and M' are stuttering path equivalent, and more precisely that for each path $\pi = g_0 e_0 g_1 e_1 \cdots$ with $g_0 = \iota$ in M there is a stuttering equivalent path $\pi' = g'_0 e'_0 g'_1 e'_1 \cdots$ with $g'_0 = \iota$ in M' such that $Evt(\pi)|_{Vis_A} = Evt(\pi')|_{Vis_A}$, i.e., π and π' have the same maximal sequence of visible events for A. To show that M' satisfies $\mathbf{AE}(A)^Z_Y$, consider an Y-joint strategy σ_A and $\pi \in out^Z_M(\iota, \sigma_A)$. As demonstrated above, there is $\pi' \in \Pi_{M'}(\iota)$ such that $\pi \equiv_s \pi'$ and $Evt(\pi)|_{Vis_A} = Evt(\pi')|_{Vis_A}$. Since $Evt_i \subseteq Vis_A$ for each $i \in A$, the same sequence of events of each Evt_i is executed in π and π'. Note that opponent reactiveness only restricts the outcome sets, and not the model itself; hence, the above reasoning applies to $Z = \text{Std}$ as well as $Z = \text{React}$. By the generalization of Lemma 3.4.2 to Y-joint strategies we get $\pi' \in out^Z_M(\iota, \sigma_A)$. Thus, by Lemma 3.4.1 we have $\pi' \in out^Z_{M'}(\iota, \sigma_A)$. □

We now show that the reduced models satisfying $\mathbf{AE}(A)^Z_Y$ preserve $\mathbf{sATL}^{*Z}_Y$ for strategy semantics $Y \in \{\text{ir}, \text{iR}\}$ and outcome types $Z \in \{\text{Std}, \text{React}\}$.

Theorem 3.4.5. *Let $A \subseteq \mathcal{A}$, $Y \in \{\text{ir}, \text{iR}\}$, $Z \in \{\text{Std}, \text{React}\}$, and let model $M' \subseteq M$ satisfy $\boldsymbol{AE}(A)^Z_Y$. For each $\mathbf{sATL}^{*Z}_Y$ formula φ, that refers only to coalitions $\hat{A} \subseteq A$, we have that $M, \iota \models^Z_Y \varphi$ iff $M', \iota' \models^Z_Y \varphi$.*

Proof. Proof by induction on the structure of φ. We show the case $\varphi = \langle\!\langle \hat{A} \rangle\!\rangle \gamma$. The cases for $\neg, \wedge$ are straightforward.

Notice that $out^Z_{M'}(\iota, \sigma_{\hat{A}}) \subseteq out^Z_M(\iota, \sigma_{\hat{A}})$, which together with the condition $\mathbf{AE}(A)^Z_Y$ implies that the sets $out^Z_M(\iota, \sigma_{\hat{A}})$ and $out^Z_{M'}(\iota, \sigma_{\hat{A}})$ are stuttering path equivalent. Hence, the thesis follows from Theorem 3.2.2. □

Together with Theorem 3.4.4, we obtain the following.

Theorem 3.4.6. *Let M be a model, and let $M' \subseteq M$ be the reduced model generated by DFS with the choice of $E(g')$ for $g' \in St'$ given by conditions* **C1, C2, C3** *and the independence relation $I_{A,PV}$. For each $\mathbf{sATL}^{*Z}_Y$ formula φ over PV, that refers only to coalitions $\hat{A} \subseteq A$, we have: $M, \iota \models^Z_Y \varphi$ iff $M', \iota' \models^Z_Y \varphi$, where $Y \in \{\text{ir}, \text{iR}\}$.*

3.4.2 Concurrency-fair sATL* with imperfect information

In Section 3.4.1, we have established a reduction scheme for **sATL*** under imperfect information, for both memoryless (ir) and perfect recall (iR) strategies. This approach, based on stuttering equivalence and the POR algorithm for **LTL**, works for the standard definition of strategy outcome (Definition 2.2.5), as well as for the restricted, opponent-reactive outcome sets (Definition 2.3.4) if the **RO** condition is assumed.

However, this does not easily extend to the notion of concurrency-fairness (**CF**) discussed in Section 2.2.4. Note that unlike opponent reactivity, **CF** does not merely restrict the outcome sets of strategies, but also the set of paths in the model. As a consequence, stuttering equivalence does not preserve **CF**, necessitating a different approach to reductions when concurrency-fair paths are involved. The good news is, Mazurkiewicz traces preserve **CF** in reduced models.

Definition 3.4.7 (Structural condition **AECF**)**.** *Consider model M and its submodel M'. Let $A \subseteq \mathcal{A}$ and $Y \in \{\mathrm{ir}, \mathrm{iR}\}$. By **AECF** we denote the property:*

$$\forall \pi \in \Pi^{\mathrm{Fair}}_{M}(\iota) \quad \exists \pi' \in \Pi^{\mathrm{Fair}}_{M'}(\iota): \quad Evt(\pi) \equiv^{\omega}_{I} Evt(\pi').$$

We first show that each set $out_M(g, \sigma_A)$ is trace-complete in the sense that with each path π such that $Evt(\pi) = w$, it contains a path over any $w' \in [w]_{\equiv^{\omega}_{I}}$.

Lemma 3.4.8. *Let $\pi \in out_M(\iota, \sigma_A)$ and $Evt(\pi) = w$. Then, $\forall w' \in [w]_{\equiv^{\omega}_{I}} \exists \pi' \in out^{\mathrm{Std}}_{M}(\iota, \sigma_A)$ such that $Evt(\pi') = w'$.*

Proof. Let M' be obtained from M by fixing $P_i(l_i) = \{\sigma_i(l_i)\}$ for each $i \in A, l_i \in L_i$, and pruning the transitions accordingly. That is, transitions of agents outside coalition A remain unchanged, while agents in A only keep those consistent with strategy σ_A. Consider the set of paths $\Pi_{M'}(\iota)$. Let w be a sequence of events obtained by traversing M' along some path π, i.e., $w = Evt(\pi)$. Following the inductive reasoning of [43, Theorem 3.3], while reading w, an arbitrary equivalent sequence $w' \in \Pi_{M'}(\iota)$ can be produced. Thus, $\Pi_{M'}(\iota)$ is trace-complete. But, from the construction of M' and in accordance with Definition 2.2.5, we have that $\Pi_{M'}(\iota) = out^{\mathrm{Std}}_{M}(\iota, \sigma_A)$, which ends the proof. □

The above lemma implies the following.

Lemma 3.4.9. *Let M be a model and M' its submodel satisfying the property **AECF**. Then, for each* ir*-strategy σ_A, $\forall \pi \in out^{\mathrm{Fair}}_{M}(\iota, \sigma_A)\ \exists \pi' \in out^{\mathrm{Fair}}_{M'}(\iota, \sigma_A)$ such that $Evt(\pi) \equiv^{\omega}_{I} Evt(\pi')$.*

Proof. Assume that $\pi \in out^{\mathrm{Fair}}_{M}(\iota, \sigma_A)$. Then there is $\pi' \in \Pi^{\mathrm{Fair}}_{M'}(\iota)$ such that $Evt(\pi) \equiv^{\omega}_{I} Evt(\pi')$ (by **AECF**). Since M' is a submodel of M, we have that $\pi' \in \Pi^{\mathrm{Fair}}_{M}(\iota)$. This implies that $\pi' \in out^{\mathrm{Fair}}_{M}(\iota, \sigma_A)$ by Lemma 3.4.8. Since $\pi' \in \Pi^{\mathrm{Fair}}_{M'}(\iota)$ by Definition 2.2.5, we obtain that $\pi' \in out^{\mathrm{Fair}}_{M'}(\iota, \sigma_A)$, which together with the fact that $Evt(\pi) \equiv^{\omega}_{I} Evt(\pi')$ completes the proof. □

Theorem 3.4.10. *Let M be a model and M' its submodel satisfying **AECF**. For each* $\mathbf{sATL}^{*\mathrm{Fair}}_{\mathrm{ir}}$ *formula φ over PV we have:*

$$M, \iota \models^{\mathrm{Fair}}_{\mathrm{ir}} \varphi \quad \textit{iff} \quad M', \iota' \models^{\mathrm{Fair}}_{\mathrm{ir}} \varphi.$$

Proof. Proof by induction on the structure of φ. We show the case $\varphi = \langle\!\langle A \rangle\!\rangle \gamma$. The cases for $\neg, \wedge$ are straightforward.

(⇒) Follows from the fact that for each ir-joint strategy σ_A we have $out^{\mathrm{Fair}}_{M'}(\iota, \sigma_A) = out^{\mathrm{Fair}}_{M}(\iota, \sigma_A) \cap \Pi^{\mathrm{Fair}}_{M'}(\iota)$, so $out^{\mathrm{Fair}}_{M'}(\iota, \sigma_A) \subseteq out^{\mathrm{Fair}}_{M}(\iota, \sigma_A)$.

(⇐) Assume that $M', \iota \models^{\mathrm{Fair}}_{\mathrm{ir}} \langle\!\langle A \rangle\!\rangle \gamma$. From the semantics, there is an ir-joint strategy σ_A such that for each $\pi \in out^{\mathrm{Fair}}_{M'}(\iota, \sigma_A)$ we have $M', \pi \models^{\mathrm{Fair}}_{\mathrm{ir}} \gamma$. In order to prove the thesis, we show that for each $\pi \in out^{\mathrm{Fair}}_{M}(\iota, \sigma_A) \setminus out^{\mathrm{Fair}}_{M'}(\iota, \sigma_A)$ we have $M, \pi \models^{\mathrm{Fair}}_{\mathrm{ir}} \gamma$. It follows from Lemma 3.4.9 and Theorem 3.2.5 that for each $\pi \in out^{\mathrm{Fair}}_{M}(\iota, \sigma_A)$ there is $\pi' \in out^{\mathrm{Fair}}_{M'}(\iota, \sigma_A)$ such that $\pi \equiv_s \pi'$. So, $M', \pi' \models^{\mathrm{Fair}}_{\mathrm{ir}} \gamma$ implies that $M, \pi \models^{\mathrm{Fair}}_{\mathrm{ir}} \gamma$. Thus, we can conclude that $M, \iota \models^{\mathrm{Fair}}_{\mathrm{ir}} \langle\!\langle A \rangle\!\rangle \gamma$. □

3.4.3 sATLK*: handling the epistemic operator

Thus far, we have demonstrated that partial order reductions for **LTL** can be adapted to **sATL***, the subset of **ATL*** without nested strategic modalities. This has several notable implications. Since the existing method for **LTL** is not new, it has already been implemented in model checkers over the years and applied in practice. These tools can now be leveraged in a new setting, that is, for the verification of strategic ability on top of purely temporal properties.

In this section, we will show that this approach can be extended further still to **sATLK***, the epistemic extension of **sATL***. We begin by augmenting the previous definition of stuttering equivalence (Definition 3.2.1) to include the subset of agents J for whom states are indistinguishable.

Definition 3.4.11 (J-stuttering equivalence)**.** *Let $J \subseteq \mathcal{A}$. Paths $\pi, \pi' \in \Pi_M(g)$ are* stuttering equivalent*, denoted $\pi \equiv_s \pi'$, if there exists a partition $B_0 = (\pi[0], \dots, \pi[i_1 - 1]),\ B_1 = (\pi[i_1], \dots, \pi[i_2 - 1]),\ \dots$ of the states of π, and an analogous partition $B'_0, B'_1, \dots$ of the states of π', such that for each $j \geqslant 0 : B_j$ and B'_j are nonempty and finite, and $V(g) \cap PV = V(g') \cap PV$ for every $g \in B_j$ and $g' \in B'_j$.*

*Paths π and π' are J-*stuttering equivalent*, denoted $\pi \equiv_s^J \pi'$, if $\pi \equiv_s \pi'$, and additionally we have that $\forall j > 0 \quad \forall g \in B_j, g' \in B'_j : \ \ g \sim_J g'$.*

*States g and g' are J-*stuttering path equivalent*, denoted $g \equiv_s^J g'$, iff for every path π starting from g, there is a path π' starting from g' such that $\pi' \equiv_s^J \pi$, and for every path π' starting from g', there is a path π starting from g such that $\pi \equiv_s^J \pi'$.*

*Models M and $M' \subseteq M$ are J-*stuttering path equivalent*, denoted $M \equiv_s^J M'$, iff they have the same initial state, and for each path $\pi \in \Pi_M(\iota_i)$, there is a path $\pi' \in \Pi_{M'}(\iota_i)$ such that $\pi \equiv_s^J \pi'$)*

Now, we prove the correctness of reduction, provided that $J \subseteq A$.

Theorem 3.4.12. *Let $J \subseteq A \subseteq \mathcal{A}$, $Y \in \{\mathrm{ir}, \mathrm{iR}\}$, $Z \in \{\mathrm{Std}, \mathrm{React}\}$, and consider model M, and its reduced model $M' \subseteq M$ generated by DFS with the choice of $E(g')$ for $g' \in St'$ given by conditions* **C1-C3***. Then, for any* **sATLK*** *formula φ over PV that refers only to coalitions $\hat{A} \subseteq A$, we have that $M, \iota \models_Y^Z \varphi$ iff $M', \iota \models_Y^Z \varphi$.*

Proof. First, note that conditions **C1-C3** remain unchanged from Section 3.3.1, where they were used in the context of reductions for $\mathbf{sATL}^*{}_Y^Z$. Thus, by Theorem 3.4.4, we have that:

(*) M and M' are stuttering path equivalent. For each path $\pi = g_0 e_0 g_1 e_1 \dots$ with $g_0 = \iota$ in M, there is a stuttering equivalent path $\pi' = g'_0 e'_0 g'_1 e'_1 \dots$ with $g'_0 = \iota$ in M' such that $Evt(\pi)|_{Vis_A} = Evt(\pi')|_{Vis_A}$, i.e., π and π' have the same maximal sequence of visible events for A.

(**) M and M' satisfy structural condition $\mathbf{AE}(A)_Y^Z$.

That is, we have $M, \iota \models_Y^Z \varphi$ iff $M', \iota_i \models_Y^Z \varphi$ for all non-epistemic φ. To extend the reasoning to any $\mathbf{sATLK}^*{}_Y^Z$ formula, we first show that the full and reduced model are also J-stuttering equivalent, which then allows to prove that epistemic subformulas are preserved in the reduced model M'. Finally, we show that these subformulas can be replaced with equivalent new propositions, effectively reducing the problem to the previously proven case for $\mathbf{sATL}^*{}_Y^Z$.

Because $J \subseteq A$ (and so all transitions of the agents in group J are visible), it follows directly from **C2** that if $E(g) \neq \mathit{enabled}(g) \backslash \{\epsilon\}$, then $Agt(\alpha) \cap J = \varnothing$ for any event $\alpha \in E(g)$. This is a direct analogue of the extra condition **CJ** from [35]. Together with (*), this implies that the full and reduced model are also J-stuttering equivalent:

(***) $M \equiv_s^J M'$.

Consider any subformula $\varphi = K_i\psi$. As per the syntax of **sATLK***, temporal operators and strategic modalities cannot be nested inside K_i, so φ is a purely epistemic formula that only contains knowledge operator(s) and propositional variables with Boolean connectives. Now, we will show it follows from (***) that epistemic subformulas are preserved in the reduced model, i.e., for any state g such that $g \equiv_s^J g'$, we have $M, g \models_Y^Z \varphi$ iff $M', g' \models_Y^Z \varphi$:

($\Rightarrow$) Assume that $M, g \models_Y^Z K_i\psi$. Let $St_\psi = \{g_\psi \in St \mid g \sim_i g_\psi\}$, and take g'_ψ such that $g' \sim_i g'_\psi$. We need to show that $M', g'_\psi \models_Y^Z \psi$. From $g \equiv_s^J g'$ and by transitivity of relation $\sim_i$, we have that $g'_\psi \in St_\psi$. So, clearly $M, g'_\psi \models_Y^Z \psi$. As $g_\psi \equiv_s^J g'_\psi$, it follows from the inductive assumption that $M', g'_\psi \models_Y^Z \psi$. Hence, $M', g' \models_Y^Z K_i\psi$.

($\Leftarrow$) Assume that $M', g' \models_Y^Z K_i\psi$. Let $St'_\psi = \{g'_\psi \in St' \mid g' \sim_i g'_\psi\}$, and take g_ψ such that $g \sim_i g_\psi$. We need to show that $M, g_\psi \models_Y^Z \psi$. Consider a path $\pi \in M$ that contains g_ψ. From (***), there is a path $\pi' \in M'$, which contains a state $g''_\psi \in St'$, such that $g_\psi \equiv_s^J g''_\psi$. By transitivity of $\sim_i$, we get that $g''_\psi \in St'_\psi$, and thus $M', g''_\psi \models_Y^Z \psi$. As $g_\psi \equiv_s^J g''_\psi$, it follows from the inductive assumption that $M, g_\psi \models_Y^Z \psi$. Hence, $M, g \models_Y^Z K_i\psi$.

From the above we get that any epistemic subformula $K_i\psi$ holds in the reduced model M' iff it holds in the corresponding state of the (J-stuttering equivalent) full model M. Now, we introduce auxiliary propositional variables to replace epistemic subformulas, including nested ones.

Consider subformulas $\varphi_0, \varphi_1, \ldots$, where $\varphi_0 = \varphi$, and for all $i > 0$, φ_i is an epistemic subformula nested in φ_{i-1}. Note that in the reduced model M', one can add a set of new propositional variables $\mathcal{PV}' = \bigcup_i \{\mathsf{sat}_{\varphi_i}\}$ to $\mathcal{PV}$, and extend the valuation function accordingly, so that we have $V' : St' \to 2^{\mathcal{PV} \cup \mathcal{PV}'}$, and sat_{φ_i} is true in state $g' \in St'$ iff $M', g' \models_Y^Z \varphi_i$. That is, for each epistemic (sub)formula φ_i, a new proposition sat_{φ_i} is added, whose valuation in each state $g' \in St'$ corresponds to the satisfaction of φ_i in that state of M'. Then, for the formula $\varphi' = \mathsf{sat}_{\varphi_0}$, it clearly holds that $M', g' \models_Y^Z \varphi'$ iff $M', g' \models_Y^Z \varphi$. Furthermore, since epistemic subformulas only refer to agents $i \in J$ and we have that $J \subseteq A$, it follows that $Vis_{A,\mathcal{PV}} = Vis_{A,\mathcal{PV} \cup \mathcal{PV}'}$ and $I_{A,\mathcal{PV}} = I_{A,\mathcal{PV} \cup \mathcal{PV}'}$. That is, replacing epistemic subformulas with new propositions in this manner does not affect the visibility or independence of events wrt. A and $\mathcal{PV}$. Hence, we also have $M', g' \models_Y^Z \varphi'$ iff $M, g \models_Y^Z \varphi$, from (*) and (**) and by Theorem 3.4.4, as φ' is a **sATL**$^{*Z}_Y$ formula. But from the construction of φ', we have $M', g' \models_Y^Z \varphi'$ iff $M', g' \models_Y^Z \varphi$, so it also holds that $M, g \models_Y^Z \varphi$ iff $M', g' \models_Y^Z \varphi$ for any **sATLK**$^{*Z}_Y$ formula φ. Thus, in particular we have that $M, \iota \models_Y^Z \varphi \quad \text{iff} \quad M', \iota \models_Y^Z \varphi$. □

3.4.4 POR for subjective strategic ability

All formal results regarding partial order reductions obtained in this chapter for **sATL*** and **sATLK*** have assumed the objective notion of strategic ability and the clause from Definition 2.2.10. We will now show they remain applicable also when subjective strategic ability is adopted, as defined in Definition 2.2.11.

Theorem 3.4.13. *Let $Y \in \{\mathrm{ir}, \mathrm{iR}\}$, $Z \in \{\mathrm{Std}, \mathrm{React}\}$, $\hat{A} \subseteq A$ and $Init_{\hat{A}} = \{\iota\} \cup \{g \in St \mid \exists_{i \in \hat{A}} : g \sim_i \iota\}$. Let M be a model with multiple initial states $Init_{\hat{A}}$, and let M' be the reduction of M generated by POR using conditions **C1-C3** for the choice of ample sets. Then, for any initial state $\iota_i \in Init_{\hat{A}}$ and any* **sATLK**$^{*Z}_Y$ *formula φ that refers only to coalition $\hat{A}$, we have that $M, \iota_i \overset{S}{\models}{}_Y^Z \varphi$ iff $M', \iota_i \overset{S}{\models}{}_Y^Z \varphi$.*

Proof. For each $\iota_i \in Init\hat{A}$, take M_i constructed by DFS starting from ι_i (i.e., with a single initial state), and let M'_i be the reduction of M_i generated by POR.

Take any path $\pi \in \Pi_M$. Clearly, $\pi \in \Pi_{M_i}$ for some $i > 0$. By Theorem 3.4.12, we have $M_i \equiv_s^J M'_i$, so there is a J-stuttering equivalent path $\pi' \in \Pi_{M'_i}$. From the construction by DFS, $\Pi_{M'} = \bigcup_{i \in Init_{\hat{A}}} \Pi_{M'_i}$. Hence, $\pi' \in \Pi_{M'}$, which implies that $M \equiv_s^J M'$. (*)

Take any joint strategy $\sigma_{\hat{A}}$. The subjective outcome of $\sigma_{\hat{A}}$ in M (resp. M') is the sum of objective outcomes of $\sigma_{\hat{A}}$ in M_i (resp. M_i'). But from $\mathbf{AE}_A$, $out^Z_{M_i}(\iota_i, \sigma_{\hat{A}}) \equiv_s out^Z_{M_i'}(\iota_i, \sigma_{\hat{A}})$. So, analogously to the reasoning for (*), it follows that $\bigcup_{i \in Init_{\hat{A}}} out^Z_{M_i}(\iota_i, \sigma_{\hat{A}}) \equiv_s \bigcup_{i \in Init_{\hat{A}}} out^Z_{M_i'}(\iota_i, \sigma_{\hat{A}})$. Hence, M and M' also satisfy $\mathbf{AE}(A)^Z_Y$. (**)

Since (*) and (**), the thesis follows from Theorem 3.4.12, as in the case for objective semantics of strategic ability. □

3.4.5 Counterexample for sATL* with perfect information

So far, we have only considered agents with imperfect information, showing that the reduction algorithm for **LTL** can be adapted to $\mathbf{sATL}^{*Z}_Y$ and $\mathbf{sATLK}^{*Z}_Y$ for memoryless (Y = ir) and perfect recall (Y = iR) strategies in this setting, with (Z = React) and without (Z = Std) assuming the condition of opponent reactivity (**RO**), and for the objective as well as subjective notion of strategic ability. Unfortunately, the same method does not extend to perfect information semantics, i.e. strategies of types Y = Ir and Y = IR. Recall from Definition 2.2.3 that under perfect information, strategies are no longer defined on the local states of agents' components in the AMAS, but on the global states of the model (IIS). Since any reduction of the latter involves pruning some global states and transitions, it intuitively follows that perfect information strategies cannot be preserved in reduced models. This can be demonstrated using a simple counterexample.

Example 3.4.14. *Consider an AMAS composed of two agents* $\{1, 2\}$ *such that:*

- $L_1 = \{l_1^1, l_1^2\}, \quad L_2 = \{l_2^1, l_2^2\},$
- $Evt_1 = \{loop, a\}, \quad Evt_2 = \{loop, b\},$
- $P_1(l_1^1) = \{a, loop\}, \quad P_1(l_1^2) = \{loop\}, \quad P_2(l_2^1) = \{b\}, \quad P_2(l_2^2) = \{loop\},$
- $T_1(l_1^1, a) = l_1^2, \quad T_2(l_2^1, b) = l_2^2.$

Figure 3.1: AMAS for the counterexample in Example 3.4.14.

Let M be the model of the AMAS defined above. Now, consider an Ir*-strategy $\sigma_{\{1,2\}}$ as follows:*

- $\sigma_1(l_1^1, l_2^1) = a, \quad \sigma_1(l_1^1, l_2^2) = \sigma_1(l_1^2, l_2^2) = \epsilon;$
- $\sigma_2(l_1^1, l_2^1) = \sigma_2(l_1^2, l_2^1) = b, \quad \sigma_2(l_1^2, l_2^2) = \epsilon.$

It is easy to see that the outcome set $out^{\mathrm{Ir}}_M((l_1^1, l_2^1), \sigma_{\{1,2\}})$ is not trace-complete. Note that $(a, b) \in I$, but while $out^{\mathrm{Ir}}_M((l_1^1, l_2^1), \sigma_{\{1,2\}})$ contains the path over $ab(loop)^\omega$, it does not contain any path over $ba(loop)^\omega$.

3.5 Experimental evaluation

In this section, we apply in practice model reductions obtained for the verification of strategic and strategic-epistemic formulas in AMAS. To that end, we employ the state-of-the-art verifier Spin, which

```
#define N 2

bool in[N];
chan go = [0] of { pid }
chan leave = [0] of { pid }

active [N] proctype train()
{
W:      go!_pid -> goto T;
T:      leave!_pid -> goto A;
A:      skip -> goto W;
}

active proctype controller()
{
        pid t;
G:      atomic { go?t -> in[t] = true; } -> goto R;
R:      atomic { leave?t -> in[t] = false; } -> goto G;
}
```

Figure 3.2: PROMELA code for the Train-Gate-Controller model.

implements **LTL** partial order reductions. Thus, besides evaluating the efficiency of reduction for **sATL*** and **sATLK***, at the same time we demonstrate a major advantage of our approach, that is, using existing tools, originally intended for linear-time logic, for reasoning about strategic ability of agents. We first introduce SPIN and its input language PROMELA.

3.5.1 The model checker SPIN

Created by Gerard Holzmann, SPIN (Simple PROMELA Interpreter) is an **LTL** model checker. The tool remains in active development since 1980, with the most recent version published in 2020. Its source code is also publicly available.

Over the years, SPIN has been successfully employed in practice for the verification of real-world systems, including mission-critical software for several space missions, e.g. the Cassini [16], Deep Space 1 [17], and Mars Exploration Rover [18] probes. It was also used to investigate Toyota car control software, and to verify transmission protocols in medical devices.

3.5.2 Input language: PROMELA

PROMELA (Process Meta Language) is the input language for SPIN. It allows for modelling concurrent processes that exchange information with one another either asynchronously, via buffered message channels of arbitrary size, or synchronously, via rendez-vous communication, which can be considered a message channel of size 0.

Example 3.5.1. *Recall the Train-Gate-Controller (TGC) AMAS and its IIS presented in the previous chapter (Examples 2.1.2 and 2.1.4). Figure 3.2 shows the* PROMELA *representation of this model. The local variable* `_pid` *and its data type* `pid` *are predefined in* PROMELA *to store the process ID, i.e. the instantiation number of an executed process. The first process with* `_pid = 0` *is always created automatically. Note the synchronised, rendez-vous communication on two channels* `go` *and* `leave`*, where operators* `!` *and* `?` *denote sending and receiving a message, respectively. In this case, the controller synchronises with one of the trains on channel* `go`*, receives its ID, and sets the Boolean variable* in_t *to true accordingly. Analogously, synchronisation occurs on the other channel whenever a train leaves the tunnel.*

Note also that PROMELA allows for easily instantiating any desired number of symmetrical processes using a single template, as in the case of trains in the above example.

3.5.3 Benchmarks

We now introduce the set of benchmarks used to assess the efficiency of **sATL*** partial order reduction in practice, as well as the formulas in the context of which the reduction is considered.

Train-Gate-Controller

The first benchmark is Train-Gate-Controller (TGC), inspired by [50, 51, 37], which was already presented and used as a running example in Chapter 2. In particular, the AMAS for TGC and its unfolding to an IIS were demonstrated in Examples 2.1.2 and 2.1.4, respectively. Furthermore, the PROMELA code for this model was shown in Figure 3.2. We scale the TGC benchmark with the number of trains; recall that the variant of TGC with n trains is denoted by TGC_n. The input formula is $\phi_1 = \langle\langle c \rangle\rangle \mathrm{G}\, \neg(\bigvee_{1 \leqslant i \neq j \leqslant n} (\mathsf{in_i} \wedge \mathsf{in_j}))$, which specifies that the controller c can ensure that different trains $t_1, \ldots, t_n$ are never simultaneously in the tunnel.

Faulty TGC

The second benchmark is Faulty TGC (FTGC), a variant of TGC proposed in [52] and inspired by [50, 53], where one of the trains is faulty, i.e. its communication with the controller is malfunctioning. As such, whenever this particular train enters or leaves the tunnel, rather than jointly execute an event shared with the controller, it may instead select a local one. The behavior of the other trains and the controller remains unchanged, i.e. it is exactly as in TGC. The same applies to the propositional variables in $\mathcal{PV}$ and their valuation. By $FTGC_n$, we denote the instance of FTGC with $n-1$ flawless trains $(t_1, \ldots, t_{n-1})$ and one faulty train t_n. The input formula ϕ_2 is the same as ϕ_1 used in the standard version of TGC, see above.

Pipeline

The third benchmark is Pipeline, adapted from [54]. It features a sequence of n processes $p_1, \ldots, p_n$, each of length m (except for p_1 and p_n, whose length is fixed at 2). Resembling a physical pipeline, they are arranged so that an output transition of p_i is always synchronized with the input transition of p_{i+1}. The input formula is $\phi_3 = \langle\langle p_i \rangle\rangle \mathrm{G}\, (\bigwedge_{1 \leqslant i \leqslant n} (\neg \mathsf{in_i}\, \mathrm{U}\, (\mathsf{out_1} \wedge \cdots \wedge \mathsf{out_{i-1}} \wedge \mathsf{in_i})))$, where p_i is the i-th process in the pipeline and propositions $\mathsf{in_i}, \mathsf{out_i}$ denote, respectively, that p_i has started processing and that it has delivered its output. Thus, ϕ_3 expresses that no process in the pipeline will start operating until the output from every previous one has been delivered.

Asynchronous Simple Voting

The fourth benchmark is Asynchronous Simple Voting (ASV), inspired by the simple model of voting and coercion from [55]. Figure 3.3 depicts the AMAS for this scenario.

There are n voters $v_1, \ldots, v_n$ participating in an election, where they choose from k candidates $1, \ldots, k$. Additionally, a single coercer agent c attempts to make voters select a specific candidate. Each voter has the choice between providing the coercer with a proof (i.e., ballot receipt) of how they voted, or refusing to do so. We denote an AMAS for the scenario by $ASV_{n,k}$. The set of propositional variables is $\mathcal{PV} = \{\mathsf{voted_{1,1}}, \ldots, \mathsf{voted_{n,k}}, \mathsf{revealed_{1,1}}, \ldots, \mathsf{revealed_{n,k}}\}$, where $\mathsf{voted_{i,j}}$ denotes that v_i voted for the j-th candidate, while $\mathsf{revealed_{i,j}}$ denotes that v_i additionally gave the coercer the proof of having voted for j. For the ASV benchmark, we used input formula $\phi_4 = \langle\langle v_i \rangle\rangle \mathrm{F}\, (\mathsf{voted_{i,a}} \wedge \neg \mathsf{revealed_{i,a}} \wedge \mathsf{revealed_{i,b}})$, expressing that voter v_i has a strategy to eventually have voted for candidate a, and have revealed a receipt for a b vote instead.

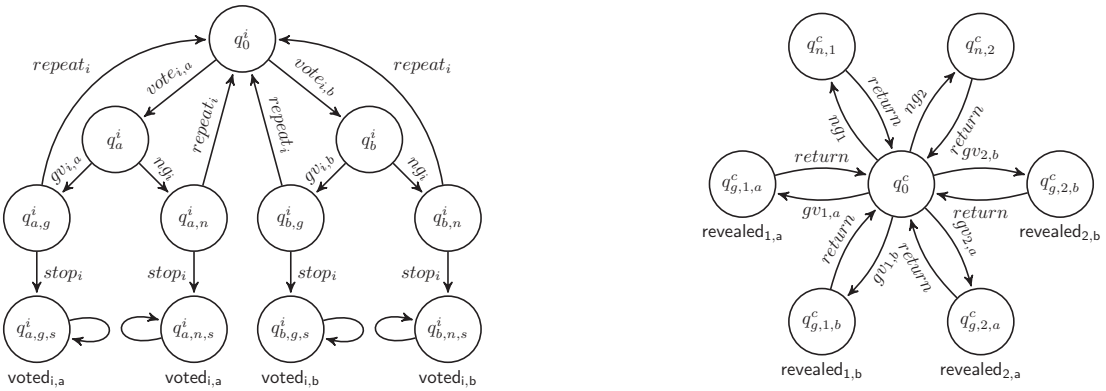


Figure 3.3: Voter v_i (left) and coercer c (right) in the ASV benchmark.

3.5.4 Results

We used the SPIN verifier to generate the state space from each benchmark's PROMELA model. This was done twice for each instance: once with the built-in partial order reduction functionality of SPIN disabled, and once with it enabled. Note that while SPIN is a model checker, we are only interested in evaluating the efficiency of reduction here. Hence, no actual model checking is performed, and the PROMELA models are adapted to the context of formulas ϕ_1, ϕ_2, ϕ_3 beforehand. In particular, any propositions not featured in the considered formula are irrelevant and thus can be safely removed from the model. These experiments also do not aim at measuring the performance of the POR algorithm in SPIN, so we are not reporting running times for specific instances.

Results: TGC

The results for TGC represent the best case scenario for partial order reduction, where the reduced model is exponentially smaller than the full one. This benchmark is perfectly suited for the reduction algorithm, see the comparison of full and reduced model with two trains in Figure 3.4.

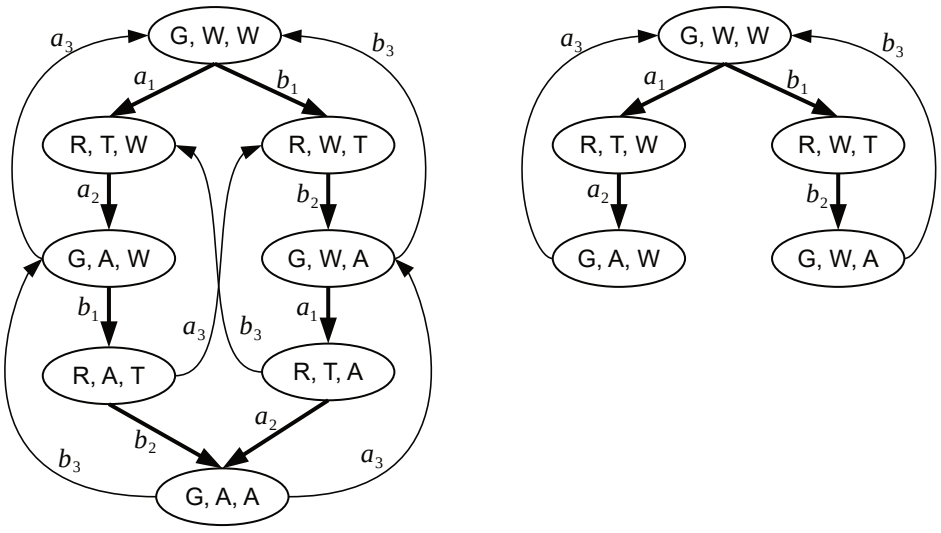


Figure 3.4: TGC_2: full model (left) and reduced model (right). Bold arrows denote visible transitions.

Our experimental results correspond with earlier estimates of the state space size: $|St_{IIS(TGC_n)}| = \mathcal{O}(2^{n+1})$ for the full model, and $|St_{M'_n}| = \mathcal{O}(n)$ for the reduced one [35]. The results are presented in Figure 3.5.

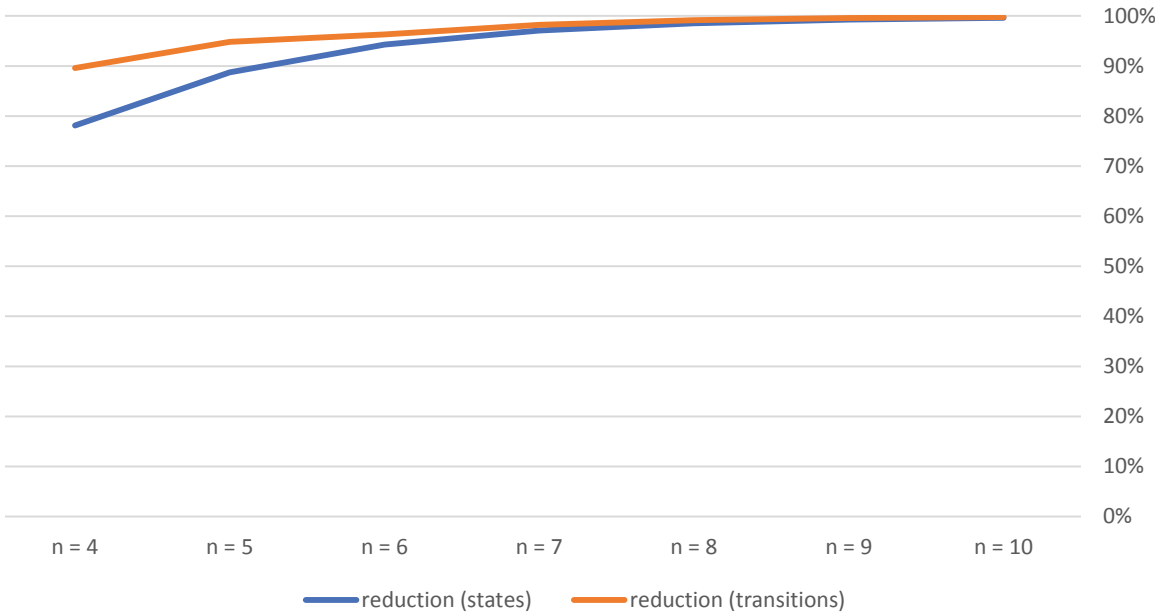


Figure 3.5: Effectiveness of reduction for the TGC benchmark with n trains.

Results: Faulty TGC

Of course, it would be unrealistic to always expect exponential reduction in practice, and indeed, this is no longer the case with FTGC. Nonetheless, the reduction remains significant, particularly for larger instances with more trains. The gains are notably larger in the transition space compared to the state space: we get a 63% reduction of the former versus a 45% reduction of the latter (for $n = 10$). This is good news, as the number of transitions is typically the decisive factor in the complexity of model checking (and it is usually much larger than the number of states). The results are presented in Figure 3.6.

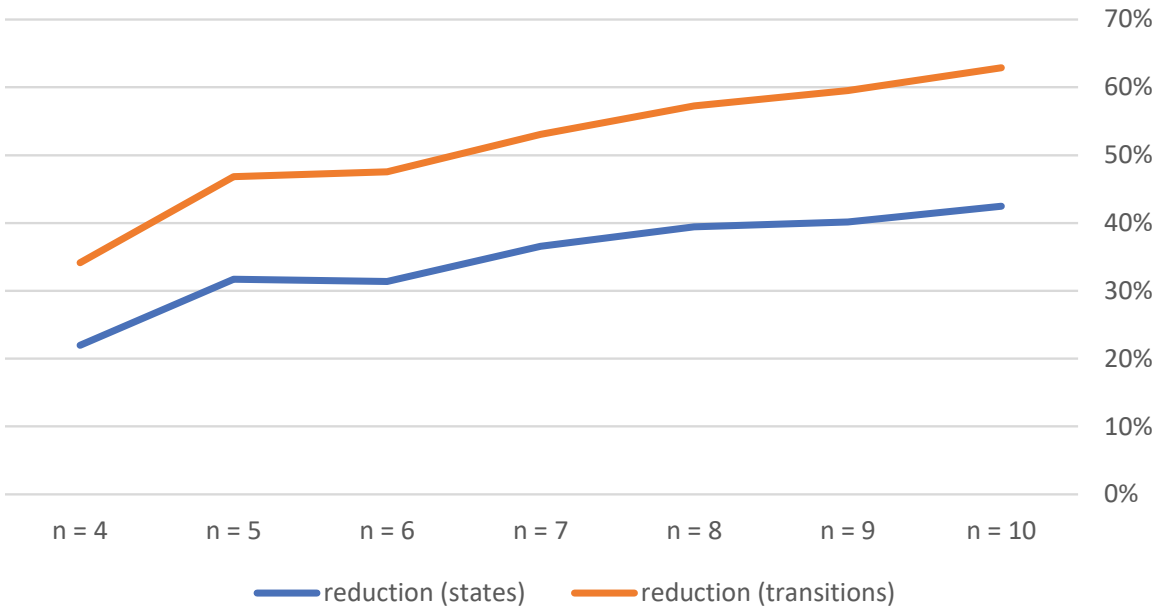


Figure 3.6: Effectiveness of reduction for the FTGC benchmark with n trains.

Results: Pipeline

While not exponential as in the case of FTGC, compared to the latter benchmark the reduction obtained for Pipeline is much higher, exceeding 90 % in larger instances. Furthermore, and no less importantly, its effectiveness actually increases with the size of the model, which is excellent news, given that the larger the state space, the more essential model reductions become in any real-world applications. Additionally, we observe scaling with both parameters here: not only the number of processes n in the pipeline, but also their average length m. The results are presented in Figure 3.7.

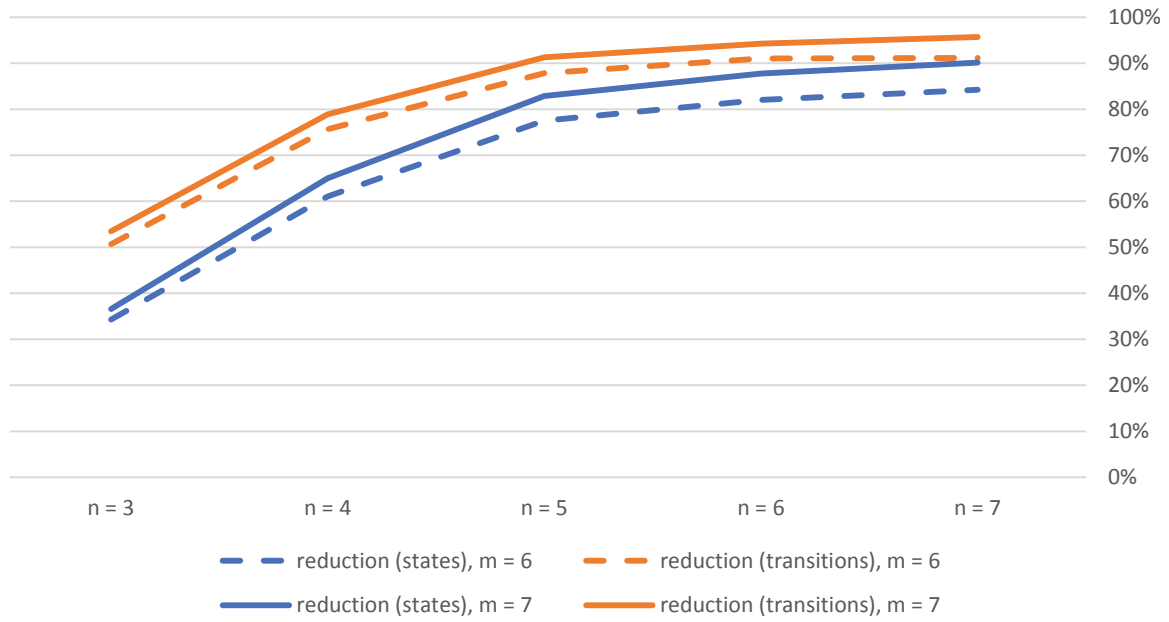


Figure 3.7: Effectiveness of reduction for the Pipeline benchmark with n processes, each of length m.

Results: ASV

Similarly to the previous benchmark, the reduction is not exponential in ASV. Since the local components of voters and the coercer are larger than the processes in Pipeline, in this case we were able to generate the IIS only up to $n = 7$ voters (plus one coercer). Hence, reduction efficiency is also somewhat lower than in the previous benchmark. Nonetheless, it also increases with the size of the model, and thus lends itself well for practical usage. Furthermore, note that the reduction of the transition space is slightly larger than that of the state space. As the former is typically the more important factor in the complexity of model checking strategic ability, this also bodes well for real-world applications. The results are presented in Figure 3.8.

3.6 Summary

In this chapter, we have established a method for partial order reduction in the verification of strategic ability in asynchronous multi-agent systems, for imperfect information strategies in several semantical settings: standard and reactive opponents, agents with and without memory, objective and subjective notion of strategic ability. The approach is a non-trivial adaptation of the existing POR algorithm for linear-time temporal logic **LTL**. As such, the proposed reductions for strategic ability immediately enjoy a number of unique advantages. Despite the higher expressivity of **sATL***, there is no increase in the computational complexity of the POR algorithm, and indeed, the efficiency of obtained reduction remains

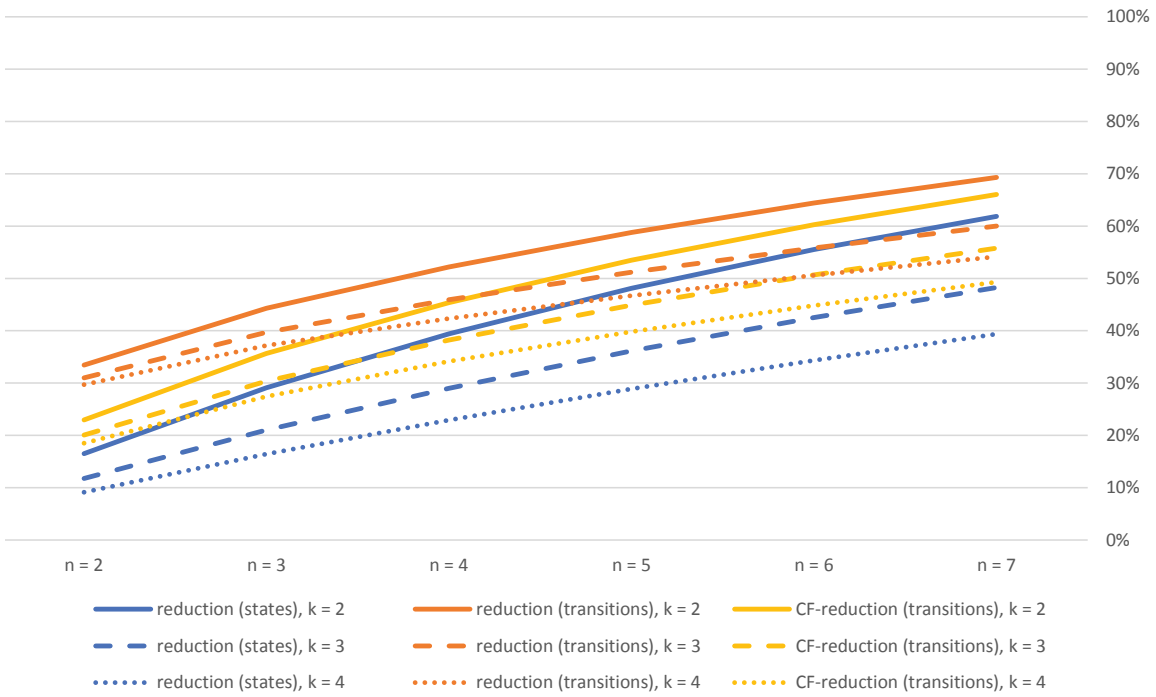


Figure 3.8: Effectiveness of reduction for the ASV benchmark with n voters and k candidates.

high. This curious case of a seemingly "free lunch", which should not exist in computational complexity, can be attributed to existing **LTL** reductions (or more precisely, the equivalences they are based on) being more discriminative than strictly required by **LTL** itself, and enough to also distinguish between formulas of **sATL*** (and even of its epistemic extension **sATLK***). Hence, it would be perhaps more appropriate, if less glamorous, to see it as "leftovers" from a previous meal. Regardless, its consequences are hard to overlook, perhaps to most important of which is that existing tools can be essentially repurposed for strategic verification straight off the shelf. As partial order reductions for **LTL** have been known for over thirty years, these are already mature projects that have been developed and optimised for decades, and have been thoroughly tested by experts in the field of formal verification on real-world systems and protocols. The Spin model checker, used for experimental evaluation of reduction for strategic ability here, is an excellent example, having been developed by Gerard Holzmann since 1998. Adapting the existing method for POR allows for leveraging years of progress in **LTL** verification in entirely new applications. For instance, the verification of an election protocol in electronic voting can involve a broad spectrum of strategic, temporal, and epistemic formulas, from the underlying cryptographic layer to the "social" properties like coercion resistance, where the notions of autonomous agents and strategic play become essential.

3.6.1 Related work

The three seminal papers on partial order reduction for **LTL** are the works of Valmari [56], Peled [54], and Godefroid [57]. Over the years, POR has been defined also for **CTL** [40], and epistemic extensions of linear and branching temporal logics [35].

The main result presented in this chapter, i.e. the adaptation of **LTL** reductions to the verification of strategic ability properties specified in **sATL***, originated in [25, 1], where AMAS and the asynchronous semantics for **ATL*** were also defined. Subsequently, it was demonstrated that this approach remains applicable in the updated AMAS execution semantics with auxiliary ϵ transitions [2], and also for **sATLK***, the epistemic extension of **sATL*** [3]. These are, to the best of our knowledge, the first and only works

that have proposed partial order reductions for strategic ability.

However, the technique of POR itself remains extensively studied today in the context of a broad range of other formalisms and applications. These include recent work on reductions for *reachability games* [58, 59], *parity games* and *parameterised Boolean equation systems* [60].

Dynamic POR (DPOR) [61] used in stateless model checking [62] aims to further reduce the number of interleavings by tracking interactions between concurrent threads or processes at runtime in order to identify backtracking points. DPOR was improved upon in [49], which replaces persistent sets with *source sets* and introduces a new mechanism called *wakeup trees* to achieve optimality, in the sense that only one scheduling per Mazurkiewicz trace is explored. Augmenting optimal DPOR with the notion of *observability* [63], i.e. introducing conditional independence relations based on future "observer" operations, requires some fundamental changes to the reduction algorithm, but in some cases can be superior to the trace-based approach.

A heuristic for detecting and extending stutter-invariant components of Büchi automata during the translation from **LTL** was proposed in [64], allowing for using **LTL** reductions with formulas that are not invariant under stuttering, in particular also those containing the next step operator X.

Chapter 4

Specialised Reductions

The material in Chapter 4 is based on the following papers to which the author of this thesis has contributed:

[4] J. Arias, C. Budde, W. Penczek, L. Petrucci, T. Sidoruk, and M. Stoelinga, "Hackers vs. Security: Attack-Defence Trees as Asynchronous Multi-agent Systems," in *Proceedings of ICFEM 2020*. Springer, 2020, pp. 3–19

[5] L. Petrucci, M. Knapik, W. Penczek, and T. Sidoruk, "Squeezing State Spaces of (Attack-Defence) Trees," in *Proceedings of ICECCS 2019*. IEEE, 2019, pp. 71–80

The author's involvement in these works includes defining the two reduction techniques in the paper [5], as well as proofs of their correctness and a comparative analysis with partial order reduction, all of which were recalled in this chapter. In the paper [4], the author was responsible for a theoretical and experimental study of how the attack times in ADTree scenarios scale with coalition size, beginning the avenue of research continued further in [6] and detailed in the next chapter of this thesis.

4.1 Introduction

Partial order reduction is a powerful technique, one of its major strengths being how universal it is. While the efficiency depends on the specific model, and more precisely, on the visibility of agents' events w.r.t. the considered coalition, the approach can in principle be applied to any asynchronous multi-agent system. However, it is often the case that a particular application of verification involves reduction and model checking of systems that are quite similar to one another, and share some common characteristics that are known beforehand. These traits may be used to define specialised reduction methods that can be expected to yield additional gains, albeit for a significantly smaller class of models. In this section, we will discuss such techniques for security scenarios translated from attack-defence trees (ADTrees) to asynchronous multi-agent systems. Note that these AMAS retain the tree topology of their original representation, so regardless of the specifics of the ADTree, synchronisation between agents always occurs in a very specific way, from the leaves to the root of the tree. This can be leveraged to obtain better model reductions.

4.2 Attack-Defence Trees (ADTrees)

Attack trees are a graphical formalism that represents attack scenarios in a tree structure, intended as a means of modelling potential threats against complex systems, and thus providing means of evaluating

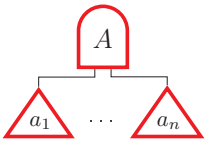
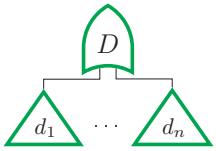
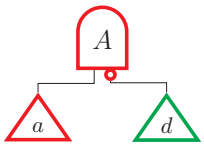
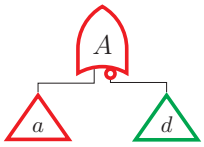
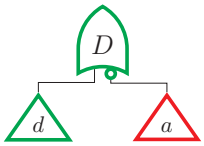
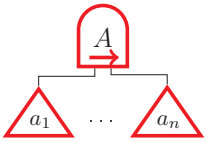
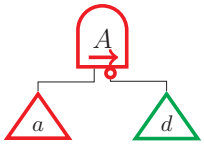
Name	Graphics	Semantics
Attack		$a \equiv$ "basic attack action a done"
Defence		$d \equiv$ "basic defence action d done"
And attack		$A \equiv$ "attacks a_1 through a_n done"
Or defence		$D \equiv$ "one of the defences d_1 through d_n done"
Counter defence		$A \equiv$ "attack a done and defence d not done"
No defence		$A \equiv$ "either attack a done or else defence d not done"
Inhibiting attack		$D \equiv$ "either defence d done or else attack a not done"
Sequential and attack		$A \equiv$ "done attack a_1, then attack a_2, ... then attack a_n"
Failed reactive defence		$A \equiv$ "done attack a and then did not do defence d"

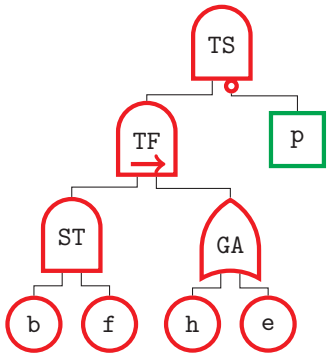
Table 4.1: ADTree constructs and their informal semantics.

their security. The root of an attack tree corresponds to the attackers' overall goal, its child nodes refine it into more specific sub-goals, all the way down to the leaves, which represent individual actions, at the highest level of granularity considered.

Optionally, nodes can be associated with numerical *attributes* denoting additional characteristics of actions, such as their associated financial cost, or the time required to perform them. Boolean functions over attributes, called *conditions*, may then be defined to provide additional flexibility in modelling the requirements for the success or failure of considered actions.

Attack-defence trees (ADTrees) extend this formalism with a representation of the defenders' counteractions, thus providing a way to model the interplay between the two opposing parties. Analogously to attacking actions, defence nodes can also be refined into sub-goals as desired. Table 4.1 lists the graphical representations and informal semantics of basic ADTree constructs.

Example 4.2.1. *Consider the ADTree scenario in Figure 4.1, featuring an attempt to steal a treasure from a museum. To achieve their goal, the thieves must first access the exhibition room, which involves bribing a guard (*b*), and forcing open the secure door (*f*). Both actions are costly and take some time. Once the room has been breached, the attacker(s) can steal the treasure (*ST*), which takes a little time for opening its display stand and putting it in a bag. If the two-thieves coalition is used, we encode in* ST *an extra €90 to hire the second thief—the computation function of the gate can handle this plurality—else* ST *incurs no extra cost. Then, the thieves are ready to flee (*TF*), choosing an escape route to get away (*GA*): this can be a spectacular escape in a helicopter (*h*), or a mundane one via the emergency exit (*e*). The helicopter is expensive but fast, while the emergency exit is slower but at no cost. Furthermore, the time to perform a successful escape could depend on the number of agents involved in the robbery. Again, this can be encoded via computation functions in gate* GA*.*



(a) ADTree

Name	Cost	Time
TS (treasure stolen)		
p (police)	€100	10 min
TF (thieves fleeing)		
ST (steal treasure)	€{0,90}	2 min
b (bribe gatekeeper)	€500	1 h
f (force arm. door)	€100	2 h
GA (get away)		
h (helicopter)	€500	3 min
e (emergency exit)		10 min

Condition for TS**:**

$init\_time(\texttt{p}) > init\_time(\texttt{ST}) + time(\texttt{GA})$

(b) Attributes of nodes

Figure 4.1: Example ADTree: treasure hunters.

As soon as the treasure room is breached (i.e. after b *and* f*, but before* ST*) an alarm goes off at the police station, so while the thieves flee the police hurries to intervene (*p*). The treasure is then successfully stolen iff the thieves have fled and the police failed to arrive or does so too late. This last possibility is captured by the condition associated with the* TS *gate (treasure stolen), which states that the arrival time of the police must be greater than the time for the thieves to steal the treasure and go away.*

4.3 Guarded Update Systems

Guarded Update Systems ($\mathcal{GUS}$) can be thought of as a simplified rendition of EAMAS (defined in the next chapter, cf. Section 5.2.1) that does not cater for agents. They are formally defined as follows.

Definition 4.3.1 (Guarded Update Systems [5]). *A* Guarded Update System *($\mathcal{GUS}$) is a tuple $M = (St, s^0, Acts, \rightarrow)$, including:*

- *a finite set of states St;*
- *an initial state $s^0 \in St$;*
- *a finite set of action names $Acts$;*
- *a transition relation $\rightarrow \subseteq St \times Acts \times G \times U \times St$, where:*
  - *G is a set of* guards*, i.e. boolean formulae over atoms of type $t \sim 0$, where t is a linear term over a finite set of integer variables $Vars$ and $\sim$ a relation such that $\sim \in \{\leqslant, =, \geqslant\}$;*
  - *U is a set of* updates*, i.e. sets of assignments of type $v_j := f(v_0, \ldots, v_k)$, where $\forall_{0 \leqslant i \leqslant k} v_i \in Vars$, $v_j \in Vars$ and f is a function whose domain and codomain are compatible with the domains of its arguments and target; it is assumed that each variable is assigned at most once per update;*

A *valuation* of *Vars* is a function $\omega \colon Vars \rightarrow \mathbb{N}$, and the set of all valuations of *Vars* is denoted by *Vals*. Furthermore, by $u(\omega) \in Vals$ we denote the valuation such that for $v_j \in Vars$:

$$u(\omega)(v_j) = \begin{cases} f(\omega(v_0), \ldots, \omega(v_k)) & \text{if } v_j := f(v_0, \ldots, v_k) \in u \\ \omega(v_j) & \text{otherwise.} \end{cases}$$

By $g(\omega)$ we denote the boolean value of the expression obtained after valuating the variables in g with ω. For brevity, we will write $s \xrightarrow[u]{g,act} s'$ instead of $(s, act, g, u, s') \in \rightarrow$, and also denote $acts(M) = Acts$.

Definition 4.3.2 (Concrete Semantics of $\mathcal{GUS}$ [5]). *Let M be a $\mathcal{GUS}$, and $\omega^0 \in Vals$ be an initial valuation of $Vars$. The* concrete semantics *of M over ω^0 is a tuple $\mathcal{CS}(M, \omega^0) = (CS, w^0, \rightarrow)$, where:*

- *$w^0 = (s^0, \omega^0)$;*
- *$CS = St \times Vals$ is the set of* concrete states*;*
- *$\rightarrow \subseteq CS \times Acts \times CS$ is the transition relation such that $(s, \omega) \xrightarrow{act} (s', \omega')$ iff $s \xrightarrow[u]{g,act} s'$ where $g(\omega)$ is true and $\omega' = u(\omega)$, for some guard g and update u.*

A run is defined as follows.

Definition 4.3.3 (Run [5]). *A* run *$\rho = t_0 act_0 t_1 act_1 \ldots$ is an infinite sequence of alternating concrete states and transitions such that for all $i \in \mathbb{N}$ we have $t_i \xrightarrow{act_i} t_{i+1}$.*

We denote the set of all runs starting from concrete state $t \in CS$ by $Runs(M, t)$. When the starting state is assumed to be initial, we write $Runs(M)$.

4.3.1 Asynchronous product of $\mathcal{GUS}$

The systems considered in this chapter are composed of modules that can share variables and synchronise over common action labels. Hence, we define the asynchronous product of $\mathcal{GUS}$.

Definition 4.3.4 (Asynchronous Product of $\mathcal{GUS}$ [5]). *Consider a $\mathcal{GUS}$ $M_i = (St_i, s_i^0, Acts_i, \rightarrow_i)$, for $i \in \{1, 2\}$. The* asynchronous product *of M_1 and M_2 is the $\mathcal{GUS}$ $M_1 || M_2 = (St_1 \times St_2, (s_1^0, s_2^0), Acts_1 \cup$*

$Acts_2, \rightarrow$), *with the transition rule defined in the usual way as follows:*

$$\frac{act \in Acts_1 \backslash Acts_2 \wedge s_1 \xrightarrow[u]{g,act}_1 s_1'}{(s_1, s_2) \xrightarrow[u]{g,act} (s_1', s_2)}$$

$$\frac{act \in Acts_2 \backslash Acts_1 \wedge s_2 \xrightarrow[u]{g,act}_2 s_2'}{(s_1, s_2) \xrightarrow[u]{g,act} (s_1, s_2')}$$

$$\frac{act \in Acts_1 \cap Acts_2 \wedge s_1 \xrightarrow[u_1]{g_1,act}_1 s_1' \wedge s_2 \xrightarrow[u_2]{g_2,act}_2 s_2'}{(s_1, s_2) \xrightarrow[u_1 \cup u_2]{g_1 \wedge g_2,act} (s_1', s_2')}$$

Note that the last rule above can be applied only if $(u_1 \cup u_2)$ is an update, i.e. each variable is assigned at most once. The above definition is naturally extended to an arbitrary number of components, where we sometimes write $||_{i=0}^{n} M_i$ instead of $M_1 || \ldots || M_n$.

4.3.2 Synchronisation topology

The synchronisation topology is a graph that records how components synchronise with one another.

Definition 4.3.5 ($\mathcal{GUS}$ Synchronisation Topology [5])**.** *The* synchronisation topology *induced by a* $\mathcal{GUS}$ $\mathcal{G} = ||_{i=0}^{n} M_i$ *is the undirected graph* $\mathcal{SG}(\mathcal{G}) = (\{M_i \mid i = 0 \ldots n\}, \mathcal{E})$*, where* $(M_i, M_j) \in \mathcal{E}$ *iff* $i \neq j$ *and* $Acts_i \cap Acts_j \neq \varnothing$*.*

In the following, we assume an asynchronous $\mathcal{GUS}$ product $\mathcal{G} = ||_{i=0}^{n} M_i$ with a tree synchronisation topology.

4.4 Translating ADTrees to $\mathcal{GUS}$

The approach to modelling ADTrees as $\mathcal{GUS}$ is compositional, with specific *transformation patterns* defined for different ADTree constructs. That is, each node of the original ADTree is modelled as a separate automaton, whose pattern corresponds to the node's type. Table 4.2 lists automata patterns for specific types of ADTree constructs. The transformations are symmetrical for attack and defence nodes, so the latter are omitted in the table.

Formally, given an ADTree $\mathcal{T}$ with a set of nodes $\{A_i \mid i = 0 \ldots n\}$, a $\mathcal{GUS}$ M_i is associated with each node A_i, and the associated synchronisation topology $\mathcal{SG}(\mathcal{T})$ is defined by replacing each node A_i of $\mathcal{T}$ with the $\mathcal{GUS}$ M_i.

A run of $||_{i=0}^{n} M_i$ describes a successful execution of $\mathcal{T}$ if it eventually stabilises on an infinite loop of executions of $\mathcal{R}\_ok$, where $\mathcal{R}$ labels the root node of the tree $\mathcal{SG}(\mathcal{T})$.

In other words, successful executions of $\mathcal{T}$ eventually reach the *good* final state. The runs that end with an infinite sequence of $\mathcal{R}\_nok$ are $\mathcal{T}$'s failures, i.e. the *bad* final state can be reached.

The attributes of an ADTree node A_i are modelled by corresponding variables in $\mathcal{GUS}$ M_i, while the conditions added to A_i are represented by guards in M_i. Variables are updated during the synchronisation between a child and its parent node, mimicking how the values of the attributes are actually used. These variables may be used in the guards of the node's ancestors to check whether the actual value that was set satisfies some condition.

In the figures, we employ message sender/recipient markings in a form of additional !/? action prefixes, which is only a syntactic sugar. It allows for an explicit synchronisation between two partners. In the particular case where there is no partner to synchronise with (root of the tree), this is a regular action.

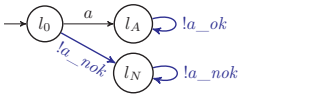
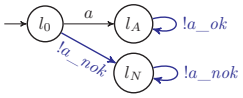
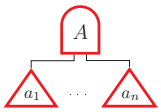
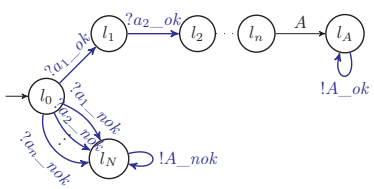
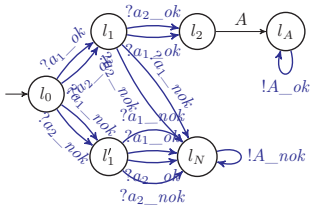
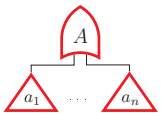
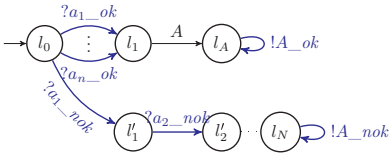
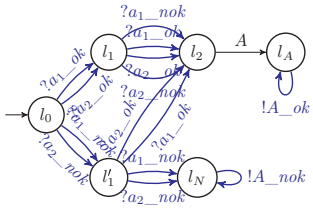
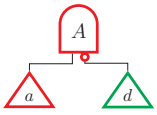
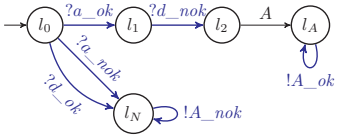
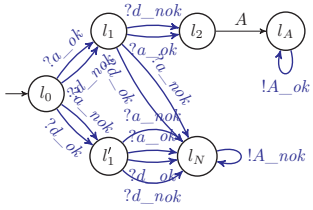
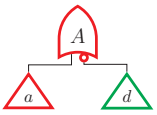
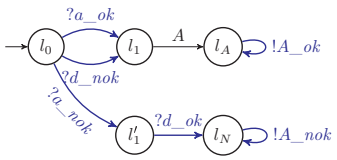
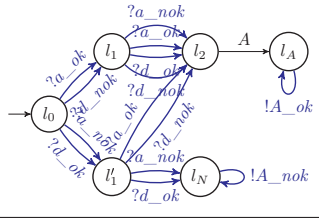
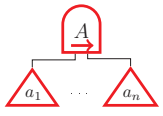
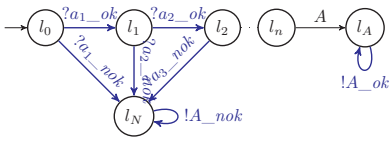
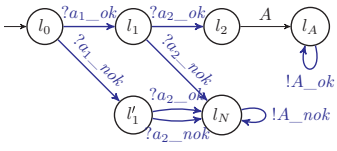
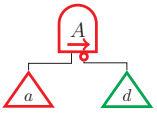
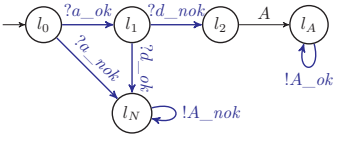
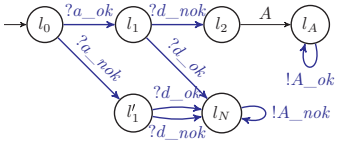
ADTree construct	Reduced Model	Full Model (2 children only)
Leaf node		
		
Conjunction/disjunction nodes		
		
		
Countering nodes		
		
		
Sequential nodes		
		
		

Table 4.2: ADTree nodes and corresponding automata patterns.

Note that the full pattern may be replaced with a reduced one; this will be discussed, alongside other approaches that mitigate state explosion, in the remainder of this chapter.

4.5 Pattern-based reduction

Recall from Section 4.4 the translation of ADTrees to $\mathcal{GUS}$. Note that Table 4.2 lists, in addition to the full automata patterns, also basic reductions that can be used instead. These transformations, called *pattern reductions*, make the concrete semantics of the resulting synchronisation topology amenable, while keeping behaviours necessary to check ADTrees properties.

Essentially, before applying the pattern reduction, nodes can receive all *ok* and *nok* messages from their children in any order, before performing their own actions and sending messages. The reduction consists in providing a predefined selection of appropriate message orders and allows for constructing agent models according to the patterns shown in Table 4.2, while preserving the ADTree properties [4, Theorem 1].

The proofs of the soundness and correctness of this reduction will be presented in the subsequent chapter, in the context of a translation to extended AMAS rather than $\mathcal{GUS}$, using the same automata patterns but representing agents.

Note also that pattern-based reduction is similar, but not identical, in its idea and application to the classical partial order reduction. We expand on this comparison in Section 4.8.

4.6 Layer-based reduction

In this section, we will introduce a model reduction scheme applicable to networks of automata whose topology of synchronisation is a tree. First, within the framework of Guarded Update Systems, we demonstrate the relevant properties of such systems. Then, we demonstrate how these properties of tree synchronisation topologies can be exploited to obtain additional model reductions, preserving reachability. We will refer to this proposed method as *layer-based reduction*.

4.6.1 Properties of tree topologies

We begin by defining the relation of *precedence* between components of a synchronisation topology. Intuitively, a child node precedes its parent if all the synchronisations between them occur before the execution of any other action by the parent node.

Definition 4.6.1 (Precedence [5])**.** *Let M_N be a node, and M_C one of its children.* M_C precedes M_N*, denoted by $M_C \hookrightarrow M_N$, if along each run $\rho \in Runs(\mathcal{G})$ no action from $Acts_N \cap Acts_C$ appears after executing an action $act \in Acts_N \backslash Acts_C$.*

For the layer-based reduction to be applicable, all child nodes must precede their parents. Such synchronisation trees are called *root-directed*.

Definition 4.6.2 (Root-directed Synchronisation Tree [5])**.** *A synchronisation tree $\mathcal{SG}(\mathcal{G})$ is* root-directed *if, for each node M_N and any of its children M_C, we have $M_C \hookrightarrow M_N$.*

The second necessary condition for layer-based reduction concerns variable updates. Specifically, runs in the product of a subtree cannot depend on updates in disjoint subtrees. We formalise this notion as *update-separability*.

Definition 4.6.3 (Update-separability [5])**.** *Let $\mathcal{SG}(\mathcal{G})$ be a root-directed synchronisation tree. $\mathcal{SG}(\mathcal{G})$ is* update-separable *if for each $v \in Vars$ the following conditions hold:*

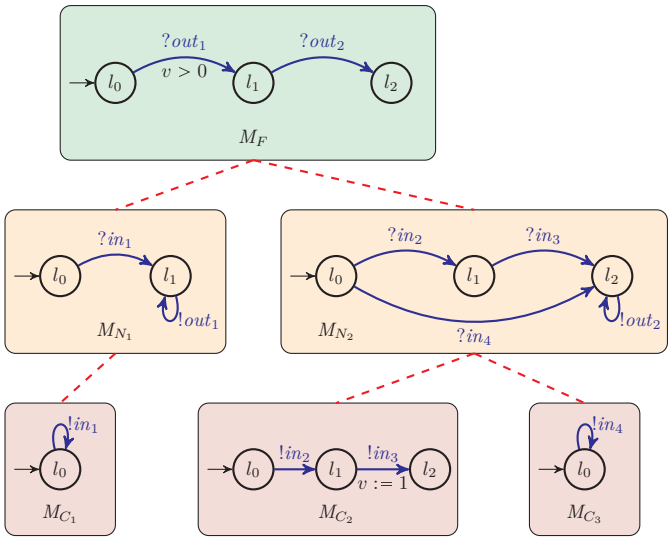


Figure 4.2: Example of a $\mathcal{GUS}$ with a tree synchronisation topology.

- *v is updated in at most one component M_v;*
- *v is tested only in guards of the ancestors of node M_v in the tree $\mathcal{SG}(\mathcal{G})$.*

Example 4.6.4. *Figure 4.2 presents a simple tree synchronisation topology with components M_F, its two children M_{N_1} and M_{N_2}, the only child M_{C_1} of M_{N_1}, and two children M_{C_2}, M_{C_3} of M_{N_2}. Note that the labeling convention of the sender's and the recipient's actions with ! and ?, respectively, only serves as "syntactic sugar". It is easy to see that the tree is root-directed. Furthermore, it is also update-separable, since the variable v is only updated at leaf M_{C_2} and read at the root M_F.*

Subtrees are defined in a straightforward manner, as being rooted in some node of the full synchronisation topology and containing only the descendants of that component. Furthermore, we formalise the notion of run *projections* on subtrees.

Definition 4.6.5 (Subtree [5])**.** *Let M_i be a node of $\mathcal{SG}(\mathcal{G})$. The* subtree *of $\mathcal{SG}(\mathcal{G})$ rooted in M_i, denoted by $\Downarrow M_i$, is the tree containing M_i and all its descendants.*

Intuitively, the projection of a run ρ on a subtree $\Downarrow M_i$, denotedby $\rho_{\Downarrow M_i}$, is obtained by keeping from ρ only the locations, transitions and variables belonging to the nodes in $\Downarrow M_i$.

Definition 4.6.6 (Projections on Subtrees [5])**.** *Let $\mathcal{SG}(\mathcal{G}) = (\mathcal{G}, \mathcal{E})$ be a synchronisation tree, M_i be a node of $\mathcal{SG}(\mathcal{G})$, $\Downarrow M_i$ a subtree rooted in M_i, and let $\rho = t_0 act_0 t_1 act_1 \ldots$ be a run in $Runs(\mathcal{G})$.*

The projection of ρ on $\Downarrow M_i$, denoted by $\rho_{\Downarrow M_i}$, is obtained by:

1. *retaining in each concrete state $t_j, j \in \mathbb{N}$ only its projection (states and variables) on $\Downarrow M_i$;*
2. *keeping only the transitions in the nodes of $\Downarrow M_i$.*

Note that for any action not in the subtree, the projected source and target states are identical, and thus these actions are safely removed from the projected run.

Lemma 4.6.7 ([5])**.** *Let $\mathcal{SG}(\mathcal{G})$ be a root-directed, update-separable tree. Let M_i be a node and $\rho \in Runs(\mathcal{G})$ be a run. Then, $\rho_{\Downarrow M_i}$ is a prefix of some run $\rho' \in Runs(\Downarrow M_i)$.*

Proof. It suffices to observe that by definition of update-separability the variables tested in $\rho_{\Downarrow M_i}$ are updated only in $\Downarrow M_i$. □

We now demonstrate that the synchronisation topologies induced by attack-defence trees are both root-directed and update-separable.

Lemma 4.6.8 ([5]). *ADTrees topologies are root-directed and update-separable.*

Proof. The root-directed property follows from the automata patterns used in the translation (cf. Table 4.2). Note that it holds for all nodes of the full patterns that synchronisations with children occurs before execution of local actions and before synchronisation with the parent. Since pattern-based reduction only prunes states and transitions without adding any new ones, clearly the same applies also to reduced patterns.

The update-separability property follows from the fact variables are only updated when synchronising with children and tested by ancestors. Analogously, this applies to full as well as reduced automata patterns. □

Having established that all necessary properties hold for $\mathcal{GUS}$ synchronisation topologies obtained by translation from ADTrees using either the full or reduced automata patterns depicted in Table 4.2, we can now outline the layer-based reduction itself.

4.6.2 Layered reduction at a single depth

In the layered construction, we consider specifically the last synchronisation of node M_N with one of its children M_C, before any other action in M_N. Let $\#_{child}(M_N)$ be the number of children of node M_N.

Definition 4.6.9 (Last synchronisations with child nodes [5]). *By the last synchronisations of M_N with its children we mean the transitions (denoted by lst), that are synchronising transitions of M_N and one of its children M_{C_j}, such that there are states s_i, s_{i+1}, s_{i+2} and another transition t of M_N which does not synchronise with any transition of its children M_{C_j}, with s_i lst s_{i+1} t s_{i+2}. The set of these transitions is denoted by $Lst_C(M_N)$.*

The reduction is applied to a single "layer", corresponding to all nodes at the same depth of the tree $\mathcal{SG}(\mathcal{G})$, as follows. Let us fix a depth $d > 0$ of $\mathcal{SG}(\mathcal{G})$. We add a fresh variable v_d, initialised with 0, which counts the total number of synchronisations between the nodes at depth d and the nodes at depth $d+1$.

We modify each node M_N at depth d by adding to the update u of any transition in $Lst_C(M_N)$ a new element $v_d := v_d + \#_{child}(M_N)$. It is a way for node M_N to notify that it has performed all synchronisations with its children.

The total number of the children of the nodes at depth d is $\#_{child}(d) = \#_{child}(M_{N_1}) + \ldots + \#_{child}(M_{N_k})$, where $\{M_{N_1}, \ldots, M_{N_k}\}$ are all the nodes at depth d in $\mathcal{SG}(\mathcal{G})$.

In the next step of the construction, we also modify each node M_N at depth d by extending the guard g of each transition t of M_N which does not synchronise with any transition of the children of M_N to $g \wedge (v_d = \#_{child}(d))$. Intuitively, this prevents any action at depth d before all synchronisations with the children are finished.

4.6.3 Layered reduction for the entire tree

In order to obtain the final result, denoted by $\mathcal{SG}^{lr}(\mathcal{G})$, the above transformation is performed for each depth $0 < d <$ height of $\mathcal{SG}(\mathcal{G})$.

Example 4.6.10 ([5]). *Let us revisit Example 4.6.4 with the layered reduction scheme. The resulting tree is shown in Figure 4.3. To that end, we add a new variable v_1 used at depth 1 in the tree. Note that in the transformed model nodes M_1 and M_2 must wait until they both fully synchronise with their children before they can synchronise with M_F.*

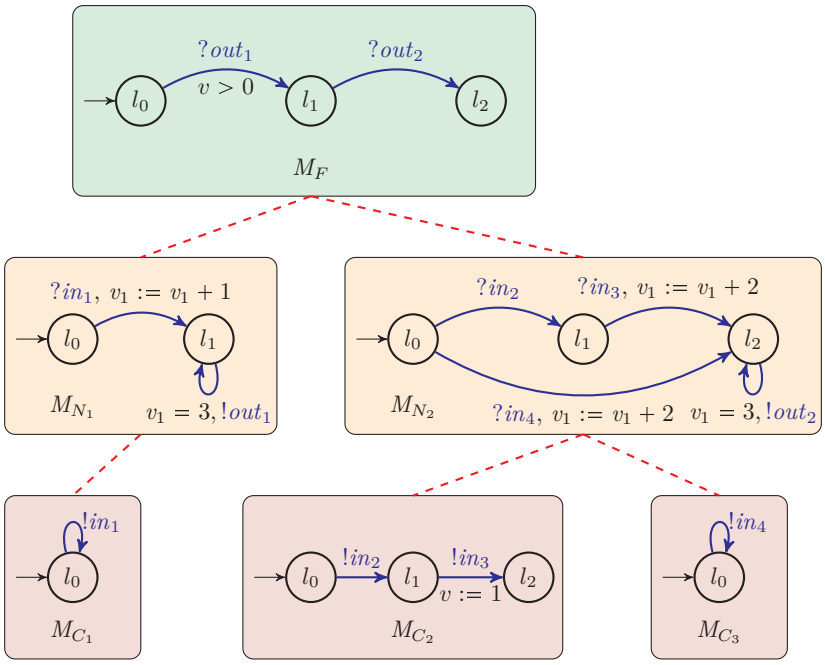


Figure 4.3: Example of layered reduction.

The following proposition states that the technique of layered reductions preserves reachability.

Proposition 4.6.11 (Layer-based reduction preserves reachability [5])**.** *Let $\mathcal{SG}(\mathcal{G})$ be a root-directed, update-separable tree. A good (bad) final state s is reachable in $\mathcal{SG}(\mathcal{G})$ iff it is reachable in $\mathcal{SG}^{lr}(\mathcal{G})$.*

Proof. It follows from Lemma 4.6.7 and the construction of $\mathcal{SG}^{lr}(\mathcal{G})$. □

Example 4.6.12 ([5])**.** *Figure 4.4 shows the state space of the $\mathcal{GUS}$ from Figure 4.3. The grey nodes and edges appear in the full state space but not in the layered reduction.*

4.7 Experimental evaluation

Two sets of experiments were conducted to assess the efficiency of the pattern- and layer- based reduction schemes.

4.7.1 Literature case studies

The first set of experiments involves three ADTree case studies from [4]. These ADTrees model specific security scenarios, some of them based on real-world occurrences, the downside being that they are not easily scalable as a result. Hence, in the second set of benchmarks we add a scalable ADTree, see Section 4.7.3.

Forestalling a software release (forestall)

Based on a real-world instance [65], this case study models an attack on the intellectual property of a company C, perpetrated by an unlawful competitor company U aiming to use the stolen code to release their own software product first, thereby gaining a significant market advantage. Following [66], software extraction from C must take place *before* U builds it into its own product, and U must furthermore deploy to market before C, which takes place after U has integrated the stolen software into its product. The ADTree and node attributes for forestall are presented in Figure 4.5.

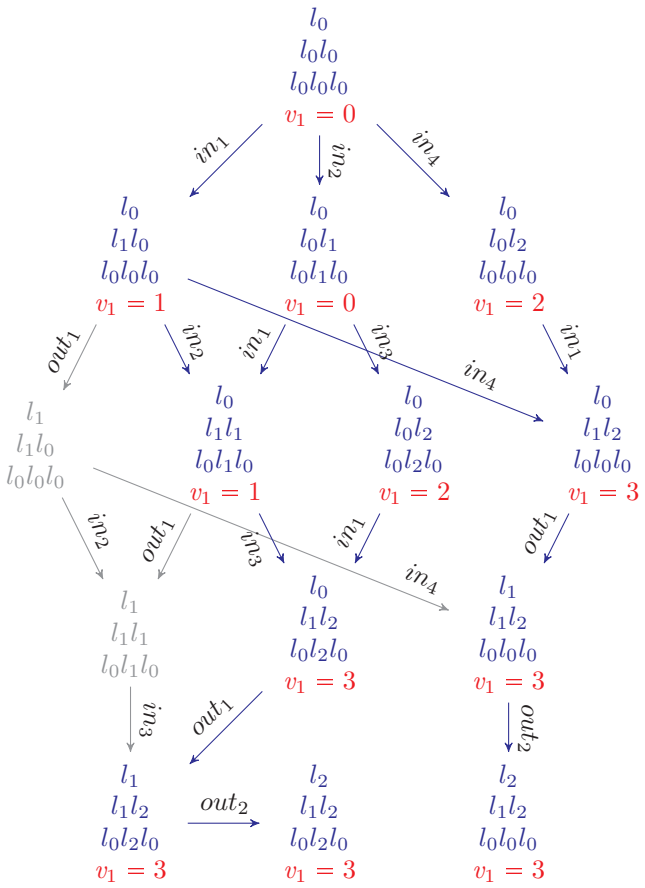


Figure 4.4: State space for the $\mathcal{GUS}$ from Figure 4.3: full (14 states, 19 edges) vs. layer-based reduction (12 states, 14 edges).

Compromising an IoT device (iot-dev)

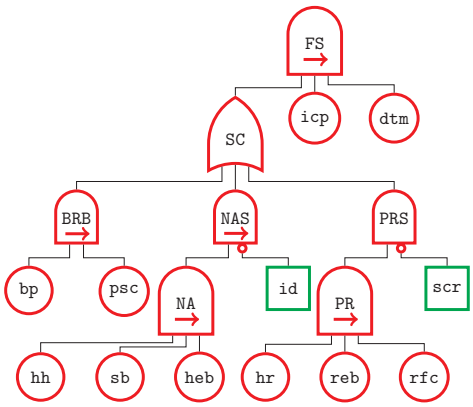
A LAN-based attack against an Internet of Things (IoT) device, originally from [67], later extended in [68], and further enriched in [4] with the addition of defence mechanisms. The attacker, having gained access to the private network (either wireless or wired LAN) and acquired the corresponding access credentials, can execute a malicious script by exploiting software vulnerabilities in the IoT device. The ADTree and node attributes for iot-dev are presented in Figure 4.6.

Obtaining admin privileges (gain-admin)

A well-known case study in the literature [69, 70, 71, 66], modelling an attempt to gain administrative privileges on a UNIX system command line interface. This requires either physical access to an already logged-in console or attacking the SysAdmin to gain remote access via privilege escalation. We note that unlike the previous two case studies, the structure of gain-admin is mostly branching, with all but one gate in the original ADTree from [69] being disjunctions. The scenario was extended in [4], firstly with the SAND gate (following [66]), and secondly, with the addition of reactive defences for certain attacks. The ADTree and node attributes for gain-admin are presented in Figure 4.7.

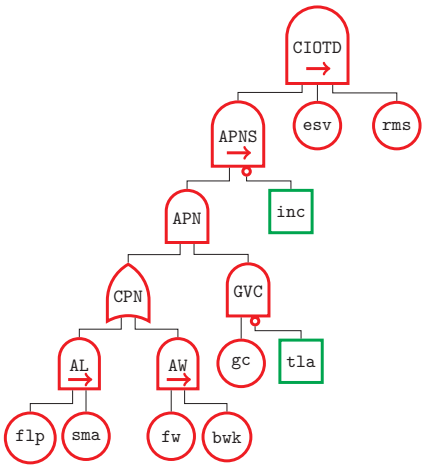
4.7.2 Case studies: results

The results are summarised in Table 4.3. For each case study, it lists the number of states, the number of transitions in the state space and its generation time in four cases:



Name	Cost	Time
FS : forestalling of softw.	€0	10 d
SC : steal code	€0	0 d
PRS : physical robbery succ.	€0	0 d
PR : physical robbery	€0	0 d
NAS : network attack succ.	€0	0 d
NA : network attack	€0	1 d
BRB : bribe	€0	3 d
icp : integr. code in prod.	€2k	15 d
dtm : deploy to market	€1k	5 d
scr : secure coding rooms	€5k	0 d
rfc : rob. finds code	€0	0 d
reb : rob. enters building	€500	3 d
hr : hire robber	€4k	10 d
id : intrusion detection	€200	1 d
heb : hacker exploits bug	€0	3 d
sb : system has a bug	€0	0 d
hh : hire hacker	€1k	20 d
psc : progr. steals code	€0	7 d
bp : bribe programmer	€2k	15 d

Figure 4.5: The ADTree and node attributes for the forestall case study.



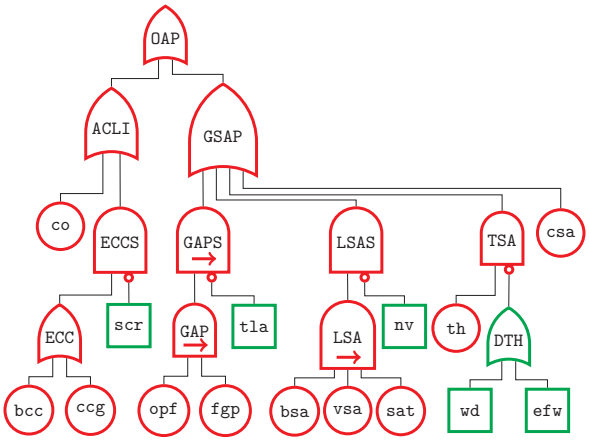
Name	Cost	Time
CIOTD : compromise IoT device	€0	0 h
APNS : access private net. succ.	€0	1 m
APN : access private net.	€0	3 m
GVC : get valid credentials	€0	0 h
CPN : connect to private net.	€0	0 h
AW : access WLAN	€0	0 h
AL : access LAN	€0	0 h
rms : run malicious script	€100	30 m
esv : exploit soft. vulnerab.	€10	1 h
inc : inform of new connect.	€5	1 m
tla : two-level authentic.	€5	1 m
gc : get credentials	€100	10 h
bwk : break WPA keys	€100	2 h
fw : find WLAN	€10	5 h
sma : spoof MAC address	€50	30 m
flp : find LAN port	€10	1 h

Figure 4.6: The ADTree and node attributes for the iot-dev case study.

- the **full** model for each ADTree node, that comprises all possible combinations of messages received from children, as in the "Full Model" column of Table 4.2;
- the model reduced with the **layers** reduction scheme introduced in Section 4.6,
- the model reduced to the **patterns** listed in the "Reduced Model" column of Table 4.2, as per the reduction scheme discussed in Section 4.5;
- the model obtained with **both** of these reductions (patterns and layers);
- the size of the version with both reductions compared with the others (highlighted in green).

The experiments demonstrate that the two reduction techniques are quite complimentary, with significant gains from applying both schemes together (i.e. patterns and layers), compared to using either reduction alone. Notably, the only way to obtain the model for the largest of test cases (gain-admin) within the timeout period was by combining the pattern- and layer-based reductions.

On its own, the pattern-based reduction performs better, which is in line with expectations, since it is by far the more universal technique of the two, equally applicable to all nodes of the ADTree irrespectively of its underlying topology of synchronisation.



Name		Cost	Time
OAP	: obtain admin privileges	€0	0 m
GSAP	: get SA password	€0	0 m
ACLI	: access c.c. CLI	€0	2 m
TSA	: trojan horse for SA	€0	0 m
DTH	: defence against trojans	€0	0 m
LSAS	: LSA successful	€0	0 m
LSA	: look over SA shoulder	€0	0 m
GAPS	: GAP successful	€0	2 m
GAP	: get admin password	€0	10 m
ECCS	: enter c.c. successful	€0	1 h
ECC	: enter computer centre	€0	0 d
csa	: corrupt Sys. Admin.	€5k	5 d
efw	: E-Mail firewall	€3k	0 m
wd	: watchdog sys. daemon	€2k	5 m
th	: trojan horse SA	€100	3 d
nv	: no-visits policy	€0	0 d
sat	: spy SA terminal	€0	30 m
vsa	: visit SA at work	€20	2 d
bsa	: befriend Sys. Admin.	€500	14 d
tla	: two-level authentic.	€5	1 m
fgp	: find guessable pass.	€0	1 d
opf	: obtain password file	€100	3 d
scr	: secure coding rooms	€5k	0 d
ccg	: c.c. guest unwatched	€100	5 d
bcc	: break-in comp. centre	€6k	2 d
co	: corrupt operator	€4k	4 d

Figure 4.7: The ADTree and node attributes for the gain-admin case study.

Model	No reduction			Layers			Patterns			Both			% size ($\|S\|$)		
	$\|S\|$	$\|T\|$	t (s)	$\|S\|$	$\|T\|$	t (s)	$\|S\|$	$\|T\|$	t (s)	$\|S\|$	$\|T\|$	t (s)	$\frac{Both}{No}$	$\frac{Both}{Layers}$	$\frac{Both}{Patterns}$
tr. hunters	558	1,452	0.322	278	452	0.129	156	339	0.082	74	89	0.041	13.26%	26.62%	47.44%
forestall	77,803	215,349	542.976	39,893	67,862	150.775	7,390	22,250	39.390	1,845	2,721	4.094	2.37%	4.62%	24.97%
iot-dev	4,349	8,280	11.280	3,145	4,050	6.510	907	2,154	2.207	371	450	0.623	8.53%	11.80%	40.90%
gain-admin	TO			TO			TO			52,923	94,570	416.431			

Table 4.3: Efficiency of the pattern- and layer-based reductions in the four case study benchmarks.

4.7.3 Scalable experiments

The second set of experiments aims at evaluating how the two reduction techniques scale with the size of input ADTrees. To that end, a generator of IMITATOR files was prepared, outputting four variants for each instance (corresponding to the cases of no reductions, just patterns, just layers, and both reduction schemes applied together). These scalable ADTrees were parameterised with:

- a fixed *number of children* for each intermediate node in the ADTree (denoted by # in Table 4.4). Note that the intermediate nodes generated are all of the AND or the OR type. These two cases lead to exactly the same results as they exhibit an identical automaton structure. Only synchronisation labels are inverted;
- the *number of nodes* in the ADTree (denoted by N in Table 4.4), indicated for information, as this is entailed by the other numbers;
- the *depth* of the generated ADTree (denoted by d in Table 4.4);
- the *width* corresponding to the number of deepmost nodes (denoted by w in Table 4.4). Nodes at some depth are used as children of nodes at the previous depth, and leaves may be added for the parent to have the specified number of children.

Figure 4.8 shows an example ADTree obtained using the generator.

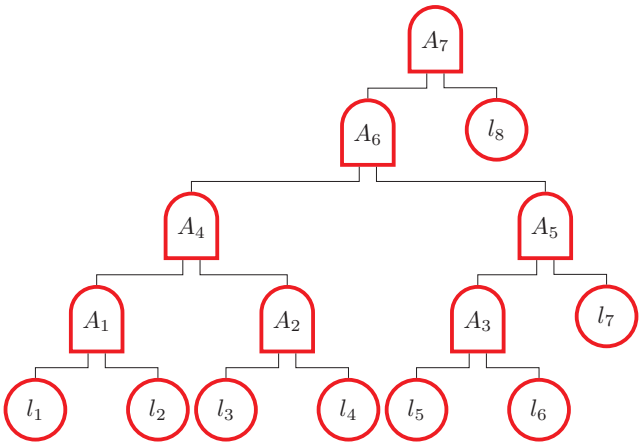


Figure 4.8: ADTree generated for 2 children per node, of depth 4 and width 6.

4.7.4 Scalable ADTrees: results

The experimental results for scalable ADTrees are reported in detail in Table 4.4, and additionally depicted in Figures 4.9 and 4.10.

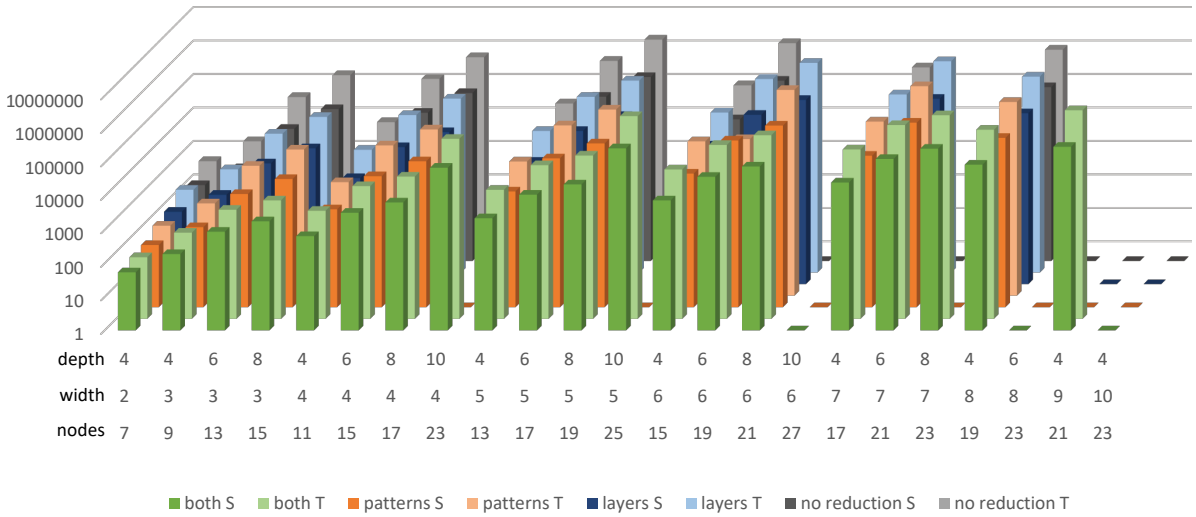


Figure 4.9: Scalability of reductions for ADTrees with 2 children (missing bars indicate timeouts).

The results confirm previous findings from non-scalable case studies, demonstrating the high complementarity of the two reductions schemes. Applying the layer-based reduction in conjunction with the reduced automata patterns can lead to huge gains, e.g. for the variant with 3 children, depth 3 and width 9, where the resulting state space that is 0.98% the size of the full one (line 27 of Table 4.4).

One interesting observation is that varying depth, for a fixed width and number of children, appears to have little impact on the reduction, observed e.g. for 2 children and width 6, at lines 3, 6, 10, 14 and 18. On the other hand, changes in the width parameter, at a fixed depth and number of children, has a huge impact, see e.g. 2 children and depth 3, at lines 2–4.

This can be attributed to increased ADTree width introducing many additional interleavings, which in turns allows for leveraging the layered reduction to great effect. Meanwhile, increasing ADTree depth has very limited impact on the number of interleavings.

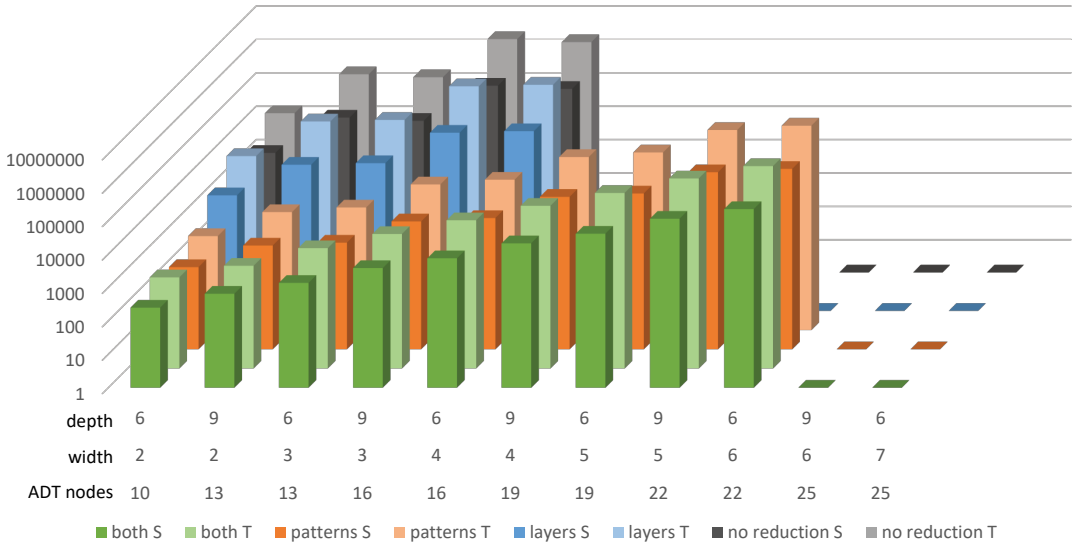


Figure 4.10: Scalability of reductions for ADTrees with 3 children (missing bars indicate timeouts).

Model				No reduction		Layers		Patterns		Both		% size			
#	N	d	w	$\|S\|$	$\|T\|$	$\|S\|$	$\|T\|$	$\|S\|$	$\|T\|$	$\|S\|$	$\|T\|$	$\frac{\|S\|_{Both}}{\|S\|_{No}}$	$\frac{\|S\|_{Both}}{\|S\|_{Layer}}$	$\frac{\|S\|_{Both}}{\|S\|_{Pattern}}$	
2	7	2	4	185	432	145	296	72	123	54	69	29.19%	37.24%	75.00%	1
2	9	3	4	587	1,698	467	1,210	246	571	190	373	32.36%	40.68%	77.23%	2
2	13	3	6	8,823	35,602	4,043	14,046	2,405	7,584	883	1,802	10.00%	21.84%	36.71%	3
2	15	3	8	34,481	160,096	11,425	44,464	6,734	23,135	1,808	3,439	5.24%	15.82%	26.84%	4
2	11	4	4	1,825	6,332	1,465	4,628	840	2,458	652	1,680	35.72%	44.50%	77.62%	5
2	15	4	6	26,725	124,708	12,385	50,480	8,184	30,773	3,256	9,167	12.18%	26.28%	39.78%	6
2	17	4	8	103,955	549,762	34,787	156,754	22,854	92,393	6,606	17,615	6.35%	18.99%	28.90%	7
2	23	4	10	TO		TO		TO		71,965	237,154				8
2	13	5	4	5,603	22,774	4,523	16,942	2,868	10,124	2,228	7,088	39.76%	49.26%	77.68%	9
2	17	5	6	80,687	428,086	37,667	176,722	27,926	121,887	11,258	38,693	13.95%	29.89%	40.31%	10
2	19	5	8	312,889	1,858,220	105,385	540,860	77,948	362,276	22,808	74,986	7.29%	21.64%	29.26%	11
2	25	5	10	TO		TO		TO		273,484	1,128,571				12
2	15	6	4	17,065	79,784	13,825	60,128	9,792	40,476	7,608	28,796	44.58%	55.03%	77.69%	13
2	19	6	6	243,085	1,446,656	114,025	606,524	95,336	473,670	38,520	155,698	15.84%	33.78%	40.40%	14
2	21	6	8	TO		318,203	1,835,398	266,084	1,397,360	78,020	303,724		24,51%	29.32%	15
2	27	6	10	TO		TO		TO		TO					16
2	17	7	4	51,707	273,994	41,987	208,546	33,432	158,372	25,976	113,996	50.23%	61.86%	77.69%	17
2	21	7	6	TO		344,123	2,049,670	325,492	1,813,652	131,564	611,188		38.23%	40.42%	18
2	23	7	8	TO		TO		TO		266,464	1,198,156				19
2	19	8	4	156,145	926,420	126,985	710,636	114,144	609,608	88,688	442,736	56.79%	69.84%	77.69%	20
2	23	8	6	TO		TO		TO		TO					21
2	21	9	4	TO		TO		TO		302,800	1,694,352				22
2	23	10	4	TO		TO		TO		TO					23
3	10	2	6	3,803	15,598	2,891	11,486	289	654	250	537	6.57%	8.64%	86.50%	24
3	13	2	9	43,387	228,362	23,779	124,754	1,283	3,424	653	1,192	1.50%	2.74%	50.89%	25
3	13	3	6	34,739	186,530	26,531	138,578	1,563	4,700	1,380	4,025	3.97%	5.20%	88.29%	26
3	16	3	9	392,531	2,577,950	216,059	1,410,182	6,771	23,024	3,846	10,667	0.98%	1.78%	56.80%	27
3	16	4	6	314,699	2,097,686	240,827	1,567,622	8,511	31,912	7,530	27,559	2.39%	3.12%	88.47%	28
3	19	4	9	TO		TO		36,777	152,390	21,117	74,504			57.41%	29
3	19	5	6	TO		TO		46,377	208,290	41,040	180,639			88.49%	30
3	22	5	9	TO		TO		200,349	978,834	115,164	491,799			57.48%	31
3	22	6	6	TO		TO		252,729	1,322,490	223,650	1,150,257			88.49%	32
3	25	6	9	TO		TO		TO		TO					33
3	25	7	6	TO		TO		TO		TO					34

Table 4.4: Scalability of the pattern- and layer-based reductions.

4.8 Comparison with POR

Between the pattern-based and layer-based reductions, the former technique is clearly closer to classical partial order reduction. Firstly, it is more universal of the two. While applied here for automata corresponding to different types of ADTree components, its approach does not require any particular prerequisites concerning the patterns. Thus, it could in principle be replicated for translations of other formalisms to automata networks, and used to employ specific reductions pertaining to the particular characteristics of different components in the original representation.

The other aspect of this similarity to POR is the way pattern-based reduction is applied. Note, in particular, that reduced automata patterns contain a single path of arbitrarily ordered transitions leading to the loop transition labelled with *A_ok* or *A_nok* for every node in the tree. This is precisely what one would expect, intuitively, from a POR algorithm: since the actions of different child nodes are independent (cf. Definition 2.2.7), it does not matter in which order they are received by the parent. The difference is that POR preserves at least one representative of every path in the full model (cf. Section 3.2), whereas the pattern-based reduction has no such requirement. This yields extra gains, allowing for higher efficiency of reduction wherever some paths may be completely truncated in accordance with the semantics of their corresponding ADTree components. Examples of such paths include any sequences where an `AND` or `SAND` node has already received one or more *nok* messages.

Finally, we also note that much like classical POR is applied on-the-fly while generating the model from its representation, the full automata patterns also never need to be used, or indeed preserved anyhere, once the reduced ones are created.

4.9 Summary

In this chapter, we have discussed two specialised method of state and transition space reduction, tailored specifically to systems that exhibit a tree topology of action synchronisations between local components. An excellent example of such models are various security scenarios that involve preparations and interactions between "attacker" and "defender" parties. Originally represented as attack-defence trees (ADTrees), these scenarios may be converted to an automata-based formalism, which allows for leveraging the power of verifiers like Spin or Imitator. As a parametric model checker, the latter tool is of particular interest here, enabling the synthesis of constraints relevant to the two opposing parties.

The translation of ADTree nodes to a network of automata, with one automaton per node, is based on specific patterns for each type of ADTrees construct. Local transitions of these automata patterns correspond to the semantics of particular ADTree components, e.g. an `AND` gate requires synchronisation from all child nodes, whereas just one suffices for `OR` etc.

It is at this point that the *pattern-based reduction* can be applied. The approach consists in replacing the automata patterns used for the translation with their reduced versions, based on the observation that in the full patterns, many paths are superfluous and can be safely pruned. This makes it analogous, but not identical, to classical partial order reduction. While POR always has to preserve a representative of each path in the full model (cf. Section 3.2), this is not necessarily the case with pattern-based reduction. Intuitively, based on the corresponding ADTree component's semantics, certain paths are known in advance to be "dead ends", safe to be truncated. For instance, an `AND` gate does not need to wait for synchronisation with all child nodes if one was already unsuccessful, since its own resulting state is already known at this point.

For the purposes of this chapter, we represented the full and reduced automata patterns as Guarded Update Systems ($\mathcal{GUS}$), and networks obtained via translation from ADTrees as an asynchronous product of $\mathcal{GUS}$. Note that this formalism does not consider agents yet and will be generalised to Extended AMAS (cf. Section 5.2.1) in Chapter 5, where the impact of both the number of agents and their assignment to specific nodes by the opposing coalitions of attackers and defenders will be discussed in detail.

In the context of Chapter 4, however, the $\mathcal{GUS}$ formalism is sufficient. We have formalised the notion of a synchronisation topology, and demonstrated important properties of tree topologies of $\mathcal{GUS}$, such as those obtained by translating ADTrees. This leads to the second reduction technique, called *layer-based reduction*. It exploits the tree topology by enforcing certain restrictions on the asynchronous interleaving of local transitions from automata patterns, so that nodes do not proceed any further (i.e., hold off executing any private actions and synchronisation with the parent node) until all nodes in the lower

"layer" all have finished and synchronised with their parents. Notably, this approach is fully compatible and complimentary to the pattern-based reduction, leading to additional gains on top of the latter as demonstrated by the experimental results.

4.9.1 Related work

ADTrees were originally conceived as Attack Trees [72, 65, 73, 71, 68, 74], and later extended to model defences and the interactions between opposing parties [75, 76, 77]. They remain a popular, extensively studied formalism [78, 79, 80], and have been implemented in a number of analysis frameworks based on, among others, Timed Automata [81, 82], Petri Nets [83], I/O-IMCs [84, 85], Bayesian Networks [86], and stochastic games [87, 88].

The translation from Attack-Defence Trees to Guarded Update Systems was introduced in [5], while the pattern-based reduction was proposed, alongside the extension of the asynchronous multi-agent formalism to EAMAS, in [4]. However, this work largely focused on the translation itself, demonstrating it using several case studies and two model checkers: Uppaal and Imitator.

The layer-based reduction was then introduced in [5], which also for the first time investigated the efficiency of both reduction techniques, applied separately as well as in conjunction.

The tool `ADT2AMAS`, presented in [89], allows for translating ADTrees, either specified as simple syntax plain text or manually constructed by the user using a graphical interface, to an extension of the AMAS formalism that will be introduced and discussed in Chapter 5 of this thesis. Furthermore, `ADT2AMAS` creates LaTeX files that can be compiled for a graphical overview of the transformation, and integrates well with Imitator, automatically generating input for the latter.

Chapter 5

Minimal Agent Scheduling

The material in Chapter 5 is based on the following papers to which the author of this thesis has contributed:

[4] J. Arias, C. Budde, W. Penczek, L. Petrucci, T. Sidoruk, and M. Stoelinga, "Hackers vs. Security: Attack-Defence Trees as Asynchronous Multi-agent Systems," in *Proceedings of ICFEM 2020*. Springer, 2020, pp. 3–19

[5] L. Petrucci, M. Knapik, W. Penczek, and T. Sidoruk, "Squeezing State Spaces of (Attack-Defence) Trees," in *Proceedings of ICECCS 2019*. IEEE, 2019, pp. 71–80

[6] J. Arias, L. Petrucci, Ł. Maśko, W. Penczek, and T. Sidoruk, "Minimal Schedule with Minimal Number of Agents in Attack-Defence Trees," in *Proceedings of ICECCS 2022*. IEEE, 2022, pp. 1–10

The author's involvement in these works includes most of the material featured in [6], in particular proposing and proving the correctness of an algorithm synthesising an optimal schedule for a minimal number of agents given an ADTree. As the paper [6] is based on previous work, it naturally refers the reduction techniques proposed and proved in [5], see Chapter 4.

5.1 Introduction

We have discussed both the general-purpose technique of partial order reduction (Chapter 3), as well as specialised methods that trade universal applicability for potentially higher efficiency in a smaller class of models that exhibit particular characteristics of their synchronisation topology (Chapter 4). In both cases, reductions were considered within the context of a preexisting representation of a model, with a fixed number of (often symmetrical) local components corresponding to individual agents.

The main factor behind the state-space explosion in models considered here is the number of components that comprise their representations, particularly so with the asynchronous execution semantics of IIS (cf. Definitions 2.1.3 and 2.3.1). Note that the interpreted interleaved system is, in most cases, exponentially larger than its corresponding AMAS [1]. With that in mind, it is clear that in any practical applications a different kind of reduction is equally essential, one that aims to minimise the number of agents in the model.

5.2 Representing agents in security scenarios

Previously, in Chapter 4, we considered specialised model reductions applicable to networks of Guarded Update Systems ($\mathcal{GUS}$) exhibiting a tree topology of synchronisation. Note that although this formalism quite closely resembles the networks of automata of AMAS, $\mathcal{GUS}$ does not cater for or represent agents. In this chapter, we extend the prior approach to reason about agents in attack-defence security scenarios. To that end, we augment the original definitions of AMAS and IIS from Chapter 2 as follows.

5.2.1 Extending AMAS to represent ADTrees

In order to translate ADTrees to AMAS formalism, it is necessary to extend the latter so as to represent elements inherent to ADTrees, in particular node attributes and Boolean conditions.

Definition 5.2.1 (Extended AMAS [4])**.** *An* Extended Asynchronous Multi-agent System (EAMAS) *is an AMAS, in which each local transition function $t \in LT = \bigcup_{i\in\mathcal{A}} T_i$ has a finite set of variables $AT_t = \{v_t^1, \ldots, v_t^k\}$* (attributes) *over a domain $D_t = d_t^1 \times \cdots \times d_t^k$.*

Let $AT = \bigcup_{t\in LT} AT_t$ and $D = \prod_{t\in LT} D_t$. Let Guards be the set of formulas of the form $\beta \sim 0$, where β is a linear expression over attributes of AT and $\sim \in \{<, \leqslant, =, \geqslant, >\}$. Let MSG be the set of all messages, FUN be all functions taking arguments in AT, and $EXP(AT, FUN)$ be linear expressions over AT and FUN. Each transition $t \in LT$ has associated:

- *a* message $f_m(t) \in (\{!, ?\} \times MSG) \cup \{\perp\}$*, which indicates whether transition t sends (marked with $!$) or receives ($?$) a message $m \in MSG = \{ok, nok\}$, or does not synchronise ($\perp$);*
- *a* guard $f_g(t) \in Guards$*, which constrains transitions;*
- *an* update function $f_t \colon AT_t \to EXP(AT, FUN)$*, which states how taking a transition modifies the associated attributes.*

Likewise, the interleaved interpreted system is extended to account for attributes, messages, guards, and update functions added in EAMAS.

Definition 5.2.2 (Extended IIS [4])**.** *Let $\mathcal{PV}$ be a set of propositional variables, $\mathbf{v} \colon AT \to D$ a valuation of the attributes, and $\mathbf{v}_0$ an initial valuation. An* extended interleaved interpreted system (EIIS) *is an EAMAS extended with the following elements:*

- *a set of* global states $St \subseteq L_1 \times \cdots \times L_n \times D$*;*
- *an* initial state $s_0 = \langle(\iota_1, \ldots, \iota_n), \mathbf{v}_0\rangle \in St$*;*
- *a* valuation function $V : St \to 2^{\mathcal{PV}}$*;*
- *a (partial)* global transition function $T \subseteq St \times Evt \times St$*, such that $\langle(l_1, \ldots, l_n, \mathbf{v}), e, (l'_1, \ldots, l'_n, \mathbf{v}')\rangle \in T$ iff either of the following two conditions is satisfied:*

  A) $|Agent(e)| = 1 \;\wedge\; \exists t_i = (l_i, e, l'_i) \in T_i$ *for* $i \in Agent(e) \;\wedge\; \forall k \in \mathcal{A}\backslash\{i\}\;\; l_k = l'_k \;\wedge\; \mathbf{v} \models f_g(t_i) \;\wedge\; \mathbf{v}' = \mathbf{v}[AT_{t_i}]$*;*

  B) $\exists i, j \in Agent(e) \;\wedge\; \exists t_i = (l_i, e, l'_i) \in T_i \;\wedge\; \exists t_j = (l_j, e, l'_j) \in T_j$*, such that* $f_m(t_i) = (!, m) \wedge f_m(t_j) = (?, m)$*; and*
  $\forall k \in \mathcal{A}\backslash\{i, j\}\;\; l_k = l'_k$*; and*

  $\mathbf{v} \models f_g(t_i) \wedge f_g(t_j)$*; and*

$\mathbf{v}' = \mathbf{v}[AT_{t_i}][AT_{t_j}]$, *where* AT_{t_i} *and* AT_{t_j} *are disjoint,*

where $\mathbf{v}[AT_{t_i}][AT_{t_j}]$ *indicates the substitution of attributes in the valuation* $\mathbf{v}$ *according to transitions* t_i *and* t_j*, that is*

$$\mathbf{v}' = \mathbf{v}\Big[\bigwedge_{v_{t_i} \in AT_{t_i}} v_{t_i} := f_{t_i}(v_{t_i})\Big]\Big[\bigwedge_{v_{t_j} \in AT_{t_j}} v_{t_j} := f_{t_j}(v_{t_j})\Big].$$

Note that in the above definition, we make an explicit assumption that events are either private or shared between exactly two agents. This is consistent with the tree topology of EAMAS models obtained from ADTrees, where synchronisation always occurs on events shared by a pair of nodes, i.e. the parent and the child. Indeed, the purpose of extending the AMAS formalism to EAMAS is specifically to represent such models. Applying it in more general scenarios would require changing the above definition of the global transition function accordingly.

5.2.2 Translation to EAMAS

The translation to EAMAS works exactly as in Section 4.4, except each node of the original ADTree, replaced with the corresponding automaton pattern from Table 4.2, now represents a single agent. The execution semantics for the resulting EAMAS is provided by EIIS. In particular, shared events allow to capture the communication between nodes: parents and children nodes need to synchronise whenever the latter report the success or failure of their action.

We now prove that the transformation of ADTrees to EAMAS based on the patterns from Table 4.2 is both *complete* and *sound*.

Theorem 5.2.3 (Completeness of the ADTree to EAMAS transformation [4])**.** *Let* $a_1, \ldots, a_n, A$ *be ADTree nodes such that* $a_1, \ldots, a_n$ *are the children of the gate* A*, and let* $M_{a_1}, \ldots, M_{a_n}, M_A$ *be their respective EAMAS models. Let* A *succeed when* $a_{i_1} \cdots a_{i_m}$ *finalise (succeeding or failing), in that order. If the EAMAS models* $M_{a_{i_j}}$ *finalise in the same order, then* M_A *transits from its initial state* l_0 *to its final state* l_A*.*

Proof. First note that if node x finalises, its EAMAS model will send *either* $!x\_ok$ or $!x\_nok$. Moreover, due to the self-loops in the end states, this happens infinitely often. Thus, if nodes $a_{i_1} a_{i_2} \cdots a_{i_m}$ finalise, actions $!a_{i_1}\_ok$, $!a_{i_2}\_ok$,..., $!a_{i_m}\_ok$ (or the corresponding $!a_*\_nok$) will be signaled infinitely often. By hypothesis, gate A succeeds when $a_{i_1} \cdots a_{i_m}$ finalise in that order. All patterns in Table 4.2 have (at least) one sequence of actions $?a_{j_1}\_ok \cdots ?a_{j_k}\_ok$ (or $?a_*\_nok$) that take it from l_0 to l_A. By the first argument, all actions in the sequence of M_A are signaled infinitely often. M_A will then transit from l_0 to l_A. □

This covers *expected* actions that a parent must receive from its children to achieve its goal. For *unexpected* sequences of signals, a parent may not react to all information from its children, e.g. a `CAND` gate that after $?a\_ok$ receives (unexpectedly) $?d\_ok$. In such scenarios, the model cannot reach its final state, entering a deadlock. This means that the model of A cannot signal its $!A\_ok$ action. Notice that this is exactly what should happen, because such unexpected sequences actually inhibit the goal of node A. To formally complete this argument, we now prove that the transformations of Table 4.2 are sound. That is, all paths (of actions) that make the model of a node A signal $!A\_ok$, correspond to an ordered sequence of finalising children of A that make it reach its goal.

Theorem 5.2.4 (Soundness of the ADTree to EAMAS transformation [4])**.** *Let* $a_1, \ldots, a_n, A$ *be ADTree nodes such that* $a_1, \ldots, a_n$ *are the children of the gate* A*, and let* $M_{a_1}, \ldots, M_{a_n}, M_A$ *be their respective EAMAS models. Let the sequence of actions* $?a_{i_1}\_s_{i_1} ?a_{i_2}\_s_{i_2} \cdots ?a_{i_m}\_s_{i_m}$ *take* M_A *from its initial state*

l_0 to its final state l_A, where $s_j \in \{ok, nok\}$. Then, the corresponding ordered success or failure of the children $a_{i_1}, \ldots, a_{i_m}$ make the ADTree gate A succeed.

Proof. First, observe that the reduced models in Table 4.2 are subsets of the *full models*—which consider all possible interleavings of synchronisation messages from child nodes. Thus, any path π in a reduced model also appears in the corresponding full model. Moreover, inspecting Tables 4.1 and 4.2 shows that, in the full model $\overline{M}_A$ of gate A, a path of actions $?a_i\_s_i$ (from children a_i of A) that transit from l_0 to l_A, encodes the ordered finalisation of these children that make the ADTree A succeed. Our hypothesis is the existence of a path π of actions in (the reduced model) M_A, that take this EAMAS from l_0 to l_A. By the first argument, π is also a path in (the full model) $\overline{M}_A$. By the second argument, π encodes the ordered finalisation of children c of A that make this gate succeed. □

5.3 Minimising number of agents

Translating attack-defence trees to the multi-agent formalism of extended AMAS has several major advantages. The first and most obvious one is being able to leverage the vast array of tools and methods that have been developed for the formal verification of multi-agent systems over the years. However, the inclusion of agents additionally allows for reasoning about the ADTree security scenarios on an entirely new level, i.e. considering the size of opposing coalitions, as well as specific assignments of individual agents to specific tasks corresponding to nodes of the original ADTree. Clearly, these aspects affect both the feasibility of attack or defence in considered scenarios and the performance metrics quantified by attributes such as attack/defence time or their associated financial cost. On the other hand, increasing the number of agents has severe consequences not only because the extra resource requirements it represents in the real world, but also from the verification standpoint, where each agents corresponds to a separate automaton in the EAMAS, leading to significantly larger models. This leads to the very interesting, clearly relevant, and non-trivial problem of optimal scheduling of agents in attack-defence scenarios. In other words, the problem is to obtain an assignment of agents to ADTree nodes, such that the attack is achieved in the lowest possible time using the minimal number of agents required.

In this section, we will present an algorithm that synthesises such an assignment for a given ADTree. We begin by describing the preprocessing phase, where the input tree will be normalised and transformed into a directed acyclic graph (DAG).

5.3.1 Normalising ADTrees

The preprocessing of the input ADTree involves transforming it into a DAG, where all actions are of the same duration. This is achieved by splitting nodes into sequences of actions with normalised duration of one time unit, mimicking the scheduling enforced by ADTrees sequential gates, and considering the different possibilities of defences. To that end, we introduce a sequential node `SEQ`, which only waits for some input, processes it and produces some output. It is depicted as a lozenge (see nodes N_1 and N_2 in Figure 5.1).

In what follows, we assume that $t_{unit} = gcf(t_{N_1} \ldots t_{N_{|ADTree|}})$, i.e. one time unit is the greatest common factor of time durations across all nodes in the input ADTree. By *time slots*, we refer to fragments of the schedule whose length is t_{unit}. That is, after normalisation, one agent can handle exactly one node of non-zero duration within a single time slot.

Note that in order to ensure backwards traceability after synthesising the optimal assignment of agents, all node labels are preserved. Wherever applicable, their new versions are appended with primes or indices.

Since all ADTree attributes are optional, it may be the case that one or more nodes have no time parameter set, and are thus considered to have a duration of 0. Such nodes play essentially a structuring role. Because they do not take any time, the following proposition is straightforward.

Proposition 5.3.1 (Scheduling zero-duration nodes [6])**.** *Nodes with duration* 0 *can always be scheduled immediately before their parent node or after their last occurring child, using the same agent in the same time slot.*

Proof. Straightforward, as such nodes do not take any time to process, thus not affecting the schedule when assigned together with the parent or child. □

Preprocessing introduces nodes similar to `SEQ` but with 0 duration, called `NULL` and depicted as trapeziums (Fig. 5.2).

The first pre-processing step prior to applying the scheduling algorithm normalises the time parameter of nodes.

Proposition 5.3.2 (Time normalisation [6])**.** *Any node* N *of duration* $t_N = n \times t_{unit}, n \neq 0$ *can be replaced with an equivalent sequence consisting of a node* N' *(differing from* N *only in its* 0 *duration) and* n `SEQ` *nodes* $N_1, \ldots, N_n$ *of duration* t_{unit}*.*

Proof. Consider a node N' identical to N except for its duration $t_{N'} = 0$. Let N' be followed by a sequence of n `SEQ` nodes, each with duration t_{unit}. Together, N' and the `SEQ` nodes form a sequence, and $t_N = t_{N'} + t_{N_1} + \cdots + t_{N_n} = n \times t_{unit}$. Furthermore, since N' only differs from N in its duration, the conditions on their children are the same, and are evaluated at the same time (i.e. immediately after synchronisation with a sufficient number of children, and before any `SEQ` node is processed). Thus, replacing N with the sequence $N'\ N_1 \ldots N_n$ preserves the behaviour and duration, independently of the number of agents processing nodes. □

Example 5.3.3. *Figure 5.1 shows the transformation of an* `AND` *node* N *with duration* $t_N = 2t_{unit}$*. Both* N_2 *and* N_1 *are of duration* t_{unit}*, while* N' *has a null duration.*

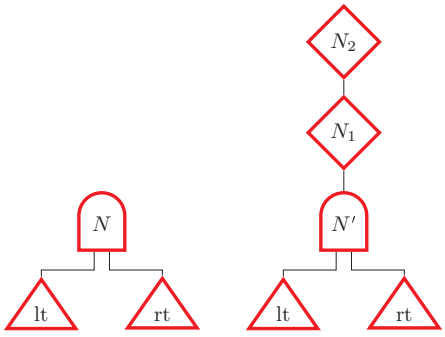


Figure 5.1: Normalising `AND` node N ($t_N = 2t_{unit}$, $t_{N'} = 0$).

Sequential nodes `SAND` enforce some scheduling. These are transformed into a sequence containing their subtrees and `NULL` nodes.

Proposition 5.3.4 (Scheduling enforcement [6])**.** *Any* `SAND` *node* N *with children subtrees* $T_1, \ldots, T_n$ *can be replaced with an equivalent sequence* $T_1, N_1, T_2, \ldots, N_{n-1}, T_n, N_n$*, where each* N_i *is a* `NULL` *node, its input is the output of* T_i *and its outputs are the leaves of* T_{i+1} *(except for* N_n *which has the same output as* N *if any).*

Proof. Each NULL node N_i occurs after its input and before its output, so the sequentiality is preserved: T_i occurs before all leaves of T_{i+1}, which in turn occur before any other action in T_{i+1}, and N_n occurs last. Moreover, since NULL nodes have 0 duration, they do not impact the timing. Finally, note that since time has been normalised, it holds that $t_{N_i} = 0$. Hence, the transformation preserves both the order of actions and the timing. □

Example 5.3.5. *Consider the* SAND *node N depicted in Figure 5.2 on the left, where all other nodes have already been processed by the time normalisation. The transformation of Proposition 5.3.4 produces the DAG on the right side of Figure 5.2, where subtrees A, B and C occur in sequence, as imposed by the* NULL *nodes N_1 and N_2 between them, and then action N_3 occurs.*

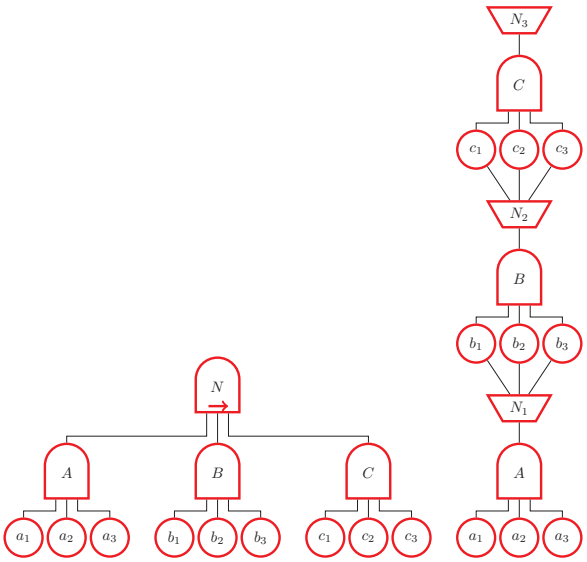


Figure 5.2: Normalising SAND node N.

5.3.2 Handling defences and conditional branches

The scheduling we are seeking to obtain will guarantee the necessary attacks are performed. Hence, when dealing with defence nodes, we can assume that all attacks are successful. However, they may not be mandatory, in which case they should be avoided so as to obtain a better scheduling of agents.

Taking into account each possible choice of defences will lead to as many DAGs representing the attacks to be performed. This allows for answering the question: "What is the minimal schedule of attacker agents if these defences are operating?"

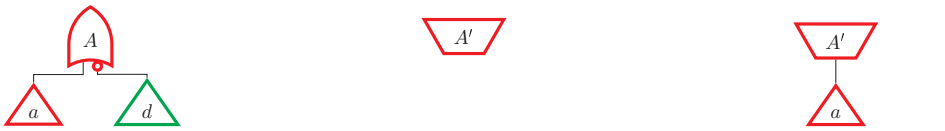
Composite defences. Defences resulting from an AND, SAND or OR between several defences are operating according to the success of their subtrees: for AND and SAND, all subtrees should be operating, while only one is necessary for OR. This can easily be computed by a boolean bottom-up labelling of nodes. Note that different choices of elementary defences can lead to disabling the same higher-level composite defence, thus limiting the number of DAGs that will need to be considered for the scheduling.

*No Defence nodes (*NODEF*).* A NODEF succeeds if its attack succeeds or its defence fails. Hence, if the defence is not operating, the attack is not necessary. Thus, the NODEF node can be replaced by a NULL node without children, and the children subtrees can be deleted. On the contrary, if the defence is operating, the attack must take place. The defence subtree is deleted, while the attack one is kept, and the NODEF node can be replaced by a NULL node, as pictured in Figure 5.3.

*Counter Defence (*CAND*) and Failed Reactive Defence (*SCAND*) nodes.* A CAND succeeds if its attack is successful and its defence is not. A SCAND behaves similarly but in a sequential fashion, i.e. the defence

takes place after the attack. In both cases, if the defence is not operating, its subtree is deleted, while the attack one is kept, and the CAND (or SCAND) node can be replaced by a NULL node, as was the case in Figure 5.3c. Otherwise, the CAND (or SCAND) node is deleted, as well as its subtrees. Moreover, it transmits its failure recursively to its parents, until a choice of another branch is possible. Thus, all ancestors are deleted bottom up until an OR is reached.

Thus, we have a set of DAGs with attack nodes only.



(a) NODEF node (b) Case d fails (c) Case d operates

Figure 5.3: Handling NODEF A.

Conditional choices in OR *nodes.* OR nodes give the choice between several series of actions, only one of which will be chosen in an optimal assignment of events. However, one cannot simply keep the shortest branch of an OR node and prune all others. Doing so minimises attack time, but not necessarily the number of agents. In particular, a slightly longer, but narrower branch may require fewer agents without increasing attack time, provided there is a longer sequence elsewhere in the DAG. Consequently, only branches that are guaranteed not to lead to an optimal assignment can be pruned, which is the case when a branch is the longest one in the entire graph. All other cases need to be investigated, leading to multiple variants depending on the OR branch executed, similar to the approach for defence nodes.

5.3.3 Example of preprocessing

Figures 5.4 and 5.5 detail the preprocessing of the treasure hunters example step by step. The time unit is one minute. Long sequences of SEQ are shortened with dotted lines. Note that when handling the defence, at step 3, we should obtain two DAGs corresponding to the case where the defence fails (see Figure 5.5b), or where the defence is successful. This latter case leads to an empty DAG where no attack can succeed. Therefore, we can immediately conclude that if the police is successful, there is no scheduling of agents.

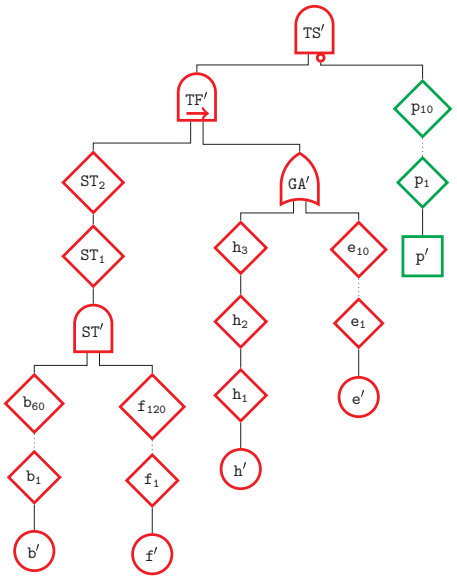
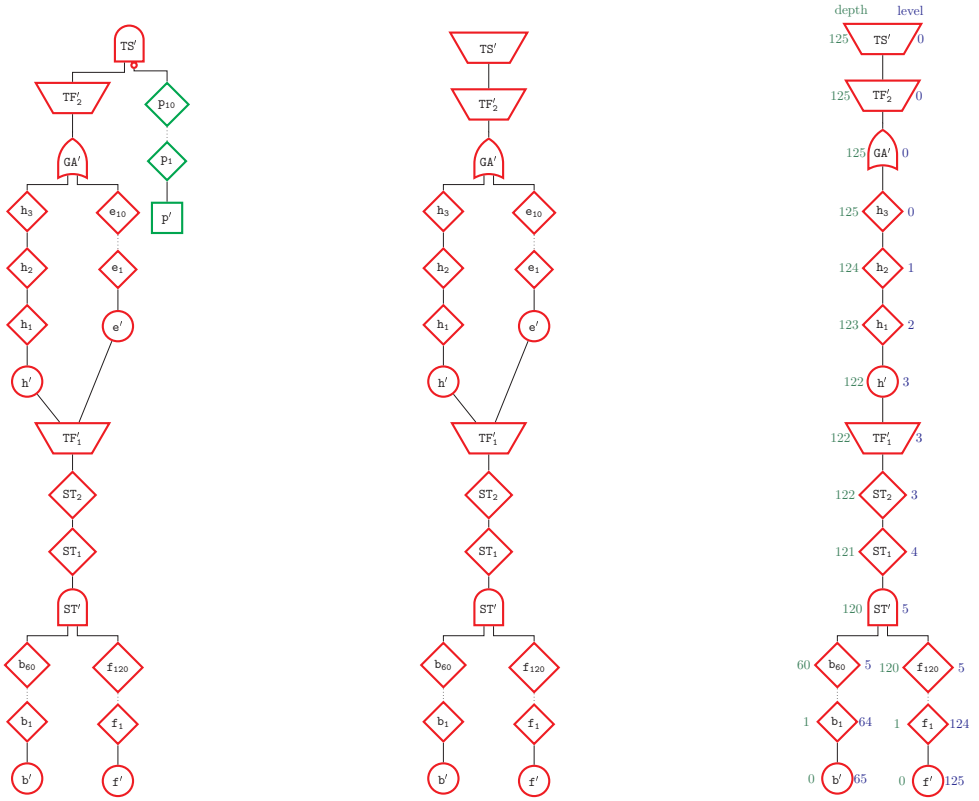


Figure 5.4: Treasure hunters ADTree: time normalisation.



(a) Scheduling enforcement

(b) Handling failed defence

(c) Applying Algs. 1-2, handling `OR` nodes

Figure 5.5: Treasure hunters ADTree: final preprocessing steps (left, middle) and initial part of the main algorithm (right).

5.4 Synthesising the minimal assignment

Following the preprocessing steps described in the previous section, the input ADTree has been transformed into a set of DAGs, each of which corresponds to one particular configuration of defence nodes' outcomes and choices made in `OR` branches in the original ADTree. Note that as per Section 5.3.1, at this stage all DAG nodes are either (i) a leaf, or of type `AND`, `OR`, or `NULL`, all with duration 0 or (ii) of type `SEQ` with duration t_{unit}. Their branches mimic the possible runs in the system.

Thus, the input of the algorithm synthesising the minimal assignment using the minimal number of agents is a set of DAGs. For each of these graphs,

The algorithm's input is a set of DAGs preprocessed as described in Section 5.3.1, corresponding to possible configurations of defence nodes' outcomes and choices of `OR` branches in the original ADTree. For each of these graphs, n denotes the number of `SEQ` nodes (all other ones have 0-duration). Furthermore, nodes (denoted by N) have some attributes: their *type*, as well as four integers *depth*, *level*, *agent* and *slot*, all initially with value 0. The values of *depth* and *level* denote, respectively, the height of a node's tallest subtree and the distance from the root (both without counting the zero duration nodes), while *agent* and *slot* store a node's assignment in the schedule.

We first compute the nodes' depth in Section 5.4.1, then compute the level of nodes in Section 5.4.2, discuss the theoretical upper and lower bounds on the number of required agents in Section 5.4.3, and finally compute an optimal scheduling in Section 5.4.4.

5.4.1 Depth of nodes

Starting from the root, the procedure DepthNode (Algorithm 1) explores the DAG in a DFS *(depth first search)* manner. During backtracking, i.e. starting from the leaves, *depth* is computed for the different types of nodes as follows:

- `LEAF` **node:** After the time normalisation, a leaf node takes 0 time. It may still be an actual leaf, in which case its total duration is also 0, since it has no children (not satisfying condition at l. 3). Alternatively, it may have a child node due to the scheduling enforcement, and then its time is the same as the one of its only child (l. 10).
- `SEQ` **node:** Its duration is one t_{unit}, which must be added to the duration of its only child to obtain the minimum time of execution from the start (l. 4–5).
- `AND` **node:** All children of an `AND` node must be completed before it occurs. Therefore, its minimal time is the maximum one of all its children (l. 6–7).
- `OR` **node:** Only one child must complete for the `OR` node to happen. Its time is thus the minimal one of all its children (l. 8–9).
- `NULL` **node:** Note that, by construction, a `NULL` node may have several parents but a single child. Its duration being null, its time is the same as the one of its only child (l. 10).

Note that the condition at l. 2 avoids a second exploration of a node which has already been handled in another branch.

Algorithm 1: DepthNode(*node*) [6]

```
1  for N ∈ child(node) do
2      if N.depth = 0 then DepthNode(N)
3  if child(node) ≠ ∅ then
4      if node.type = SEQ then
5          node.depth ← N.depth + 1, s.t. {N} = child(node)
6      else if node.type = AND then
7          node.depth ← max({N.depth | N ∈ child(node)})
8      else if node.type = OR then
9          node.depth ← min({N.depth | N ∈ child(node)})
10     else node.depth ← N.depth, s.t. {N} = child(node)
```

5.4.2 Level of nodes

Levels are assigned recursively, starting with the root, using a DFS. The procedure LevelNode (Algorithm 2) computes nodes' levels. It first assigns the node's level (l. 1) according to the call argument. Note that in case of multiple parents (or ancestors with multiple parents), the longest path to the root is kept.

Algorithm 2: LevelNode($node, l$) [6]

```
1  node.level ← max(l, node.level)
2  for N ∈ child(node) do
3      if node.type = SEQ then LevelNode(N, l + 1)
4      else LevelNode(N, l)
```

Algorithm 3: MinSchedule(DAG_set) [6]

```
1  output = ∅
2  while DAG_set ≠ ∅ do
3      Pick DAG ∈ DAG_set
4      if DAG.n = 0 then continue                          ▷ Skip empty DAGs
5      DepthNode(root(DAG))                                ▷ Compute depth of nodes
6      DAG ← DAG\{N | ¬N.keep}
7      LevelNode(root(DAG), 0)                             ▷ Compute level of nodes
8      slots ← root(DAG).depth
9      lower_bound ← ⌈DAG.n / slots⌉ − 1
10     max_agents ← max_j(|{N : N.type = SEQ ∧ N.level = j}|)   ▷ Max. level width (concur.
           SEQ nodes)
11     upper_bound ← max_agents
12     curr_output = ∅
13     while (upper_bound − lower_bound > 1) do
14         agents ← lower_bound + ⌊(upper_bound − lower_bound) / 2⌋
15         (candidate, n_remain) ← Schedule(DAG, slots, agents)
16         if n_remain = 0 then                            ▷ Candidate schedule OK
17             upper_bound ← agents
18             curr_output ← candidate
19         else lower_bound = agents                       ▷ Candidate schedule not OK
20     if upper_bound = max_agents then
21         (curr_output, _) ← Schedule(DAG, slots, max_agents)
22     ZeroAssign(DAG)
23     output ← output ∪ curr_output
24     DAG_set ← DAG_set\DAG
25 return output
```

5.4.3 Bounds on the number of agents

The upper bound on the number of agents is obtained from the maximal width of the preprocessed DAG, i.e. the maximal number of `SEQ` nodes assigned the same value of *level* (that must be executed in parallel to ensure minimum time).

The minimal attack time is obtained from the number of levels l in the preprocessed DAG. Note that the longest path from the root to a leaf has exactly l nodes of non-zero duration. Clearly, none of these nodes can be executed in parallel, therefore the number of time slots cannot be smaller than l. Thus, if an optimal schedule of $l \times t_{unit}$ is realisable, the n nodes must fit in a schedule containing l time slots. Hence, the lower bound on the number of agents is $\lceil \frac{n}{l} \rceil$. There is, however, no guarantee that it can be achieved, and introducing additional agents may be necessary depending on the DAG structure, e.g. if there are many parallel leaves.

5.4.4 Minimal schedule

The algorithm for obtaining a schedule with the minimal attack time and also minimising the number of agents is given in Algorithm 3. Input DAGs are processed sequentially, a schedule returned for each one. Not restricting the output to the overall minimum allows to avoid "no attack" scenarios where the time is 0 (e.g. following a defence failure on a root `NODEF` node). Furthermore, with information on the repartition of agents for a successful minimal time attack in all cases of defences, the defender is able to decide which defences to enable according to these results (and maybe the costs of defences).

The actual computation of the schedule is handled by the function Schedule (Algorithm 4). Starting from the root and going top-down, all `SEQ` nodes at the current level are added to set S. The other nodes

Algorithm 4: SCHEDULE($DAG, slots, agents$) [6]

```
1  l ← 0, slot ← slots, S ← ∅, n_remain ← DAG.n
2  while n_remain > 0 and slot > 0 do
3      agent ← 1
4      S ← S ∪ {N | N.type = SEQ ∧ N.level = l}
5      if ∃_{N∈S}, s.t. N.depth < slots − slot then
6          return ∅, n_remain
7      while agent ⩽ agents and S ≠ ∅ and
         (Pick N ∈ S, s.t. ∀_{N'∈S} N.depth ⩾ N'.depth ∧ ∀_{N':N'.slot=slot} N' ∉ ancestors(N)) ≠ ∅ do
8          N.agent ← agent
9          N.slot ← slot
10         agent ← agent + 1, n_remain ← n_remain − 1
11         S ← S\{N}
12     ReshuffleSlot(slot, agent − 1)
13     l ← l + 1, slot ← slot − 1
14 output ← ⋃_{N∈DAG} {(N.agent, N.slot)}
15 return output, n_remain
```

at that level have a null duration and can be scheduled afterwards with either a parent or child. An additional check in l. 5 ensures that non-optimal variants (whose longest branch exceeds a previously encountered minimum) are discarded without needlessly computing the schedule. Nodes in S are assigned an agent and time slot, prioritising those with higher *depth* (i.e. taller subtrees), as long as an agent is available. Assigned nodes are removed from S, and any that remains (e.g. when the bound was exceeded) is carried over to the next level iteration. Note that at this point it is possible for a parent and child node to be in S concurrently, but since higher *depth* takes precedence, they will never be scheduled in the wrong order. In such cases, an extra check in the while loop avoids scheduling both nodes to be executed in parallel.

Algorithm 4 calls function ReshuffleSlot after the complete assignment of a time slot at l. 12 to ensure consistent assignment of sub-actions of the same ADTree node. Note that depending on *depth*, a sub-action may be moved to the next slot, creating an interrupted schedule where an agent stops an action for one or more time units to handle another. Alternatively, agents may collaborate, each handling a node's action for a part of its total duration. Such assignments could be deemed unsuitable for specific scenarios, e.g. defusing a bomb, in which case manual reshuffling or adding extra agent(s) is left to the user's discretion.

At this point, either the upper or the lower bound on the number of agents is adjusted, depending on whether the resulting schedule is valid (that is, there are no nodes left to assign at the end). Scheduling is then repeated for these updated values until the minimal number of agents is found (i.e. the two bounds are equal).

After the complete computation for a given DAG, l. 22 calls ZeroAssign in order to obtain assignments for all remaining nodes, i.e. those of zero duration. Functions ReshuffleSlot and ZeroAssign are detailed in Sections 5.4.5 and 5.4.6, respectively.

Although this algorithm assumes the minimal time is of interest, it can be easily modified to increase the number of time slots, thus synthesising the minimal number of agents required for a successful attack of any given duration.

5.4.5 Uniform assignment for SEQ nodes

A separate subprocedure, given in Algorithm 5, swaps assigned agents between nodes at the same level so that the same agent handles all `SEQ` nodes in sequences obtained during the time normalisation step

(i.e. corresponding to a single node in the original ADTree).

Algorithm 5: ReshuffleSlot($slot, num\_agents$) [6]

```
1 for agent ∈ {1..num_agents} do
2     current_node ← N, s.t. N.agent = agent ∧ N.slot = slot
3     par_agent ← parent(current_node).agent
4     if par_agent ≠ agent ∧ par_agent ≠ 0 then
5         if ∃N' ≠ current_node, s.t.  N'.agent = par_agent ∧ N'.slot = slot then
6             N'.agent ← agent                                  ▷ Swap with N' if it exists
7             N'.slot ← slot
8         current_node.agent ← par_agent
9         current_node.slot ← slot
```

Proposition 5.4.1 ([6])**.** *Reshuffling the assignment by swapping the agents assigned to a pair of nodes in the same slot does not affect the correctness of the scheduling.*

Proof. First, note that the procedure does not affect nodes whose parents have not yet been assigned an agent (l. 4). Hence, reshuffling only applies to `SEQ` nodes (since the assignment of 0 duration nodes occurs later in the main algorithm MinSchedule). Furthermore, changes are restricted to pairs of nodes in the same time slot, so swapping assigned agents between them cannot break the execution order and does not affect the schedule correctness. □

5.4.6 Assigning nodes without duration

After all non-zero duration nodes have been assigned and possibly reshuffled at each level, Algorithm 6 handles the remaining nodes.

Our choice here stems from the ADTree gate the node originates from. We first assign zero-duration nodes to the same agent and the same time slot as their parent if the parent is a `SEQ` node (l. 2–6).

Of the remaining ones, nodes of type `NULL`, `OR` and `LEAF` get the same assignment as their only child if any, or as their parent if they have no child (l. 8–19). The latter case may happen for `NULL` when handling defences as in e.g. Figure 5.3b, and for `LEAF` nodes with originally a null duration. `AND` nodes are assigned the same agent and time slot as the child that occurs last (l. 20–30).

Note that in all cases the agents (and time slots) assigned to zero duration nodes are the same as those of their immediate parents or children. Hence, no further reshuffling is necessary.

Proposition 5.4.2 ([6])**.** *Adding nodes of zero duration to the assignment in Algorithm 6 does not affect the correctness of the scheduling.*

Proof. Since all remaining nodes have zero duration, no extra agents or time slots are necessary. In all cases, the zero duration node is assigned with either its immediate parent or child, preserving the correct execution order. Consider possible cases at each step of the algorithm:

- l. 2–6: Nodes with a `SEQ` parent are the final nodes of sequences obtained during time normalisation. Clearly, they can be assigned the same agent and time slot as their immediate parent without affecting the schedule.
- l. 8–19: `OR` nodes: in each DAG variant (see Section 5.3.2), they are guaranteed to have a single child node and can be scheduled together with this child provided the corresponding sub-DAG has some duration.

Algorithm 6: ZeroAssign(DAG) [6]

```
1  S ← {N | N.agent = 0}                                   ▷ Nodes not assigned yet
2  for node ∈ S do
3      if N ∈ parent(node) ∧ N.type = SEQ then
4          node.agent ← N.agent
5          node.slot ← N.slot
6          S ← S\{node}
7  while S ≠ ∅ do
8      for node ∈ S s.t. node.type ∈ {NULL, OR, LEAF} do
9          if N.agent ≠ 0 s.t. N ∈ child(node) then
10             node.agent ← N.agent
11             node.slot ← N.slot
12             S ← S\{node}
13         if (child(node) = ∅
14         ∨(N.depth = 0 s.t. N ∈ child(node))) then
15             parent_node ← N ∈ parent(node) s.t.   ∀_{N'∈parent(node)} N.slot ⩽ N'.slot
16             if parent_node.agent ≠ 0 then
17                 node.agent ← parent_node.agent
18                 node.slot ← parent_node.slot
19                 S ← S\{node}
20     for node ∈ S s.t. node.type = AND do
21         if node.depth = 0 ∧ parent(node).agent ≠ 0 then
22             node.agent ← parent(node).agent
23             node.slot ← parent(node).slot
24             S ← S\{node}
25         if node.depth ≠ 0
26         ∧∀_{N∈child(node)} (N.agent ≠ 0 ∨ N.depth = 0) then
27             child_node ← N ∈ child(node) s.t.   ∀_{N'∈child(node)} N.slot ⩾ N'.slot
28             node.agent ← child_node.agent
29             node.slot ← child_node.slot
30             S ← S\{node}
```

NULL and LEAF nodes have a single child if any and are handled analogously to OR, being assigned the same agent as their child. Note that LEAF nodes can have gotten this child during e.g. the scheduling enforcement step (see Proposition 5.3.4).

OR, NULL and LEAF nodes with no child or a child sub-DAG with no duration are assigned as their parent. If a NULL node has several parents due to sequence enforcement, it gets the same assignment as its parent that occurs first.

- l. 20–30: In case all children are never able to get an assignment, i.e. they are subtrees of null duration and can be identified with a depth 0, the AND node gets the same assignment as its parent.

  Otherwise, AND nodes are also scheduled together with one of their children. Note that the AND condition is satisfied only if all its longest children have completed, therefore the one that occurs last, i.e. has the biggest time slot, is chosen (l. 20–30). Furthermore, note that since children subtrees with a null duration are discarded, such children of the AND node have already been assigned an agent at that point.

The pathological case of a full ADTree with no duration is not handled since the algorithm is not called for such DAGs. □

5.4.7 Complexity and correctness

We now consider the algorithm's complexity and prove that it achieves its intended goal.

Proposition 5.4.3 ([6]). *Algorithm 3 is in $\mathcal{O}(kn^2 \log n)$, where k is the number of input DAG variants, and n their average number of nodes.*

Proof. Initially, DepthNode, and LevelNode each visit all DAG nodes, hence $2n$ operations. In Schedule, the outer while loop (l. 2) iterates over nodes of non-zero duration; its inner loop and ReshuffleSlot both operate within a single time slot. Overapproximating these numbers by n puts the function at no more than n^2 operations. The schedule computation is repeated at most $\log n$ times in a divide-and-conquer strategy (l. 13).

Finally, ZeroAssign visits all zero duration nodes (again overapproximated by n), performing at most $2n$ iterations for each, for a total of $2n^2$. Thus, the complexity of processing a single DAG is $\mathcal{O}(2n + n^2 \log n + 2n^2) = \mathcal{O}(n^2 \log n)$, and $\mathcal{O}(kn^2 \log n)$ for the whole input set.

Note that as per Section 5.3.1, the preprocessing step introduces a number of additional nodes in resulting DAGs. However, since that factor depends on the structure and attributes of the original ADTree rather than its size, it is treated as a constant in the consideration of complexity. □

Thus, while the complexity of the scheduling algorithm itself is quadratic, it is executed for k input DAG variants, where k is exponential in the number of `OR` and defence nodes in the ADTree.

Proposition 5.4.4 ([6]). *The assignments returned by Algorithm 3 are correct and use the minimal number of agents for each variant $DAG \in DAG\_set$ to achieve the attack in minimal time.*

Proof. Let L denote the number of levels in an input variant $DAG \in DAG\_set$, and L_i the set of nodes at the i-th level. We need to show that the resulting assignment is 1) *correct*, and 2) *optimal* in both schedule length and number of agents.

1) Schedule assigns time slot 1 to leaves at the bottom level, subsequent slots to their ancestors, and finally the last one L to the root node. Thus, the execution order of nodes in DAG is correct. Furthermore, it is guaranteed that there are enough agents to handle all nodes by increasing *agents* accordingly after an invalid assignment with unassigned nodes is discarded (l. 14), and that any nodes executed in parallel (i.e. at the same level) are assigned to different agents (l. 10). Note also that the while loop at l. 13 of MinSchedule is guaranteed to terminate as the value of *agents* is refined from its theoretical bounds in a divide-and-conquer strategy.

2) Since the number of time slots is fixed at L (Algorithm 3, l. 8), i.e. the minimal value that follows directly from the structure of DAG as its longest branch (note that $L = root(DAG).depth$), it follows that the total attack time $L \times t_{unit}$ is always minimal.

To show that the number of agents is also minimal, consider the assignment of nodes at each level L_i of DAG. The case for the top level L_0 is trivial: it only contains the root node, which cannot be executed in parallel with any other and thus can be assigned to any agent. By induction on subsequent levels L_i, we can show agents are also optimally assigned at each one. Suppose that the assignment of agents and time slots for all nodes down to and including level L_i is optimal. At L_{i+1}, there are two possibilities to consider. If $|L_{i+1}| \leqslant agents$, some agents are idle in this time slot. However, this assignment cannot be improved upon: note that any lower values of *agents* would have already been checked during earlier cycles of the while loop (l. 13), and found to produce an invalid schedule where some nodes are left without any agent asssigned (l. 16).

Conversely, if $|L_{i+1}| > agents$, some nodes will be carried over to L_{i+2}. Similarly, it follows from the divide-and-conquer scheme in which the final value of *agents* is obtained (l. 13) that decreasing the number of agents further is impossible without adding an extra slot instead.

Therefore, the assignment up to level L_{i+1} cannot be improved and is optimal for a schedule containing L time slots. Note that subsequently executed subprocedures ReshuffleSlot and ZeroAssign do not affect this in any way, since neither adds extra agents or time slots.

Thus, for any $DAG \in DAG\_set$, schedule length is fixed at its theoretical minimum, and the optimality of agent assignment for this minimal length follows from the fact time slots are filled exhaustively wherever possible, but using the lowest number of agents that does not leave unassigned nodes (i.e. an invalid schedule). Since all input DAG variants are equivalent to the original ADTree w.r.t. scheduling (by Propositions 5.3.1, 5.3.2 and 5.3.4), it also holds that the assignment is optimal for the original ADTree. □

5.4.8 Example of scheduling

We now apply these algorithms to the treasure hunters example. Figure 5.5c shows the output of the three initial subprocedures. The depth of nodes assigned by Algorithm 1 is displayed in green. The branch corresponding to attack `e` has been pruned as per Section 5.3.2. Levels assigned by Algorithm 2 are displayed in blue. Finally, the agents assignment computed by Algorithm 3 is shown in Table 5.1.

slot \ agent	1	2
125	h_3, GA', TF'_2, TS'	
124	h_2	
123	h_1, h'	
122	ST_2, TF'_1	
121	ST_1, ST'	
120	f_{120}	b_{60}
...	...	...
61	f_{61}	b_1, b'
60	f_{60}	
...	...	
1	f_1, f'	

Table 5.1: Treasure hunters ADTree: assignment of Algorithm 3.

5.5 Experimental results

The algorithm presented in this chapter has been implemented in `ADT2AMAS`, an open source tool written in written in `C++17`. Input ADTrees can be loaded from plain text files using a simple syntax, or specified manually by the user via an intuitive graphical interface. In addition to translating them to the EAMAS formalism and synthesising an optimal schedule for the minimal number of agents, the tool additionally allows for exporting the algorithm's intermediary steps (i.e. the preprocessing detailed in Section 5.3.1) as LaTeX figures for easy visualisation of the transformation of the ADTree to a DAG. The architecture of `ADT2AMAS` is described in more detail in [89].

5.5.1 Benchmarks

The algorithm was evaluated on a number of benchmarks, detailed below.

Literature case studies

The first set of benchmarks includes the three case studies already discussed in Section 4.7.1: forestall, iot-dev, gain-admin, whose ADTrees are presented in Figure 4.5, Figure 4.6, and Figure 4.7, respectively.

For forestall, there are 4 possible cases depending on which defence nodes are active, however the DAG corresponding to no defences is actually the same as the one where only `id` is active, leaving 3 unique variants. Of these, all have an optimal schedule using only 1 agent, with different attack time depending on the variants induced by active defences: 43 days for the no defence (or `id` only) case, 54 days if only `scr` is active, and 55 days if both defences occur. Note that while a single agent is sufficient to achieve minimal attack time in this benchmark, the synthesised schedules are still useful as they point out specific attacks that must be performed in order to achieve optimality. The results are reported in Table 5.2.

(a) DAG (a)

slot \ agent	1
1	$\mathtt{hr}', \mathtt{hr}_1$
2	$\mathtt{hr}_2$
...	...
9	$\mathtt{hr}_9$
10	$\mathtt{PR}'_1, \mathtt{hr}_{10}$
11	$\mathtt{reb}', \mathtt{reb}_1$
12	$\mathtt{reb}_2$
13	$\mathtt{FS}'_1, \mathtt{PR}'_2, \mathtt{PR}'_3, \mathtt{PRS}', \mathtt{SC}', \mathtt{reb}_3, \mathtt{rfc}'$
14	$\mathtt{icp}', \mathtt{icp}_1$
15	$\mathtt{icp}_2$
...	...
27	$\mathtt{icp}_{14}$
28	$\mathtt{FS}'_2, \mathtt{icp}_{15}$
29	$\mathtt{dtm}', \mathtt{dtm}_1$
30	$\mathtt{dtm}_2$
...	...
33	$\mathtt{dtm}_5$
34	$\mathtt{FS}'_3, \mathtt{FS}_1$
35	$\mathtt{FS}_2$
...	...
43	$\mathtt{FS}_{10}$

(b) DAG (b)

slot \ agent	1
1	$\mathtt{hh}', \mathtt{hh}_1$
2	$\mathtt{hh}_2$
...	...
19	$\mathtt{hh}_{19}$
20	$\mathtt{NA}'_1, \mathtt{NA}'_2, \mathtt{hh}_{20}, \mathtt{sb}'$
21	$\mathtt{heb}', \mathtt{heb}_1$
22	$\mathtt{heb}_2$
23	$\mathtt{heb}_3$
24	$\mathtt{FS}'_1, \mathtt{NA}'_3, \mathtt{NAS}', \mathtt{NA}_1, \mathtt{SC}'$
25	$\mathtt{icp}', \mathtt{icp}_1$
26	$\mathtt{icp}_2$
...	...
38	$\mathtt{icp}_{14}$
39	$\mathtt{FS}'_2, \mathtt{icp}_{15}$
40	$\mathtt{dtm}', \mathtt{dtm}_1$
41	$\mathtt{dtm}_2$
...	...
44	$\mathtt{dtm}_5$
45	$\mathtt{FS}'_{3,\mathtt{FS}_1}$
46	$\mathtt{FS}_2$
...	...
54	$\mathtt{FS}_{10}$

(c) DAG (c)

slot \ agent	1
1	$\mathtt{bp}', \mathtt{bp}_1$
2	$\mathtt{bp}_2$
...	...
14	$\mathtt{bp}_{14}$
15	$\mathtt{BRB}'_1, \mathtt{bp}_{15}$
16	$\mathtt{psc}', \mathtt{psc}_1$
17	$\mathtt{psc}_2$
...	...
22	$\mathtt{psc}_7$
23	$\mathtt{BRB}'_2, \mathtt{BRB}_1$
24	$\mathtt{BRB}_2$
25	$\mathtt{BRB}_3, \mathtt{FS}'_1, \mathtt{SC}'$
26	$\mathtt{icp}', \mathtt{icp}_1$
27	$\mathtt{icp}_2$
...	...
39	$\mathtt{icp}_{14}$
40	$\mathtt{FS}'_2, \mathtt{icp}_{15}$
41	$\mathtt{dtm}', \mathtt{dtm}_1$
42	$\mathtt{dtm}_2$
...	...
45	$\mathtt{dtm}_5$
46	$\mathtt{FS}'_{3,\mathtt{FS}_1}$
47	$\mathtt{FS}_2$
...	...
55	$\mathtt{FS}_{10}$

Table 5.2: Assignment for `forestall`.

For iot-dev, note that only 1 of 4 possible cases (no active defences) leads to a DAG, since we have that `tla` causes the failure of `GVC`, which in turn makes `APN` and then `APNS` fail, independent of `inc`. Thus, the attack necessarily fails. This is also the case if defence `inc` is active. The only way for the attack to succeed is that all defences fail, in which case the optimal schedule uses 2 agents and the attack takes 694 minutes. On the other hand, the algorithm's output provides crucial information to the defending party, namely that activating any one of the two defences is sufficient to prevent any attack. The results are reported in Table 5.3.

For gain-admin, the largest ADTree of the three case studies, there are as many as 16 possible combinations of active defences. However, it turns out they are all covered by just 3 cases: `scr` not active, `scr` active but not `DTH`, and both of them active. The fastest attack for those cases can be scheduled in in 2942, 4320 and 5762 minutes, respectively. As in the case of forestall, each requires only a single agent, with the algorithm still useful for determining particular attacks to be performed by that agent. The results are reported in Table 5.4.

Additional benchmarks

Additionally, the algorithm was tested on several smaller examples, not corresponding to or based on specific real-world security scenarios, but designed specifically to exhibit particular features and characteristics of the schedules.

slot \ agent	1	2
1	$\texttt{gc}', \texttt{gc}_1$	
2	$\texttt{gc}_2$	
...	...	
510	$\texttt{gc}_{510}$	
511	$\texttt{gc}_{511}$	$\texttt{flp}', \texttt{flp}_1$
512	$\texttt{gc}_{512}$	$\texttt{flp}_2$
...	...	...
569	$\texttt{gc}_{569}$	$\texttt{flp}_{59}$
570	$\texttt{gc}_{570}$	$\texttt{AL}'_1, \texttt{flp}_{60}$
571	$\texttt{gc}_{571}$	$\texttt{sma}', \texttt{sma}_1$
572	$\texttt{gc}_{572}$	$\texttt{sma}_2$
...	...	...
599	$\texttt{gc}_{599}$	$\texttt{sma}_{29}$
600	$\texttt{GVC}', \texttt{gc}_{600}$	$\texttt{AL}'_2, \texttt{CPN}', \texttt{sma}_{30}$
601	$\texttt{APN}', \texttt{APN}_1$	
602	$\texttt{APN}_2$	
603	$\texttt{APN}_3$	
604	$\texttt{APNS}', \texttt{APNS}_1, \texttt{CIoTD}'_1$	
605	$\texttt{esv}', \texttt{esv}_1$	
606	$\texttt{esv}_2$	
...	...	
663	$\texttt{esv}_{59}$	
664	$\texttt{CIoTD}'_2, \texttt{esv}_{60}$	
665	$\texttt{rms}', \texttt{rms}_1$	
666	$\texttt{rms}_2$	
...	...	
693	$\texttt{rms}_{29}$	
694	$\texttt{CIoTD}'_3, \texttt{rms}_{30}$	

Table 5.3: Assignment for `iot-dev`.

(a) DAG (a)

slot \ agent	1
1	$\texttt{bcc}', \texttt{bcc}_1$
2	$\texttt{bcc}_2$
...	...
2879	$\texttt{bcc}_{2879}$
2880	$\texttt{ECC}', \texttt{bcc}_{2880}$
2881	$\texttt{ECCS}', \texttt{ECCS}_1$
2882	$\texttt{ECCS}_2$
...	...
2940	$\texttt{ECCS}_{60}$
2941	$\texttt{ACLI}', \texttt{ACLI}_1$
2942	$\texttt{ACLI}_{2,\texttt{OAP}'}$

(b) DAG (b)

slot \ agent	1
1	$\texttt{th}', \texttt{th}_1$
2	$\texttt{th}_2$
...	...
4319	$\texttt{th}_{4319}$
4320	$\texttt{GSAP}', \texttt{OAP}', \texttt{TSA}', \texttt{th}_{4320}$

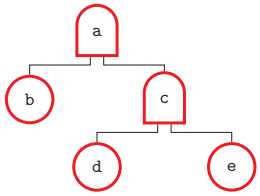
(c) DAG (c)

slot \ agent	1
1	$\texttt{co}', \texttt{co}_1$
2	$\texttt{co}_2$
...	...
5760	$\texttt{co}_{5760}$
5761	$\texttt{ACLI}', \texttt{ACLI}_1$
5762	$\texttt{ACLI}_{2,\texttt{OAP}'}$

Table 5.4: Assignment for `gain-admin`.

The first one, denoted by interrupted, demonstrates that the algorithm can produce an interleaved execution of two attacks `b` and `e`, assigned to the same agent. The ADTree, node attributes, and obtained assignment are reported in Figure 5.6.

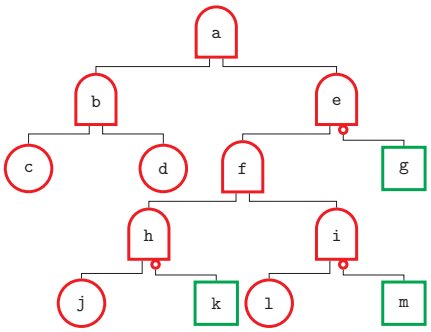
The second example, last, provides a succession of nodes with zero duration ($\texttt{a}'$, $\texttt{e}'$, $\texttt{f}'$, $\texttt{h}'$ and $\texttt{i}'$), showing that they are handled as expected. The ADTree, node attributes, and obtained assignment are reported in Figure 5.7.



Name	Time
a	0 m
b	2 m
c	1 m
d	4 m
e	3 m

slot \ agent	1	2
1	$\mathtt{d}', \mathtt{d}_1$	$\mathtt{e}', \mathtt{e}_1$
2	$\mathtt{d}_2$	$\mathtt{b}', \mathtt{b}_1$
3	$\mathtt{d}_3$	$\mathtt{e}_2$
4	$\mathtt{d}_4$	$\mathtt{e}_3$
5	$\mathtt{c}', \mathtt{c}_1$	$\mathtt{a}', \mathtt{b}_2$

Figure 5.6: Interrupted schedule example (`interrupted`) and the obtained assignment.



Name	Time
a	0 m
b	1 m
c	1 m
d	3 m
e	0 m
f	0 m
g	0 m
h	0 m
i	0 m
j	1 m
k	0 m
l	1 m
m	0 m

slot \ agent	1	2
1	$\mathtt{d}', \mathtt{d}_1$	
2	$\mathtt{d}_2$	$\mathtt{i}', \mathtt{l}', \mathtt{l}_1$
3	$\mathtt{d}_3$	$\mathtt{c}', \mathtt{c}_1$
4	$\mathtt{a}', \mathtt{b}', \mathtt{b}_1$	$\mathtt{e}', \mathtt{f}', \mathtt{h}', \mathtt{j}', \mathtt{j}_1$

Figure 5.7: Last example (`last`) and the obtained assignment.

5.6 Summary

In this chapter, we have looked at model reductions from a different point of view, i.e. firstly, aiming to minimise the number of *agents* in the system and secondly, to optimally assign them to specific tasks. Thus far, these aspects were not considered in this thesis: both the classical partial order reduction (Chapter 3) as well as the specialised pattern- and layer-based techniques for tree topologies (Chapter 4) were always applied within the context of a given *model* (and a particular **sATL*** formula). Note that, crucially, the asynchronous interleaving of agents' private events in AMAS means that their models, or Interleaved Interpreted Systems (cf. Definition 2.1.3), increase exponentially in size with the number of agents in the AMAS. Hence, even though experimental results demonstrate the efficiency of the aforementioned reduction techniques on a number of benchmarks, the practical verification of strategic ability in asynchronous systems may still prove unfeasible unless the number of components (and corresponding agents) that generate the model can be reduced first.

Clearly, any such approach needs to take into account the specifics of a particular model or class of models, at least to a certain extent. Here, we continued to focus on systems with a tree synchronisation topology, previously investigated in Chapter 4). In this setting, the non-trivial optimisation problem of synthesising the minimal schedule using the minimal number of agents for a given ADTree can be considered. Tackling it, building upon the notion of Guarded Update Systems (cf. Section 4.3), we have first generalised the latter to Extended AMAS (EAMAS), since agents are now considered. Then, we have demonstrated an algorithm that takes an ADTree and, after several preprocessing steps, produces an optimal action assignment using the minimal number of agents.

5.6.1 Related work

As in the previous chapter, attack-defence trees were extensively referred to throughout this one; thus, for general literature on this formalism, we refer the reader to Section 4.9.1. The problem of minimal scheduling for minimal number of agents in ADTrees was discussed in [6], using the translation to EAMAS automata patterns, corresponding to individual agents, proposed in [4].

While considering agents in this context is a new aspect, from the perspective of optimisation problems there is clear relevance to earlier work, in particular on parallel program scheduling with task precedence [90, 91]. Since the algorithm's input are preprocessed DAGs with normalised time, the problem falls into the Unit Computational Cost (UCC) category, whereas it can also be classified as No Communication (NC) graph scheduling, due to the fact the exchange of information between agents or nodes in all considered graphs is assumed to be instantaneous and does not incur any additional cost.

UCC problems can be effectively solved for tree-like structures [92], but cannot be directly applied to a set of DAGs. Although a polynomial solution for interval-ordered DAGs was proposed [93], that algorithm does not guarantee the minimal number of agents.

For NC problems, a number of heuristic algorithms using list scheduling were proposed [90], including Highest Levels First with No Estimated Times (HLFNET), Smallest Co-levels First with no Estimated Times (SCFNET), and Random, where nodes in the DAG are assigned priorities randomly. Variants assuming non-uniform node computation times are also considered, but are not applicable to the problem solved in this chapter. Furthermore, this class of algorithms does not aim at finding a schedule with the minimal number of processors or agents. On the other hand, known algorithms that include such a limit, i.e. for the Bounded Number of Processors (BNP) class of problems, assume non-zero communication cost and rely on the clustering technique, reducing communication, and thus schedule length, by mapping nodes to processing units. Hence, these techniques are not directly applicable.

The algorithm described in this chapter can be classified as list scheduling with a fusion of HLFNET and SCFNET heuristics, but with additional restriction on the number of agents used. The length of a schedule is determined as the length of the critical path of a graph. The number of minimal agents needed for the schedule is found with bisection.

Branching schedules analogous to the variants discussed in Section 5.3.1 have been previously explored, albeit using different models that either include probability [94] or require an additional DAG to store possible executions [95]. Zero duration nodes are also unique to the ADTree setting.

To the best of our knowledge, this is the first work dealing with agents in this context. Rather, scheduling in multi-agent systems typically focuses on agents' *choices* in cooperative or competitive scenarios, e.g. in models such as BDI [96, 97].

Finally, we note that the problem considered in this chapter can also be tackled within the framework of a general theory, as opposed to using a specialised algorithm. An approach using rewriting logic and the Maude verifier [98] has been proposed in [99], with a declarative, easily extensible model based on a set of rules, the detailed discussion of which falls outside the scope of this thesis. However, we note that although the specialised algorithm presented here typically outperforms Maude, the latter remains performant enough to be of practical use, and, more importantly, brings other unique advantages. In particular, the declarative approach allows for easily extending the concept with additional constraints, and other ADTree performance metrics than time and cost, including the possibility of a multi-objective optimisation goals.

Chapter 6

Conclusions

6.1 Summary

In this thesis, we have tackled the problem of model reduction in systems involving agents that asynchronously interleave their actions. The problem of state explosion is inherent to such asynchronous systems, making their formal verification a major challenge, and thus rendering efficient methods of reducing the state and transition spaces all but essential.

At the same time, the need for formal verification is clear, as increasingly complex systems have become prevalent in nearly all aspects of our daily lives, including applications that can be considered mission critical, where erroneous design may directly impact human welfare or even life. The aspect of strategic ability of agents, considered in this thesis, adds yet another layer of complexity compared to purely temporal, linear or branching-time logics. The question is no longer just *what* the evolution of a system in time can or will look like, but *who* can make it evolve in a particular way, and *how* can they achieve that goal.

In practical applications, electronic voting serves as an excellent setting to demonstrate the usefulness of model checking alternating-time temporal logic **ATL*** in asynchronous systems. From the point of view of formal verification, e-voting protocols and procedures are at the intersection of very relevant aspects, both technical (e.g. cryptography) and, even more so, social (e.g. trustworthiness, verifiability, resistance to various forms of coercion).

Verifying such properties is a hard computational problem, not just because of the state explosion, but first and foremost, due to the addition of strategic reasoning on top of the formalism of temporal logic. This makes our main technical result notable for a number of reasons: it demonstrates that the existing reduction technique for **LTL**, a purely temporal logic which cannot express properties involving the strategic abilities of agents, remains applicable to **ATL*** as well.

6.2 Directions for Future Research

While we have obtained notable theoretical results, in particular regarding partial order reduction, which can be adapted from **LTL** to a subset of **ATL*** at no extra cost in computational complexity, this by no means exhausts potential avenues for further research.

Firstly, *symbolic model checking* of **ATL*** could be considered, particularly in the context of potential adaptation of existing partial order reductions schemes, analogously to the approach discussed in Chapter 3. Secondly, other logical formalisms that allow for reasoning about strategic ability could be considered, with Strategy Logic (SL) being a prime example. Furthermore, the applicability of partial

order reductions could be investigated for timed logics, either with discrete or continuous time, in particular those combining the notions of time and strategic ability. Here, the recently proposed STCTL and its subset SCTL [100] are excellent candidates, as under certain assumptions (memoryless strategies with imperfect information) they offer higher expressivity than comparable logics without increasing the complexity of model checking.

As for the investigation of specialised reduction techniques for ADTrees, and the minimisation of the number of required agents, it remains a possibility to consider leveraging SAT- or SMT-solvers to produce the set of variants induced by different configurations of `OR` and defence nodes, leaving the algorithm to perform only the scheduling itself (which is already proven to be done optimally). Of course, the downside is that the algorithm remains a highly specialised solution applicable exclusively to ADTrees, and currently only to the attack time metric. Thus, extensions to other metric such as cost could be proposed, as well as formulating this optimisation problem within the framework of a general theory, such as the approach utilising rewriting logic and the Maude verifier [99], trading off the higher efficiency of the specialised algorithm for applicability to a much more general class of problems.

Bibliography

[1] W. Jamroga, W. Penczek, T. Sidoruk, P. Dembinski, and A. Mazurkiewicz, "Towards Partial Order Reductions for Strategic Ability," *Journal of Artificial Intelligence Research*, vol. 68, pp. 817–850, 2020.

[2] W. Jamroga, W. Penczek, and T. Sidoruk, "Strategic Abilities of Asynchronous Agents: Semantic Side Effects and how to tame them," in *Proceedings of KR 2021*, 2021, pp. 368–378.

[3] D. Kurpiewski, W. Jamroga, Ł. Maśko, Ł, Mikulski, W. Pazderski, W. Penczek, and T. Sidoruk, "Verification of Multi-Agent Properties in Electronic Voting: A Case Study," in *Proceedings of AiML 2022*. College Publications, 2022, pp. 531–556.

[4] J. Arias, C. Budde, W. Penczek, L. Petrucci, T. Sidoruk, and M. Stoelinga, "Hackers vs. Security: Attack-Defence Trees as Asynchronous Multi-agent Systems," in *Proceedings of ICFEM 2020*. Springer, 2020, pp. 3–19.

[5] L. Petrucci, M. Knapik, W. Penczek, and T. Sidoruk, "Squeezing State Spaces of (Attack-Defence) Trees," in *Proceedings of ICECCS 2019*. IEEE, 2019, pp. 71–80.

[6] J. Arias, L. Petrucci, Ł. Maśko, W. Penczek, and T. Sidoruk, "Minimal Schedule with Minimal Number of Agents in Attack-Defence Trees," in *Proceedings of ICECCS 2022*. IEEE, 2022, pp. 1–10.

[7] E. M. Clarke and E. A. Emerson, "Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic," in *Logics of Programs*. Berlin: Springer, 1982, pp. 52–71.

[8] J. P. Queille and J. Sifakis, "Specification and Verification of Concurrent Systems in CESAR," in *International Symposium on Programming*. Berlin: Springer, 1982, pp. 337–351.

[9] A. Pnueli, "The Temporal Semantics of Concurrent Programs," in *Semantics of Concurrent Computation*, G. Kahn, Ed. Berlin: Springer, 1979, pp. 1–20.

[10] A. N. Prior, *Time and Modality*. Greenwood Press, 1955.

[11] H. Kamp, "Tense Logic and the Theory of Linear Order," Ph.D. dissertation, University of California, 1968.

[12] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill, "Sequential Circuit Verification Using Symbolic Model Checking," in *Proceedings of the 27th ACM/IEEE Design Automation Conference*. IEEE Computer Society Press, 1990, pp. 46–51.

[13] K. Havelund and N. Shankar, "Experiments in Theorem Proving and Model Checking for Protocol Verification," in *Proceedings of FME '96*. Berlin: Springer, 1996, pp. 662–681.

[14] E. M. Clarke, S. Jha, and W. R. Marrero, "Verifying Security Protocols with Brutus," *ACM Transactions on Software Engineering and Methodology*, vol. 9, no. 4, pp. 443–487, 2000.

[15] T. Witkowski, N. Blanc, D. Kroening, and G. Weissenbacher, "Model Checking Concurrent Linux Device Drivers," in *Proceedings of ASE '07*. ACM, 2007, p. 501–504.

[16] F. Schneider, S. M. Easterbrook, J. R. Callahan, and G. J. Holzmann, "Validating Requirements for Fault Tolerant Systems using Model Checking," in *Proceedings of ICRE '98*. IEEE Computer Society, 1998, pp. 4–13.

[17] K. Havelund, M. Lowry, and J. Penix, "Formal Analysis of a Space Craft Controller Using SPIN," *IEEE Transactions on Software Engineering*, vol. 27, no. 8, pp. 749–765, 2001.

[18] G. J. Holzmann and R. Joshi, "Model-Driven Software Verification," in *Proceedings of the 11th International SPIN Workshop*, ser. Lecture Notes in Computer Science, vol. 2989. Springer, 2004, pp. 76–91.

[19] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic Model Checking: 10ˆ20 States and Beyond," in *Proceedings of LICS '90*. IEEE Computer Society, 1990, pp. 428–439.

[20] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic Model Checking Without BDDs," in *Proceedings of TACAS 1999*. Springer, 1999, pp. 193–207.

[21] R. Alur, T. A. Henzinger, and O. Kupferman, "Alternating-time Temporal Logic," *J. ACM*, vol. 49, no. 5, pp. 672–713, 2002.

[22] K. Chatterjee, T. A. Henzinger, and N. Piterman, "Strategy Logic," in *Proceedings of CONCUR '07*. Springer, 2007, p. 59–73.

[23] R. Alur, T. A. Henzinger, and O. Kupferman, "Alternating-time Temporal Logic," in *Proceedings of COMPOS '97*, ser. Lecture Notes in Computer Science, vol. 1536. Springer, 1997, pp. 23–60.

[24] M. J. Osborne and A. Rubinstein, *A Course in Game Theory*. Cambridge, Massachusetts: MIT Press, 1994.

[25] W. Jamroga, W. Penczek, P. Dembinski, and A. Mazurkiewicz, "Towards Partial Order Reductions for Strategic Ability," in *Proceedings of AAMAS '18*. ACM, 2018, pp. 156–165.

[26] P. Schobbens, "Alternating-time Logic with Imperfect Recall," *Electronic Notes in Theoretical Computer Science*, vol. 85, no. 2, pp. 82–93, 2004.

[27] N. Bulling and W. Jamroga, "Comparing Variants of Strategic Ability: How Uncertainty and Memory Influence General Properties of Games," *Journal of Autonomous Agents and Multi-Agent Systems*, vol. 28, no. 3, pp. 474–518, 2014.

[28] L. Priese, "Automata and Concurrency," *Theoretical Computer Science*, vol. 25, no. 3, pp. 221 – 265, 1983.

[29] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[30] R. Milner, *A Calculus of Communicating Systems*, ser. Lecture Notes in Computer Science. Springer, 1980, vol. 92.

[31] J. A. Bergstra and J. W. Klop, "Algebra of Communicating Processes with Abstraction," *Theoretical Computer Science*, vol. 37, pp. 77–121, 1985.

[32] R. Milner, J. Parrow, and D. Walker, "A Calculus of Mobile Processes, I," *Information and Computation*, vol. 100, no. 1, pp. 1–40, 1992.

[33] W. Jamroga, B. Konikowska, D. Kurpiewski, and W. Penczek, "Multi-valued Verification of Strategic Ability," *Fundam. Informaticae*, vol. 175, no. 1-4, pp. 207–251, 2020.

[34] M. Knapik, É. André, L. Petrucci, W. Jamroga, and W. Penczek, "Timed ATL: Forget Memory, Just Count," *Journal of Artificial Intelligence Research*, vol. 66, pp. 197–223, 2019.

[35] A. Lomuscio, W. Penczek, and H. Qu, "Partial Order Reductions for Model Checking Temporal-epistemic Logics over Interleaved Multi-agent Systems," *Fundamenta Informaticae*, vol. 101, no. 1-2, pp. 71–90, 2010.

[36] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi, *Reasoning about Knowledge*. MIT Press, 1995.

[37] W. van der Hoek and M. J. Wooldridge, "Tractable Multiagent Planning for Epistemic Goals," in *Proceedings of AAMAS '02*. ACM, 2002, pp. 1167–1174.

[38] W. Jamroga and M. Tabatabaei, "Preventing Coercion in E-Voting: Be Open and Commit," in *Proceedings of E-Vote-ID 2016*, ser. Lecture Notes in Computer Science, vol. 10141. Springer, 2016, pp. 1–17.

[39] M. Tabatabaei, W. Jamroga, and P. Y. A. Ryan, "Expressing Receipt-Freeness and Coercion-Resistance in Logics of Strategic Ability: Preliminary Attempt," in *Proceedings of PrAISeECAI 2016*. ACM, 2016, pp. 1:1–1:8.

[40] R. Gerth, R. Kuiper, D. Peled, and W. Penczek, "A Partial Order Approach to Branching Time Logic Model Checking," *Information and Computation*, vol. 150, pp. 132–152, 1999.

[41] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999.

[42] A. Mazurkiewicz, "Basic Notions of Trace Theory," in *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, ser. Lecture Notes in Computer Science, vol. 354. Springer, 1988, pp. 285–363.

[43] D. A. Peled, "Partial Order Reduction: Model-Checking Using Representatives," in *Proceedings of MFCS'96*. Springer, 1996, pp. 93–112.

[44] D. F. P. Cartier, *Problemes combinatoires de commutation et rearrangements*. Springer, 1969.

[45] A. Mazurkiewicz, "Concurrent Program Schemes and Their Interpretations," *DAIMI Report Series*, vol. 6, no. 78, 1977.

[46] ——, "Trace Theory," in *Petri Nets: Applications and Relationships to Other Models of Concurrency*, ser. Lecture Notes in Computer Science, vol. 255. Springer, 1987, pp. 278–324.

[47] D. Peled, "Combining Partial Order Reductions with On-the-fly Model-Checking," in *Proceedings of CAV '94*, ser. Lecture Notes in Computer Science, vol. 818. Springer, 1994, pp. 377–390.

[48] G. Holzmann, "On-the-fly Model Checking," *ACM Computing Surveys*, vol. 28, no. 4es, p. 120–es, 1996.

[49] P. A. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas, "Source Sets: A Foundation for Optimal Dynamic Partial Order Reduction," *Journal of the ACM*, vol. 64, no. 4, 2017.

[50] R. Alur, T. A. Henzinger, and M. Y. Vardi, "Parametric Real-time Reasoning," in *ACM Symposium on Theory of Computing, 1993*. ACM, 1993, pp. 592–601.

[51] R. Alur, T. Henzinger, F. Mang, S. Qadeer, S. Rajamani, and S. Tasiran, "MOCHA User Manual," in *Proceedings of CAV '98*, ser. Lecture Notes in Computer Science, vol. 1427, 1998, pp. 521–525.

[52] M. Knapik, A. Meski, and W. Penczek, "Action Synthesis for Branching Time Logic," *ACM Transactions on Embedded Computing Systems*, vol. 14, pp. 1–23, 2015.

[53] F. Belardinelli, A. V. Jones, and A. Lomuscio, "Model Checking Temporal-Epistemic Logic Using Alternating Tree Automata," *Fundamenta Informaticae*, vol. 112, no. 1, pp. 19–37, 2011.

[54] D. Peled, "All from One, One for All: on Model Checking Using Representatives," in *Proceedings of CAV '93*. Berlin: Springer, 1993, pp. 409–423.

[55] W. Jamroga, M. Knapik, and D. Kurpiewski, "Fixpoint Approximation of Strategic Abilities under Imperfect Information," in *Proceedings of AAMAS '17*. ACM, 2017, pp. 1241–1249.

[56] A. Valmari, "Stubborn Sets for Reduced State Space Generation," in *Advances in Petri Nets 1990*. Berlin: Springer, 1991, pp. 491–515.

[57] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Berlin: Springer, 1996.

[58] F. M. Bønneland, P. G. Jensen, K. G. Larsen, M. Muñiz, and J. Srba, "Partial Order Reduction for Reachability Games," in *Proceedings of CONCUR 2019*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, pp. 23:1–23:15.

[59] ——, "Stubborn Set Reduction for Two-Player Reachability Games," *Logical Methods in Computer Science*, vol. 17, no. 1, 2021.

[60] T. Neele, T. A. C. Willemse, W. Wesselink, and A. Valmari, "Partial-Order Reduction for Parity Games and Parameterised Boolean Equation Systems," *International Journal on Software Tools for Technology Transfer*, vol. 24, no. 5, pp. 735–756, 2022.

[61] C. Flanagan and P. Godefroid, "Dynamic Partial-Order Reduction for Model Checking Software," in *Proceedings of POPL '05*. ACM, 2005, pp. 110–121.

[62] P. Godefroid, "Model Checking for Programming Languages Using VeriSoft," in *Proceedings of POPL '97*. ACM, 1997, p. 174–186.

[63] S. Aronis, B. Jonsson, M. Lång, and K. Sagonas, "Optimal Dynamic Partial Order Reduction with Observers," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2018, pp. 229–248.

[64] S. Kan, Z. Huang, Z. Chen, W. Li, and Y. Huang, "Partial Order Reduction for Checking LTL Formulae with the Next-time Operator," *Journal of Logic and Computation*, vol. 27, no. 4, pp. 1095–1131, 2016.

[65] A. Buldas, P. Laud, J. Priisalu, M. Saarepera, and J. Willemson, "Rational Choice of Security Measures via Multi-parameter Attack Trees," in *Critical Information Infrastructures Security*. Springer, 2006, pp. 235–248.

[66] R. Kumar, E. Ruijters, and M. Stoelinga, "Quantitative Attack Tree Analysis via Priced Timed Automata," in *Proceedings of FORMATS 2015*, ser. Lecture Notes in Computer Science, vol. 9268. Springer, 2015, pp. 156–171.

[67] M. Steiner and P. Liggesmeyer, "Qualitative and Quantitative Analysis of CFTs Taking Security Causes into Account," in *Computer Safety, Reliability, and Security*. Springer, 2015, pp. 109–120.

[68] R. Kumar, S. Schivo, E. Ruijters, B. M. Yildiz, D. Huistra, J. Brandt, A. Rensink, and M. Stoelinga, "Effective Analysis of Attack Trees: A Model-Driven Approach," in *Fundamental Approaches to Software Engineering*. Springer, 2018, pp. 56–73.

[69] J. D. Weiss, "A System Security Engineering Process," in *Proceedings of the 14th National Computer Security Conference*, 1991, pp. 572–581.

[70] A. Jürgenson and J. Willemson, "Computing Exact Outcomes of Multi-parameter Attack Trees," in *Proceedings of OTM 2008*. Springer, 2008, pp. 1036–1051.

[71] B. Kordy, L. Piètre-Cambacédès, and P. Schweitzer, "DAG-based Attack and Defense Modeling: Don't Miss the Forest for the Attack Trees," *Computer Science Review*, vol. 13-14, pp. 1–38, 2014.

[72] S. Mauw and M. Oostdijk, "Foundations of Attack Trees," in *Proceedings of ICISC 2005*. Springer, 2006, pp. 186–198.

[73] R. Jhawar, B. Kordy, S. Mauw, S. Radomirović, and R. Trujillo-Rasua, "Attack Trees with Sequential Conjunction," in *ICT Systems Security and Privacy Protection*. Springer, 2015, pp. 339–353.

[74] C. Salter, O. Saydjari, B. Schneier, and J. Wallner, "Toward a Secure System Engineering Methodology," in *Proceedings of NSPW '98*. ACM, 1998, pp. 2–10.

[75] B. Kordy, S. Mauw, S. Radomirović, and P. Schweitzer, "Foundations of Attack-Defense Trees," in *Proceedings of FAST 2010*, ser. Lecture Notes in Computer Science, vol. 6561. Springer, 2011, pp. 80–95.

[76] ——, "Attack–Defense Trees," *Journal of Logic and Computation*, vol. 24, no. 1, pp. 55–87, 2014.

[77] Z. Aslanyan and F. Nielson, "Pareto Efficient Solutions of Attack-Defence Trees," in *Principles of Security and Trust*, ser. Lecture Notes in Computer Science. Springer, 2015, vol. 9036, pp. 95–114.

[78] A. Bossuat and B. Kordy, "Evil Twins: Handling Repetitions in Attack-Defense Trees - A Survival Guide," in *Proceedings of GraMSec 2017*, ser. Lecture Notes in Computer Science, vol. 10744. Springer, 2017, pp. 17–37.

[79] B. Kordy and W. Widel, "On Quantitative Analysis of Attack-Defense Trees with Repeated Labels," in *Proceedings of POST 2018*, ser. Lecture Notes in Computer Science, vol. 10804. Springer, 2018, pp. 325–346.

[80] B. Fila and W. Widel, "Efficient Attack-Defense Tree Analysis Using Pareto Attribute Domains," in *Proceedings of CSF 2019*. IEEE, 2019, pp. 200–215.

[81] O. Gadyatskaya, R. R. Hansen, K. G. Larsen, A. Legay, M. C. Olesen, and D. B. Poulsen, "Modelling Attack-Defense Trees Using Timed Automata," in *Formal Modeling and Analysis of Timed Systems*. Springer, 2016, vol. 9884, pp. 35–50.

[82] R. Kumar and M. Stoelinga, "Quantitative Security and Safety Analysis with Attack-Fault Trees," in *Proceedings of HASE 2017*. IEEE, 2017, pp. 25–32.

[83] Dalton, Mills, Colombi, and Raines, "Analyzing Attack Trees Using Generalized Stochastic Petri Nets," in *Proceedings of the 2006 IEEE Information Assurance Workshop*, 2006, pp. 116–123.

[84] F. Arnold, D. Guck, R. Kumar, and M. Stoelinga, "Sequential and Parallel Attack Tree Modelling," in *Computer Safety, Reliability, and Security*. Springer, 2015, pp. 291–299.

[85] R. Kumar, D. Guck, and M. Stoelinga, "Time Dependent Analysis with Dynamic Counter Measure Trees," *CoRR*, vol. abs/1510.00050, 2015.

[86] M. Gribaudo, M. Iacono, and S. Marrone, "Exploiting Bayesian Networks for the Analysis of Combined Attack Trees," *Electronic Notes in Theoretical Computer Science*, vol. 310, pp. 91–111, 2015.

[87] Z. Aslanyan, F. Nielson, and D. Parker, "Quantitative Verification and Synthesis of Attack-Defence Scenarios," in *Proceedings of CSF 2016*. IEEE, 2016, pp. 105–119.

[88] H. Hermanns, J. Krämer, J. Krčál, and M. Stoelinga, "The Value of Attack-Defence Diagrams," in *Proceedings of POST 2016*, ser. Lecture Notes in Computer Science, vol. 9635. Springer, 2016, pp. 163–185.

[89] J. Arias, W. Penczek, L. Petrucci, and T. Sidoruk, "ADT2AMAS: Managing Agents in Attack-Defence Scenarios," in *Proceedings of AAMAS '21*. ACM, 2021, pp. 1749–1751.

[90] T. L. Adam, K. M. Chandy, and J. R. Dickson, "A Comparison of List Schedules for Parallel Processing Systems," *Communications of the ACM*, vol. 17, no. 12, p. 685–690, 1974.

[91] Y. Kwok and I. Ahmad, "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors," *ACM Computing Surveys*, vol. 31, no. 4, pp. 406–471, 1999.

[92] T. C. Hu, "Parallel Sequencing and Assembly Line Problems," *Operations Research*, vol. 9, no. 6, pp. 841–848, 1961.

[93] C. H. Papadimitriou and M. Yannakakis, "Scheduling Interval-Ordered Tasks," *SIAM Journal on Computing*, vol. 8, no. 3, pp. 405–409, 1979.

[94] D. F. Towsley, "Allocating Programs Containing Branches and Loops Within a Multiple Processor System," *IEEE Transactions on Software Engineering*, vol. 12, no. 10, pp. 1018–1024, 1986.

[95] H. El-Rewini and H. H. Ali, "Static Scheduling of Conditional Branches in Parallel Programs," *Journal of Parallel and Distributed Computing*, vol. 24, no. 1, pp. 41–54, 1995.

[96] I. Nunes and M. Luck, "Softgoal-based Plan Selection in Model-driven BDI Agents," in *Proceedings of AAMAS '14*. IFAAMAS, 2014, pp. 749–756.

[97] M. Dann, J. Thangarajah, Y. Yao, and B. Logan, "Intention-Aware Multiagent Scheduling," in *Proceedings of AAMAS '20*. IFAAMAS, 2020, pp. 285–293.

[98] S. Eker, J. Meseguer, and A. Sridharanarayanan, "The Maude LTL Model Checker and Its Implementation," in *Proceedings of SPIN '03*. Springer, 2003, p. 230–234.

[99] J. Arias, C. Olarte, L. Petrucci, Ł. Maśko, W. Penczek, and T. Sidoruk, "Optimal Scheduling of Agents in ADTrees: Specialised Algorithm and Declarative Models," *CoRR*, vol. abs/2305.04616, 2023.

[100] J. Arias, W. Jamroga, W. Penczek, L. Petrucci, and T. Sidoruk, "Strategic (Timed) Computation Tree Logic," in *Proceedings of AAMAS '23*. ACM, 2023, pp. 382–390.

Index