Model Checking and Compositional Reasoning for Multi-Agent Systems

Andrew Vaughan Jones

2015
The copyright of this thesis rests with the author and is made available under a Creative Commons Attribution-Non Commercial-No Derivatives licence. Researchers are free to copy, distribute or transmit the thesis on the condition that they attribute it, that they do not use it for commercial purposes and that they do not alter, transform or build upon it. For any reuse or distribution, researchers must make clear to others the license terms of this work.
Declaration

I herewith certify that all material in this dissertation that is not my own work has been properly acknowledged.

Andrew Vaughan Jones
Abstract

Multi-agent systems are distributed systems containing interacting autonomous agents designed to achieve shared and private goals. For safety-critical systems where we wish to replace a human role with an autonomous entity, we need to make assurances about the correctness of the autonomous delegate.

Specialised techniques have been proposed recently for the verification of agents against mentalistic logics. Problematically, these approaches treat the system in a monolithic way. When verifying a property against a single agent, the approaches examine all behaviours of every component in the system. This is both inefficient and can lead to intractability: the so-called state-space explosion problem.

In this thesis, we consider techniques to support the verification of agents in isolation. We avoid the state-space explosion problem by verifying an individual agent in the context of a specification of the rest of the system, rather than the system itself. We show that it is possible to verify an agent against its desired properties without needing to consider the behaviours of the remaining components.

We first introduce a novel approach for verifying a system as a whole against specifications expressed in a logic of time and knowledge. The technique, based on automata over trees, supports an efficient procedure to verify systems in an automata-theoretic way using language containment.

We show how the automata-theoretic approach can be used as an underpinning for assume-guarantee reasoning for multi-agent systems. We use a temporal logic of actions to specify the expected behaviour of the other components in the system. When performing modular verification, this specification is used to exclude behaviours that are inconsistent with the concrete system.

We implement both approaches within the open-source model checker mcmas and show that, for the relevant properties, the assume-guarantee approach can significantly increase the tractability of individual agent verification.
“An algorithm that forgets its past is doomed to repeat it”

– Unknown.
Acknowledgements

My initial thanks of course have to go to Alessio Lomuscio for our discussions during my doctoral research and for providing me with the chance to conduct research at Imperial.

My thanks also go to Franco Raimondi and Marek Sergot, my examiners. Their exhaustive comments have been invaluable in improving my thesis.

I am also extremely grateful to Wojciech Penczek who originally enlightened me to the existence of alternating automata, and their applications to model checking for branching-time logics. Wojciech’s suggestions very much influenced a large part of the direction of this thesis.

I was fortunate enough have a collaboration with Francesco Belardinelli at the early stages of my Ph.D. on which parts of this thesis are built – this was extremely beneficial, and I am thankful to still consider Francesco a collaborator.

Where would these acknowledgements be without thanks to my parents – for their love, emotional and, of course, financial support.

As always, the best and most important are relegated to the end of the list: my uncountable thanks (no, they cannot be mapped to $\mathbb{N}$!) to Kim – for her constant support, love, proof-reading, endless hours spent not having fun, just to get me to the end . . .

. . . time to go diving!
Contents

List of Tables ........................................................................... 15
List of Figures ........................................................................... 17

1 Introduction ................................................................. 19
  1.1 Motivation ................................................................. 19
  1.2 Definition of the Problem ............................................ 23
    1.2.1 Aims and Objectives ........................................... 23
  1.3 Contributions ............................................................ 24
  1.4 Relevance and Applications .......................................... 26
  1.5 Collaboration and Publications ...................................... 27
  1.6 Summary of Contents .................................................. 27

2 Related Work ................................................................. 29
  2.1 Compositional Verification ............................................. 29
    2.1.1 Assume-Guarantee Reasoning ............................... 30
    2.1.2 Learning-based Assume-Guarantee Reasoning ...... 32
  2.2 Logics for Reasoning about Computations ....................... 35
    2.2.1 Modular Model Checking and Module Checking ..... 35
    2.2.2 Model Checking with Path Criterion .................... 36
    2.2.3 Local and Modular Specifications ......................... 38
    2.2.4 Automata-Theoretic Verification for Branching Structures ... 39
  2.3 Reducing the State Space for Multi-Agent Systems .......... 41
    2.3.1 Symmetry Reduction ........................................... 41
    2.3.2 Partial Order Reduction ....................................... 42
    2.3.3 Parametric Verification ....................................... 42
    2.3.4 Existential Abstraction ....................................... 42
    2.3.5 Modular Verification for Multi-Agent Systems .......... 43
    2.3.6 Modular Verification for Multi-Modal Epistemic Logic ... 45
  2.4 Identified Shortcomings and Potential Opportunities .......... 45
# Background and Preliminaries

3 Contents

## Modal Logic and Kripke Structures
3.1 Propositional Modal Logic
3.1.1 Semantics and Abbreviations
3.1.3 Multi-Modal Logic

## Temporal Logics
3.2 Transition Systems
3.2.2 Linear Temporal Logic
3.2.3 LTL Semantics and Abbreviations
3.2.4 Computational Tree Logic
3.2.5 CTL Semantics and Abbreviations
3.2.6 Further Logics

## Reasoning about Multi-Agent Systems
3.3.1 Interpreted Systems
3.3.2 Compositions

## Automata
3.4.1 Automata on Infinite Words
3.4.2 Automata on Infinite Trees
3.4.3 Expressivity of Automata

## Model Checking
3.5.1 For LTL
3.5.2 For CTL
3.5.3 For CTLK

# Automata-Theoretic Verification for Temporal-Epistemic Logic

4 Contents

## Introduction
4.1

## From Interpreted Systems to Trees
4.2.1 High-level Methodology
4.2.2 Example – Unwinding an Interpreted System
4.2.3 CTLK over Trees

## Weak Epistemic Alternating Automata
4.3

## From CTLK to Automata
4.4.1 Example – $L_m$ to WEAA

## Constructing the Product Automaton
4.5.1 Example – The Product Automaton $A_{P,\phi}$

## Concluding Remarks
4.6

# Compositional Verification for Multi-Agent Systems

5 Contents

## Introduction
5.1

## A Modular Formalisation for Interpreted Systems
5.2.1 Interaction-Defined Interpreted Systems
5.2.2 Logics over IDIS
7.2 Future Work .............................................................. 192
  7.2.1 “Many Agent Systems” .......................................... 192
  7.2.2 Application of Automata Learning Frameworks .............. 194
  7.2.3 Truly “Modular” Interpreted Systems ......................... 195
  7.2.4 Richer Assumptions ....................................................... 196
  7.2.5 Symbolic Rabin Fairness .............................................. 198
  7.2.6 Symbolic Encoding of the Product Automaton ............... 199
7.3 Closing Remarks ................................................................. 199

References .............................................................................. 201
List of Tables

<table>
<thead>
<tr>
<th>Table Number</th>
<th>Table Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3</td>
<td>Common Acronyms for Automata</td>
<td>63</td>
</tr>
<tr>
<td>3.4</td>
<td>The structure of the transition relation for an alternating automata for a</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td>CTL formula</td>
<td></td>
</tr>
<tr>
<td>4.1</td>
<td>The transition function $\delta$ in the automaton $A_{D,W}$</td>
<td>87</td>
</tr>
<tr>
<td>5.1</td>
<td>Satisfaction of a $\forall$CTLK formula in $M_{(i,\phi)}$</td>
<td>124</td>
</tr>
<tr>
<td>5.2</td>
<td>The transition function $\delta$ for a LEAA based on a $\forall$CTLK formula</td>
<td>127</td>
</tr>
<tr>
<td>6.1</td>
<td>The rules for constructing the data structure $tf$ for a formula $\psi$</td>
<td>138</td>
</tr>
<tr>
<td>6.2</td>
<td>Gossip Protocol Specifications</td>
<td>141</td>
</tr>
<tr>
<td>6.3</td>
<td>Model Checking The Gossip Protocol</td>
<td>142</td>
</tr>
<tr>
<td>6.4</td>
<td>Faulty Train-Gate-Controller Specifications</td>
<td>144</td>
</tr>
<tr>
<td>6.5</td>
<td>Model Checking The Train-Gate-Controller</td>
<td>144</td>
</tr>
<tr>
<td>6.6</td>
<td>Boolean Operators Over Three Values</td>
<td>157</td>
</tr>
<tr>
<td>6.7</td>
<td>Acceptance Criteria</td>
<td>159</td>
</tr>
<tr>
<td>6.8</td>
<td>Train-Gate-Controller: Comparison 1 – Memory and Time</td>
<td>167</td>
</tr>
<tr>
<td>6.9</td>
<td>Train-Gate-Controller: Comparison 1 – BDD Variables and Reachable States</td>
<td>168</td>
</tr>
<tr>
<td>6.10</td>
<td>Train-Gate-Controller: Comparison 2 – BDD Variables and Reachable States</td>
<td>169</td>
</tr>
<tr>
<td>6.11</td>
<td>Train-Gate-Controller: Comparison 2 – Memory and Time</td>
<td>169</td>
</tr>
<tr>
<td>6.12</td>
<td>Contract Violation Criteria for the Insurance Company</td>
<td>171</td>
</tr>
<tr>
<td>6.13</td>
<td>Satisfaction of Guarantees in the Software Development Protocol</td>
<td>176</td>
</tr>
<tr>
<td>6.14</td>
<td>Software Development Protocol: Guarantee Check – States, Time and Memory</td>
<td>177</td>
</tr>
<tr>
<td>6.15</td>
<td>Software Development Protocol: Assumption Check – Time</td>
<td>179</td>
</tr>
<tr>
<td>6.16</td>
<td>Software Development Protocol: Assumption Check – Memory</td>
<td>180</td>
</tr>
</tbody>
</table>
6.18 Software Development Protocol: Assumption Check – BDD Variables and Representable States .......................... 183
List of Figures

2.1 A Typical Assume-Guarantee Inference Rule ............................ 31
2.2 Learning Assumptions for Assume-Guarantee Reasoning ............... 33

3.1 Equivalence between $\omega$-automata. ................................. 63
3.2 Non-deterministic Büchi Automaton for $Gp$. .......................... 68
3.3 Deterministic Rabin Automaton for $Gp$. ............................... 68

4.1 An example of an interpreted system ................................... 80
4.2 The epistemic alternating tree automaton “unwinding” of the
   interpreted system in Figure 4.1 ....................................... 82
4.3 A sub-tree of the automaton $A_{P,\varphi}$ ................................ 93

5.1 An excerpt of an ISPL evolution function ............................... 100
5.2 The temporal evolution for the exemplary agent $Agent_1$. .......... 106
5.3 The temporal evolution for the exemplary agent $Agent_1$. .......... 107
5.4 The temporal evolution for the exemplary agent $Agent_2$. .......... 108
5.5 The temporal evolution for the exemplary agent $Agent_1$. .......... 110
5.6 The temporal evolution for the exemplary agent $Agent_2$. .......... 110
5.7 The temporal evolution for the exemplary agent $Agent_3$. .......... 111
5.8 The DRA $A_{\varphi}$ for $FGa$. ............................................. 118
5.9 The agent $Simple$. ...................................................... 119
5.10 Invalid Runs and Epistemic Formulae. ............................... 128
5.11 The simple interaction-defined interpreted system model $\mathcal{M}$ .... 130
5.12 Inference rules for linear-branching assume-guarantee .............. 131
5.13 Translation of inference rules, shown in Figure 5.12, to their model
   checking equivalents .................................................... 131
5.14 Inference rules for $n$-agent linear-branching assume-guarantee .... 133

6.1 Train-Gate-Controller Example .......................................... 143
6.2 LTL2DSTAR workflow for converting a given A-LTL formula $\phi$ in an ISPL file ................................................................. 148
6.3 The SPIN never claim for the NBA of the formula $G(p \rightarrow FGq)$ generated by LTL2BA ......................................................... 149
6.4 A visualisation of the NBA from Figure 6.3 ................................. 150
6.5 The GraphViz dot diagram for the DRA of $G(p \rightarrow FGq)$ automatically generated by LTL2DSTAR .............................................. 151
6.6 A partial ISPL snippet for a deterministic Rabin observer agent ...... 152
6.7 A partial ISPL snippet for a deterministic Rabin property agent ...... 153
6.8 An example model with two invalid successors and one valid successor 158
6.9 Workflow using agr-mcmas .................................................... 162
6.10 Interactions in the Train-Gate-Controller Model .............................. 164
6.11 Example ISPL for a Single Train .............................................. 165
6.12 Property environment for a train ............................................. 166
6.13 Interactions in the Software Development Protocol ......................... 171
6.14 ISPL for the Insurance Company ............................................. 174
6.15 Modified Property Environment for the Insurance Company Assumption 175

7.1 Inference rules for $n$-agent, multi-assumption linear-branching assume-guarantee reasoning .............................................. 192
7.2 Inference rules for $n$-agent, multi-assumption linear-branching assume-guarantee-based model checking .............................. 193
Chapter 1
Introduction

1.1 Motivation

Multi-agent systems (MAS) are systems composed of multiple components, or agents, which exhibit autonomous behaviour, as well as social abilities such as learning and cooperation. So-called rational agents are those with the ability to achieve their goals without being given an explicit procedure on how to do so, as well as making decisions that “make sense”.

Given the autonomous behaviour of individual agents, it is common for systems that contain few agents (or even a single agent) interfacing with an environment to be classed as an “autonomous system”. These so-called autonomous systems have gained extreme prominence within recent years, especially in the context of vehicles. Examples include autonomous surface vehicles (ASVs), unmanned aerial vehicles (UAVs) and autonomous underwater vehicles (AUVs).

In light of the rational and autonomous facets that these agents possess, it has become common to specify properties about the mental attitudes of these agents. This has given rise to the use of mentalistic logics such as epistemic logic (the logic of knowledge), deontic logic (the logic of obligations), doxastic logic (the logic of belief) and coalitional logic (logics dealing with strategies and what groups of agents can achieve) when wanting to reason about the behaviours of agents.

In this thesis, we particularly focus on the use of epistemic logic and in trying to decide what an agent can “know” about its own behaviour and the environment in which it is situated.

The formal verification of computer systems is a set of techniques oriented to enable the designers to provide assurances about quality and correctness. Model checking is one such approach; it is an automated technique for reasoning about the behaviours of a system with respect to a property specified in a given logic. The problem of reasoning about the knowledge of agents in a given multi-agent system is known as temporal-epistemic model checking.
While considerable work has been carried out in the field of model checking for multi-agent systems [Lomuscio and Penczek, 2012], the current state-of-the-art still falls short of enabling engineers to verify industrial-scaled multi-agent systems. It therefore remains of paramount importance to either develop new and novel techniques, or improve those that already exist, such that model checking toolkits are able to tackle complex multi-agent systems.

One problem that limits the wider application of model checking is the state-space explosion problem. This is where the number of states of the system increases exponentially with the number of components comprising that system. Furthermore, the complexity of modern systems is starting to push the limit of what can feasibly be verified with current methods [Emerson, 2008].

The state-space explosion problem has a particularly adverse effect in multi-agent systems, as they are inherently based on multiple, distributed components interacting. While also having to overcome the state-space explosion, the verification of multi-agent systems also has to cater for specialised procedures enabling the reasoning about the mental attitude of each agent. Although recent research has led to considerable advances in the verification of agent-based systems [Lomuscio and Penczek, 2012], we are—once again—reaching the limits of what can be feasibly verified when dealing with the system as a whole.

One solution to the state-space explosion problem is the family of techniques that fall under the umbrella of “compositional reasoning” [de Roever et al., 1998]. The aim of these techniques is to support verification in a divide-and-conquer manner. By taking advantage of the natural decompositions of a system into its components, and then verifying these components in isolation, it is possible to avoid the aforementioned combinatorial blow-up.

A proposed solution.

Given that each agent in a multi-agent system is typically developed in isolation, it would be practical to also verify each agent in this manner. This is analogous to how modern software is constructed: components are constructed without knowing the exact implementation details of the other components. Each component is, however, constructed with the knowledge of a specification of the other components’ interface and expected behaviour. In modern software development, such a specification would be the application programming interface (API) of the components we wish to interact with.

Assume-guarantee reasoning is one particular approach to compositional verification [Berezin et al., 1998; Clarke et al., 1999]. Under assume-guarantee reasoning, components are examined in isolation and verified against a guarantee in the context of an assumption. The assumption is an assertion about the rest of the system in which the component is situated; as with traditional software, this assumption could be based
on the specification of the system in which the agent (or component) is expected to execute. By comparison, the guarantee is a logical specification of the component’s requirements, expressing its desired mental evolution over time. Therefore, in this context, the verification task is to ascertain if a given component, when acting in any environment satisfying the given assumption, meets the guarantee.

Consider the parallel execution of an arbitrary agent \( \text{Agent} \) with an environment \( \text{Env} \); we denote by \( \text{Env} \| \text{Agent} \) this composition. Under this parallel execution, the agent and the environment “communicate” by observation of the actions that the other performs.

Further, we can take the specification of \( \text{Env} \), and formalise this as an assumption \( \text{Ass} \), such that the behaviours of \( \text{Env} \) are a subset of (or equal to) the behaviours of \( \text{Ass} \) (i.e., \( \text{Ass} \) is an abstraction—in the sense that it can exhibit more behaviours—of \( \text{Env} \)). It follows that, as the \( \text{Agent} \)’s behaviour depends on the behaviour of \( \text{Env} \) (due to \( \text{Agent} \) observing \( \text{Env} \)’s actions), composing the \( \text{Agent} \) with a component that has more possible behaviours will lead to more potential behaviours in the \( \text{Agent} \).

In such a scenario, we therefore have the following (where “Behaviours” encapsulates the temporal evolution of a component or a composition):

\[
\text{Behaviours (Env)} \subseteq \text{Behaviours (Ass)} \\
\Downarrow \\
\text{Behaviours (Env}\|\text{Agent)} \subseteq \text{Behaviours (Ass}\|\text{Agent)}
\]

That is if \( \text{Ass} \) can do everything that \( \text{Env} \) can do, then it follows that every sequence of behaviours of \( \text{Agent} \) composed with \( \text{Env} \) will also be a sequence of behaviours in the composition between \( \text{Agent} \) and \( \text{Ass} \).

Further, if the behaviours exhibited by \( \text{Ass} \) composed with \( \text{Agent} \) (i.e., \( \text{Ass}\|\text{Agent} \)) are a subset of the “good” or desirable behaviours of the system then, following the transitivity of the subset operator, we have that all of the behaviours of \( \text{Env}\|\text{Agent} \) should also be “good”.

More precisely:

\[
\text{Behaviours (Env)} \subseteq \text{Behaviours (Ass)} \\
\text{Behaviours (Agent}\|\text{Ass)} \subseteq \text{Good_Behaviours} \\
\Downarrow \\
\text{Behaviours (Agent}\|\text{Env)} \subseteq \text{Good_Behaviours}
\]

Under the assume-guarantee paradigm, we can therefore decompose the verification problem into two parts:

1. Checking that the behaviours of the environment are a subset of those permitted by the assumption
2. Checking that the agent, when composed with its environmental assumption, satisfies its own requirements

To make this approach practical, the size of the assumption, when represented as a finite-state transition system, should be significantly smaller than the size of the environment. While this may lead to a possible over-calculation of the behaviours of the agent, this is still an easier problem than calculating the full behaviours for the original composition. By composing the “agent under test” with the assumption rather than the genuine environment, this also allows us to effectively hide the other components in the system. This further increases the scalability and efficacy of the verification approach.

While there have been numerous alternative approaches to alleviate the state-space explosion problem in MAS-based verification by attempting to reduce the size of the input system [Lomuscio and Penczek, 2012], they fall under two common categories:

- Support for homogeneous components – under approaches such as symmetry reduction [Cohen et al., 2009a; Cohen et al., 2009b] or parametric verification [Kouvaros and Lomuscio, 2013a; Kouvaros and Lomuscio, 2013b], we can reduce the state-space for instances that contain many replicated agents of the same type, by removing duplicated identical components
- Variable hiding/slicing – techniques such as existential abstraction [Cohen et al., 2009c; Lomuscio et al., 2010b] work by clustering states together and, by “hiding” information in the system, reduce its overall size

However, with compositional reasoning, we are able to break apart the verification problem even with two heterogeneous components (compared to parametric verification), and we also verify the actual agent program and not a transformed version with fewer states (compared to existential abstraction).

Verifying an agent exactly as specified is extremely useful as it allows us to be more certain that an error identified by the decision procedure may be genuine, rather than due to an oversimplification in the verification methodology (e.g., as can be the case in existential abstraction [Clarke et al., 1994]). However, while a positive result carries across to the full composition in the assume-guarantee paradigm, a negative result is still somewhat problematic: this could either be a genuine problem in the component, or the assumption selected may have been too weak to exclude erroneous behaviours that do not exist in the genuine system.

Still, by focusing on reusable components, which should meet their specifications (i.e., the guarantee) in any environment meeting certain criteria (i.e., the assumption), it is clear that a negative result here means the component does not satisfy its requirements in all environments that the agent may potentially be situated in.
1.2 Definition of the Problem

The family of techniques classed as “modular reasoning” attempt to alleviate the state-space explosion problem by dealing with each component individually, and therefore avoiding the combinatorial explosion when taking these components in composition. However, so far very little attention has been paid to applying compositional methods to either multi-agent systems or to richer mentalistic logics (e.g., temporal-epistemic logics). This is the area that this thesis aims to explore.

In traditional “reactive systems” verification, all of the information needed to calculate the “next-state” relation used for both building the state-space and verifying temporal formulae can be derived automatically from the program text. However, this is not the case for epistemic logic. To verify an epistemic formula, the “next-state” relation for knowledge can only be derived given a concrete local configuration for the agent and the set of all reachable states. As such, verifying temporal-epistemic formulae—even in a non-compositional way—introduces a selection of theoretical issues that need to be overcome.

While some techniques do exist for automatically performing compositional verification—none supporting temporal-epistemic logic—most techniques are manual. This requires the user to provide an implementation of the assumption, rather than using a specification directly [Henzinger et al., 1998; Henzinger et al., 2002]. As such, it is necessary not only for the user to implement a correct assumption, but the tools do not endeavour to show if the real environment is a refinement of the selected assumption (i.e., by automatically showing that all behaviours of the environment are contained within the assumption).

To this end, this thesis also considers an approach to the verification of multi-agent systems using automata. Model checking via Büchi automata, originally considered by Vardi et al. [Vardi and Wolper, 1986], is one of the foremost approaches in the verification of reactive systems and constitutes the basis of the model checker SPIN [Holzmann, 2004]. The original work [Vardi and Wolper, 1986] covered linear-time only; this was extended to branching-time logic in [Kupferman et al., 2000], where the formalism of alternating tree automata (ATA) was applied. Consequently, we investigate the use of alternating tree automata for the verification of multi-agent systems.

1.2.1 Aims and Objectives

Given the current shortcomings of the state-of-the-art, this thesis proposes an advancement of the existing compositional reasoning techniques to support not only the multi-agent paradigm, but also their associated knowledge-based properties.

The overall aim of this thesis is to test the following research hypothesis:
24 1 Introduction

Research Hypothesis

Model checking techniques based on modular reasoning will make the verification of multi-agent systems more tractable when compared to the existing monolithic approaches.

In our hypothesis, we have three criteria for tractable: shrinking the number of reachable states, lowering the time taken for verification and reducing the maximum memory required.

In this vein, this thesis will develop methods of compositional reasoning, i.e., divide-and-conquer approaches, for the verification of multi-agent systems that support suitable mentalistic logics. This work aims to have both a theoretical contribution (i.e., a formal and correct basis of compositional reasoning of agents), as well as an implementation of an automated verification tool for multi-agent systems, supporting techniques for ameliorating the state-space explosion problem.

The high-level objectives of this thesis are as follows:

i Investigate novel techniques for the verification of multi-agent systems, based in part on a notion of modularity.
ii Implement selected techniques as either new tools or extensions to existing tools.
iii Compare the effectiveness of the solutions using benchmarks selected from the literature.

1.3 Contributions

The research outcomes and contributions presented in this thesis are:

i Two novel techniques for the verification of multi-agent systems (one based on automata and one based on modularity)
ii A new tool (ETAV) and three tool extensions (two extending MCMAS-1.0: AT-MCMAS and AGRMCMAS, and one extending LTL2DSTAR: DRA2ISPL)
iii Experimental results obtained by comparing the various extensions to an existing state of the art model checker for multi-agent systems

On our route to a flexible approach for modular reasoning, we introduce a novel verification method based on automata. Unlike existing approaches adopted currently for the verification of multi-agent systems that use symbolic representations (e.g., based on binary decision diagrams [Raimondi and Lomuscio, 2005; Raimondi and Lomuscio, 2004] or via a reduction to Boolean satisfiability [Penczek and Lomuscio, 2003a]), the automata-based approach [Vardi and Wolper, 1986] supports a language-theoretic
means for the verification of multi-agent systems. Similar to techniques for linear-temporal logic, we note that checking a temporal-epistemic formula can be transformed to checking the language non-emptiness for an automaton representing a composition of the system and the formula.

Furthermore, we break down the main contributions as follows:

1. Methodological

- We introduce a novel automata-theoretic model checking approach for multi-agent systems.
- We present a modular reasoning-based approach to verifying an individual agent in an isolated manner against both parts of an assume-guarantee rule.

2. Theory

- We provide proofs of correctness for the automata-theoretic approach of Chapter 4 (both in the conversion of formulae to automata, Theorem 4.1, and in the correctness of model checking transformed to language non-emptiness, Theorem 4.2).
- We also prove the correctness of preservation of various temporal and temporal-epistemic formulae when applying compositional verification in a multi-agent systems context (Theorem 5.1, Theorem 5.2, Theorem 5.3, Theorem 5.4, Theorem 5.5 and Proposition 5.5).
- We give an algorithm for checking the acceptance of a path with respect to both an assumption and a guarantee (Algorithm 2).

3. Conceptual

- We identify the classes of multi-agent systems where it is possible to apply modular-style reasoning to individual agents.
- We suggest a new logic for specifying assumptions based on actions, as well as identifying a suitable fragment for guarantees.

4. Implementations

We developed and implemented various toolkits to evaluate experimentally the theory presented in the thesis. Namely:

- **etav** – a novel model checker developed to perform explicit-state automata-theoretic verification of multi-agent systems.
- **at-mcmas** – an extension of **mcmas-1.0** [Lomuscio et al., 2009] to perform “hybrid-state” automata-theoretic verification of multi-agent systems.
- **agr-mcmas** – an extension of **at-mcmas** (Section 6.4.1) to support the modular verification of multi-agent systems (i.e., for the verification of the two “parts” of an assume-guarantee rule).
1. **DRAISPL** – an extension of *LTL2DSTAR* [Klein and Baier, 2006] to support the generation of both “assumption enforcing” environments and “temporal observer” agents from linear-time action-based specifications.

As an outcome of implementing the developed theory, we draw attention to *agr-mcmas*’s ability to verify individual agents in MAS against formulae specified in a linear-time logic using action-based propositions.

5. **Empirical**

- We show that, for some specific classes of problems, explicit-state model checking can scale better than symbolic model checking.
- We provide an empirical justification of Vardi’s statement [Vardi, 1995] that, while the complexity of modular reasoning might seem prohibitively high, when applied to practical problems, this worst-case complexity may be avoided.

### 1.4 Relevance and Applications

The work in this thesis has pertinence to the classes of multi-agent systems that generate state-spaces too large to verify directly using traditional verification techniques. As multi-agent systems are usually loosely-coupled this generally means that they inherently generate large state-spaces. When compared to hardware verification, where the synchronous communication between multiple components can remove a large proportion of possible states, the loose-coupling of multi-agent systems does not remove these possible sequences of behaviours.

Furthermore, it is common in the software development world to design software with an expectation of how it will be used. This expectation is similar to our use of assumptions here. As such, a modular approach, similar to that which we introduce, allows for the verification of an individual agent earlier in the development process as we do not require the rest of the system to provide assurances about the correctness of this single agent.

Therefore, being able to deal with the verification of multi-agent systems in a modular way (even on a restricted class of systems) is of definite benefit. This directly allows for the verification of *re-deployable* agents, that do not have to be re-proven every time they are deployed in a requisite environment.
1.5 Collaboration and Publications

The theory presented in Chapter 4 extends work done in collaboration with Francesco Belardinelli and Alessio Lomuscio [Belardinelli et al., 2010; Belardinelli et al., 2011]. Francesco Belardinelli assisted with the proofs of Theorem 4.1 and Theorem 4.2.

The remaining contributions of this thesis—the modular verification framework of Chapter 5 and the implementation and evaluation of this technique (Sections 6.3 to 6.5.2)—are my own work.

1.6 Summary of Contents

We now summarise the remaining contents of the thesis.

In the next chapter, Chapter 2, we briefly survey the work related to the core themes of this thesis. We conclude this chapter by highlighting some of the shortcomings of these works that this thesis hopes to address.

In the subsequent chapter, Chapter 3, we then cover the theoretical prerequisites on which this thesis depends.

In Chapter 4 we introduce an original automata-theoretic approach for the verification of multi-agent systems. We present epistemic alternating tree automata, an extension of alternating tree automata [Kupferman et al., 2000], and use them to represent specifications in the temporal-epistemic logic CTLK (Section 3.3.2.4). We prove that model checking a memory-less interpreted system against a CTLK property can be reduced to checking the language non-emptiness of the composition of two epistemic tree automata.

In Chapter 5 we present a novel technique for the compositional model checking of multi-agent systems. Compositional techniques take a “divide-and-conquer” approach to verification and attempt to alleviate the state-space explosion problem by verifying each component in isolation. In this chapter, we extend Vardi’s linear-branching assume-guarantee paradigm [Vardi, 1995] to support action-based linear-time assumptions and state-based branching-time temporal-epistemic guarantees. Action-based linear-time assumptions are used to specify the interaction between a component and its environment. By comparison, branching-time temporal-epistemic guarantees are used to specify the behaviour of the “agent under test”. We prove that for agent-local assumptions and guarantees the modular approach is sound.

Chapter 6 presents the implementation of the theory for the two proceeding chapters. We report on the experimental implementations and discuss preliminary results. We evaluate the effectiveness of the technique using three real-life scenarios: a gossip protocol, the (faulty) train-gate-controller and a contract-regulated software development protocol.
Finally, Chapter 7 compares and contrasts with existing methods from the literature, and concludes.
Chapter 2
Related Work

In this chapter we explore the existing literature related to the central themes of this thesis. We begin by surveying approaches for modular verification of reactive systems (Section 2.1); in this context, we are concerned with properties pertaining only to the evolution of the system over time, and not richer properties such as knowledge. For these temporal-only specifications, either a “maximal tableau” for the formula is constructed (Section 2.1.1.1), or modern learning-based approaches are applied to automatically construct a candidate abstraction (Section 2.1.2).

Subsequently (Section 2.2), we introduce extensions to well-established semantics for certain temporal logics, that—while not always identified by the original authors—have a potential application in compositional reasoning/modular verification. In Section 2.2.3, we discuss some of the related literature surrounding a notion of component-local specifications, which we build upon when it comes to constructing specifications for modular reasoning in Chapter 5. These component-local specifications are useful, as they are immediately amenable to modular verification, without pertaining to components other than those being verified.

In the penultimate section of this chapter (Section 2.3), we discuss recent approaches to alleviate the state-space explosion problem in the context of multi-agent systems. We also survey related works that investigate modular-esque reasoning in a multi-agent systems context.

Finally, to conclude the chapter (Section 2.4) we identify some shortcomings of the state-of-the-art that we endeavour to address as part of this thesis.

2.1 Compositional Verification

Compositional reasoning [de Roever et al., 1998] is a “divide-and-conqueror” approach to verification. Rather than verifying the system as a whole and using alternative methods to alleviate the state space explosion problem (e.g., efficient data structures), the idea
is to try and tackle individual components in isolation. In doing so, the combinatorial explosion problem can (hopefully) be avoided.

Concisely, compositional reasoning can be summarised as follows:

*The specification of a program should be verified on the basis of specifications of its constituent components, without knowledge of the interior construction of those components.*

– Job Zwiers [Zwiers, 1989]

That is, a component should be verifiable in the context of a formal specification of the other parts of the system, rather than the concrete components themselves. Under such a focus, we use these specifications to effectively hide the *implementational specifics* of the other components. This is potentially advantageous as it is hoped that the specifications of the components are, ideally, significantly smaller than the components themselves. By verifying the component when “composed” with the specifications, these compositional techniques endeavour to ameliorate the state-space explosion problem, by reducing the size of the model to be considered when performing verification.

However, any modular technique should hope to be *sound*. Following a set of inferences (e.g., similar to those shown in the next section), if the modular technique decides that a formula is true on the full structure, then it should be a *valid inference* and the result should be correct.

It is often the case (as is with this work) that *completeness* will not be considered [Namjoshi and Trefler, 2010]. This is where compositional verification can support both positive and negative inferences over the concrete system.

### 2.1.1 Assume-Guarantee Reasoning

One compositional reasoning technique that has been studied widely is that of *assume-guarantee reasoning* [Berezin *et al.*, 1998; Clarke *et al.*, 1999]. The crux of assume-guarantee reasoning is the tuple

\[ \langle A \rangle M \langle G \rangle \]

which has the following interpretation:

*Informally, a component \( M \) satisfies \( \langle A \rangle M \langle G \rangle \) if the environment of \( M \) violates \( A \) before the component \( M \) fails to satisfy \( G \).*

– de Roever *et al.* [de Roever *et al.*, 2001]

In such a rule, we denote by \( A \) the *assumption* and by \( G \) the *guarantee*. In a modular verification task, the specification \( A \) is used to constrain the behaviours of \( M \) to only those that adhere to \( A \). The guarantee \( G \) is therefore the “high-level” specification that we wish to show is satisfied by \( M \). If the environment can also be shown to satisfy \( A \)
and $M$ satisfies $G$ in the context of $A$ (given certain restrictions on the syntax of $G$), then $M$ should satisfy $G$ in any environment also satisfying $A$.

### 2.1.1.1 Assume-Guarantee Model Checking

Using assume-guarantee reasoning, it is therefore possible that the verification task can be divided into its natural decompositions [Berezin et al., 1998]. That is, the verification task can be decomposed into a verification task of consistent parts, under the assumption that the other components in the system behave in a certain fashion. In this context, a typical assume-guarantee proof strategy is shown in Figure 2.1.

\[
\begin{array}{c}
\langle \text{true} \rangle M_1 \langle \psi \rangle \\
\langle \psi \rangle M_2 \langle \phi \rangle \\
\langle \text{true} \rangle M_1 \parallel M_2 \langle \phi \rangle
\end{array}
\]

**Fig. 2.1. A Typical Assume-Guarantee Inference Rule**

The inference rules in Figure 2.1 attempt to establish the following:

- That $M_1$ trivially satisfies the assumption $\psi$, by further using the assumption “true” (i.e., under no assumption)
- And that $M_2$ satisfies $\phi$ in the context of an environment that satisfies $\psi$
- Finally, if the above hold, then it is a valid inference that the composition of $M_1$ and $M_2$ (denoted $M_1 \parallel M_2$) should also satisfy $\phi$

The advantage of such a verification approach is that it is never necessary to consider the composition of $M_1 \parallel M_2$. Instead, it is only necessary to first check that $M_1$ satisfies $\psi$ (in isolation) and then check that $\phi$ is satisfied by $M_2$ under the assumption $\psi$.

Berezin et al. [Berezin et al., 1998] continue with a discussion on how to implement the proof rule in Figure 2.1, using the “simulation pre-order” ($\preceq$) from [Grumberg and Long, 1994]. Two models are related under the pre-order, written $M \preceq M'$, if $M'$ has more behaviours than $M$. That is, $M'$ can simulate the behaviour of $M$ (but possibly do more).

By considering only the universal fragment of CTL (Section 5.2.4.1), the authors are able to use the maximal model for the formula. A model $M$ satisfies a universal property $\phi$ (i.e., $M \models \phi$) iff the model $M$ is a refinement in the pre-order ($\preceq$) of the tableau construction of greatest model $M_\phi$ such that $M_\phi \models \phi$. From [Grumberg and Long, 1994], the authors note that we have the following:

$$\forall \phi \text{ in the universal fragment } \exists M_\phi \text{ such that } M \models \phi \text{ iff } M \preceq M_\phi.$$
That is, under the simulation pre-order $\preceq$ from [Grumberg and Long, 1994], there exists a maximal model for a formula $\varphi$ such that all models satisfying $\varphi$ are simulated by this maximal model.

Using the notion of maximal models, we further have:

$\langle \varphi \rangle M \langle \psi \rangle$ if and only if $M \models M_\varphi \models \psi$

Finally, to implement the inference rule shown in Figure 2.1, the authors [Grumberg and Long, 1994] use the following:

$\langle \text{true} \rangle M' \langle \varphi \rangle$ if and only if $M' \preceq M_{\varphi}$

$\langle \varphi \rangle M \langle \psi \rangle$ if and only if $M \models M_{\varphi} \models \psi$

$\langle \text{true} \rangle M \models M' \langle \psi \rangle$ if and only if $M \models M' \models \psi$

As shown above, it is noted that $\preceq$ holds if and only if $\models$ holds; therefore the check $M' \preceq M_{\varphi}$ can be performed using model checking to establish $M' \models \varphi$. Consequently, it follows that all of the above steps can be performed using model checking. The soundness of this approach was shown in [Grumberg and Long, 1994; Berezin et al., 1998].

### 2.1.2 Learning-based Assume-Guarantee Reasoning

In 2003, Cobleigh et al. [Cobleigh et al., 2003] introduced the application of learning-based frameworks to compositional reasoning. Concisely, their idea was to apply an off-the-shelf learning algorithm to automatically generate assumptions in an iterative, incremental and automated fashion. The high-level methodology attempted in these works [Cobleigh et al., 2003; Nam and Alur, 2006; Cobleigh et al., 2008] is shown in Figure 2.2 (reproduced from [Giannakopoulou and Pasareanu, 2013]).

At a high-level, and as shown in Figure 2.2, the $L^*$ learning algorithm (Section 2.1.2.1) is used to construct incrementally an assumption for assume-guarantee reasoning. The technique starts by learning those strings $s$ that keep $M_1$ “safe” relative to the property $P$. Eventually, $L^*$ will generate a “candidate assumption” $A_i$ encapsulating all known “safe” strings. If $M_1$, when composed with $A_i$, is also safe with respect to $P$, it is then necessary to check if $A_i$ is a valid assumption to abstract the module $M_2$. If either of these two checks fail, the evidence generated by the model checker for this failure can be used to refine and re-learn the next candidate assumption.

#### 2.1.2.1 $L^*$ Algorithm

The $L^*$ algorithm was originally introduced by Angluin [Angluin, 1987], while the complexity bounds and efficiency of the original procedure were later improved by Rivest and Schapire [Rivest and Schapire, 1993].
2.1 Compositional Verification

\begin{align*}
\text{conjecture:} & \quad A_i \\
\text{true} & \text{false} \\
\text{+} & \\
\text{cex} & \\
\text{true permissive?} & \\
\text{c} & \\
\text{false} & \\
\text{true} & \\
\text{false} & \\
\end{align*}

\begin{align*}
\langle A \rangle_{M_1} (P) & \\
\langle \text{true} \rangle_{M_2} (A) & \\
\langle \text{true} \rangle_{M_1} \parallel M_2 (P) & \\
\langle \text{true} \rangle_{M_2} (A_i) & \\
\text{false} & \\
\text{+} & \\
\text{cex} & \\
\text{true} & \\
\text{false} & \\
\text{true} & \\
\text{false} & \\
\text{true} & \\
\text{false} & \\
\end{align*}

\text{query: string} s

\begin{align*}
\langle s \rangle_{M_1} (P) & \\
\langle \text{true} \rangle_{M_1} \parallel M_2 (P) & \\
\langle \text{true} \rangle_{M_2} (A_i) & \\
\langle \text{true} \rangle_{M_1} \parallel M_2 (P) & \\
\langle \text{true} \rangle_{M_2} (A_i) & \\
\text{false} & \\
\text{+} & \\
\text{cex} & \\
\text{true} & \\
\text{false} & \\
\text{true} & \\
\text{false} & \\
\text{true} & \\
\text{false} & \\
\text{true} & \\
\text{false} & \\
\end{align*}

\text{Fig. 2.2. Learning Assumptions for Assume-Guarantee Reasoning}

$L^*$ learns an unknown regular language $U$, over an alphabet $\Sigma$, and produces a deterministic finite automaton (DFA) $C$, such that the language accepted by $C$ is $U$.

To learn $U$, the $L^*$ algorithm interacts with a “minimally adequate teacher” (teacher). The algorithm queries the teacher with two kinds of questions:

1. **Membership Query** – this asks, given a string $\sigma \in \Sigma^*$, whether $\sigma \in U$. When $\sigma \in U$ the teacher returns true, and false in all other cases.

2. **Equivalence Query** – an equivalence query takes the form of a conjecture by $L^*$, i.e., the algorithm generates a possible candidate DFA $C$ and asks if $L(C)$, the language accepted by $C$, is the same as $U$. When $L(C) = U$ the teacher returns true. When it returns false, the teacher also returns a counterexample, which is an element in the symmetric difference between $L$ and $U$ (i.e., a member of $((L(C) \setminus U) \cup (U \setminus L(C)))$).

$L^*$ collects information about strings which are, or are not, members of the language $U$. This is done by building an observation table containing the set of suffixes and prefixes that have been learnt to be included in the language.

2.1.2.2 Assume-Guarantee Reasoning for Labelled Transition Systems

Various authors [Cobleigh et al., 2003; Cobleigh et al., 2008; Pasareanu et al., 2008] have applied $L^*$-based learning for assumptions to the problem of compositional verification for labelled transition systems (LTS) and trace-based properties. For further details on LTS, we refer the reader to [Magee and Kramer, 1999].
In the following, let \( \mathcal{A} \) be a set of universal observable actions and let \( \pi \) denote an error state. Trivially, for a given safety property, when an LTS enters an error state, then it is in violation of the property.

**Properties.**

An LTS that contains no \( \pi \) states is called a “safety LTS”. A safety property is specified as a safety LTS \( P \), where the language of \( \mathcal{L}(P) \) specifies all of the safe behaviours over a set of actions \( \alpha P \).

An LTS \( M \) “satisfies” the safety LTS \( P \) iff:

\[
\forall \sigma \in \mathcal{L}(M) \ (\sigma \downarrow \alpha P) \in \mathcal{L}(P)
\]

i.e., all of the traces of \( M \), when restricted (denoted \( \downarrow \alpha P \) in the above) to the alphabet of \( P \), are also traces of \( P \).

To check an LTS safety property \( P \), an “error LTS” \( P_{err} \) is constructed, which traps violations of \( P \) in the error state \( \pi \). Given an LTS safety property \( P = \langle Q, \alpha P, \delta, q_0 \rangle \), \( P_{err} = \langle Q \cup \{ \pi \}, \alpha P_{err}, \delta', q_0 \rangle \), where \( \alpha P_{err} = \alpha P \) and:

\[
\delta' = \delta \cup \{(q, a, \pi) \mid a \in \alpha P \text{ and } \not\exists q' \in Q : (q, a, q') \in \delta\}
\]

That is, the transition relation \( \delta' \) for the safety LTS \( P_{err} \) is the (potentially non-serial) transition relation for \( P \), but for every transition that does not exist in \( P \), \( P_{err} \) transitions to the error state \( \pi \).

To check the conformance of \( M \) to \( P \), the parallel composition of \( M \parallel P_{err} \) is constructed, and then it is verified if \( M \parallel P \) is true by checking if \( \pi \) is not reachable in \( M \parallel P_{err} \).

**Learning-based assume-guarantee reasoning.**

To check if an assume-guarantee triple \( \langle A \rangle M \langle P \rangle \) holds for a component \( M \), where both \( A \) and \( P \) are given as safety LTS, it is possible to simply check if the state \( \pi \) is accessible in \( A \parallel M \parallel P_{err} \). If \( \pi \) is not reachable, then \( \langle A \rangle M \langle P \rangle \) is satisfied. The learning-based approaches consider the application of learning in the context of the following assume-guarantee rule with safety properties:

\[
\frac{\langle \text{true} \rangle M_1 \langle A \rangle}{\langle A \rangle M_2 \langle P \rangle} \quad \frac{\langle \text{true} \rangle M_1 \parallel M_2 \langle P \rangle}{\langle \text{true} \rangle M_1 \parallel M_2 \langle P \rangle}
\]

as per Section 2.1.1.1.

The learning algorithm \( L^* \) is applied to learn a candidate assumption \( A \) representing \( M_1 \). This abstraction \( A \) must be strong enough to allow \( M_2 \), when composed with \( A \),
to still satisfy $P$. That is, if $A$ is too much of an abstraction of $M_1$ (i.e., it allows more potential behaviours), then this could lead $M_2$ to be potentially unsafe.

To find a suitable assumption $A$ of $M_1$, $L^*$ constructs an assumption $A$ over the alphabet $\mathcal{Act}$, such that the string $t \in \mathcal{Act}^*$ is in the language of $A$ iff $\langle t \rangle M_2 \langle P \rangle$. As before $\langle t \rangle M_2 \langle P \rangle$ holds iff $\pi$ is unreachable in $t || M_2 \parallel P_{err}$.

$L^*$ can then be used to iteratively learn an assumption $A$, over the alphabet $\mathcal{Act}$, which is:

- strong enough to constrain the behaviour of $M_2$ such that $\langle A \rangle M_2 \langle P \rangle$ holds, and
- weak enough such that $\langle true \rangle M_1 \langle A \rangle$ holds.

The learned assumption therefore contains all traces of $M_1$, abstracted to $\mathcal{Act}^*$, that prevent $M_2$ from violating $P$.

While the techniques of Cobleigh et al. were implemented in the Ltsa tool [Magee and Kramer, 1999], there are a number of extensions to these works, taken in different contexts. For example, Gupta et al. [Gupta et al., 2008] apply SAT-based algorithms and Alur et al. [Alur et al., 2005a; Nam and Alur, 2006; Nam et al., 2008] apply the technique to learn assumptions in the context of safety properties in NuSMV.

2.2 Logics for Reasoning about Computations

We now discuss some logical formalisms related to the theme of modular verification. We begin by surveying approaches based on modular model checking and module checking (Section 2.2.1) using the standard semantics for various temporal logics. In Section 2.2.2, we then consider non-standard semantics of temporal logics. We highlight that these semantics are of interest as they potentially support modular verification.

2.2.1 Modular Model Checking and Module Checking

We now introduce module and modular model checking.

2.2.1.1 Modular Model Checking

The idea of modular model checking [Kupferman and Vardi, 1997a; Kupferman and Vardi, 2000] is very closely akin to the compositional approach of Long [Grumberg and Long, 1994] presented in the previous section. The authors demonstrate how an automata-theoretic verification for fair model checking can be suitably extended to support space-efficient verification of a CTL* guarantee against the maximal model of a CTL* assumption.
The authors present both a space-efficient model checking procedure based on automata, as well as investigating the complexity of the various approaches.

A specific case of modular model checking is that of linear-branching modular model checking, which was introduced in [Vardi, 1995]. In the linear-branching paradigm, Vardi considers the verification of a branching-time guarantee in the context of a linear-time assumption. We investigate this further in Chapter 5.

### 2.2.1.2 Module Checking

Modular model checking was further advanced to module checking in [Kupferman and Vardi, 1996; Kupferman and Vardi, 1997b; Kupferman et al., 2001]. In these works, the authors consider verification of a given component in a completely chaotic environment [Roscoe et al., 1996; Sidorova and Steffen, 2001; Leino and Logozzo, 2005]. This was called “open satisfaction”. For an “open system” $M$ (i.e., a system that receives input/can communicate with its environment), the “open satisfaction” problem is to decide if $M$ satisfies a formula $\phi$ for all possible environments $E$.

In [Kupferman and Vardi, 1996], the authors consider module checking in the context of compositional reasoning. That is, for an assumption $\psi$ and for all environments $E$ such that $E|\psi = \psi$, does $M||E|\psi = \phi$?

In a similar way to their previous works for modular model checking, the authors only consider the complexity of the underlying approach, rather than developing a practical and implementable verification technique.

### 2.2.2 Model Checking with Path Criterion

There have been approaches for verifying CTL properties against “path criteria”. It is common for these to be applied when verifying CTL properties, and using the criterion to decide which paths should be considered when performing path quantification.

The works of both [Kupferman and Vardi, 2006] and [Niebert et al., 2008] consider similar settings. In [Niebert et al., 2008], the authors introduce EmCTL* (“Embedded CTL*”). In Embedded CTL*, the branching-time operators for quantifying over paths contain an additional argument of an LTL formula. If $\phi$ is an LTL formula, then $\forall\mathfrak{P}X\psi$ is an EmCTL* formula, where the satisfaction of $\psi$ at the next state is only checked for paths that satisfy $\phi$. Using such an approach, it is therefore possible to ignore paths that do not satisfy some categorisation of “good” paths in the model.

Kupferman and Vardi [Kupferman and Vardi, 2006] introduce an earlier work to EmCTL*, called mCTL* for “memoryful CTL*”. Unlike EmCTL*, which introduces new modalities for path quantification, mCTL* presents new semantics for path quantification based on the special proposition present. When checking the formula $EF \phi$ in mCTL*
at a state \( s \), the approach of Kupferman and Vardi is to first generate the new proposition \( \text{present} \) such that it holds only at \( s \), and then to check if there exists a path starting at the initial state that satisfies \( F \varphi \). To obtain the regular semantics for CTL\(^*\), the proposition \( \text{present} \) can be used as follows:

\[
EF \varphi \equiv EF (\text{present} \land F \varphi)
\]

Where the left-hand expression is a standard CTL formula, and the right-hand is its mCTL\(^*\) equivalent. It clearly follows that the formula \( EF (\text{present} \land \ldots) \) is sufficient for us to “return to” the state \( s \) in order to begin evaluating \( F \varphi \). Checking this formula under the mCTL\(^*\) semantics then means that \( \varphi \) is only checked at the state \( s \).

Therefore, an immediate embedding of EmCTL\(^*\) into mCTL\(^*\) is possible:

\[
\exists \varphi \psi \equiv \exists (\varphi) \land F (\text{present} \land \psi)
\]

It is argued however [Niebert et al., 2008] that, although mCTL\(^*\) subsumes EmCTL\(^*\), the complexity of the model checking procedure for EmCTL\(^*\) is lower and therefore a native approach for EmCTL\(^*\) can be favourable.

Related to these works is the approach for MCTL [Josko, 1987] (“Modular CTL”). Josko’s presentation of MCTL is directed towards modular verification, although his language of assumptions is not that of a full LTL, as is supported in [Kupferman and Vardi, 2006; Niebert et al., 2008]. By comparison, in MCTL, it is only possible to use nested until formulae, without negations, and Boolean expressions over propositions, nested under a top level globally formula. When checking a CTL “guarantee” in MCTL, the quantification of paths occurs only over those paths that satisfy the assumption. While semantics are provided, as well as hinting to the complexity of the technique and suggesting the application to modular reasoning, unlike [Kupferman and Vardi, 2006; Niebert et al., 2008], Josko [Josko, 1987] does not provide an algorithmic approach for verifying an MCTL formula.

### 2.2.2.1 Modular Verification for Software using Filters

Following from the above, and still while considering the ability to “mask” undesirable behaviours from the state-space, the works [Dwyer and Pasareanu, 1998; Pasareanu et al., 1998; Pasareanu et al., 1999] consider the idea of model checking in the context of filters. These filters can be used to “refine” (via filtering out) a generated state-space to assist in model checking.

These works consider using an LTL formula (and later a CTL formula) to remove those states from an analysis that do not meet the specification. As such, in an assume-guarantee context, an LTL filter can be used as the assumption and then it can prune states from the component under test that are not considered with the formula.
To support this, they consider a restricted class of safety formulae. For this class of formulae, the authors note [Pasareanu et al., 1999] that it is possible to obtain a deterministic finite automaton that corresponds to the maximal tableau of a formula. The finite automaton is obtained by first constructing the non-deterministic Büchi automaton for the formula, then by “ignoring the acceptance conditions” [Pasareanu et al., 1999], the authors obtain a finite automaton.

In this finite automaton, it is claimed that every path in the tableau trivially satisfies the formula. Consequently, it is possible to perform assume guarantee reasoning by composing the component under test with the tableau, and then performing model checking using the standard approaches.

### 2.2.3 Local and Modular Specifications

As we further motivate in Chapter 5, when performing modular verification, it is a sensible restriction to only consider specifications defined over components we wish to verify. We now discuss some works with a similar motivation.

In [Engelhardt et al., 1998], the authors present an approach for reasoning about individual agents using local propositions. Using an observation we also use in Chapter 5, it is shown that the verification of knowledge can be suitably reformulated under a logic of “local propositions”, where these propositions can be generated “at run-time”. For example, when checking knowledge it is necessary to consider all possible states where an agent has a current configuration; if it is possible to assert a proposition $p_i$ that only holds at states satisfying this configuration, then it is possible to check $G(p_i \rightarrow \varphi)$. That is, do all states satisfying $p_i$ also satisfy the epistemic subformula being verified?

In an epistemic setting, this framework is the basis of [Hoek and Wooldridge, 2002] and is used to implement epistemic model checking on top of the model checker spin [Holzmann, 1997; Holzmann, 2004]. Furthermore, the work of [van Ditmarsch et al., 2012] considers the schemas and logical validities in such a logic.

In [Caleiro et al., 2005], the authors introduce DTL, a distributed temporal logic, containing component local specifications. As an integral part of the logic, is it possible to specify properties that pertain to an individual agent in the system using the syntax:

$$\text{@}_i [\mathcal{L}_i]$$

where $\text{@}_i$ is read “at the agent $i$”, and $\mathcal{L}_i$ is the set of $i$-local formulae with the syntax:

$$\mathcal{L}_i ::= \text{Act}_i \mid \text{Prop}_i \mid \bot \mid \mathcal{L}_i \rightarrow \mathcal{L}_i \mid \mathcal{L}_i V \mathcal{L}_i \mid \mathcal{L}_i S \mathcal{L}_i \mid \text{@}_j [\mathcal{L}_j]$$

where $\text{Act}_i$ is a set of propositions over $i$’s actions and $\text{Prop}_i$ is a set of propositions over $i$’s states.
It can be seen that, although components’ local specifications are able to “branch” to other components using the operator $@_j$, a majority of the specification language pertains to only the current agent $i$. We note that in [Caleiro et al., 2005], a large number of presented specifications are purely $i$-local, and therefore do not make use of the $@_j$ modality in a $L_i$ formula.

In a similar context [Filippidis et al., 2012] considers “Local LTL” formulae for synthesising LTL controllers for individual components based on their private specifications. Although specifications are provided individually, it is possible to ensure that the overall system can achieve the desired goals by allowing agents to communicate their synthesised controller to other agents.

In the context of assume-guarantee, [Lomuscio et al., 2013] makes explicit the application of local specifications to modular verification. The work places a restriction on the modular framework to verify a global specification $\varphi$ which is the conjunction of a component-specific guarantees. Therefore, the component-specific guarantee $\varphi_i$ is a local specification defined only over the component $M_i$. Additionally, component-specific assumptions, manually specified as Büchi automaton (Section 3.4) are used to check if each component satisfies its guarantee. It then immediately follows that if each component separately satisfies its guarantee, and if the top-level specification is a conjunction of all of these formulae, the full composition trivially meets its requirements.

Furthermore, in a temporal-epistemic context, Russo [Russo, 2011] considers a multitude of agent-local specifications, applied to both multi-agent card games and to a software development protocol example we consider ourselves later in this thesis (Section 6.5.2).

### 2.2.4 Automata-Theoretic Verification for Branching Structures

We now consider automata-theoretic approaches for various branching structures and logics. As we present in Section 3.3, knowledge forms a branching structure, and cannot be verified in the same way as linear-time properties.

#### 2.2.4.1 Automata-Theoretic Verification of Knowledge

We begin by discussing, to the best of our knowledge, the only approach for applying automata theory to the verification of temporal-epistemic formulae.

In [van der Meyden and Shilov, 1999], the authors consider an automata-based approach for the verification of multi-agent systems against linear-time temporal-epistemic formulae. They adopt the semantics of synchronous perfect recall [Fagin et al., 1995]. This is where all agents have a common, synchronous and observable clock – as well as being endowed with the ability to recall all possible states. To achieve the verification
of this, the authors present a notion of “forests of trees”, where each forest emanating from a node in the tree is the set of states that a particular agent considers possible.

To verify a linear-time temporal-epistemic formula, the authors suggest using an infinite number of Büchi automata, each of which can be used to verify if a subtree satisfies a given formula. As such, for each instance where a knowledge operator needs to be evaluated in the tree, \( n \) Büchi automata (one for each subtree) need to be created and their acceptance checked. When descending through a formula, each time an epistemic subformula is considered, then its nesting depth is decreased by one. As such, while a potentially infinite number of Büchi automata are required, this is bounded by the nesting depth of the formula, as a new forest is only required for each epistemic modality that occurs in the formula.

Unfortunately, as the authors consider the synchronous perfect recall semantics with common knowledge they show that various aspects of the model checking procedure are undecidable. Furthermore, the authors themselves state that “the algorithm [to check if a formula is satisfied in an environment] is not yet fully operational” (i.e., the procedure is not complete/finished in algorithmic terms). This is due to needing to calculate the set of states where an LTL formula \( \varphi \) holds, which was not well established at the time of writing.

We also note that the approach is not a purely language-theoretic (i.e., based on language containment) method for deciding if a given multi-agent system satisfies a formula. While Büchi automata are used to check the satisfaction of the linear-time formulae, it is a procedure external to the automata that is used to check the epistemic modalities.

We note that Mohalik et al. [Mohalik and Ramanujam, 2010] consider automata as a foundational structure for representing and capturing epistemic knowledge. While they allude to the benefits of verification of epistemic properties by automata—and that it is a little explored field—they do not consider their theory in such a setting.

### 2.2.4.2 Symbolic Automata-based CTL Model Checking

In [Qian and Nymeyer, 2006], Qian et al. present a method for checking language emptiness of a certain class of automata (Section 3.4.2.3) using binary decision diagrams (BDDs) [Bryant, 1986; McMillan, 1992] rather than standard, explicit-state methods.

BDDs are reduced, canonical representations of Boolean formulae. Importantly, they allow for the efficient calculation of operations such as \textit{and}, \textit{or} and \textit{negation}. There exists a large body of existing work in applying symbolic methods (in particular BDDs) to set-theoretic approaches to model checking. These approaches have been shown to be very efficient [Burch et al., 1992].

As will be shown in Chapter 4, model checking a structure against a formula can be translated to checking the (non-)emptiness of the product automaton for the formula.
and the structure. Qian et al. [Qian and Nymeyer, 2006] apply a method of “state recording” to emulate the double depth-first search approach [Holzmann et al., 1997] of explicit-state model checkers to decide if a path is accepting or not.

Using BDDs, they construct the “AND/OR graph” for the non-emptiness problem of the product structure, and then perform reachability checking to find a solution for the graph. In their work, the authors prove the following: the AND/OR graph for product automaton of $K$ and $\varphi$, $A_K,\varphi$, has a solution graph iff the language of $A_K,\varphi$ is non-empty (i.e., the structure $K$ satisfies $\varphi$).

The idea behind state-recording is to duplicate the set of state variables. These duplicates can then be used to detect if a state has been visited previously in an accepting run. This is similar to the strongly-connected component detection in explicit-state approaches.

Their technique, implemented into the symbolic (i.e., BDD-based) model checker nuSMV, achieved results suggesting the automata-theoretic approach can be preferable to the standard set manipulation approaches.

2.3 Reducing the State Space for Multi-Agent Systems

We now briefly survey some recent developments in the area of state-space reductions for verifying multi-agent systems. We purposefully exclude approaches dealing with efficient representation of the state-space (e.g., SAT-based or BDD-based methods), and concern ourselves with approaches that reduce the input state-space to be verified.

2.3.1 Symmetry Reduction

Symmetry reduction for MAS [Cohen et al., 2009a; Cohen et al., 2009b] attempts to reduce the size of the state-space by identifying isomorphic subgraphs of the state-space due to replicated components. As such, while the traces of the components may be different, they are automorphic based on the reordering of local states in the set of global states. By taking these duplicated behaviours into consideration, it is therefore possible to collapse the size of the reachable states.

Following the so-called counterpart semantics, under symmetry reduction it is possible to demonstrate that the abstract system and its concrete counterpart satisfy the same set of formulae. This is because the automorphism preserves the relations between states in both the concrete structure and the reduced one.
2.3.2 Partial Order Reduction

The idea behind partial order reduction for multi-agent systems [Lomuscio et al., 2010a] is that not all interleavings of actions are of interest when considering formulae without the “next-time” (X, AX or EX) operator. Therefore, to avoid the state-space explosion problem, it is possible to construct a model that does not contain all of the interleavings for the independent actions of all of the agents in the model.

In this setting, the authors are only able to consider the class of interleaved (asynchronous), and not synchronous, multi-agent systems that we consider in this thesis.

2.3.3 Parametric Verification

Parametric verification for MAS was studied [Kouvaros and Lomuscio, 2013a; Kouvaros and Lomuscio, 2013b] by looking at the possibility of identifying a cut-off for temporal-epistemic model checking. Given a family of homogeneous agents (i.e., identical), the parametric technique will identify a cut-off parameter k, such that if a composition containing k agents satisfies a formula, then a composition containing any number of agents (even an infinite number) will also satisfy the formula. Verifying a model containing a cut-off number of agents allows the technique to decide the satisfaction of a formula irrespective of the number of agents in that composition.

It is noted that parametric verification induces a simulation pre-order between the larger model and its cut-off; therefore, the technique is not complete (i.e., an answer of “false” could be either due to a fault in the model or due to the simulation not being strong enough).

2.3.4 Existential Abstraction

Existential abstraction looks at clustering states under a quotient structure, and computing an over-approximation of the state-space. That is, by sacrificing preciseness in the calculation of the exact state-space, efficiencies can be obtained.

By clustering local states of the agents together, and inserting a local transition between two clustered states if there exists a transition in that agent’s unclustered equivalents, we observe that the behaviour of the abstract agent is an over-approximation of the behaviour of the concrete one.

In [Cohen et al., 2009c; Lomuscio et al., 2010b], it is proven that, if an abstract model satisfies a universal temporal-epistemic property, then so does its concrete equivalent. However, a negative result in the abstract domain does not carry across to the concrete
domain: the formula may either be genuinely false or the over-approximation may have included spurious behaviours.

### 2.3.4 Abstraction Refinement

Given an over-approximation of the state-space, if a formula is false, it is therefore necessary to decide if this formula was genuinely false, or if it was caused by the abstraction methodology. To address this deficiency, the approach of *counterexample guided abstraction refinement* (CEGAR) has been advanced to a MAS-based setting by various authors [Jones and Lomuscio, 2011; Zhou et al., 2011; Koleini et al., 2013].

When performing CEGAR, two steps are added to the abstraction methodology:

1. **Counterexample validation** – this is where a possibly spurious counterexample is checked for validity against the concrete components. If the counterexample is shown to be valid, then the process can terminate with a definitive *false* result.
2. **Abstraction refinement** – however, if the formula is shown to be spurious, the clustering function used to preform abstraction is modified to remove the possibility of this spurious behaviour.

These two steps can be incrementally performed until either: 
   a) the method returns *true*; 
   b) the method returns *false* and the counterexample is shown to be valid; or 
   c) the refinement reaches the original concrete model, and therefore one final check can be performed to generate a definitive answer.

### 2.3.5 Modular Verification for Multi-Agent Systems

We now discuss some of the works that aim to look at modular verification of multi-agent systems. We note that, to the best of our knowledge, there exist no works applying the assume-guarantee paradigm to the verification of multi-agent systems against the rich properties we consider in this thesis (Section 3.3.2.4).

**Hierarchical verification.**

In 2002, Engelfriet et al. [Engelfriet et al., 2002] introduced an approach for compositional verification against formulae in *temporal multi-epistemic logic*. However, this work considers the compositional verification properties in the face of *aggregation* rather than *parallel composition* (i.e., of properties from hierarchies of structures, rather than concurrent systems). Furthermore, the compositional verification approach also follows a *proof theoretic* method; rather than verifying components in the face of an environmental abstraction (e.g., an assumption in the assume-guarantee paradigm), they...
show that a specification holds at a higher level in the hierarchy if it is *logically entailed* by the formulae shown to hold lower in the hierarchy.

Furthermore, this work also does not adopt the *de facto* standards for epistemic logic (interpreted systems [Fagin *et al.*, 1995], Section 3.3). Additionally, the authors highlight that: “epistemic operators cannot be nested, and cannot be applied to temporal formulae”.

**Safety verification of MAS.**

More recently, El-Zaher *et al.* [El-Zaher *et al.*, 2012] considered the compositional verification of a multi-agent system for a platoon convoy example. While considering the approach in a multi-agent systems setting, the authors only consider the specification and verification of safety properties of the system.

The authors look at how the procedure of compositional verification for safety formulae can be effectively chained. The process looks at showing if the $i$-th component satisfies a safety property against a hypothesis discharged by the $(i-1)$-th component. If the $i$-th component is safe, then a new hypothesis is generated that can be used in the verification of the $(i+1)$-th component.

As we note, this process is only applicable to a limited selection of safety properties.

**Proof-theoretic approaches.**

The works [Dennis *et al.*, 2013] and [Fisher *et al.*, 2013] consider the verification of complex multi-agent systems against environment assumptions, representing possibilities of an agent’s incoming precepts. They consider a model of “open satisfaction”, where model checking is used to identify the correlations between precepts. Once these relationships have been identified, they are then “hard coded” into the model to allow for the verification of more complex properties.

Theorem proving is used to check the desired properties for a given agent $A_i$ and an environment $E$. That is, under “open satisfaction” it is first shown that $A_i$—the agent under test—satisfies $\varphi$ and then it is further assumed that $E$—the environment—also satisfies a formula $\psi$. Next, a theorem prover is used to show that $\vdash (\varphi \land \psi) \Rightarrow \xi$ (i.e., $\varphi \land \psi$ *logically entails* $\xi$). Finally it is then possible to derive that $A_i$ further satisfies $\xi$.

**Co-operative satisfaction.**

In 2014, the notion of *co-operative satisfaction* was investigated in a multi-agent systems context [Partovi and Lin, 2014]. In this work, the authors look at verifying the formula

$$\phi_G = \bigvee_{c=1}^r \phi_c \land \bigwedge_{k=1}^r \phi_k$$
in isolation, where $\tau$ and $\bar{k}$ are cardinality of two sets of formulae. However, rather than attempting to ensure that each component in isolation satisfies $\phi_G$, the authors look at how to iterate over all of the subformulae in $\phi_G$, to then identify which of these subformulae each component can satisfy.

To apply this in a modular fashion, the authors consider a prescriptive method of deriving an assumption that supports the guarantee. That is, given a potential set of satisfiable subformulae for a component, they generate an assumption applicable to the component to allow it to satisfy those formulae.

If the component does satisfy the selected subformulae, in the context of a derived assumption, the environment for that component is then checked to see if it does satisfy the construct assumption. Such a requirement puts a tight restriction on the properties the environment must satisfy, to ensure that the individual component can satisfy parts of its guarantee.

### 2.3.6 Modular Verification for Multi-Modal Epistemic Logic

Compositional reasoning for epistemic logic is considered in [Aceto et al., 2012] (dubbed multi-modal logic in [Aceto et al., 2013]). However, this is not agent-based composition, but composition of the underlying models representing a full composition of agents. This means it is possible to verify properties in two Kripke structures independently and then derive further properties that hold on the composition of these two structures.

The approach requires that both models contain the same “agent identifiers” (i.e., in the multi-modal logic, we have the same set of modalities). The composition of the two structures is the cross product of the state-space, removing any states that are inconsistent due to clashing atomic propositions. Consequently, the composition does not reflect the composition of individual components.

The work also does not consider temporal formulae (i.e., the consider modal logics only, without any richer notion of time) and the authors themselves state that it is not comparable to the interpreted systems semantics we study in this thesis.

### 2.4 Identified Shortcomings and Potential Opportunities

We now identify a number of shortcomings in the work presented in this chapter.

One of the major shortcomings is in the lack of development of an implementation of a generic assume-guarantee approach. For example, while the approaches based on learning can automatically generate a candidate assumption, the underlying learning algorithms used can only generate a deterministic finite automaton. Such automata may
actually be larger (due to being deterministic) than the component that it is abstracting. Furthermore, given the nature of the learning approach (i.e., based on feeding a counterexample back to the learner), these approaches are restricted to safety properties (i.e., those properties that are expressed as reachability only).

For approaches that support manually-generated assumptions, then it is possible to support a richer logic for guarantees, but, in all of the approaches attempted practically, the assumptions must be in the safety fragment of the specification. Again, this is a serious short-coming.

Furthermore, we note that no works exist that support the automatic verification of multi-agent systems against temporal-epistemic guarantees in the presence of assumptions. We consider the closest approach to be that of [Dennis et al., 2013]. However, in this case, the authors consider properties specified in a logic for belief, desires and intentions, and they use proof-theoretic approaches to show the satisfaction of the chosen guarantees (i.e., they do not use a further model checking procedure). Using a proof-theoretic means significantly reducing the set of properties that are verifiable, as the “final” property must be logically entailed from a set of weaker formulae that have been verified. This is due to the fact that the final property does not consider the construction of the system under verification, but can only follow derivations allowable by the underlying logic.

While [Pasareanu et al., 1999] does introduce an approach for constructing the maximal tableau for a formula it is, as noted, restricted to the safety fragment. Beyond this and the work in mocha (as discussed in Chapter 7), to the best of our knowledge, there is no tooling support for the verification of assume-guarantee triples. If both parts of an assume-guarantee triple are specified as LTL formulae, the assume-guarantee rule can easily be checked as an implication between the assumption and the guarantee.

From a temporal-epistemic logic perspective, we also identify that the automata-theoretic verification approaches are significantly under-developed. Apart from the work [van der Meyden and Shilov, 1999]—where the authors themselves note the technique is not finished—there are no direct techniques for the verification of temporal-epistemic properties using automata. We note that while [Hoek and Wooldridge, 2002] does enable the verification of linear-time temporal-epistemic formulae, but this approach cannot be extended to the branching-time case, and does not handle epistemic logic as a “first class citizen” (it only supports epistemic modalities via a reduction of the problem to propositional atoms).

Finally, in the context of model checking against paths with “path criteria”, we note that, as with a lot of the techniques covered so far, there are no implementations of the techniques. Furthermore, apart from the work of [Josko, 1987], there has been no modern focus on the use of path criterion when applied to modular verification/assume-guarantee reasoning. It should be completely without surprise that these logics are strongly grounded in temporal-only logic, without the consideration of any epistemic modalities.
Chapter 3
Background and Preliminaries

In this chapter we present the necessary background and prerequisites of the material we will build upon in this thesis. Primarily, we will use this chapter to introduce and define the notation that will be used throughout the rest of this thesis. The material presented below summarises [Fagin et al., 1995; Hughes and Cresswell, 1996; Clarke et al., 1999; Huth and Ryan, 2004; Baier and Katoen, 2008].

3.1 Modal Logic and Kripke Structures

Modal logics extend propositional logics via the use of operators that can be used to express forms of necessity or possibility.

3.1.1 Propositional Modal Logic

Given a set $AP = \{p, q, \ldots\}$ of atomic propositions, the syntax of a typical (uni-)modal logic is as follows:

$$\phi ::= p \mid \neg \phi \mid \phi \lor \phi \mid \Box \phi$$

where $p \in AP$.

The formula $\Box \phi$ has the common reading of “$\phi$ is necessary”. As we will show in the subsequent sections, we can assign various meanings to necessary, one of which corresponds to a notion of knowing the formula $\phi$. 

47
3.1.2 Semantics and Abbreviations

The simplest and most common semantics for modal formulae are Kripke structures (see, e.g., [Huth and Ryan, 2004]).

**Definition 3.1.** A Kripke structure is a tuple $M = \langle W, R, V \rangle$, where:

- $W$ is a finite, non-empty set of states
- $R \subseteq W \times W$ is an accessibility relation
- $V : W \rightarrow 2^{AP}$ is a labelling function, which maps states to atomic propositions in $AP$

The accessibility relation $R$ can also be thought of as a function $R : W \rightarrow 2^W$. That is, given a state $s \in W$, $R(s) \subseteq W$ returns the set of successors for $s$.

We can now interpret a given modal formula against semantics for Kripke structures. We write $M, w \models \phi$ when a given formula is satisfied (i.e., $\phi$ evaluates to true) at a state $w$ in the structure $M$.

$$M, w \models p \quad \text{iff} \quad p \in V(w)$$

$$M, w \models \neg \phi \quad \text{iff} \quad \text{it is not the case that} \ M, w \models \phi$$

$$M, w \models \phi \lor \psi \quad \text{iff} \quad M, w \models \phi \text{ or } M, w \models \psi$$

$$M, w \models \Box \phi \quad \text{iff} \quad \forall w' \in W, wRw' \text{ implies } M, w \models \phi$$

The formula $\Box \phi$ is true at a state $w$ iff the formula $\phi$ is satisfied at all states that are accessible via $R$ from $w$. Importantly, if $\exists w' \in W$ s.t. $wRw'$ then $w \models \Box \phi$, for any formula $\phi$.

### 3.1.2.1 Abbreviations

It is useful to define some common abbreviations:

- $true \equiv p \lor \neg p$
- $\phi \land \phi' \equiv \neg (\neg \phi \lor \phi')$
- $false \equiv p \land \neg p$
- $\phi \rightarrow \phi' \equiv \neg \phi \lor \phi'$
- $\Diamond \phi \equiv \neg \Box \neg \phi$

The expression $\Diamond \phi$ is read as “$\phi$ is possible”. For clarity, the semantics for $\Diamond \phi$ are as follows:

$$M, w \models \Diamond \phi \quad \text{iff} \quad \exists w' \in W, wRw' \text{ and } M, w \models \phi$$
Intuitively, a state $w$ satisfies the formula $\Diamond \varphi$ if there exists at least one state accessible through $R$ that satisfies $\varphi$. It follows that $\Diamond$ is the dual of $\Box$; that is, a state satisfies $\Diamond \varphi$ if it is not the case that all successors satisfy the negation of $\varphi$. We note that it is possible for a state $w$ to satisfy both $\Diamond \neg \varphi$ and $\Diamond \varphi$.

### 3.1.3 Multi-Modal Logic

While propositional modal logic contains a single necessity operator, when it comes to dealing with interpretations of the modality in many settings, it is useful to consider multi-modal logics.

**Definition 3.2.** Multi-modal logic

Given an $n \in \mathbb{N}^+$, an $n$-ary modal logic is a modal logic containing $n$ necessity operators, and has the following syntax:

$$\varphi ::= p \mid \neg \varphi \mid \varphi \lor \varphi \mid \Box_1 \varphi \mid \ldots \mid \Box_n \varphi$$

where $p \in AP$.

That is, a multi-modal logic of degree $n$ is a modal logic containing $n$ separate modal operators.

**Definition 3.3.** Multi-modal Kripke Structures

An $n$-ary Kripke structure is as follows:

$$K = \langle S, R_1, \ldots, R_n, V \rangle$$

As with the uni-modal case, we denote by $S$ the set of states, $V$ the valuation function, and $R_i$ the $i$-th accessibility relation.

**Definition 3.4.** Multi-Modal Satisfaction

Given a formula $\varphi$ in a multi-modal logic, the satisfaction in a multi-modal Kripke structure $K$ at a state $s$ is defined as follows.

- $K, s \models p$ iff $p \in V(s)$
- $K, s \models \neg \varphi$ iff it is not the case that $K, s \models \varphi$
- $K, s \models \varphi \lor \psi$ iff $K, s \models \varphi$ or $K, s \models \psi$
- $K, s \models \Box_i \varphi$ iff $\forall s' \in S, sR_is'$ implies $K, s \models \varphi$

We note that all of the abbreviations for the uni-modal case hold in the multi-modal case. Notably, we have:

$$\Diamond_i \varphi \equiv \neg \Box_i \neg \varphi$$
3.2 Temporal Logics

Modal logics (of sorts) can also be used to reason about time. In the simplest setting the operator $\square$ can be used to represent “at the next state”, where the accessibility relation represents the flow of time through a system.

In this section, we present two temporal logics: linear temporal logic (LTL) and computational tree logic (CTL). The difference between these logics is that LTL considers time as a linear flow, where the next state is uniquely defined given a run of the system, while CTL uses a branching model of time where each state may have many possible successors.

We begin by introducing the notion of a transition system.

3.2.1 Transition Systems

A transition system is an extension to a Kripke structure as follows. A transition system is a tuple $T = \langle W, R, W_0, V \rangle$, where:

- $W$ is a set of states
- $R \subseteq W \times W$ is a transition relation
- $W_0 \subseteq W$ is a set of initial states
- $V : W \rightarrow 2^\mathbb{AP}$ is a valuation function mapping states to the propositions that state satisfies

To reason about logics related to the flow of time, it is desirable to speak about the unwinding of a transition system into a set of paths. Given a transition system $T$, we wish to speak about the set of paths or runs that $T$ induces. In what follows, we assume that $R$ is serial, that is:

$$\forall w \in W, \exists w' \in W, wRw'$$

Informally: there exists at least one successor for each state in $W$.

By way of notation, for a finite set $X$, for an infinite string $x \in X^\omega$, we denote by $x(i) \in X$ the $i$-th element in $x$.

**Definition 3.5. Path**

A path $\pi \in W^\omega$, is an infinite series of states in $W$, such that $\pi(0) \in W_0$ and for all $i \geq 0$, $(\pi(i), \pi(i+1)) \in R$. As $R$ is serial, this means that each run of $T$ must be infinite (i.e., that there exist no “dead end” states that have no successors).
3.2 Temporal Logics

3.2.2 Linear Temporal Logic

We now introduce Linear Temporal Logic (LTL) [Pnueli, 1977; Clarke et al., 1999; Baier and Katoen, 2008], a temporal logic which is interpreted over runs of the transition system.

3.2.2.1 LTL Syntax

The syntax of an LTL formula $\phi$ is given inductively as follows:

$$\phi ::=: p | \neg \phi | \phi \lor \psi | X \phi | \phi U \psi$$

where $p \in AP$.

The formula $X \phi$ is read “at the next state $\phi$”, and $\psi U \phi$ is read “$\psi$ until $\phi$.

Given a path $\pi$ in $T$, such that $\pi(0) \in W_0$, we denote by $\pi_i$ the infinite suffix of $\pi$ starting at the state $\pi(i) \in W$.

3.2.3 LTL Semantics and Abbreviations

The satisfaction ($\models$) of an LTL formula $\phi$, with respect to an infinite run $\pi$ in a transition system $T$, is defined as follows:

$$\pi \models p \quad \text{iff} \quad p \in V(\pi_0)$$

$$\pi \models \neg \phi \quad \text{iff} \quad \text{it is not the case that } \pi \models \phi$$

$$\pi \models \phi \lor \psi \quad \text{iff} \quad \pi \models \phi \text{ or } \pi \models \psi$$

$$\pi \models X \phi \quad \text{iff} \quad \pi_1 \models \phi$$

$$\pi \models \phi U \psi \quad \text{iff} \quad \exists i \geq 0, \text{ s.t. } \pi_i \models \psi \text{ and } \forall j < i, \pi_j \models \psi$$

Intuitively, a run $\pi$ satisfies the LTL formula $\psi U \phi$ iff the formula $\psi$ holds up until the formula $\phi$ holds. Similarly, it follows that a run $\pi$ satisfies $X \phi$ if the suffix of the path starting at the next state satisfies $\phi$.

3.2.3.1 LTL Abbreviations

The temporal operators $G$ (“always”, globally) and $F$ (“eventually”, future) can be further defined as:

$$F \phi \equiv true U \phi$$

$$G \phi \equiv \neg F \neg \phi$$
The definition of $F\varphi$ is naturally expressed as an until formula, as until formulae require the consequent to eventually hold at some point on the run. Furthermore, as true holds at all states, the definition of $F$ is intuitive.

### 3.2.3.2 Closure of an LTL formula

The *closure* $cl(\varphi)$ of an LTL formula is the set of all subformulae of a given formula including itself. For example, the closure of $\varphi = XpUXq$ is:

$$cl(\varphi) = \{XpUXq, Xp, Xq, p, q\}$$

### 3.2.4 Computational Tree Logic

Computational Tree Logic (CTL) [Clarke et al., 1986; Clarke et al., 1999; Baier and Katoen, 2008] is a branching-time logic and is able to express the existence of, and properties upon, runs of a system.

#### 3.2.4.1 CTL Syntax

The inductive syntax of CTL as is as follows:

$$\varphi ::= p | \neg \varphi | \varphi \lor \psi | EX\varphi | EG\varphi | E[\varphi U \psi]$$

The readings are as follows:

- $EX\varphi$ – “there exists a next state that satisfies $\varphi$”
- $EG\varphi$ – “there exists a path along where $\varphi$ holds globally”
- $E[\varphi U \psi]$ – “there exists a path where $\psi$ holds until $\varphi$”.

### 3.2.5 CTL Semantics and Abbreviations

Unlike LTL, the satisfaction of a CTL formula is given with respect to a state $s \in W$:

- $s \models p$ iff $p \in V(s)$
- $s \models \neg \varphi$ iff it is not the case that $s \models \varphi$
- $s \models \varphi \lor \psi$ iff $s \models \varphi$ or $s \models \psi$
- $s \models \varphi \land \psi$ iff $s \models \varphi$ and $s \models \psi$
- $s \models EX\varphi$ iff $\exists \pi, \text{ s.t. } \pi_0 = s \text{ and } \pi_1 \models \varphi$
3.2 Temporal Logics

\[ s \models EG \varphi \iff \exists \pi, \text{ s.t., } \pi_0 = s \text{ and } \forall m \geq 0, \pi_m \models \varphi \]

\[ s \models E [\varphi U \psi] \iff \exists \pi \pi_0 = s, \text{ s.t., } \exists m \geq 0, \pi_m \models \psi \text{ and } \forall j < m, \pi_j \models \varphi \]

3.2.5.1 CTL Abbreviations

Given the syntax of CTL, we define the further temporal operators:

\[ AX \varphi \equiv \neg EX (\neg \varphi) \]

\[ EF \varphi \equiv E [trueU \varphi] \]

\[ AG \varphi \equiv \neg EF (\neg \varphi) \]

\[ A [\varphi U \psi] \equiv \neg E [\neg \psi U \neg \varphi \land \neg \psi] \land EG \neg \psi \]

\[ AF \varphi \equiv A [trueU \varphi] \]

In contrast to their existential counterparts, an “A” formula is read as “for all paths”. This can easily be seen in the case of \( AX \varphi \); if it is not the case that there exists a successor satisfying \( \neg \varphi \), then it must be the case that all successors satisfy \( \varphi \).

3.2.5.2 Closure of a CTL formula

As for LTL, we define the closure of a CTL formula to be the formula itself and all of its constituent subformulae.

For example, the closure of \( \varphi = A [AX p U EX q] \) is:

\[ cl(\varphi) = \{A [AX p U EX q], AX p, EX q, p, q\} \]

3.2.6 Further Logics

For completeness in later in the thesis, we now introduce three further logics: CTL\(^*\), ACTL and ECTL.

3.2.6.1 CTL\(^*\)

The logic CTL\(^*\) is the branching-time logic that subsumes both CTL and LTL [Emerson and Halpern, 1986]. The logic is consisting of both path and state formulae, where path formulae come from LTL and state formulae come from CTL.
The two types of formulae are defined as follows.

- **State formulae:**
  \[ \varphi ::= p \mid \neg \varphi \mid \varphi \land \varphi \mid A (\psi) \mid E (\psi) \]
  Where \( p \) is an atomic proposition and \( \psi \) is a path formula.

- **Path formulae:**
  \[ \psi ::= \varphi \mid \neg \psi \mid \psi \land \psi \mid \psi U \psi \mid G \psi \mid F \psi \mid X \psi \]
  Where \( \varphi \) is a path state formula.

We refer the reader to [Emerson and Halpern, 1986] for the exact semantics of CTL\(^*\).

It is clear from the above definition that CTL\(^*\) subsumes both CTL and LTL.

### 3.2.6.2 ACTL and ECTL

The two logics ACTL and ECTL are themselves subfragments of the logic CTL. ACTL is called the **universal fragment** of CTL and ECTL the **existential fragment**.

The fragments have the following syntax.

- **Universal fragment:**
  \[ \varphi ::= p \mid \neg p \mid AX \varphi \mid A [\varphi U \varphi] \mid AG \varphi \mid AF \varphi \]

- **Existential fragment:**
  \[ \varphi ::= p \mid \neg p \mid EX \varphi \mid E [\varphi U \varphi] \mid EG \varphi \mid EF \varphi \]

It is important to note that in ACTL and ECTL that negation can only be applied to propositional atoms. As such, ACTL can only be used to specify properties over all runs, while ECTL can only be used to express the existence of runs.

### 3.3 Reasoning about Multi-Agent Systems

To be able to reason about the mental and temporal evolution of the agents in multi-agent systems, we require computationally grounded semantics [Wooldridge, 2000; van der Hoek and Wooldridge, 2003]. In this thesis, we will focus on a particular semantics for representing and reasoning about knowledge and time in multi-agent systems: the **interpreted systems** formalism [Parikh and Ramanujam, 1985; Fagin et al., 1995].
3.3 Reasoning about Multi-Agent Systems

3.3.1 Interpreted Systems

In interpreted systems each agent is endowed with a set of local states representing that agent’s internal state. This internal state could be, for example, an assignment to all of the variables that agent is (metaphorically) composed of; the current status of an internal knowledge base, etc. More simply, it can be seen as the current configuration of the agent.

Each agent also has a repertoire of local actions that it is able to perform. These actions could represent a physical action an agent could take (e.g., “move forward”) or could correlate to a communicative action the agent could take (e.g., Alice might be able to perform the action “send msg to Bob”).

Furthermore, each agent additionally has a (non-deterministic) local protocol that governs which actions an agent can perform in a given state (e.g., if the local state of the agent is “broken”, the protocol of the agent may prescribe the agent to perform the action “repair” or “request_help”).

Finally, agents also have an evolution function. This function stipulates how each agent’s local state is updated, between the current value and the “next assignment”. Rather than being a strictly local function, and to facilitate “communication” between the agents, an agent’s evolution function is defined with respect to all of the actions of all of the agents in the system; we call an action at the system level (i.e., where it contains one action per agent) as a global action. We note that, unlike the protocol function, an agent’s evolution function is deterministic (i.e., there can be only next-state given a current local state and a global action). To exemplify, Bob’s evolution function might contain: \{empty\} $\xrightarrow{\pi}$ \{msg\}, where $\pi$ is the global action in which Bob performs “receive” and Alice performs “send msg.Bob”, while “empty” and “msg” are both possible assignments in Bob’s local state.

Given a set of agents $Ag = \{1, \ldots, n\}$, we define an individual agent formally as follows.

**Definition 3.6. (Agent).** An agent is a tuple $\langle L_i, Act_i, P_i, E_i \rangle$, with elements:

- $L_i$ – the set of local states
- $Act_i$ – the set of local actions
- $P_i : L_i \rightarrow 2^{Act_i}$ – the protocol function
- $E_i : L_i \times \Pi_{j \in Ag} Act_j \rightarrow L_i$ – the evolution function

In addition to the set $Ag$, interpreted systems often include a special agent $e$ called the environment, defined as per Definition 3.6. The environment, while being syntactically and semantically the same as a “normal” agent, is used to encapsulate the rest of the agents in the system. For example, in the Alice and Bob example, the environment might be used to model the channel that Alice and Bob use to communicate. Alternatively, the evolution function of the environment might be used to capture the “side effects”
that an agent’s actions might (conceivably) have upon its physical environment. The environment can therefore be used to capture exogenous changes in a multi-agent system.

3.3.2 Compositions

We now consider two different presentations for the same notion of composition for the set of agents $Ag$ in an interpreted system. One presentation follows a state-based presentation, while the other follows a run-based presentation.

In what follows, a joint action $\text{Act} \subseteq \text{Act}_1 \times \cdots \times \text{Act}_n \times \text{Act}_e$ is a system-level action that is performed by all agents synchronously.

3.3.2.1 State-based Composition

We begin by considering the set of all possible global states

$$G \subseteq L_1 \times \cdots \times L_n \times L_e$$

which is a subset of the Cartesian product of the local states for all agents in the system. A single global state $(l_1, \ldots, l_n, l_e) \in G$ represents an instantaneous configuration of all the agents in the system.

The function $l_i : G \rightarrow L_i$ is a projection of an individual agent’s local state from a given global state. Without ambiguity, we also denote by $l_i \in L_i$, an arbitrary local state; the context will disambiguate.

The transition relation $T \subseteq G \times \text{Act} \times G$ defines the temporal evolution of the system. Given two global states $g$ and $g'$, $(g, g') \in T$ iff there exists a joint action $a_1, \ldots, a_n$, such that for all $i \in Ag$, $a_i \in P_i(l_i(g))$ and $E_i(l_i(g), a_1, \ldots, a_n) = l_i(g')$. That is, there exists a temporal transition between two states iff there exists a global action that is consistent with the protocols of each agent at the source state, and, in the evolution for each agent, the source local state and selected global action lead to the destination local state. We assume seriality of this relation, i.e., every global state has at least one successor.

Given an initial state $\iota \in G$, the protocols for each agent and the global transition function, these can induce an infinite structure representing all of the possible computations of the system. A path $\pi = (t, g_1, \ldots)$ is an infinite sequence of global states such that $\forall k \geq 0 \ (g_k, g_{k+1}) \in T$. $\pi(k)$ is the $k^{th}$ global state of the path $\pi$, whilst $\Pi(g)$ is the set of all paths starting at the given state $g \in G$.

The epistemic accessibility relation $\sim_i \subseteq G \times G$ represents that two global states are indistinguishable for that agent. Formally, $(g, g') \in \sim_i$ iff $l_i(g) = l_i(g')$. That is, two global states are indistinguishable to the agent $i$ if the agent’s local state is the same at
these two states. This means that, given only the information in its local state, the agent is unable to tell the related two states apart. For two states \( g, g' \) such that \( g \sim_i g' \), it is possible that \( l_j(g) \neq l_j(g') \) (i.e., the local state \( j \) is not the same between \( g \) and \( g' \)); in this instance, this means that \( i \) is oblivious to, or is ignorant of, the current state of \( j \).

As we will show in what follows, the indistinguishability relation for each agent can be used to interpret a modal operator encapsulating the knowledge of a given agent.

**Definition 3.7. Model**

A model of an interpreted system is a tuple

\[ M_{IS} = \langle G, i, T, \sim_1, \ldots, \sim_n, V \rangle \]

where:

- \( G \) is the set of reachable states accessible from \( i \) via \( T \)
- \( i \in G \) is an initial state
- \( T \) is the relation as defined above
- \( \sim_i \subseteq G \times G \) is the indistinguishability relation for the agent \( i \in Ag \)
- \( V \) is a mapping of global states to the propositional variables that hold at that state, i.e., \( V : G \to 2^{AP} \).

### 3.3.2.2 Run-based Presentation

We now introduce a second presentation for the composition of \( m \) agents in a multi-agent system. Compared to the previous section, the alternative approach presented below is more focused on the runs of the system, compared to the states.

Nonetheless, we denote by \( S \subseteq L_1 \times L_2 \times \ldots \times L_m \) the set of global states of the MAS. To represent the temporal evolution of the MAS we consider the flow of time \( \mathbb{N} \) of the natural numbers.

**Definition 3.8.** A run in an interpreted system is a function \( \rho : \mathbb{N} \to S \) that intuitively represents a possible evolution of the MAS.

That is, given \( n \in \mathbb{N} \) and a run \( \rho \), \( \rho(n) \in S \) is the \( n \)-th global state in the run.

**Definition 3.9.** An interpreted system (IS) is a tuple \( P = \langle R, s^0, V \rangle \), where:

(i) \( R \) is a non-empty set of runs (Def. 3.8)
(ii) \( s^0 \in S \) is the initial state, i.e., \( s^0 = \rho(0) \) for all \( \rho \in R \)
(iii) \( V : S \to 2^{AP} \) is an assignment for the propositional variables in \( AP \)

In what follows we assume without loss of generality that for every \( s \in S \), there exist \( \rho \in R \) and \( n \in \mathbb{N} \) such that \( s = \rho(n) \), i.e., \( S \) is the set of all reachable states. We refer to a pair \( (\rho, n) \) as a point in \( P \) and we write \( \Pi \) for the set of all points in \( P \). If
We now interpret a multi-modal logic as a logic of knowledge. That is, each relation \( \sim_i \) follows:

- \( \sim_i \) is defined such that \( (p,n) \sim_i (p',n') \) iff \( p(n) = p'(n') \); while \( p|_n \) is the sequence of states \( p(0), \ldots, p(n) \) for the prefix of \( p \). Sometimes we do not distinguish between a point and the associated state when it is clear from the context.

### 3.3.2.3 Logic of Knowledge

We now interpret a multi-modal logic as a logic of knowledge. In this logic, the indistinguishability relations \( \sim_i \) can be used to interpret the knowledge modality \( K_i \). That is, each relation \( \sim_i \) is classed as an epistemic accessibility relation for the modal operator \( K_i \).

For a set of agents \( |Ag| = m \), we denote by \( L_m \) the logic with the following grammar:

\[
\varphi ::= p \mid \neg \varphi \mid \varphi \lor \varphi \mid K_i \varphi \mid E_G \varphi \mid C_G \varphi
\]

where \( p \in AP, i \in Ag \) and \( \Gamma \subseteq Ag \neq \emptyset \).

We provide the following readings of the epistemic formulae in \( L_m \):

- \( K_i \varphi \) – “agent \( i \) knows \( \varphi \)”
- \( E_G \varphi \) – “everybody in the group \( \Gamma \) knows \( \varphi \)”
- \( C_G \varphi \) – “\( \varphi \) is common knowledge between the agents in \( \Gamma \)”

Furthermore, for \( n \in \mathbb{N} \), we define \( E^0_G \varphi = \varphi \) and \( E^{n+1}_G \varphi = E_G E^n_G \varphi \).

We note that there is a further modality, distributed knowledge, denoted \( D_G \), which is commonly used when reasoning about multi-agent systems. However we do not address distributed knowledge in this thesis.

The semantics of a formula \( \varphi \) in \( L_m \) given an interpreted system \( \mathcal{P} = \langle \mathcal{R}, \delta^0, I \rangle \) is given below.

**Definition 3.10.** The satisfaction relation \( \models \) for \( \varphi \in L_m \) and \( (p,n) \in \mathcal{P} \) is defined as follows:

\[
\begin{align*}
(\mathcal{P}, p, n) \models p & \quad \text{iff} \quad p \in V(p(n)) \\
(\mathcal{P}, p, n) \models \neg \psi & \quad \text{iff} \quad \text{it is not the case that} \ (\mathcal{P}, p, n) \models \psi \\
(\mathcal{P}, p, n) \models \psi \lor \psi' & \quad \text{iff} \quad (\mathcal{P}, p, n) \models \psi \text{ or } (\mathcal{P}, p, n) \models \psi' \\
(\mathcal{P}, p, n) \models K_i \psi & \quad \text{iff} \quad (p, n) \sim_i (p', n') \text{ implies } (\mathcal{P}, p', n') \models \psi \\
(\mathcal{P}, p, n) \models E_G \psi & \quad \text{iff} \quad \text{for all } i \in G, (\mathcal{P}, p, n) \models K_i \psi \\
(\mathcal{P}, p, n) \models C_G \psi & \quad \text{iff} \quad \text{for all } k \in \mathbb{N}, (\mathcal{P}, p, n) \models E^k_G \psi
\end{align*}
\]

We note that the modalities \( E_G \) and \( C_G \) can be expressed as modalities expressed over the following relations:

\[
\begin{align*}
\mathcal{R} & = \{ \rho \mid \text{the logic with the following grammar:} \} \\
\end{align*}
\]
\[ R^* \equiv \left( \bigwedge_{i \in \Gamma} R_i^* \right) \]

Where \( R^* \) represents the transitive closure of the relation \( R \).

### 3.3.2.4 CTL + \( L_m \) = Branching-Time Multi-Modal Logic CTLK

We now introduce the temporal-epistemic logic CTLK, which is the fusion logic [Kurucz, 2006] of computation tree logic with the multi-modal epistemic logic \( L_m \) for the set \( Ag = \{ 1, \ldots, m \} \) of agents. We provide this language with a formal semantics in terms of interpreted systems [Fagin et al., 1995].

The models of interpreted systems can be used to reason about a branching-time temporal-epistemic logic. The language CTLK is built from a countable set of propositional variables \( AP \) and using the following syntax:

\[
\varphi, \psi ::= p \mid \neg \varphi \mid \varphi \lor \psi \mid EX \varphi \mid EG \varphi \mid E [\varphi U \psi] \mid K_i \varphi
\]

where \( i \in Ag \). The epistemic modality \( K_i \varphi \) is read as “agent \( i \) considers it possible that \( \varphi \)”. We define \( EF \varphi \) as \( E [true U \varphi] \). The duals are as follows: \( AX \varphi \equiv \neg EX \neg \varphi, AF \varphi \equiv \neg EG \neg \varphi \) and \( AG \varphi \equiv \neg EF \varphi \). The dual of the epistemic modality for “possibility” is “knowledge”; \( K_i \varphi \) is defined as \( \neg K_i \neg \varphi \), and is read as “agent \( i \) knows \( \varphi \)”.

We define the connectives \( \land, \lor, \rightarrow \) and the propositional constants \( true \) and \( false \) as standard.

We note that \( EX \phi \) is defined as \( \neg AX \neg \phi \). The linear-time operator \( U \) is dual to \( U \), that is, \( A\phi U \phi' \) is defined as \( \neg E \neg \phi \neg U \neg \phi' \), and \( E\phi U \phi' \) as \( \neg A \neg \phi \neg U \neg \phi' \). We will sometimes refer to \( U \) as \( R \); the “release” operator. The operator \( C_G \) is dual to \( C_G \), i.e., \( C_G \phi \) is a shorthand for \( \neg C_G \neg \phi \). Also, the operators \( AG, AF, EG \) and \( EF \) are defined as standard. Finally, \( E_G \phi \) is defined as \( \bigwedge_{i \in G} K_i \phi \).

The \( U \)-formulae in CTLK are the formulae of the form \( A\phi U \phi' \) or \( E\phi U \phi' \) for some \( \phi, \phi' \in CTLK \); the \( U \), \( K_i \) and \( C_G \)-formulae are defined similarly.

### 3.3.2.5 State-based Satisfaction

Given a model \( \mathcal{M}_{15} \) (Def. 3.7), a global state \( g \) and two CTLK formulae \( \varphi \) and \( \psi \), satisfaction of \( \varphi \) and \( \psi \) at a global state \( g \) in a model \( \mathcal{M}_{15} \), written \( \mathcal{M}_{15}, g \models \varphi \) (or, for brevity, \( g \models \varphi \)), is defined as follows:

\[
g \models \varphi \iff p \in V(g)
\]
We now introduce automata on infinite objects. In particular, we will focus on automata on infinite words and on infinite trees.

### 3.4 Automata

We now introduce automata on infinite objects. In particular, we will focus on automata on infinite words and on infinite trees.

A CTLK formula $\phi$ is valid in a model $\mathcal{M}_I = (G, I, T, \sim_1, \ldots, \sim_n, V)$ iff $\mathcal{M}_I, I \models \phi$, i.e., $\phi$ is true in the initial state of a model.

#### 3.3.2.6 Run-based Satisfaction

Following the run-based presentation of interpreted systems (Section 3.3.2.2), we also provide semantics for CTLK in this context.

**Definition 3.11.** The satisfaction relation $\models$ for $\phi \in$ and $(\rho, n) \in \mathcal{P}$ is defined as follows:

\[
\begin{align*}
(\mathcal{P}, \rho, n) \models p & \quad \text{iff } p \in V(\rho(n)) \\
(\mathcal{P}, \rho, n) \models \neg \psi & \quad \text{iff it is not the case that } (\mathcal{P}, \rho, n) \models \psi \\
(\mathcal{P}, \rho, n) \models \psi \lor \psi' & \quad \text{iff } (\mathcal{P}, \rho, n) \models \psi \text{ or } (\mathcal{P}, \rho, n) \models \psi' \\
(\mathcal{P}, \rho, n) \models AX \psi & \quad \text{iff for all runs } \rho', \rho'|_n = \rho|_n \text{ implies } (\mathcal{P}, \rho', n+1) \models \psi \text{ and for all } k', n \leq k' < k \text{ implies } (\mathcal{P}, \rho', k') \models \psi \\
(\mathcal{P}, \rho, n) \models A\psi U \psi' & \quad \text{iff for some run } \rho', \rho'|_n = \rho|_n \text{ and there is } k \geq n, (\mathcal{P}, \rho', k) \models \psi' \text{ and for all } k', n \leq k' < k \text{ implies } (\mathcal{P}, \rho', k') \models \psi \\
(\mathcal{P}, \rho, n) \models K_i \psi & \quad \text{iff } (\rho, n) \sim_i (\rho', n') \text{ implies } (\mathcal{P}, \rho', n') \models \psi \\
(\mathcal{P}, \rho, n) \models C_G \psi & \quad \text{iff for all } k \in \mathbb{N}, (\mathcal{P}, \rho, n) \models E_k^G \psi
\end{align*}
\]

The truth conditions for $\land, \lor, \rightarrow, \text{true}, \text{false}, EX, AU, EU, K_i$ and $C_G$ are defined from those above. A formula $\phi$ is true on an IS $\mathcal{P}$ iff it is satisfied at $(\rho, 0)$ such that $\rho(0) = s^0$.

### 3.4 Automata

We now introduce automata on infinite objects. In particular, we will focus on automata on infinite words and on infinite trees.
By way of motivation, and as we will show later in this chapter for both linear-time and branching-time logics, automata can be used to decide if a given model satisfies a formula or not.

We begin by introducing standard non-deterministic automata, and then present the theory related to alternating automata, which further generalise non-deterministic automata.

### 3.4.1 Automata on Infinite Words

Given a fixed and finite alphabet \( \Sigma = \{a, b, \ldots\} \), we denote by \( \Sigma^* \) the set of finite strings and by \( \Sigma^\omega \) the set of infinite strings [Farwer, 2002]. Given an infinite word \( \alpha \in \Sigma^\omega \), we denote by \( \alpha(i) \in \Sigma \) the \( i \)-th letter in the string \( \alpha \).

As \( \Sigma \) is finite but \( \Sigma^\omega \) is infinite, any word \( \alpha \in \Sigma^\omega \) will contain an infinite number of occurrences of a subset of \( \Sigma \). Consequently, we define:

\[
\text{Inf} \ (\alpha) = \{ a \in \Sigma \mid \text{for infinitely many } i, \ \alpha(i) = a \}
\]

to be the set of letters occurring infinitely often in \( \alpha \).

**Definition 3.12.** (\( \omega \)-automata). An \( \omega \)-automaton—or a finite automaton on infinite words—is a tuple

\[ \mathcal{A} = (Q, \Sigma, \delta, Q_0, \text{Acc}) \]

where:

- \( Q \) is a finite set of states
- \( \Sigma \) is the alphabet of the automaton
- \( \delta \subseteq Q \times \Sigma \rightarrow 2^Q \) is the transition function
- \( Q_0 \subseteq Q \) is a set of initial states
- \( \text{Acc} \) is the acceptance condition

The set \( \delta(q,a) \) is the set of states which the automaton \( \mathcal{A} \) “moves into” when it is in the state \( q \) and sees the letter \( a \). Given an \( \omega \)-automaton \( \mathcal{A} = (Q, \Sigma, \delta, q_0, \text{Acc}) \) and a string \( \alpha \in \Sigma^\omega \), a run of \( \mathcal{A} \) is a sequence \( r = q_0, q_1, \ldots \), with \( q_0 \in Q_0 \) and \( q_{n+1} \in \delta(q_n, \alpha(n)) \).

In the theory of automata over finite strings, a string is accepted if it reaches a set of “final” states in the automaton. However, the notion of final states in infinite automata is not possible: there cannot be a “final” state given an infinite input string. Consequently, the acceptance condition \( \text{Acc} \) needs to be suitably adapted for infinite strings.

As a matter of notation, in what follows, for an automaton \( \mathcal{A} \) we denote by \( \mathcal{L}(\mathcal{A}) \) the language, or greatest set of infinite strings that \( \mathcal{A} \) accepts. Furthermore, we also write \( \mathcal{A}^X \) for the automaton \( (\Sigma, S, X, \delta, \text{Acc}) \), \( X \in S \). That is, \( \mathcal{A}^X \) is the automaton \( \mathcal{A} \) with the initial state \( X \).
Definition 3.13. (Büchi automata.) A Büchi automaton \( A = (Q, \Sigma, \delta, Q_0, \text{Acc}) \) is a \( \omega \)-automaton, where \( \text{Acc} \subseteq Q \). A string \( \alpha \in \Sigma^\omega \) is accepted by \( A \) iff there exists a run \( \rho \) of \( A \) for \( \alpha \), such that:

\[
\text{Inf}(\rho) \cap \text{Acc} \neq \emptyset.
\]

It follows that a string \( \alpha \) is accepted by \( A \) iff the corresponding run \( \rho \) visits a state in \( \text{Acc} \) infinitely often.

A further acceptance condition is a co-Büchi acceptance condition [Löding and Thomas, 2000]. Given co-Büchi acceptance condition \( \text{Acc} \), a run \( \rho \) is co-Büchi accepting iff:

\[
\text{Inf}(\rho) \cap \text{Acc} = \emptyset.
\]

That is, a “run is accepting iff it visits states from the accepting set only finitely often” [Kupferman et al., 2004].

We also consider Rabin automata, where, unlike Büchi automata, the acceptance condition is a set of pairs of states:

Definition 3.14. (Rabin automata.) A Rabin automaton \( A = (Q, \Sigma, \delta, Q_0, \text{Acc}) \) is a \( \omega \)-automaton, where \( \text{Acc} = \langle (L_0, U_0), \ldots, (L_n, U_n) \rangle \), such that \( \forall n < |\text{Acc}|, L_n \subseteq Q \) and \( U_n \subseteq Q \). A string \( \alpha \) is accepted by \( A \) iff there exists a run \( \rho \) on \( A \) for \( \alpha \), such that:

\[
\exists (L_i, U_i) \in \text{Acc} \text{ such that } \text{Inf}(\rho) \cap L_i \neq \emptyset \text{ and } \text{Inf}(\rho) \cap U_i = \emptyset.
\]

For each Rabin pair \((L_i, U_i) \in \text{Acc}\), we denote by \( L_i \) the set of “good” states and \( U_i \) by the set of “bad” states. A run of \( A \) is accepted if there is at least one Rabin pair such that the run infinitely often enters the good states of the pair and visits the bad states only finitely often.

So far, we have not constrained the form of \( \delta \); that is, as it returns an element in the powerset of \( Q \) (i.e., \( 2^Q \)), an automaton can have many choices of next state given a current state and an input letter. We therefore define the following classes of automata.

Definition 3.15. (Deterministic Automata.)

An \( \omega \)-automaton is deterministic iff

\[
\forall a \in \Sigma, \forall q \in Q, |\delta(q, a)| \leq 1
\]

If an \( \omega \)-automaton is deterministic, this means that given a current state \( q \) and an input letter \( a \), the next state is uniquely determined.

Definition 3.16. (Non-deterministic automata.)

An \( \omega \)-automaton is non-deterministic iff

\[
\exists a \in \Sigma, \exists q \in Q, |\delta(q, a)| > 1
\]

That is, for at least one state \( q \in Q \), and for at least one input letter \( a \in \Sigma \), there are possibly many successor states.
We now present some common acronyms that we will use throughout this thesis; these can be seen in Table 3.3.

<table>
<thead>
<tr>
<th>Name</th>
<th>Acronym</th>
</tr>
</thead>
<tbody>
<tr>
<td>NBA</td>
<td>Non-deterministic Büchi Automata</td>
</tr>
<tr>
<td>DBA</td>
<td>Deterministic Büchi Automata</td>
</tr>
<tr>
<td>NRA</td>
<td>Non-deterministic Rabin Automata</td>
</tr>
<tr>
<td>DRA</td>
<td>Deterministic Rabin Automata</td>
</tr>
</tbody>
</table>

Furthermore, the equivalence and expressiveness of various deterministic and non-deterministic automata can be seen in Figure 3.1; we write $A \Rightarrow B$ to mean that $B$ is strictly more expressive than $A$ (i.e., automata of the type $B$ can express more infinite languages than those in $A$) and $A \Leftrightarrow B$ if $A$ and $B$ are equally expressive.

<table>
<thead>
<tr>
<th>deterministic Büchi ⇒ non-deterministic Büchi</th>
</tr>
</thead>
<tbody>
<tr>
<td>deterministic Rabin</td>
</tr>
<tr>
<td>non-deterministic Rabin</td>
</tr>
</tbody>
</table>

Fig. 3.1. Equivalence between $\omega$-automata.

Figure 3.1 shows us, for example, that a non-deterministic Büchi automata is strictly more expressive than a deterministic Büchi automata [Roggenbach, 2002]. We also observe that deterministic Rabin automata are strictly more expressive than deterministic Büchi automata.

### 3.4.2 Automata on Infinite Trees

We now introduce finite automata over infinite trees [Visser et al., 1997; Kupferman and Grumberg, 1996; Visser and Barringer, 2000; Kupferman et al., 2000; Penczek and Pótrola, 2006].
3.4.2.1 Trees

A tree is a connected, directed, rooted (and possibly infinite) graph which has a single root denoted $\varepsilon$; every other, non-root node has a unique parent – if $s$ is the parent of $t$, then there exists an edge in the tree from $s$ to $t$. The node $\varepsilon$ does not have a parent.

In a tree $\tau$, the degree of a node $x$, denoted by $d(x)$, is the number of successors of $x$ in $\tau$. A leaf node is a node which has no successors (i.e., $d(x) = 0$). A tree is leafless if every node has at least one child.

A tree $\tau$ over $\mathbb{N}$ is a subset of $\mathbb{N}^*$ (i.e., a finite sequence in $\mathbb{N}$). If $x \cdot i \in \tau$, where $x \in \mathbb{N}^*$ and $i \in \mathbb{N}$, then $x \in \tau$ and for all $0 \leq i' < i$, $x \cdot i' \in \tau$. That is, if $x$ is the parent of $x \cdot i$, then $x \cdot i$ must have siblings $x \cdot j$, for all $0 \leq j < i$.

Example 3.1. Consider a tree containing a node $0 \cdot 3 \cdot 2$. This means that the tree must also contain nodes $0 \cdot 3 \cdot 1$ and $0 \cdot 3 \cdot 0$, as well $0 \cdot 2$, $0 \cdot 1$, $0 \cdot 0$ and $0$. Furthermore, the degree $d(0)$ of the node 0 is 4, while the degree $d(0 \cdot 3)$ is 3.

A path $\pi$ in a tree $\tau$ is a sequence of nodes starting with $\varepsilon$, such that either $x$ is a leaf node, or there exists a unique $i \in \mathbb{N}$ such that $x \cdot i \in \pi$. For a finite path, then the last state in $\pi$ is a leaf node in $\tau$. In the same way as for word automata, we denote by $\inf(\pi)$ the set of nodes $x$ which appear infinitely often in a path $\pi$.

Given a set of degrees $D \subset \mathbb{N}$, then a tree $\tau$ is a $D$-tree if $\tau$ is a tree over $\mathbb{N}$ and $\forall x \in \tau, d(x) \in D$.

Given a finite alphabet $\Sigma$, a $\Sigma$-labelled tree is a pair $(\tau, T)$, where $\tau$ is a tree and $T$ is the function $T : \tau \to \Sigma$ and that maps nodes of $\tau$ to letters in $\Sigma$. A path $\pi = x_0, x_1, \ldots$ defines a word $T(\pi) = T(x_0), T(x_1) \ldots$ composed of all the letters of each node.

3.4.2.2 Tree Automata

We now consider automata on labelled leafless $D$-trees.

Definition 3.17. Non-deterministic Tree Automata

A non-deterministic tree automaton $A = (\Sigma, D, S, s_0, \delta, \text{Acc})$ is a tuple where:

- $\Sigma$ is an finite alphabet
- $D \subset \mathbb{N}$ is a set of branching degrees
- $S$ is a set of states
- $S_0 \subseteq S$ is a set of initial states
- $\delta : S \times \Sigma \times D \to 2^S$ is a transition function, such that $\delta(s, a, k) \subseteq S^k$ for each $s \in S$, $a \in \Sigma$ and $k \in D$. For a degree $k$, $\delta(, a, k)$ returns a set of states with cardinality $k$.
- $\text{Acc}$ is an acceptance condition.
3.4 Automata

Intuitively, when an automaton $A$ is in a state $s$ and reads a node $x$ with degree $k$ of a tree $\tau$, it non-deterministically chooses a $k$-arity tuple $(s_1, \ldots, s_k) \in \delta(s, T(x), k)$ and makes $k$ copies of itself. Each copy moves into a node $x \cdot i$ and to a state $s_i$, for $i = 1, \ldots, k$.

A run $r: \tau \rightarrow S$ of an automaton $A$ over a tree $\langle \tau, T \rangle$ is a tree where the root is labelled with an initial state in $s_0$ and every other node is labelled with $N^* \times S$. Therefore, $r$ can be seen as a $\Sigma$-labelled tree $\langle \tau, T_r \rangle$ such that $\Sigma_r = N^* \times S$ and where $\langle \tau, T_r \rangle$ satisfies the following:

- $\varepsilon \in \tau_r$ and $T(\varepsilon) = (\varepsilon, s_0)$.
- The labels of a node, and its successors (therefore every transition in $\tau_r$) must obey the transition function $\delta$.

It can be seen that each node of $r$ corresponds to a node in the tree $\tau$. A node of $r$, labelled with $\langle x, s \rangle$, describes a copy of the automaton that reads a node $x$ of $\tau$ in the state $s$ of the automaton $A$.

**Example 3.2.** Consider the run $\langle T, r \rangle$ of an automaton $A$ over a tree $\langle T, V \rangle$, such that $r(0) = q, V(0) = a$ and $d(0) = 2$. Furthermore, suppose that transition function contains the following:

$$\delta(q, a, 2) = \{ (q_1, q_2), (q_3, q_4) \}$$

It then follows that at the next level of the tree either there are nodes $r(0 \cdot 0) = q_1$ and $r(0 \cdot 1) = q_2$, or there are nodes $r(0 \cdot 0) = q_3$ and $r(0 \cdot 1) = q_4$. That is, the node 0 labelled with $q$ either has successors $0 \cdot 0$ labelled with $q_1$ and $0 \cdot 1$ labelled with $q_2$, or it has successors $0 \cdot 0$ labelled with $q_3$ and $0 \cdot 1$ labelled with $q_4$. \[\triangle\]

As with infinite word automata, a tree automata accepts a tree if and only if there exists a run that accepts it. We denote by $\mathcal{L}(A)$ the set of $\Sigma$-labelled trees that $A$ accepts.

### 3.4.2.3 Alternating Tree Automata

Alternating automata generalise the behaviour of non-deterministic automata – while non-deterministic automata can only express existential choice, alternating automata can also express universal choice.

For a given set $X$ of variables, $B^+(X)$ represents the set of positive Boolean formulae over $X \cup \{ \text{true}, \text{false} \}$ constructed using $\land$ and $\lor$. Given $Y \subseteq X$, the set $Y$ satisfies a formula $\theta \in B^+(X)$ if $\theta$ is satisfied when assigning all the variables in $Y$ (and therefore occurring in $\theta$) to true and those in $X \setminus Y$ to false.

An alternating tree automata is similar to a non-deterministic tree automata, but where the transition function is a partial function:

$$\delta: S \times \Sigma \times D \rightarrow B^+(N \times S)$$
where $\delta(s, a, k) \in B^+ (\{1, \ldots, k\} \times S)$ for each $s \in S$, $a \in \Sigma$ and $k \in D$. A conjunction in $\delta$ represents universal choice, while disjunction represents existential choice.

**Example 3.3.** For example, consider the following entry in the transition relation as follows:

$$\delta(s, a, 2) = ((1, s_1) \lor (2, s_2)) \land ((1, s_3) \lor (2, s_1))$$

The right hand expression is the transition that is selected when the automaton is in the state $s$ when the letter $a$ is read and when the input tree has a branching degree of two. As such, the nodes at next level of the tree of the automaton include $(1, s_1)$ or $(2, s_2)$ and include $(1, s_3)$ or $(2, s_1)$.

\[\triangle\]

A run $r$ of an alternating Büchi tree automaton $A = \langle \Sigma, D, S, s_0, \delta, F \rangle$, on a $\Sigma$-labelled, leafless $D$-tree $T = \langle \tau, T_R \rangle$ results in a $\tau \times S$-labelled tree. A node in $r$, labelled with $\langle x, s \rangle$, describes a copy of the automaton that reads the node $x$ of $\tau$ in the state $s \in S$.

The labels of a node, and its children, in a run $r$ have to satisfy the transition function of $A$. When $r$ is a $\Sigma_r$-labelled tree $\langle \tau_r, T_r \rangle$, where $\Sigma_r = \tau \times S$, $\langle \tau_r \times T_r \rangle$ satisfies the following:

1. $T_r(e) = (e, s_0)$.
2. Given $y \in \tau_r$, $T_r(y) = (x, s)$, $d(x) = k$ and $\delta(s, T(x), k) = \theta$. Then, there exists a set $Q = \{(c_1, s_1), \ldots, (c_n, s_n)\} \subseteq \{1, \ldots, k\} \times S$ such that:
   - $Q$ satisfies $\theta$
   - $\forall 1 \leq i \leq n$, $y \cdot i \in \tau_r$ and $T_r(y \cdot i) = (x \cdot c_i, s_i)$

For an alternating Büchi tree automaton, a run is accepting if all its infinite paths satisfy the Büchi acceptance condition $F$.

We now introduce two extensions to alternating automata: weak alternating tree automata and hesitant alternating tree automata. The former will be used in Chapter 4 and the latter in Chapter 5.

**3.4.2.4 Weak Alternating Tree Automata**

A weak alternating tree automata (WAA) [Kupferman et al., 2000] is a Büchi alternating tree automata such that there exists a partition of the states $S$ into disjoint sets $S_1, \ldots, S_n$ such that for each set, either $S_i \subseteq F$ or $S_i \cap F = \emptyset$. When $S_i \subseteq F$, $S_i$ is an accepting set; when $S_i \cap F = \emptyset$, $S_i$ is a rejecting set.

Additionally, there exists a partial order $\leq$ over the partitions, such that for each $s \in S_i$, $s' \in S_j$ and $s' \in \delta(s, a, k)$ for some $a \in \Sigma$ and $k \in D$. $S_j \leq S_j$. As such, each transition in $\delta$ either transitions into a state contained in the same set, or moves into a set lower in the partial-order.
Eventually, a run of a WAA will end up trapped in a given $S_i$. Such a run is accepting iff $S_i \subseteq F$.

### 3.4.2.5 Hesitant Alternating Tree Automata

Hesitant Alternating Tree Automata (HAA) [Kupferman et al., 2000; Penczek and Pólrola, 2006] are alternating Büchi tree automata that have a restricted transition relation, but the acceptance condition is stronger. Each $S_i$ in the partial order is classified as either transient, existential or universal. The transition relation is restricted as follows:

- If $S_i$ is transient, then $\forall s \in S_i, \delta(s,_,_)$ contains no elements of $S_i$.
- If $S_i$ is existential, then $\forall s \in S_i, \delta(s,_,_)$ only contains disjunctively related elements of $S_i$.
- If $S_i$ is universal, then $\forall s \in S_i, \delta(s,_,_)$ only contains conjunctively related elements of $S_i$.

An infinite run of a HAA will eventually get trapped in either an existential or a universal partition of the states.

The acceptance condition of a HAA is a pair of states $\langle G, B \rangle$, similar to a single Rabin pair. Given the trapped set of states ($S_i$), a path $r$ satisfies the acceptance condition iff $S_i$ is universal and $inf(r) \cap B = \emptyset$ or $S_i$ is a existential set and $inf(r) \cap G \neq \emptyset$.

### 3.4.3 Expressivity of Automata

We now discuss the potential of certain forms of automata to express either other classes, or being able to express the language of a temporal formula.

#### 3.4.3.1 From LTL to NBA

It is well known that it is possible to construct an automaton $A_\phi$, where the number of states in the automaton is less than $2^{O(|\phi|)}$, such that the language of $A_\phi$ (denoted $L(A_\phi)$) is exactly the computations that satisfy $\phi$. We refer the interested reader to [Gerth et al., 1996; Wolper et al., 1983; Vardi and Wolper, 1986; Vardi, 2007; Vardi, 1996].

We reproduce the standard theorem below:

**Theorem 3.1. LTL to BA** [Vardi and Wolper, 1994; Wolper, 2000]

Given an LTL formula $\phi$, there exists a Büchi automaton $A_\phi = \langle Q, \Sigma, \delta, Q_0, Acc \rangle$, with $|Q| = 2^{O(|\phi|)}$, $\Sigma = 2^{Prop}$ and where $L(A_\phi)$ is exactly those runs $\pi$ such that $\pi \models \phi$. 
3.4.3.2 From NBAs to DRAs

As summarised in Figure 3.1, we have that DRAs and NBAs are equally expressive. As such, one can efficiently construct a DRA from an NBA [Klein and Baier, 2006]. This approach follows Safra’s construction [Roggenbach, 2002]. Consequently, we can translate an LTL formula to a DRA, by first constructing its NBA and then determinising this into a DRA.

For example, consider the NBA constructed for the formula $Gp$ shown in Figure 3.2. If a transition does not exist for a given input letter (e.g., $q$), then the infinite path is implicitly rejecting.

![Fig. 3.2. Non-deterministic Büchi Automaton for $Gp$.](image)

Using a determinisation procedure, the DRA corresponding to the NBA shown in Figure 3.2 can be seen in Figure 3.3. We note that the acceptance condition of this automaton is as follows:

$$\text{Acc} = \{ \langle 1, 1 \rangle \}$$

That is, it is only paths that infinitely often visit $\top$ and do not visit $\bot$ that are accepted.

![Fig. 3.3. Deterministic Rabin Automaton for $Gp$.](image)

3.5 Model Checking

Model Checking [Clarke et al., 1999; Baier and Katoen, 2008; Clarke and Emerson, 1982; Merz, 2001] is the automated procedure for verifying if a (finite) transition system
satisfies a property in a given logic. We note that the “model” in model checking is not referring to checking if the given input system satisfies the property; it is checking if the input model is a model of the formula and then therefore satisfies the formula. Model checking is therefore a procedure for deciding if a given input represents a possible model of the given formula we wish to verify.

### 3.5.1 For LTL

We begin by reviewing the standard automata-theoretic approach for verifying LTL.

#### 3.5.1.1 Büchi Automata and LTL

As noted in Section 3.4.3.1, given an LTL formula $\varphi$ there exists a translation between the formula and an infinite Büchi word automata [Vardi and Wolper, 1994; Wolper, 2000].

As such, the model checking problem for LTL can be reduced to checking the non-emptiness of the intersection between the language of the (automaton representing the) system and the language of the automaton representing the negation of the formula. Formally:

$$M \models_{\text{LTL}} \varphi \iff L(M) \cap L(A_{\neg \varphi}) = \emptyset.$$  

The automata-theoretic verification of an LTL formula is as follows:

1. Given a model $M$ and a property $\varphi$
2. Construct the automaton $A_{\neg \varphi}$ for the formula $\neg \varphi$
3. Take the cross-product of the model $M$ and the automaton $A_{\neg \varphi}$, denoted by $A_{M,\neg \varphi}$.

   By construction, we have:

   $$L(A_{M,\neg \varphi}) = L(M) \cap L(A_{\neg \varphi})$$

   That is, $A_{M,\neg \varphi}$ accepts exactly those strings that are contained in both $M$ and $\neg \varphi$.

4. Check $L(A_{M,\neg \varphi})$ for emptiness, i.e., the language of $A_{M,\neg \varphi}$ accepts no input.

The correctness is easy to observe: imagine that there exists an infinite string $\sigma \in L(A_{M,\neg \varphi})$. This means that there exists a sequence of $M$ such that this behaviour is also accepted by $A_{\neg \varphi}$. If this is the case, then it means that this behaviour does not satisfy $\varphi$, and therefore it is not the case that all behaviours of $M$ satisfy $\varphi$.

We further note that such automata-theoretic verification can be performed on-the-fly [Gerth et al., 1996]. This is where the product automaton $A_{M,\neg \varphi}$ is constructed at the same time as exploring the state-space of $M$. As such, if a rejecting path is found in $M$, the procedure can terminate early without having to construct the full state-space of $M$. 

---

**Note:** The above text is a reproduction without further changes or annotations. The content is provided as is for the purpose of this exercise.
3.5.2 For CTL

There are two potential ways to verify a CTL formula: an automata-theoretic method or a set-based approach. As the latter parts of this thesis (i.e., Chapters 4 and 5) rely heavily on the automata-theoretic presentation, we present this in detail below.

For details on the set-based approach, we refer the interested reader to [Huth and Ryan, 2004].

3.5.2.1 Model Checking CTL with Automata

The automata-theoretic approach for CTL model checking is very similar to that for LTL [Vardi, 2006]. Given a CTL formula $\phi$, and a Kripke structure $K$ with branching degrees $D$ (Sec. 3.4.2.1):

1. Construct the alternating automaton representing $A_D, \phi$
2. Construct the product alternating automaton $A_K, \phi = K \times A_D, \phi$
3. Check if $L(A_K, \phi)$ is non-empty

Tree automata and CTL.

It has been shown that there exists a linear-time translation from CTL formulae to alternating automata [Vardi, 2006; Visser et al., 1997], where each state of the automaton for $\phi$ represents a sub-formula of $\phi$.

Given a CTL formula in negation normal form $\phi$ and a finite set of branching degrees $D \subset \mathbb{N}$ (Sec. 3.4.2.1), it is possible to build an automaton $A_\phi = \langle \Sigma, D, S, S_0, \delta, F \rangle$ [Vardi, 2006; Visser et al., 1997] where $\Sigma = 2^D$, such that $L(A_\phi)$ is exactly the set of $D$-trees satisfying $\phi$. The set $S$ of states consists of all the sub-formulae of $\phi$; the initial state is $\phi$. Furthermore, the set of accepting states $F$ are the formulae $A U$ in the closure of the formula.

The transition relation can be seen in Table 3.4.

Example 3.4. Consider the formula $\phi = EF p$. Following the presentation earlier in this chapter, we know that the closure of the formula is:

$$cl(\phi) = \{EF p, p\}$$

As $\phi$ only contains a single proposition, we also have its alphabet as:

$$\Sigma = \{\emptyset, \{p\}\}$$
Table 3.4. The structure of the transition relation for an alternating automata for a CTL formula

\[
\begin{align*}
\delta(p, \sigma, k) &= \top \text{ if } p \in \sigma \\
\delta(p, \sigma, k) &= \bot \text{ if } p \notin \sigma \\
\delta(\neg p, \sigma, k) &= \top \text{ if } p \notin \sigma \\
\delta(\neg p, \sigma, k) &= \bot \text{ if } p \in \sigma \\
\delta(\varphi \land \psi, \sigma, k) &= \delta(\varphi, \sigma, k) \land \delta(\psi, \sigma, k) \\
\delta(\varphi \lor \psi, \sigma, k) &= \delta(\varphi, \sigma, k) \lor \delta(\psi, \sigma, k) \\
\delta(\text{AX} \varphi, \sigma, k) &= \delta(\varphi, \sigma, 0) \\
\delta(\text{AX} \varphi, \sigma, k) &= \delta(\varphi, \sigma, k) \land \left( \bigvee_{c=0}^{k-1} (c, \varphi) \right) \\
\delta(E [\varphi U \psi], \sigma, k) &= \delta(\psi, \sigma, k) \lor \left( \delta(\varphi, \sigma, k) \land \bigvee_{c=0}^{k-1} (c, \varphi U \psi) \right) \\
\delta(A [\varphi U \psi], \sigma, k) &= \delta(\psi, \sigma, k) \lor \left( \delta(\varphi, \sigma, k) \land \bigwedge_{c=0}^{k-1} (c, \varphi U \psi) \right) \\
\delta(E [\varphi R \psi], \sigma, k) &= \delta(\psi, \sigma, k) \lor \left( \delta(\varphi, \sigma, k) \lor \bigvee_{c=0}^{k-1} (c, \varphi R \psi) \right) \\
\delta(A [\varphi R \psi], \sigma, k) &= \delta(\psi, \sigma, k) \lor \left( \delta(\varphi, \sigma, k) \lor \bigwedge_{c=0}^{k-1} (c, \varphi R \psi) \right) \\
\end{align*}
\]

It now remains to define the transition relation \( \delta \) for the alternating automaton \( \mathcal{A}_\varphi = (\Sigma, D, S, S_0, \delta, F) \). As above, we have its alphabet \( \Sigma \) and its states \( S \), its initial state (i.e., \( S_0 = \varphi \)), and its acceptance condition is \( F = \emptyset \) (i.e., the automaton for \( EFp \) should accept no infinite strings).

The transition relation is given below, where \( q \) is an arbitrary state in \( S \):

\[
\begin{array}{c|c|c}
q & \delta(q, \emptyset, k) & \delta(q, \{p\}, k) \\
\hline
EFp & \bigvee_{c=0}^{k-1} (c, EFp) & \text{true} \\
\hline
p & \text{false} & \text{true} \\
\end{array}
\]

When the automaton reads \( p \), irrespective of the current state, the transition relation returns \text{true} – so its work is over. However, when the automaton reads a state in which \( p \) does not hold, it sends a copy of itself to all successor states, and then checks the disjunction of those copies.

We note that, following Table 3.4, the transition relation for a formula is amalgamated with the transition relation for its constituent subformulae (i.e., the whole transition relation can be constructed by substituting in-place for all subformulae). While Table 3.4 shows that \( EU \) formulae contains many parts, we only have a single clause.
in $\delta(EFp, \emptyset, k)$. This is due to the fact that $\delta(p, \emptyset, k)$ (i.e., the transition relation for the expansion of $p$) returns $false$ and $\delta(true, \emptyset, k)$ (i.e., the transition relation for the expansion of $true$) returns $true$.

\[\triangle\]

**Model checking CTL with trees.**

Model checking for CTL, when using trees, is reduced to the inclusion between the runs of a structure and the language of the tree automata which describes the given property.

A Kripke structure $K = (W, w_0, R, V)$ can be viewed as a $W$-labelled tree $(\tau_K, T_K)$ that represents the unwinding of $K$ from $w_0$. For a node $w \in W$, $d(w)$ is the number of $R$ successors – we denote by $succ_R(w) = \langle w_1, \ldots, w_{d(w)} \rangle$ the list of successors of $w$. We denote by $D = \{d(w) : w \in W\}$, the set of degrees for states for a Kripke structure $K$, and therefore $\tau_K$ is a $D$-tree. Furthermore, as long as $K$ is finite, then $D$ is finite. We define $\tau_K$ and $T_K$ as follows [Vardi, 2006; Visser et al., 1997]:

- $e \in \tau_K$ and $T_K(e) = w_0$.
- For $y \in \tau_K$ and $succ_R(T_K(y)) = \langle w_1, \ldots, w_k \rangle$, $\forall 1 \leq i \leq k$:
  - $y \cdot i \in \tau_K$. And,
  - $T_K(y \cdot i) = w_i$.

We denote by $(\tau_K, V \cdot T_K)$, the $2^{AP}$-labelled $D$-tree defined by $V \cdot T_K(y) = V(T_K(y))$, for $y \in \tau_K$.

Assume that $A_{D, \varphi}$ is the alternating tree automaton that accepts the $D$-trees which satisfy $\varphi$. It follows that a tree $(T_K, V \cdot T_K)$ is accepted by $A_{D, \varphi}$ iff $K \models \varphi$.

The product of $A_{D, \varphi}$ and $(T_K, V \cdot T_K)$ is a Büchi tree automaton over a 1-letter alphabet, which is empty iff $A_{D, \varphi}$ accepts $(T_K, V \cdot T_K)$. We denote the product automata as the weak alternating automaton $A_{K, \varphi} = (\langle a \rangle, D, W \times S, \delta, \langle w_0, \varphi \rangle, G)$, where [Vardi, 2006; Visser et al., 1997]:

- Assume $s \in S$, $w \in W$ and $succ_R(w) = \langle w_1, \ldots, w_k \rangle$ and $\rho(s, V(w), k) = \theta$. The transition relation for $A_{K, \varphi}$ is $\delta(\langle w, s \rangle, a, k) = \theta'$ where $\theta'$ is obtained by replacing $(c, s')$ in $\theta$ with $(c, \langle w_c, s' \rangle)$.
- $G = W \times F$

### 3.5.3 For CTLK

We now present a set-theoretic approach to the verification of multi-agent systems. By $[\varphi]$ we denote the set of all states in the structure satisfying the formula $\varphi$, that is, for a given interpreted system and its set of global states $G$:

$[\varphi] \equiv \{g \in G \mid g \models \varphi\}$
For the set-based algorithms for the temporal-only fragment, we refer the interested reader to [Huth and Ryan, 2004].

3.5.3.1 Model Checking Epistemic Subformulae

For an epistemic subformula, the set of states satisfying $K_i\varphi$ can be calculated using Algorithm 1 [Raimondi and Lomuscio, 2005], shown below. The inner call to the procedure SATCTLK is a recursive call to a high-level procedure that recursively calculates $\llbracket\varphi\rrbracket$ for all subformulae [Huth and Ryan, 2004]. We note that SAT is called by SATCTLK at the top level.

\begin{algorithm}
\caption{$\text{SAT}_K(\varphi : \text{FORMULA}, i : \text{AGENT}) : \text{set of STATE}$}
\begin{algorithmic}[1]
\STATE $X \leftarrow \text{SAT}_{\text{CTLK}}(\neg\varphi)$
\STATE $Y \leftarrow \text{pre}_K(X, i)$
\STATE return $G \setminus Y$
\end{algorithmic}
\end{algorithm}

Algorithm 1 works on the dual of $K_i$. It starts by finding all states that satisfy $\neg\varphi$ (i.e., the call to SATCTLK). It then constructs the epistemic pre-image of these states for the given agent (i.e., $\text{pre}_K(X, i)$ returns the set of states $Y = \{ g \mid \exists g' \in X, g \sim_i g' \}$ such that each state $g \in Y$ is epistemically related to at least one state in $X$). In Algorithm 1, the call to $\text{pre}_K$ calculates the set of all of the states that are epistemically related to a state where $\neg\varphi$ holds. It follows that if a state is epistemically related to a state satisfying $\neg\varphi$ then it is not possible for that state to satisfy $K_i\varphi$ (i.e., if there exists one related state satisfying $\neg\varphi$, it cannot be the case that all related states satisfy $\varphi$). By taking the set difference of this set with the set of all reachable states, we then obtain the set of states that are not epistemically related to a state where $\neg\varphi$ holds. As such, this final set is the states where $K_i\varphi$ holds (i.e., there is no state in $\neg Y$ that is epistemically related to a state where $\neg\varphi$ holds).

3.5.3.2 MCMAS and ISPL

MCMAS [Lomuscio et al., 2009] is a symbolic model checker for multi-agent systems based on BDDs. We note that BDDs are out of the scope of this thesis, and therefore are not covered. We refer the interested reader to [Huth and Ryan, 2004]. Importantly, MCMAS supports a syntax for specifying agents in an interpreted system. This language is called ISPL – or interpreted systems programming language.

We highlight the necessary parts of the ISPL syntax below.
Agent definition.

Each agent contains a definition of its local state $L_i$, actions $Act_i$, protocol function $E_i$ and evolution function $P_i$. Intuitively, they are mapped into the following sections of a per-agent definition:

- **Vars**
  The “vars” section is used to define the local states, or variables, of an agent. This entry takes a list of variable definitions; the types that are supported in ISPL are ranged integers, enumeration types and Booleans.

For example, the following defines the local state of an agent containing an enumerated variable state and a ranged integer counter.

```plaintext
1 Vars:
2   state: { wait, in, away };
3   counter: 0..10;
4 end Vars
```

- **Actions**
  The actions definition is simply a declaration of all possible actions that an agent can perform; given as list.

Below is the definition of an action set for an agent with four actions named accordingly:

```plaintext
1 Actions = { enter, leave, return, signal };
```

- **Protocol**
  The protocol function take a list of pre-condition guards, specified over the agent’s local state, such that if they hold, the corresponding actions can then be performed. Following the definition of the protocol function $P_i$, the protocol function in ISPL is non-deterministic, and therefore one guard can have many possible actions.

An example of a protocol block given the two previous definitions can be seen below:

```plaintext
1 Protocol:
2   state = wait : { signal, enter };
3   state = in : { leave };
4   state = away: { return };
5 end Protocol
```

- **Evolution**
  The evolution function takes a list of assignments, along with a set of “triggering conditions”. These triggering conditions can be specified over the current local state
of the agent, and any actions of any other agent in the system. Unlike the protocol function, the evolution function should be deterministic, and to that end, only one line in the protocol function can be enabled at any one instance (i.e., given a unique local state, and a global action, the next local state is uniquely defined by a single protocol line).

The following fragment illustrates how the previous parts can be used to define the evolution function:

```plaintext
Evolution:

state = in and counter = counter + 1 if
  counter < 10 and state = wait
  and Action = enter
  and Environment.Action = enter_1;
  ...

state = wait if state = wait
  and (!(Environment.Action = enter_1)
  or Action = signal);

end Evolution
```

**System-level definition.**

Given the definition of an agent as above, the definition of the higher-level interpreted system in ISPL uses the following syntax. We note that the specification of the whole interpreted system needs to support multiple agents, as well as initial states, the valuation function for propositional atoms and the formulae to be evaluated over the system.

- **Agent**
  The set of agents can be given as a list of agents as specified above. Each agent declaration delimited between `Agent name` and `end Agent`.

```plaintext
Agent Train1
  ...
end Agent
```

- **InitStates**
  The initial states of the model are specified as a Boolean expression over the local states of each agent in the system. Multiple initial states can be specified by using disjunction.

```plaintext
InitStates
  Train1.state = away;
end InitStates
```

- **Evaluation**
Similar to initial states, an evaluation is a Boolean expression stating at which states a propositional formula is satisfied in the model. Any states meeting the Boolean expression of a proposition satisfy that proposition.

```plaintext
Evaluation
  train1_in_tunnel if Train1.state = in;
end Evaluation
```

- **Formulae**
  The formulae block is used to specify the formulae to be evaluated over the reachable states of the model.

```plaintext
Formulae:
  AG(!train1_in_tunnel or AX(K(Train1, !train1_in_tunnel)));
end Formulae
```
Chapter 4
Automata-Theoretic Verification for Temporal-Epistemic Logic

In this chapter, we introduce a technique for automata-theoretic verification of multi-agent systems against specifications in a branching-time temporal-epistemic logic. As we will see, the material presented as part of this chapter will be used as an underpinning for the approach put forward in Chapter 5. We demonstrate the effectiveness of the technique itself in Section 6.2.2 as part of the implementation and evaluation chapter (Chapter 6).

4.1 Introduction

Recently, the automatic verification of multi-agent systems by model checking has been given considerable attention. The objective is to develop efficient methodologies to check automatically whether a multi-agent system of interest meets its specifications. Several techniques have been proposed over the past few years, ranging from bounded model checking [Huang et al., 2010], to symbolic model checking [Lomuscio et al., 2009; Gammie and van der Meyden, 2004], and partial order reduction [Lomuscio et al., 2010a].

While the approaches above have demonstrated their value, they are each insufficient to tackle the complexity of scenarios arising from the industry. It is therefore of utmost importance to explore novel methodologies that may help, either individually or in combination with existing techniques.

One method that has received little attention so far from the epistemic logic community is that of automata-based model checking. Yet, model checking via Büchi automata, originally explored in the seminal paper by Vardi et al. [Vardi and Wolper, 1986], is one of the leading approaches in verification of reactive systems and constitutes the basis of the well-known model checker spin [Holzmann, 2004]. The original work [Vardi and Wolper, 1986] covered linear-time only; this was extended to branching-time logic in [Kupferman et al., 2000], where the formalism of alternating tree automata (ATA) was applied.
In this chapter we explore the extent to which automata-based model checking can be adopted in an epistemic setting. Specifically, we work with CTLK as a specification language, and interpreted systems [Fagin et al., 1995] as the underlying semantics. As per Chapter 3, we adopt memory-less interpreted systems, or systems with no memory.

We first extend automata over infinite trees to general structures that can be used in the verification of temporal-epistemic properties of multi-agent systems (Section 4.2). We then give a uniform translation of CTLK specifications into these automata (Section 4.4). This enables us to define a suitable notion of an automata product and give an emptiness condition for satisfaction (Section 4.5). We conclude in Section 4.6.

4.2 From Interpreted Systems to Trees

We begin by presenting epistemic alternating tree automata, which are a generalisation of alternating tree automata (Section 3.4.2.3), with support for multi-modal logics (Section 3.1.3). Alternating automata were first introduced [Muller and Schupp, 1987] as a generalisation of non-deterministic automata and have been used [Kupferman et al., 2000] to define an automata-theoretic technique to model check branching-time logics (e.g., CTL).

Where the set $A$ of agents is finite, we define $A_t$ as the set $A \cup \{t\}$ containing the agents in $A$ plus the temporal index $t$. Given a MAS containing $m$ agents, it follows that $|A_t| = m+1$. In what follows, for some set $V$, we write $V^*$ for a sequence of elements in $V$.

**Definition 4.1.** An $A_t$-tree is a set $T \subseteq (\mathbb{N} \times A_t)^*$ such that if $x \cdot (c, j) \in T$ for $x \in (\mathbb{N} \times A_t)^*$ and $(c, j) \in \mathbb{N} \times A_t$ then

(i) $x \in T$
(ii) $x \cdot (c', j) \in T$, for all $0 \leq c' < c$

The sequences in $T$ are called nodes, the empty sequence $\varepsilon$ is the root of $T$. For $x \in T$, the nodes $x \cdot (c, j)$ are the $j$-successors of $x$. The number of $j$-successors of $x$ is called the $j$-degree of $x$ and is denoted by $d_j(x)$; the vector of all successor degrees of $x$ in $A_t$ is denoted by $\vec{d}(x) = \{d_t(x), d_1(x), \ldots, d_n(x)\}$. A node is classed as a leaf node if it has no successors.

**Definition 4.2.** A path in a tree $T$ is a non-empty set $\pi \subseteq T$ such that for every $x \in \pi$, either $x$ is a leaf or there exists a unique $(c, j) \in \mathbb{N} \times A_t$ such that $x \cdot (c, j) \in \pi$. A temporal path is a path where $j = t$; while an epistemic path in $\Gamma$ ($\Gamma \subseteq A$) is a path where $j \in \Gamma$.

For any path $\pi$, $\pi^n$ represents the $n$-th element in the path, for $n \in \mathbb{N}$. Given an alphabet $\Sigma$, a $\Sigma$-labelled tree is a pair $(T, V)$ where $T$ is a tree and $V : T \rightarrow \Sigma$ is a
define the tree $\langle \text{succeedors of } \text{s} \rangle$, we can construct the alternating automaton $A$ when $V$ can intuitively be seen as an assignment of propositional variables to nodes. Given an interpreted system $\mathcal{P} = (\mathcal{R}, s^0, I)$, we can define a tree $\langle T_\mathcal{P}, V \rangle$ with $\Sigma = \Pi$ such that:

1. $V(\varepsilon) = (\rho, 0)$, where $\rho(0) = s^0$
2. if $V(x) = (\rho, n)$ and $s_0, \ldots, s_k$ are all $s'$ such that $s' \sim_i \rho(n)$ for $i \in A$, then for $0 \leq c \leq k$, $V(x \cdot (c, i)) = (\rho_c, n_c)$ for some $\rho_c$ and $n_c$ such that $\rho_c(n_c) = s_c$
3. if $V(x) = (\rho, n)$ and $\rho_0, \ldots, \rho_k$ are all $\rho' \in \mathcal{R}$ such that $\rho'|_n = \rho|_n$, then $V(x \cdot (c, t)) = (\rho_c, n + 1)$ for $0 \leq c \leq k$

By Clause 2 of the above, we have that a node $x$ in the tree $\langle T_\mathcal{P}, V \rangle$ contains $i$ successors when $V(x) = \rho(n)$ has $i$ successors; and by Clause 3, $x$ has temporal successors when $V(x) = \rho(n)$ has temporal successors.

The tree $\langle T_\mathcal{P}, V \rangle$ is not unique given the IS $\mathcal{P}$, as by (2) above for each state $s_c \in \mathcal{S}$ there might be more than one point $(\rho_c, n_c)$ such that $\rho_c(n_c) = s_c$. However, we can recover uniqueness by defining a mapping $V_\mathcal{P} : T_\mathcal{P} \rightarrow \mathcal{S}$, i.e., where $V_\mathcal{P}$ is a mapping from nodes in the tree to states in $\mathcal{S}$, such that $V_\mathcal{P}(x) = s$ iff $V(x) = (\rho, n)$ and $\rho(n) = s$. It is straightforward to check that given an IS $\mathcal{P}$, the labelled tree $\langle T_\mathcal{P}, V_\mathcal{P} \rangle$ is indeed unique.

### 4.2.1 High-level Methodology

Given the IS $\mathcal{P}$ and a state $s \in \mathcal{S}$ in $\mathcal{P}$, we take $\text{succ}_R(s) \subseteq \mathcal{S}$ to be the set of temporal successors of $s$, and $\text{succ}_J(s) \subseteq \mathcal{S}$ to be the set of $j$ successors of $s$. Therefore, we can define the tree $\langle T_\mathcal{P}, V_\mathcal{P} \rangle$ as follows:

1. $\varepsilon \in T_\mathcal{P}$ where $V_\mathcal{P}(\varepsilon) = s^0$
2. For $y \in T_\mathcal{P}$ where $V_\mathcal{P}(y) = s$ and $\text{succ}_R(s) = \langle s_0, \ldots, s_n \rangle$, then for $0 \leq c \leq n$ we have $y \cdot (c, t) \in T_\mathcal{P}$ and $V_\mathcal{P}(y \cdot (c, t)) = s_i$
3. For $y \in T_\mathcal{P}$ where $V_\mathcal{P}(y) = s$ and $\text{succ}_J(s) = \langle s_0, \ldots, s_n \rangle$, then for $0 \leq c \leq n$ we have $y \cdot (c, j) \in T_\mathcal{P}$ and $V_\mathcal{P}(y \cdot (c, j)) = s_i$

Let $\varphi$ be a CTLK formula and $D \subseteq \mathbb{N}$ be a set of branching degrees. Suppose that we can construct the alternating automaton $A_{D, \varphi}$ such that $\mathcal{L}(A_{D, \varphi})$ accepts exactly those $D$-trees that satisfy the formula $\varphi$. Now consider the automaton $A_{\mathcal{P}, \varphi}$, which is the “product automaton” of the interpreted system $\mathcal{P}$ with the automaton $A_{D, \varphi}$, such that

$$\mathcal{L}(A_{\mathcal{P}, \varphi}) = \mathcal{L}(A_{D, \varphi}) \cap \{\langle T_\mathcal{P}, V_\mathcal{P} \rangle\}$$
That is, the language of $A_{P, \varphi}$ is either the single tree $\langle T_P, V_P \rangle$ and $P \models \varphi$, or the language is empty and $P \not\models \varphi$.

As such, given a formula $\varphi$ and an interpreted system $P$ with branching degrees in $D$, the automata-theoretic approach proceeds as follows:

1. Construct the alternating automaton $A_{D, \varphi}$
2. Construct the alternating automaton $A_{P, \varphi} = P \times A_{D, \varphi}$, representing the product of $P$ and $A_{D, \varphi}$
3. If $L(A_{P, \varphi}) \neq \emptyset$ return “true”, otherwise return “false”

In the remainder of this chapter, we show how to construct $A_{D, \psi}$ in Section 4.4 and how to construct $A_{P, \varphi}$ in Section 4.5. Finally, we prove correctness of the overall procedure in the proof of Theorem 4.2.

### 4.2.2 Example – Unwinding an Interpreted System

A simple interpreted system with two agents, 1 and 2, is shown in Figure 4.1. We use solid lines to represent temporal transitions and indexed, dashed lines to represent the epistemic indistinguishability relations for agents 1 and 2. Formally, this can be represented as the IS $P = \langle R, w_0, I \rangle$ such that:

- $w_0$ is the initial state
- the local state $l_1(w_0)$ of agent 1 in $w_0$ is the same as its local state $l_1(w_2)$ in $w_2$
- the set of runs $R = \{ \rho_1, \rho_2 \}$, where $\rho_1(0) = w_0$ and $\rho_1(n) = w_1$ for all $n \geq 1$, and $\rho_2(n) = w_2$ for all $n \geq 0$

Further, we assign the proposition $p$ to all states in $P$, i.e., for all $w$, $I(w) = \{ p \}$.

![Diagram](image_url)

**Fig. 4.1.** An example of an interpreted system

By way of an illustration, we see in Figure 4.2 that $d_i(w_0) = 2$ (the branching degree of the world $w_0$ in the epistemic direction $i$ is 2) and that $d_i(w_1) = 1$ (that $w_0$’s branching degree in the temporal direction is 1).

The set of global states $S = \{ w_0, w_1, w_2 \}$. Figure 4.2 shows the $S$-labelled tree $\langle T_P, V_P \rangle$ unwinding of $P$. Each node in the tree is of the form “$x, V(x)$”, representing the node of the tree $T$ (i.e., $x \in (\mathbb{N} \times A_i)^*$) along with the mapping in $V$ of that node to a state in $S$ (i.e., $V(x) \in S$). The interpreted system of Figure 4.1 is cyclic (i.e., it contains
reflexive loops), therefore its corresponding unwinding is an infinite tree. We only show a truncated part of the tree; the infinite part of the tree is represented with dotted lines.
Fig. 4.2. The epistemic alternating tree automaton "unwinding" of the interpreted system in Figure 4.1
4.2. From Interpreted Systems to Trees

4.2.3 CTLK over Trees

We can now present the satisfaction of a CTLK formula over a tree.

**Definition 4.3.** Given a tree \( \langle T, V \rangle \) with \( \Sigma = 2^{AP} \) (i.e., where \( V : T \rightarrow 2^{AP} \)), we can define a satisfaction relation \( \models \) for \( \phi \in \mathcal{L}_m \) and \( x \in T \) as follows:

\[
\begin{align*}
(T, x) \models p & \quad \text{iff } p \in V(x) \\
(T, x) \models \neg \psi & \quad \text{iff } (T, x) \not\models \psi \\
(T, x) \models \psi \rightarrow \psi' & \quad \text{iff } (T, x) \not\models \psi \text{ or } (T, x) \models \psi' \\
(T, x) \models AX \psi & \quad \text{iff for all temporal paths } \pi, \text{ for all } n \in \mathbb{N}, \text{ if } \pi^n = x \text{ then } (T, \pi^{n+1}) \models \psi \\
(T, x) \models A \psi U \psi' & \quad \text{iff for all temporal paths } \pi, \text{ for all } n \in \mathbb{N}, \text{ if } \pi^n = x \text{ then there is } k \geq n \\
& \quad \text{ such that } (T, \pi^k) \models \psi', \text{ and for all } k', n \leq k' < k \text{ implies } (T, \pi^{k'}) \models \psi \\
(T, x) \models E \psi U \psi' & \quad \text{iff for some temporal path } \pi, \text{ for some } n \in \mathbb{N}, \text{ if } \pi^n = x \text{ and there is } k \geq n \\
& \quad \text{ such that } (T, \pi^k) \models \psi', \text{ and for all } k', n \leq k' < k \text{ implies } (T, \pi^{k'}) \models \psi \\
(T, x) \models K \psi & \quad \text{iff for all } 0 \leq c < d_\pi(x), \text{ } (T, x \cdot (c,i)) \models \psi \\
(T, x) \models C_\pi \psi & \quad \text{iff for all epistemic paths } \pi \in \Gamma, \text{ for all } n \in \mathbb{N}, \\
& \quad \text{ if } \pi^n = x \text{ then for all } k \geq n, \text{ } (T, \pi^k) \models \psi
\end{align*}
\]

Given an IS \( \mathcal{P} \), we know that there is a tree \( \langle T, V \rangle \) with \( \Sigma = \Pi \). If we identify each \( (\rho, n) \in \Pi \) with \( \{ p \in AP \mid p \in I(\rho(n)) \} \subseteq 2^{AP} \), then \( \langle T, V \rangle \) is also a tree with \( \Sigma = 2^{AP} \), and we can prove the following.

**Lemma 4.1.** For every \( \phi \in \mathcal{L}_m \), for \( V(x) = (\rho, n) \),

\[
(T, x) \models \phi \iff (\mathcal{P}, \rho, n) \models \phi
\]

**Proof.** Straightforward by induction on the length of \( \phi \). \( \Box \)

From Lemma 4.1 it follows that we have \( (T, \varepsilon) \models \phi \) iff \( \phi \) is true in the IS \( \mathcal{P} \), i.e., the initial state of \( \mathcal{P} \) satisfies \( \phi \) (\( s_0 \models \phi \)).

Obviously, the correspondence between IS and trees is not one-to-one: there are many trees that cannot be represented as an IS. This is indeed the case every time a node has no successors (in IS we require both runs \( \rho \) and the indistinguishability relation \( \sim_\mathcal{I} \) to be serial). However, in the following discussion we will focus on the class \( T \) of trees \( \langle T, V_\mathcal{P} \rangle \) for some interpreted system \( \mathcal{P} \).
4.3 Weak Epistemic Alternating Automata

We now introduce epistemic alternating tree automata. Given a set $D \subseteq \mathbb{N}$, a $D$-tree is an $A_r$-tree in which all the nodes have degrees in $D$.

We remind the reader that $B^+(X)$ is the set of positive Boolean formulae over the set $X$, and including the constants $\text{true}$ and $\text{false}$. For instance, for $AP = \{p, q\}$, $B^+(AP)$ includes $p \land q$, $p \lor q$, $p \land \neg p$.

**Definition 4.4.** An *epistemic alternating tree automaton* (EATA) is a tuple $A = \langle \Sigma, D, Q, \delta, q_0, A_r, F \rangle$ such that:

(i) $\Sigma$ is a finite alphabet
(ii) $D \subseteq \mathbb{N}$ is a set of branching degrees
(iii) $A_r$ is a set of agent and temporal directions
(iv) $Q$ is a set of states
(v) $q_0 \in Q$ is the initial state
(vi) $F \subseteq Q$ is the set of accepting states
(vii) $\delta : Q \times \Sigma \times D^{|A_r|} \to B^+(\mathbb{N} \times A_r \times Q)$ is the transition function

We restrict $\delta$, such that if the atom $(c, j, q')$ appears in $\delta(q, \sigma, k_1, \ldots, k_m)$, then $0 \leq c < k_j$. That is, the transition relation for an epistemic or temporal modality is restricted to the maximal the branching degree for that direction.

When the automaton is in state $q$ and reads a node that is labelled by $\sigma$ and has $k_j$ $j$-successors, it applies the transition $\delta(q, \sigma, k_1, \ldots, k_m)$. In what follows we denote the tuple $(k, k_1, \ldots, k_m)$ as $\vec{k}$.

An EATA is a generalisation of an alternating tree automaton in that the successors of a node are indexed according to the elements in $A_r$. ATA [Kupferman et al., 2000] are a special case of EATA, in which $|A_r| = 1$, i.e., $A_r$ contains only the temporal index $t$.

A run of an EATA $A$ over a tree $\langle T, V \rangle$ is a tree $\langle T_r, r \rangle$ in which the root is labelled by $(e, q_0)$ and every other node is labelled by an element in $(\mathbb{N} \times A_r)^* \times Q$. Each node of $T_r$ corresponds to a node of $T$. However, each node of $T$ can correspond to many nodes of $T_r$.

**Definition 4.5.** A run $\langle T_r, r \rangle$ is a $\Sigma_r$-labelled tree where $\Sigma_r = (\mathbb{N} \times A_r)^* \times Q$ and $\langle T_r, r \rangle$ satisfies the following:

(i) $e \in T_r$ and $r(e) = (e, q_0)$
(ii) Let $y \in T_r$ with $r(y) = (x, q)$. If $\delta(q, V(x), d(x)) = \theta \in B^+(\mathbb{N} \times A_r \times Q)$ then there are (possibly empty) sets $S_j = \{(c_0, j, q_0), \ldots, (c_n, j, q_n)\} \subseteq \{0, \ldots, d_j(x) - 1\} \times \{j\} \times Q$ such that the following hold:

(a) the assignment that assigns $\text{true}$ to all the atoms in $\bigcup_{i \in A_r} S_j$ satisfies $\theta$
(b) for $0 \leq i \leq n$ we have $y \cdot (i, j) \in T_r$ and $r(y \cdot (i, j)) = (x \cdot (c_i, j), q_i)$
Note that if, for some \( y \), the transition function \( \delta \) has value \text{true}, then \( y \) need not have successors. Also, \( \delta \) can never have the value \text{false} in a run. We use the same term \text{run} both for IS and for EATA to be consistent with the literature [Fagin et al., 1995; Kupferman et al., 2000]; the context will disambiguate.

A run \( \langle T_r, r \rangle \) is accepting if all its infinite paths satisfy the acceptance condition. In this chapter, we consider a Büchi acceptance condition. As per Chapter 3, given a run \( \langle T_r, r \rangle \) and an infinite path \( \pi \subseteq T_r \), let \( \inf(\pi) \subseteq Q \) be the set of \( q \in Q \) such that there are infinitely many \( y \in \pi \) for which \( r(y) \in (\mathbb{N} \times \mathcal{A}_t)^* \times \{ q \} \), that is, \( \inf(\pi) \) contains exactly all the states that appear infinitely often in \( \pi \). The acceptance condition is defined as follows:

- A path \( \pi \) satisfies a Büchi acceptance condition \( F \subseteq Q \) if and only if \( \inf(\pi) \cap F \neq \emptyset \)

An automaton accepts a tree if and only if there exists a run that accepts it. We denote by \( L(A) \) the set of all \( \Sigma \)-labelled trees that \( A \) accepts.

**Definition 4.6.** An epistemic alternating word automaton is an epistemic alternating tree automaton over infinite words, such that \( D = \{ 1 \} \) and \( |\mathcal{A}_t| = 1 \). Formally, we define an EATA over infinite words as \( A = (\Sigma, Q, \delta, q_0, F) \) where \( \delta : Q \times \Sigma \to B^+(Q) \).

As alternating word automata have degree 1, we omit the parameter \( D \) from the definition.

The model checking procedure for CTLK considers weak epistemic alternating automata (WEAAs), an extension of weak alternating automata as first introduced [Muller et al., 1986].

**Definition 4.7.** A weak epistemic alternating automaton (WEAA) is an EATA such that:

- there is a partition of \( Q \) into disjoint sets \( Q_1, \ldots, Q_n \) such that for \( 1 \leq j \leq n \), either \( Q_j \subseteq F \), and \( Q_j \) is an accepting set, or \( Q_j \cap F = \emptyset \), and \( Q_j \) is a rejecting set
- there is a partial order \( \leq \) on the collection \( Q_j \) such that for every \( q \in Q_j \), and \( q' \in Q_j \) occurring in \( \delta(q, \sigma, \vec{k}) \) for some \( \sigma \in \Sigma, \vec{k} \in D^{\mathcal{A}_t} \), we have \( Q_j \leq Q_i \)

Thus, transitions from a state in \( Q_i \) lead to states in either the same \( Q_i \) or a lower one. It follows that every infinite path of a run of a WEAA ultimately gets trapped within some \( Q_i \). The path then satisfies the acceptance condition if and only if \( Q_i \) is an accepting set. We call the partition of \( Q \) the weakness partition, and we call the partial order over the sets of the weakness partition the weakness order. In the following section, we show how the partial order is related to the closure of a formula.

**4.4 From CTLK to Automata**

In this section we provide the construction of a weak alternating automaton \( A_{D, \psi} \) that accepts all \( D \)-trees in \( T \) satisfying a given CTLK formula \( \psi \in L_{\text{m}} \). In the next section
the automaton $A_{D, P}$ will be used to construct the product word automaton $A_{P, \psi}$ for the formula $\psi$ and a given IS $P$. We will then prove that the language $L(A_{P, \psi})$ is non-empty iff the tree $\langle T_P, V_P \rangle$ is accepted by $A_{D, P}$, i.e., iff $\psi$ is true in $P$. By extending Theorems 3.1 and 4.7 in [Kupferman et al., 2000] we can show that all these steps can be performed in linear-time in the size of $\phi$ and $P$.

First, we remark that by using de Morgan’s laws and the definitions of operators $\overline{U}$, $\overline{K}_i$ and $\overline{C}_\Gamma$, we can draw the negation inwards, such that it applies only to propositional variables. That is, given a formula $\phi$ in CTLK, it is possible to rewrite the formula in negation normal form, such that negation is only applied to atomic propositions.

We remind the reader that the closure $cl(\psi)$ of a formula $\psi \in L_m$ as follows:

- $\psi \in cl(\psi)$
- Let $\square$ be any of operators $\neg$, $AX$, $EX$, $K_i$, $K_i$, $C_\Gamma$ or $C_\Gamma$. If $\square \phi \in cl(\psi)$ then $\phi \in cl(\psi)$
- Let $\uparrow$ be any of operators $\rightarrow$, $AU$, $EU$, $A\Upsilon$ or $E\Upsilon$. If $\phi \uparrow \phi' \in cl(\psi)$ then $\phi, \phi' \in cl(\psi)$

**Theorem 4.1.** Given a CTLK formula $\psi \in L_m$ and a set $D \subset \mathbb{N}$, we can construct in linear-time a WEA $A_{D, \psi} = \langle 2^{AP}, D, cl(\psi), \delta, \psi, A_t, F \rangle$ such that the $D$-tree $\langle T_P, V_P \rangle$ is in $L(A_{D, \psi})$ iff $\psi$ is true in $P$.

**Proof.** The WEA $A_{D, \psi} = \langle 2^{AP}, D, cl(\psi), \delta, \psi, A_t, F \rangle$ is such that $\Sigma = 2^{AP}$, $Q = cl(\psi)$ and $q_0 = \psi$. Further, the set $F$ of accepting states consists of all the $U$, $K_i$ and $C_\Gamma$-formulae in $cl(\psi)$. For $\sigma \in 2^{AP}$ and $\hat{k} \in D|A_t|$ the transition function $\delta$ is defined as in Table 4.1.
Table 4.1. The transition function \( \delta \) in the automaton \( A_{\forall \psi} \)

\[
\delta \left( p, \sigma, \bar{k} \right) = \begin{cases} 
true & \text{if } p \in \sigma \\
false & \text{if } p \notin \sigma 
\end{cases}
\]

\[
\delta \left( \neg p, \sigma, \bar{k} \right) = \begin{cases} 
true & \text{if } p \notin \sigma \\
false & \text{if } p \in \sigma 
\end{cases}
\]

\[
\delta \left( \phi_1 \ast \phi_2, \sigma, \bar{k} \right) = \delta \left( \phi_1, \sigma, \bar{k} \right) \ast \delta \left( \phi_2, \sigma, \bar{k} \right), \text{ for } \ast \in \{ \land, \lor \}
\]

\[
\delta \left( \bigwedge_{c=0}^{k_i-1} (c, t, \phi) \right) = \bigwedge_{c=0}^{k_i-1} (c, i, \phi) \\
\delta \left( \bigvee_{c=0}^{k_i-1} (c, t, \phi) \right) = \bigvee_{c=0}^{k_i-1} (c, i, \phi)
\]

\[
\delta \left( \phi_1 U \phi_2, \sigma, \bar{k} \right) = \delta \left( \phi_2, \sigma, \bar{k} \right) \lor \left( \delta \left( \phi_1, \sigma, \bar{k} \right) \land \bigwedge_{c=0}^{k_i-1} (c, t, \phi_1 U \phi_2) \right)
\]

\[
\delta \left( \phi_1 U^\perp \phi_2, \sigma, \bar{k} \right) = \delta \left( \phi_2, \sigma, \bar{k} \right) \land \left( \delta \left( \phi_1, \sigma, \bar{k} \right) \lor \bigwedge_{c=0}^{k_i-1} (c, t, \phi_1 U^\perp \phi_2) \right)
\]

\[
\delta \left( \phi_1 U \phi_2, \sigma, \bar{k} \right) = \delta \left( \phi_2, \sigma, \bar{k} \right) \lor \left( \delta \left( \phi_1, \sigma, \bar{k} \right) \land \bigwedge_{c=0}^{k_i-1} (c, t, (E \phi_1 U \phi_2)) \right)
\]

\[
\delta \left( \phi_1 U^\perp \phi_2, \sigma, \bar{k} \right) = \delta \left( \phi_2, \sigma, \bar{k} \right) \land \left( \delta \left( \phi_1, \sigma, \bar{k} \right) \lor \bigwedge_{c=0}^{k_i-1} (c, t, (E \phi_1 U^\perp \phi_2)) \right)
\]

\[
\delta \left( \bigwedge_{i \in \Gamma} (c, i, \phi_1 U \phi_2) \right) = \bigwedge_{i \in \Gamma} (c, i, \phi_1 U \phi_2)
\]

\[
\delta \left( \bigvee_{i \in \Gamma} (c, i, \phi_1 U^\perp \phi_2) \right) = \bigvee_{i \in \Gamma} (c, i, \phi_1 U^\perp \phi_2)
\]
Each formula \( \phi \in cl(\psi) \) constitutes a (singleton) set \( \{ \phi \} \) in the weakness partition. The weakness order is defined by \( \{ \phi_1 \} \preceq \{ \phi_2 \} \) iff \( \phi_1 \in cl(\phi_2) \). Since each transition of the automaton from a state \( \phi \) leads to states in \( cl(\phi) \), the weaknesses conditions hold. For example, for the formula \( \phi = \mathcal{A}\phi_1 \mathcal{U}\phi_2 \), we note that \( \phi \) occurs both as part of the transition function \( \delta \) from \( \phi \) and in \( cl(\phi) \) – this therefore preserves the weakness condition.

We now prove the correctness of this construction. By Lemma 4.1 the formula \( \psi \) is true in \( P \) iff \( \langle T_p, e \rangle \models \psi \). So it is left to prove that the \( D \)-tree \( \langle T_p, V_P \rangle \) is in \( L(\mathcal{A}_{D,\psi}) \) iff \( \langle T_p, e \rangle \models \psi \). We first prove that \( \mathcal{A}_{D,\psi} \) is sound, that is, given an accepting run \( \langle T_r, r \rangle \) of \( \mathcal{A}_{D,\psi} \) over the tree \( \langle T_p, V_P \rangle \), we show that for every \( y \in T_r \) such that \( r(y) = (x, \phi) \) we have that \( \langle T_p, x \rangle \models \phi \). Thus, we have \( \langle T_p, e \rangle \models \psi \). The proof is by induction on the structure of \( \phi \).

If \( \phi \) is an atomic proposition and \( r(y) = (x, p) \) then \( \delta(p, V_P(x), \bar{d}(x)) = \text{true} \) iff \( p \in V_P(x) \), i.e., iff \( \langle T_p, x \rangle \models \phi \). The cases where \( \phi \) is \( \phi_1 \land \phi_2 \), \( \phi_1 \lor \phi_2 \), \( AX\phi_1 \), or \( EX\phi_1 \) follow easily, by the induction hypothesis, from the definition of \( \delta \).

Consider now the case of \( \phi \) equal to \( \phi_1 \mathcal{U}\phi_2 \) (resp. \( E\phi_1 \mathcal{U}\phi_2 \)). As \( \langle T_r, r \rangle \) is an accepting run, it visits the state \( \phi \) finitely often only. Since \( \mathcal{A}_{D,\psi} \) keeps inheriting \( \phi \) as long as \( \phi_2 \) is not satisfied, then it is guaranteed, by the definition of \( \delta \) and the induction hypothesis, that along all paths (resp. some path) \( \phi_2 \) eventually holds and \( \phi_1 \) holds until then. Finally, consider the case of \( \phi \) equal to \( \phi_1 \mathcal{U}\phi_2 \) or \( E\phi_1 \mathcal{U}\phi_2 \). By the definition of \( \delta \) and the induction hypothesis, either \( \phi_2 \) always holds or until both \( \phi_2 \) and \( \phi_1 \) hold.

If \( \phi = K_i \phi_1 \) and \( r(y) = (x, K_i \phi_1) \) then

\[
\delta(K_i \phi_1, V_P(x), \bar{d}(x)) = \bigwedge_{c=0}^{d_i(x)-1} (c, i, \phi_1) \land \bigwedge_{c=0}^{d_i(x)-1} (c, i, K_i \phi_1)
\]

Thus, for all \( 0 \leq k < d_i(x) \), and \( c = k \) we have \( r(y \cdot (k, i)) = (x \cdot (c, i), \phi_1) \). By induction hypothesis, \( \langle T_p, x \cdot (c, i) \rangle \models \phi_1 \) for \( 0 \leq c < d_i(x) \), and therefore \( \langle T_p, x \rangle \models K_i \phi_1 \). The case of \( \phi = \mathcal{T}_i \phi_1 \) is similar.

If \( \phi \) is equal to \( C_T \phi \) and \( r(y) = (x, C_T \phi) \) then

\[
\delta(C_T \phi, V_P(x), \bar{d}(x)) = \delta(\phi, V_P(x), \bar{d}(x)) \land \bigwedge_{i \in I} \left( \bigwedge_{c=0}^{k_i-1} (c, i, C_T \phi) \right)
\]

Thus, if \( \pi \) is an epistemic path for \( I \) such that \( \pi^n = x \), then by induction hypothesis for all \( k \geq n \), we have that \( \langle T_p, \pi^k \rangle \models \phi \). Therefore, \( \langle T_p, x \rangle \models C_T \phi \). The case of \( \phi = \mathcal{C}_T \phi_1 \) is similar.

We now prove that \( \mathcal{A}_{D,\psi} \) is complete, that is, if \( \langle T_p, V_P \rangle \) is a \( D \)-tree such that \( \langle T_p, e \rangle \models \psi \), then \( \mathcal{A}_{D,\psi} \) accepts \( \langle T_p, V_P \rangle \). In fact, we show that there exists an accepting run \( \langle T_r, r \rangle \) of \( \mathcal{A}_{D,\psi} \) over \( \langle T_p, V_P \rangle \) defined as follows: the run starts at the initial state, so \( e \in T_r \) and \( r(e) = (e, \psi) \), and it proceeds maintaining the invariant that for every \( y \in T_r \), if \( r(y) = (x, \phi) \) then \( \langle T_p, x \rangle \models \phi \). Since \( \langle T_p, e \rangle \models \psi \), the invariant holds for \( y = e \). Also,
by the semantics of CTLK and the definition of $\delta$, the run can always proceed such that all the successors $y \cdot (k, j)$ of a node $y$ that satisfy the invariant have $r(y \cdot (k, j)) = (x', \phi')$ with $\langle T_P, x' \rangle \models \phi'$. Finally, the run always tries to satisfy eventualities of $U$-formulae. Thus, whenever $\phi$ is of the form $A\phi_1 U \phi_2$ or $E\phi_1 U \phi_2$ and $\langle T_P, x \rangle \models \phi_2$, it proceeds according to $\delta(\phi_2, V_P(x), d(x))$. It is easy to see that all the paths in such $\langle T_r, r \rangle$ are either finite or reach a state associated with a $U$, $K_i$- or $C_G$-formula and remain thereafter. Thus, $\langle T_r, r \rangle$ is accepting.

Finally, it is easy to check that the construction of the automaton $A_{D, \psi}$ can be performed in linear-time in the size of $\psi$. □

Note that the particular way the $\delta$ function is defined for $K_i \phi$, and the loss of symmetry with $K_i \phi$, depends on the fact that the epistemic indistinguishability relation is an equivalence relation. It is straightforward to define the $\delta$ function for the epistemic $E_{I_1}$ modality in terms of the $\delta$ function for $K_i$ for $i \in \Gamma$.

### 4.4.1 Example – $L_m$ to WEAA

We now demonstrate the technique by showing the translation of a $L_m$ formula to a weak epistemic alternating automaton. We take the formula

$$\phi = AG(C_{\{1,2\}} K_2 p)$$

To translate $\phi$ into a WEAA, we require the formula in negation-normal form with all abbreviations expanded; so we use

$$\varphi = A(\text{false} U C_{\{1,2\}} K_2 p)$$

The closure of $\varphi$ is

$$cl(\varphi) = \{\varphi, C_{\{1,2\}} K_2 p, K_2 p, p\}$$

which is the set $Q$ of states in $A_{D, \varphi}$. The accepting states $F$ are $\{\varphi, C_{\{1,2\}} K_2 p, K_2 p\}$. The alphabet of $A_{D, \varphi}$ has only the letter $p$; therefore the transitions are over $2^{|p|}$ (i.e., $\Sigma = \{\emptyset, \{p\}\}$).

We formally define

$$A_{D, \varphi} = (\{\emptyset, \{p\}\}, D, \{\varphi, C_{\{1,2\}} K_2 p, K_2 p, p\}, \delta, \varphi, \{1, 2\}, \{\varphi, C_{\{1,2\}} K_2 p, K_2 p\})$$

where the transition relation $\delta$ is as follows:
We now introduce the notion of the follows:

\[\langle D, \varphi \rangle, \langle \varphi, P \rangle, \langle \varphi, \mathcal{K} \rangle, \langle \mathcal{K}, K \rangle, \langle K, \delta \rangle, \langle \delta, q \rangle, \langle q, \emptyset \rangle, \langle \emptyset, k \rangle\]

\[
\begin{array}{|c|c|}
\hline
q & \delta(q, \{p\}, \delta) \\
\hline
\varphi & \bigwedge_{c=0}^{k_3-1} (c, i, \varphi) \land \bigwedge_{c=0}^{k_2-1} (c, 2, p) \land \bigwedge_{i \in \{1, 2\}} \bigwedge_{c=0}^{k_1-1} (c, i, C_{\{1, 2\}}K_2p) \\
\mathcal{K}_2p & \bigwedge_{c=0}^{k_2-1} (c, 2, p) \land \bigwedge_{c=0}^{k_2-1} (c, 2, K_2p) \\
\mathcal{K}_2p & \bigwedge_{c=0}^{k_2-1} (c, 2, p) \\
p & true & false \\
\hline
\end{array}
\]

In the state \(\varphi\) the automaton expects that \(\varphi\) recursively holds in all temporal successors and that both \(K_2p\) and \(C_{\{1, 2\}}K_2p\) hold in all epistemically related states. This is highlighted in Rows 2 and 3 of the transition relation above.

### 4.5 Constructing the Product Automaton

We now introduce the notion of the product automaton for a given CTLK formula and an interpreted system \(\mathcal{P}\). We remind the reader that the product automaton \(A_{\mathcal{P}, \varphi}\) should accept the language \(L(A_{\mathcal{P}, \varphi} \cap \{(T_P, V_P)\})\) and therefore should be non-empty if \(\mathcal{P}\) satisfies \(\varphi\).

Let \(A_{\mathcal{D}, \varphi} = (2^p, D, Q, \mathcal{L}, q_0, A, F)\) be an epistemic alternating tree automaton that accepts all the \(D\)-trees in \(T\) that satisfy \(\varphi\), as constructed in the previous section. Let \(\mathcal{P} = (\mathcal{R}, s^0, I)\) be an interpreted system such that the degrees of \((T_P, V_P)\) are in \(D\). We introduce the weak epistemic alternating word automaton \(A_{\mathcal{P}, \varphi} = (\{a\}, \Pi \times Q, \delta, ((\rho, 0), q_0), F)\) such that \(\rho(0) = s_0\) and \(\delta, F\) are defined as follows:

- **Epistemic:** Let \(q \in Q_\varphi, (\rho, n) \in \Pi, \{s' \in S | s' \sim_i \rho(n)\} = \{s_0,i, \ldots, s_{d_i(\rho(n))−1,i}\}\), where \(s_0,i, \ldots, s_{d_i(\rho(n))−1,i}\) is set the \(\text{succ}_{c_i}(\rho(n))\) is the set of \(i\) successors for \(\rho(n)\) and \(\delta_\varphi(q, I(\rho(n)), \tilde{d}(\rho(n))) = \theta\). Then \(\delta_\varphi((\rho, n), q, a) = \theta', \) where \(\theta'\) is obtained from \(\theta\) by replacing each atom \((c_{i,j}, q_j)\) in \(\theta\) by the atom \((\rho_{c_{i,j}, i, n_{c_{i,j}}}, q_j)\) for some point \((\rho_{c_{i,j}, i, n_{c_{i,j}}}) = s_{c_{i,j}}\).

- **Temporal:** Let \(q \in Q_\varphi, (\rho, n) \in \Pi, \{\rho' \in R | \rho'\mid_n = \rho\mid_n\} = \{\rho_0,i, \ldots, \rho_{d_\varphi(\rho(n))−1,i}\}\), where \(\rho_0,i, \ldots, \rho_{d_\varphi(\rho(n))−1,i}\) is the set \(\text{succ}_{\varphi}(\rho(n))\) of \(i\) successors for \(\rho(n)\) and let \(\delta_\varphi(q, I(\rho(n)), \tilde{d}(\rho(n))) = \theta\). Then \(\delta_\varphi((\rho, n), q, a) = \theta', \) where \(\theta'\) is obtained from \(\theta\) by replacing each atom \((c_{i,j}, q_j)\) in \(\theta\) by the atom \((\rho_{c_{i,j}, i, n+1}, q_j)\).

The acceptance condition \(F\) is defined according to the acceptance condition \(F_\varphi\) of \(A_{\mathcal{D}, \varphi}\). If \(F_\varphi \subseteq Q_\varphi\) is a Büchi condition, then \(F = \Pi \times F_\varphi\) is also a Büchi condition.
It is easy to see that if $A_{D,\psi}$ is a WEAA with a weakness partition $\{Q_1, \ldots, Q_n\}$, then so is $A_{P,\psi}$ with a partition $\{\Pi \times Q_1, \ldots, \Pi \times Q_n\}$.

We remark that the word automaton $A_{P,\psi}$ defined above is not unique given $P$. However, we can recover uniqueness by considering states in $S$ rather than points in $\Pi$. Thus, the product automaton of $A_{D,\psi}$ and $P$ is defined as the weak epistemic alternating word automaton $A_{P,\psi} = \langle \{a\}, S \times Q_\psi, \delta, (s^0, q_0), F \rangle$ where $F = S \times F_\psi$ and if $\delta((\rho, n), q, a) = \theta$ the $\delta((\rho(n), q), a) = \theta'$, where $\theta'$ is obtained from $\theta$ by replacing each atom $(\rho(n), q)$ in $\theta$ by $(\rho(n), q)$.

**Observation 1.** Tree automata to word automata.

We expand on the notion of translating the tree automaton to a word automaton. It follows that transforming every node in $A_{D,\psi}$

$$((c, j), q) \Rightarrow (s_j, q)$$

where $s_j$ is the $c$-th $j$-successor ($j \in A_i$) in $P$, converts the alternating tree automaton $A_{D,\psi}$ for the formula into the alternating word automaton $A_{P,\psi}$. This is because $A_{P,\psi}$ no longer has a set of degrees $D$, and simply has a fixed set of successors.

To this end, each disjunction and conjunction in the formula over nodes in the tree (i.e., over each $c$ in $d_j$, $j \in A_i$), becomes a disjunction/conjunction of successors in the word automaton.

The automaton $A_{P,\psi}$ is also over one letter, i.e., $\Sigma = \{a\}$, as each node in $A_{P,\psi}$ exactly encapsulates a given state in the tree $\langle P \cup V, P\rangle$ and therefore also in the interpreted system $P$. Consequently, the automaton $A_{P,\psi}$ does not have to “read” a letter from $P$ – this information is already captured by the automaton by construction.

**Theorem 4.2.** $L(A_{P,\psi})$ is nonempty iff $\psi$ is true in $P$.

**Proof.** We show that $L(A_{P,\psi})$ is nonempty if and only if $A_{D,\psi}$ accepts the tree $\langle P \cup V, P\rangle$ built from the IS $P$ as shown in Section 4.4. Since $A_{D,\psi}$ accepts exactly all the $D$-trees in $T$ that satisfy $\psi$, and since all the degrees of $P$ are in $D$, the latter holds if and only if $\psi$ is true in $P$. Given an accepting run of $A_{D,\psi}$ over $\langle P \cup V, P\rangle$, we construct an accepting run of $A_{P,\psi}$. Also, given an accepting run of $A_{P,\psi}$, we construct an accepting run of $A_{D,\psi}$ over $\langle T \cup P, V \rangle$.

Assume first that $A_{D,\psi}$ accepts $\langle P \cup V, P\rangle$. Thus, there exists an accepting run $\langle T_r, r \rangle$ of $A_{D,\psi}$ over $\langle P \cup V, P\rangle$. Recall that $T_r$ is labelled with $(\mathbb{N} \times A_i)^* \times Q_\psi$. A node $y \in T_r$ with $r(y) = (x, q)$ corresponds to a copy of $A_{D,\psi}$ that is in the state $q$ and reads the tree obtained by unwinding $P$ from $V_P(x)$. Consider the tree $\langle T_r, r' \rangle$ where $T_r$ is the tree obtained from $T_r$ by the function $f$ as follows. Suppose that $\delta_\psi(q, V_P(x), \bar{d}(x)) = \theta$ and there are (possibly empty) sets $S_j = \{(c_0, j, q_0), \ldots, (c_{n_j}, j, q_{n_j})\} \subseteq \{0, \ldots, d_j(x) - 1\} \times \{J\} \times Q$ such that $\bigcup_{j \in A_i} S_j$ satisfies $\theta$, and for $0 \leq i < n_j$, we have $y \cdot (i, j) \in T_r$ and $r(y \cdot (i, j)) = (x \cdot (c_j, j), q_t)$. Then,

- $f(\varepsilon) = \varepsilon$
As in the previous direction, we can check that
\[
 f(y \cdot (i, j)) = f(y) \cdot \left( \sum_{f < j} n_f + i \right)
\]
The tree $T_r$ is labelled with $0^* \times S \times Q_r$ and for every $y \in T_r$ with $r(y) = (x, q)$, we have
\[
 r'(f(y)) = (0^{|x|}, V_P(x), q).
\] We show that $(T_r, r')$ is an accepting run of $A_{P, V}$. In fact, since $F = S \times F_V$, we only need to show that $(T_r, r')$ is a run of $A_{P, V}$; this follows from the definition of $\delta$. Acceptance follows from the fact that $(T_r, r)$ is accepting.

Assume now that $A_{P, V}$ accepts $a^\omega$. Thus, there exists an accepting run $(T_r, r)$ of $A_{P, V}$. Recall that $T_r$ is labelled with $0^* \times S \times Q_r$. Consider the tree $(T_r', r')$ labelled with $(\mathbb{N} \times A_i)^* \times Q_r$, where $T_r'$ and $r'$ are obtained from $T_r$ and $r$ by means of a function $g : T_r \rightarrow T_r'$ as follows:

- $g(\varepsilon) = \varepsilon$ and $r'(\varepsilon) = (\varepsilon, q_0)$
- if $y \cdot c \in T_r$, $r'(g(y)) \in \{x\} \times Q_r$, $r(y \cdot c) = (0^{|x|+1}, s, q)$ and $i, j$ are such that $V_P(x \cdot (i, j)) = s$, then $g(y \cdot c) = g(y) \cdot (i, j)$ and $r'(g(y \cdot c)) = (x \cdot (i, j), q)$

As in the previous direction, we can check that $(T_r', r')$ is an accepting run of $A_{P, V}$ over $(T_P, V_P)$.

By Theorem 4.7 in [Kupferman et al., 2000] we know that the 1-letter non-emptiness problem for weak alternating automata is decidable in linear-time. This concludes the automata-theoretic model checking procedure for CTLK.

Observation 2. In the proof of Theorem 4.2, the function $f$ transforms a node as follows:
\[
 f(y \cdot (i, j)) = f(y) \cdot \left( \sum_{f < j} n_f + i \right)
\]
This takes the sum of all indices prior to the current index, such that we have a consistent number for all indices for all directions in $A_i$. Furthermore, we have that $n_j = |d_j|$, i.e., $n_j$ is the degree for the direction $j$.

As $T_r$ is a word automaton, this also means that each node is labelled with $0^n$, where $n$ is the length of the corresponding node in the tree automaton $T_r$. When we have $n = |x|$, this represents the depth in the tree that the current node sits at (e.g., $x = (1, i) \cdot (0, t)$, $|x| = 2$).

### 4.5.1 Example – The Product Automaton $A_{P, \phi}$

Using the approach developed so far, it can be shown that the language of the product automaton obtained from the composition of $A_{P, \phi}$ and the tree unwinding of the IS from Figure 4.1 is non-empty.

Figure 4.3 shows a sub-tree of the full product automaton. This sub-tree shows the accepting sub-tree for the formula

\[
 AX \left( A \left[ \text{false} U C_{\{1,2\}} \left( K_2 (p) \right) \right] \right)
\]
starting at the world $w_1$ from Figure 4.1. While this sub-tree contains 13 nodes, the full product automaton contains 31 nodes and has been omitted for brevity.

\[
\begin{align*}
&\text{Fig. 4.3. A sub-tree of the automaton } A_{\mathcal{P}, \varphi} \\
\end{align*}
\]

Branches that reach a recurring node (e.g., the node “$w_1, K_2(p)$” that occur twice in Figure 4.3) are underlined. The nodes that appear between the first and second occurrence of a recurring node make a path in the tree that is subsequently checked against the acceptance condition. For example, the set of states that occur infinitely often between the first and second occurrence of the node “$w_1, K_2(p)$” are “$w_1, K_2(p)$” and “$w_1, p \land K_2(p)$”. This path is accepted as the intersection of this path and accepting states $F = \{ \varphi, C_{\{1,2\}}K_2p, K_2p \}$ of the formula $\varphi$ is non-empty.

Furthermore, by the weakness partition of $A_{\mathcal{P}, \varphi}$, we have that $w_1, K_2(p) \in \mathcal{S} \times F$ and therefore it follows that the node $w_1, K_2(p)$ is accepting.

4.6 Concluding Remarks

In this chapter we have presented an automata-theoretic methodology for verifying multi-agent systems against specifications in temporal-epistemic logic. Although automata
To achieve this, we extended the relevant notions of automata and provided a sound translation from the logic CTLK into automata, thereby providing a model checking algorithm.

The translation from CTLK to alternating automata as presented in this chapter is an extension of the alternating automata framework as proposed by Kupferman et al. [Kupferman et al., 2000]. The extension here was in supporting multiple directions and therefore being able to support multi-modal logics (such as CTLK).

The automata-theoretic approach proposed here is not a golden bullet in the verification of multi-agent systems. For the so-called existential fragment of CTLK [Penczek and Lomuscio, 2003a], the approach presented could provide a fruitful avenue for on-the-fly model checking [Gerth et al., 1996]. Under such a framework, only the parts of the system necessary to demonstrate the satisfaction of the formula are explored; space savings could be made by avoiding parts of the model unnecessary for demonstrating the validity of the formula.

In this chapter, we presented the first approach for model checking multi-agent systems against branching-time temporal-epistemic properties using automata. Unlike the well-established approaches for the linear-time temporal-only formulae, our approach uses tree automata rather than word automata. We demonstrated how it is possible to unwind an interpreted system into a tree, such that the language of this tree is accepted by a tree automata for the formula iff the original interpreted system satisfies the formula. Later, in Chapter 6, we demonstrate an implementation of this technique on standard examples from the literature.

In the next chapter (Chapter 5) we suitably extend the framework presented so far to support a method of modular reasoning for multi-agent systems.
Chapter 5
Compositional Verification for Multi-Agent Systems

In this chapter, we demonstrate that new and potentially efficient verification approaches can have a fundamental underpinning in the automata-theoretic approach that Chapter 4 introduced. We introduce a method for verifying an agent in isolation against an agent-local formula, without considering the composition of the agent with the rest of the system. By encoding information about the rest of the system into a linear-time formula, the automata-theoretic approach can then disregard paths in the abstract composition using an extended acceptance condition. Later, in Chapter 6, we present an implementation of the technique, and compare its efficiency against the state-of-the-art in MAS-based model checking (\texttt{mcmas-1.0} [Lomuscio et al., 2009], Section 3.5.3.2) and an implementation of the approach from Chapter 4.

5.1 Introduction

The family of techniques classed as “compositional reasoning” allow for the verification of a system in a “divide-and-conquer” manner, avoiding the construction of the system’s full composition. One such compositional technique is assume-guarantee reasoning [de Roever et al., 1998]. With this paradigm, components are verified against a specification (the “guarantee”) under the premise that the rest of the system satisfies another property (the “assumption”). As of yet, little work has been done to apply either compositional methods or assume-guarantee reasoning to the verification of multi-agent systems.

Despite the lack of attention paid to compositional reasoning for multi-agent systems, they present an ideal setting for divide-and-conquer model checking. Individual agents are often designed to act in an autonomous manner to achieve their design goals by interacting with their environment. Hence the ability to specify properties of a single agent is desirable, e.g., that it meets its individual design goals. However, with current techniques, reasoning about the behaviour of an individual agent requires the costly
calculation of the transition relation and of the reachable states for all of the agents in the system. This leads to the infamous state-space explosion problem, where the number of states in the composed system is exponential in the number of agents. Consequently, the ability to reason about the mentalistic evolution of individual agent in a situated environment is clearly beneficial.

The seminal work of Vardi [Vardi, 1995] advocates that for assume-guarantee reasoning, as the assumption in an assume-guarantee pair is a specification over all the interactions between a component and its environment, the assumption is naturally expressed as a linear-time formula. Furthermore, Vardi also suggests [Vardi, 1995] that the guarantee should be specified as a universal branching-time formula, in order to state properties of all computations and not the existence of a particular computation.

Most approaches to assume-guarantee reasoning look at components that use variables to communicate, or labelled transition systems that use synchronisation over common actions. This has given rise to two orthogonal approaches to the assume-guarantee problem: those focused on communication variables [Nam and Alur, 2006] and those focused on shared actions [Cobleigh et al., 2003].

However, in agent-based systems it is common to design agents that communicate via the observation of global actions [Fagin et al., 1995] (in the reactive systems literature, this is referred to as “rendezvous communication”, and is a similar communication model as used in networks of automata), while their specifications are usually state-based.

In this setting, each agent can perform a set of actions that may be entirely distinct from every other agent in the system, and there is no notion of forced synchronisation over common actions. As such, synchronisation occurs by agents observing actions that are performed by other members of the system. Consequently, we present a compositional approach that relies upon the temporal specification of all valid interactions to determine—without constructing the composition of all of the agents—if a state-based branching-time temporal-epistemic formula is valid for an individual agent. Therefore, in an agent-based setting for assume-guarantee reasoning we use linear-time assumptions, specified over actions, and branching-time guarantees, specified over state-based propositions.

Our approach can be likened to a hybrid approach between module checking [Kupferman et al., 2001; Basu et al., 2007] and assume-guarantee reasoning. Unlike assume-guarantee, module checking allows for the verification properties of open systems. In an open systems interpretation, a component $A$ satisfies a requirement $\varphi$ (written $A \models_\omega \varphi$) if it satisfies the requirement when composed with all possible environments. Module checking can be summarised as below [Basu et al., 2007]:

$$A \models_\omega \varphi \iff \forall E (A|E \models \varphi)$$
where $E$ represents a possible environment that $A$ can be situated in, $\parallel$ denotes synchronous composition and $\models_o$ is “open satisfaction”.

In the approach we present, we look at a variant of module checking under guarantees. That is,

$$A \models \psi \phi \iff \forall E \models \psi \implies A \parallel E \models \phi$$

where $\models \psi \phi$ denotes the satisfaction of the guarantee $\phi$ with respect to the assumption $\psi$. We can then use inference rules to reason about the satisfaction of each agent’s specifications in a global composition.

To verify an individual agent against a temporal-epistemic guarantee and w.r.t. a linear-time assumption, we extend the automata-theoretic approach of Vardi [Vardi, 1995]. In the approach we present, the validity of the guarantee in a given component is reduced to checking the non-emptiness of a tree automaton representing the product of the assumption (translated to a Rabin automaton), the guarantee (represented as a tree automaton) and the tree unwinding of the agent. The approach is sound: a positive result for the compositional approach is preserved in the full composition.

It is worth observing that the approach proposed does not check the composition of the agent with all possible environments satisfying the assumption. Furthermore, using an extension to the automata-theoretic approach proposed in Chapter 4, we compose the agent with an environment likened to a tableau or maximal model for the formula. To this end, we only verify the agent against one composition, but take the assumption into consideration when performing the verification of the guarantee.

### Overall approach.

We now outline the high-level approach:

- The user identifies an action-based specification $\phi$ that can suitably abstract the environment of a given component
- The user then verifies the environment against this specification
- If the verification succeeds, the user then verifies the component “under test” against its own state-based requirements (e.g., $\psi$), but in the context of $\phi$ rather than the environment itself
- Should the verification of the component succeed, it can be inferred that the composition of the component and its environment satisfies both $\psi$ and $\phi$

To support this, we introduce a modular framework for individual agents (IDIS, Section 5.2.1), a method to suitably verify agents in isolation (universal environments, Section 5.3.1) and a way to verify an agent and a specification in the context of an environmental assumption property (property closure environments, Section 5.3.2.1).

The rest of this chapter is structured as follows. In the following section, Section 5.2, we present the necessary preliminaries for our technique; Section 5.3 introduces a method and semantics for verifying assumptions and guarantees; Section 5.4 proposes
a novel approach for verifying assume guarantee rules using alternating automata;
Section 5.5 builds on these three sections to introduce our sound approach for reasoning
about assume-guarantee rules; and finally, in Section 5.6, we conclude the chapter.

5.2 A Modular Formalisation for Interpreted Systems

In this section we reformalise the preliminaries outlined earlier in this thesis to introduce
a focus on modularity. We note that considering the underlying theory at an individual
level, allows us to specify formulae in a modular fashion, which is then amenable to
compositional verification.

For example, consider the CTLK formula

\[ \phi = AG(\lnot bad) \]

This formula expresses that a “bad” state is never reachable in the system. An example
of a “bad” condition could be \( bad = ag_1.crit \land ag_2.crit \), and then \( \phi \) would be the
specification of a mutual exclusion condition on any model that satisfies it.

However such a property is not ideally suited to modular verification. The property
refers to the behaviour of (i.e., it contains atomic propositions defined over) multiple
components; if we were to verify this formula over a single agent, e.g., agent 1, what
should we hope to learn?

As such, we take inspiration from works such as [Filippidis et al., 2012] (Sec-
tion 2.2.3) and look at agent-local specifications. These specifications are used to reason
about the behaviour (and knowledge, as we will show) of an individual agent only. To
facilitate this, when specifying the desired behaviour of agent \( i \), we only allow the
specification to contain propositions from \( i \) and the knowledge modality \( K_i \) for that
agent. Temporal-epistemic specifications of this kind are “introspective”: they refer to
an agent’s ability to know certain properties about its own behaviour over time.

One vital question remains though: what can you gain by performing temporal-
epistemic model checking when only dealing with local propositions? It follows directly
from the semantics of CTLK that the formula \( \phi_i = K_i p_i \) is satisfied in a local state \( g \) only
if \( p_i \in V_i (l_i(g)) \). This is due to the definition of satisfaction for \( K_i \psi \) as being quantified
over all global states in which \( i \) is in the same state. As \( g \sim_i g' \iff l_i(g) = l_i(g') \) and
\( g' \) satisfies \( p_i \iff p_i \in V_i (l_i(g')) \), then we see that such a formula can be simplified by
removing any instance of the modality \( K_i \) from the formula \( \phi \).

However, and as we will demonstrate in this chapter, this is valid only for the purely
propositional-epistemic case. If we introduce temporal modalities into our formulae
(e.g., \( \phi'_i = (p_i \rightarrow AXK_i AXq_i) \)), then such reductions are no longer correct.

While branching and linear-time logics differ on their view of the future (i.e., branch-
ing with multiple possibilities, linear with only one; compare: “Peircean” vs. “Ock-
hamist” time [Reynolds, 2002; Hodkinson and Reynolds, 2006]), they both consider the history of a run to be a necessity (i.e., the past is fixed and is linear). As such, when verifying purely temporal specifications, they allow us to reason about what is necessary given a current past from the current state.

However, when we include epistemic modalities in our specification language, this allows us to leave our current “past”: we are left reasoning for a given state about all possible futures irrespective of the “current” history that took us to that state originally. In such a way, it is clear that reducing specifications by stripping the knowledge modality for formulae based on local propositions is not correct.

For example, it is possible that for a given state \( g = (\ldots, s_i, \ldots) \), \( p_i \) will always hold at any next state from this point (i.e., \( g \) satisfies \( AX p_i \)). By comparison, on a different history that reaches a state \( g' = (\ldots, s_i, \ldots) \) we might not be guaranteed that \( p_i \) holds at all successor states. However, as \( l_i(g) \equiv l_i(g') \equiv s_i \), we have that \( g \sim_i g' \) and therefore \( g \models AX p_i \) but not \( g \models K_i AX p_i \).

This is what makes modular temporal-epistemic reasoning of interest, and why it is not possible to reduce the modular verification of temporal-epistemic formulae by syntactic transformation alone (as might be the intuition, e.g., by dropping the modality \( K_i \) from any formula, which holds in a propositional-epistemic case).

### 5.2.1 Interaction-Defined Interpreted Systems

We begin by presenting interaction-defined interpreted systems (IDIS). As with interpreted systems (presented in Chapter 3), we assume that an interpreted system consists of \( n \) agents and an additional agent, \( e \), representing the environment. For simplicity, we adopt a presentation supporting only a single initial state for the whole model.

IDIS are composed of a set of named agents with local propositions; each agent is a tuple \( A_{id} = (id, L_{id}, l_{id}^0, Act_{id}, P_{id}, E_{id}, AP_{id}, V_{id}) \) where:

- \( id \) is a unique identifier
- \( L_{id} \) is a set of local states
- \( l_{id}^0 \in L_{id} \) is an initial state
- \( Act_{id} \) is the set of actions in \( id \)’s repertoire
- \( P_{id} : L_{id} \rightarrow 2^{Act_{id}} \) is a non-deterministic protocol function denoting the set of actions that \( id \) can perform in a given local state
- \( E_{id} : L_{id} \times (id', act_{id})^m \rightarrow L_{id} (m \in \mathbb{N}) \) is a deterministic evolution function
- \( AP_{id} \) is a set of agent-local propositions
- \( V_{id} : L_{id} \rightarrow 2^{AP_{id}} \) is an agent local interpretation function, mapping local states to local propositions

We also note that these named agents have a unique identifier (i.e., \( id \)) and that the evolution function takes a set of agents paired with their respective actions. Unlike
agents in IS, in IDIS the evolution function is loosely defined and is more closely representative of how agents are specified in the interpreted systems programming language (Section 3.5.3.2). Such a presentation allows for the definition of evolution rules that only specifies what the agent is concerned with, rather than specifying all possible upwards combinations, as is required in standard interpreted systems.

Example 5.1. Interpreted Systems vs. ISPL

Consider the snippet of ISPL (Section 3.5.3.2) shown in Figure 5.1. This snippet expresses that the agent $Agent_i$ transitions from the state $state_n$ to the state $state_m$, when the agent itself performs the action $act_1$.

```
Agent Agent_i

... Evolution:
...
  state = state_m if state = state_n and Action = act_1;
...
end Evolution

end Agent
```

Fig. 5.1. An excerpt of an ISPL evolution function

To specify this evolution line in standard interpreted systems, we would need to specify an individual entry for each possible action that is compatible with $Agent_i$ performing $act_1$. Consider the set:

$$Act_{compatible} = \left\{ (act_1 \times Act_G) \mid Act_G \in \prod_{j \in \{Ag \backslash Agent_i\}} Act_j \right\}$$

This set specifies all combinations of global actions such that $Agent_i$ has performed the action $act_1$. To specify the evolution rule shown in Figure 5.1, the whole definition of the evolution line would need to contain the following, explicitly expanded:

$$\{(state_n, \pi \rightarrow state_m) \mid \pi \in Act_{compatible}\}$$

That is, every combination of actions containing $act_1$, initiating at the state $state_n$ should have the successor $state_m$. 
In comparison, in the IDIS framework, all that would be necessary to specify is:

\[
\{(\text{state}_n, (\text{Agent}_i, \text{act}_1) \rightarrow \text{state}_m)\}
\]

That is, as the snippet in Figure 5.1 only refers to the action act\(_1\) in Agent\(_i\), it is only necessary to refer to this agent and this action in the definition of the protocol function.

In IDIS, a joint/global action \(\overline{A} = (id, act_id)^n (n \in \mathbb{N})\) is a set of pairs of names and actions representing which agent performed what action. Given an IDIS containing \(n\) agents, a global action \(\overline{A}\) contains \(n\) entries, one for each agent.

An evolution rule \(e = (l, \overline{A} \rightarrow l') \in E_{id}\) can be executed in a global state \(g\), and with respect to a given global action \(\overline{A}'\), if \(l_{id}(g) = l\) and \(\overline{A} \subseteq \overline{A}'\) (i.e., all of the named action pairs in \(A\) are present in \(A'\)). As such, we require that all of the global actions referred to in an agent’s evolution function are consistent.

For a single evolution line \(e \in E_{id}\), we write \(e \downharpoonright 1\) for the source state and \(e \downharpoonright 2\) for the set of necessary relevant actions. We therefore have:

\[
\forall e \in E_{id}, \exists e' \in E_{id} \text{ such that } e \downharpoonright 1 = e' \downharpoonright 1 \text{ and } e \downharpoonright 2 \subseteq e' \downharpoonright 2
\]

That is, there exists no two evolution rules starting at the same local state such that one evolution refers to a subset of the global actions for the other. In the remainder of this chapter, we only consider IDIS containing agents with consistent evolution functions.

**Example 5.2. Evolution Consistency**

Consider an evolution function for a given agent that contains the following:

\[
\{(\text{state}_n, ((\text{Agent}_i, \text{act}_1), (\text{Agent}_j, \text{act}_2)) \rightarrow \text{state}_m) \}
\]

Now consider any global actions in the set:

\[
\text{Acts} = \left\{ (\text{act}_1 \times \text{act}_2 \times \text{Act}_G) \bigg| \right. \begin{array}{c}
\text{Act}_G \in \\
\prod_{k \in (\text{Ag}) \setminus (\text{Agent}_i \cup \text{Agent}_j)} \text{Act}_k
\end{array} \left. \right\}
\]

It is clear that both parts of the above evolution hold given this action (i.e., in the state \(\text{state}_n\), both \(\text{state}_m\) and \(\text{state}_o\) are potential successors for any action in \(\text{Acts}\)). However, as we require determinism in the protocol, the IDIS framework does not support these contrasting evolution rules.
It is now useful to provide a definition of well formedness for an IDIS.

**Definition 5.1.** Well-defined Composition

Given a set of named agents $Ag$, the composition of $Ag$ is well-defined iff

1. $\forall id \in Ag, \forall e \in E_{id}, \forall (id', act_{id'}) \in e \, \mid 2$ then $id' \in Ag$ and $act_{id'} \in Act_{id'}$
2. $\forall id \in Ag, \forall l \in L_{id}, \forall A = \prod_{id' \in Ag} (id', act_{id'}) s.t. \exists e \in E_{id}$ such that $e \downarrow 1 = l$ and $e \downarrow 2 \subseteq \overline{A}$
3. For each agent $A_i \in Ag$, $A_i$’s evolution function is consistent

Clause 1 specifies that all agents and all actions referred to in $id$’s evolution function exist in $Ag$ and are in that agent’s repertoire; Clause 2 specifies that for all local states and possible joint actions, there exists an applicable evolution rule (i.e., $E_{id}$ is a total function) and there is no deadlock in the system.

Given a well-defined IDIS, it is straightforward to construct a standard interpreted system for the agents in $Ag$. This can be done by extending any uses of a partial named global action in an individual agent’s evolution line to be defined, against a full global action. As previously required, we have that all evolution lines in IDIS are upwards consistent (i.e., if two evolution lines start at the same state, a superset of actions must also transition to any subset’s destination state); therefore we can pad any partial action with extra agents without adding any inconsistencies.

A model of an interaction-defined interpreted system is a tuple

$$\mathcal{M}_{\text{IDIS}} = (G, G_0, R, \sim_1, \ldots, \sim_n, \sim_e)$$

where:

- $G \subseteq \prod_{i \in Ag} L_i$ is a set of reachable global states
- $G_0 \subseteq \prod_{i \in Ag} l_i^0$ is the global initial state
- $R \subseteq G \times G$ is a transition relation such that:
  $$\forall (g, g') \in T \iff \exists \overline{a} \in \prod_{j \in Ag} (id_j, Act_j) s.t. \forall i \in Ag, \overline{a} |_i \in P_i(l_i(g)) \text{ and } l_i(g') = E_i(l_i(g), \overline{a})$$
- $\sim_i \subseteq G \times G$ is an indistinguishability relation such that:
  $$\forall (g, g') \in \sim_i \iff l_i(g) = l_i(g')$$

As per Definition 5.1, we have that the relation $T$ is serial, i.e., all states have at least one successor; this implicitly restricts us to deadlock-free models.

Given the set $Ag$, we wish to talk about the paths in $\mathcal{M}_{\text{IDIS}}$. A path $\pi = g_0, \overline{a}_0, g_1, \overline{a}_1 \ldots$ in $\mathcal{M}_{\text{IDIS}}$ is an interleaved sequence of global states and actions such that $\forall k \geq 0, \forall i \in Ag, \overline{a}_k |_i \in P_i(l_i(g_k))$ and $l_i(g_{k+1}) = E_i(l_i(g_k), \overline{a})$. As $T$ is serial, we assume that all paths
are infinite. We write \( \Pi(g) \) for the set of paths beginning at the global state \( g \). When verifying properties over \( \mathcal{M}_{IDS} \), we restrict the states \( G \) to be those that are reachable.

In what follows, for convenience, we will write \( \sigma \) to represent a path restricted only to actions and \( \sigma(n) \) to represent the \( n \)th global action in \( \sigma \). Additionally, as we never require interleaved paths, we will write \( \pi \) to represent a path restricted only to states and \( \pi(n) \) to represent the \( n \)th global state in \( \pi \). Given \( \pi \), we write \( \pi|_\sigma \) to represent the path of actions that correspond with the path \( \pi \) of states.

**Interaction sets.**

We are now interested in understanding the set of external influences on agents, such that these can be suitably abstracted when performing modular verification.

**Definition 5.2. Interaction Set**

The interaction set \( \text{int}(i) \subseteq \mathbb{A}_g, i \in \mathbb{A}_g \) is defined as follows:

\[
\text{int}(i) = \{ j | \exists e_i \in E_i, \exists a_j \in \text{Act}_j \text{ such that } (id_j, a_j) \in e_i \downarrow \}
\]

Informally, \( \text{int}(i) \) defines the set of agents with which \( i \) directly interfaces, i.e., the choice of no successor state depends upon an action \( a_j \in \text{Act}_j, j \not\in \text{int}(i) \). Nonetheless, our definition allows for \( i \) to interface with a group of agents.

**Example 5.3. Interaction Set**

We now revisit the partial evolution function as shown in Example 5.2; this example is reproduced below and is assumed to be part of \( \text{Agent}_i \)’s evolution function.

\[
\{ \\
\left( \text{state}_n, (\text{Agent}_i, \text{act}_{i1}), (\text{Agent}_j, \text{act}_{j2}) \right) \rightarrow \text{state}_m \\
\left( \text{state}_n, (\text{Agent}_i, \text{act}_{i1}) \rightarrow \text{state}_o \right) \\
\}
\]

Given the above, we have that \( \text{Agent}_i \)’s interaction set is as follows:

\[
\text{int}(\text{Agent}_i) = \{ \text{Agent}_i, \text{Agent}_j \}
\]

That is, as the protocol function refers to actions from \( \text{Agent}_i \) and \( \text{Agent}_j \) these agents are contained in \( \text{Agent}_i \)’s interaction set.

\( \triangle \)

For ease of presentation, we assume that \( i \) is always in \( \text{int}(i) \) (i.e., that the agent always interacts with itself). Furthermore, for brevity and w.l.o.g., we omit the agent itself when referring to the interaction set.
5.2.2 Logics over IDIS

We can use interaction-defined interpreted systems models to reason about the satisfaction of formulae in various logics. In this chapter we are interested in formulae defined in the action-based linear-time logic A-LTL and the state-based branching-time temporal-epistemic logic CTLK.

5.2.2.1 Action-Based Linear Temporal Logic

A-LTL is a linear-time temporal logic defined over actions, i.e., over the actions of the agents in $Ag$. The grammar of A-LTL is the same as for LTL, as per Chapter 3, except the satisfaction of a proposition is defined over actions instead of states. For clarity, we present the necessary details below.

In this chapter, we only consider A-LTL formulae in negation-normal form (i.e., with negation only applied to action-atoms); the syntax of a formula $\psi$ in A-LTL in negation normal form is as follows:

$$\psi ::= a_i \mid \neg a_i \mid \psi \land \psi \mid \psi \lor \psi \mid X \psi \mid \psi U \psi \mid \psi U^\psi$$

where $i \in Ag$ and $a_i \in Act_i$.

For a given action-path $\sigma$, we define the satisfaction of an A-LTL formula $\psi$ (written $\sigma \models^{A-LTL} \psi$) below:

- $\sigma \models^{A-LTL} a_i$ iff $\sigma(0) \upharpoonright i = a_i$
- $\sigma \models^{A-LTL} \neg a_i$ iff $\sigma(0) \upharpoonright i \neq a_i$
- $\sigma \models^{A-LTL} \psi \land \varphi$ iff $\sigma \models^{A-LTL} \psi$ and $\sigma \models^{A-LTL} \varphi$
- $\sigma \models^{A-LTL} \psi \lor \varphi$ iff $\sigma \models^{A-LTL} \psi$ or $\sigma \models^{A-LTL} \varphi$
- $\sigma \models^{A-LTL} X \psi$ iff $\sigma(1) \models^{A-LTL} \psi$
- $\sigma \models^{A-LTL} \psi U \varphi$ iff $\exists n \geq 0$ s.t. $\sigma(n) \models^{A-LTL} \varphi$ and $\forall j, 0 \leq j < n, \sigma(j) \models^{A-LTL} \psi$
- $\sigma \models^{A-LTL} \psi U^\varphi$ iff $\forall k \geq 0$ s.t. $\sigma(k) \not\models^{A-LTL} \varphi$ or $\exists 0 \leq j < k$ s.t. $\sigma(j) \models^{A-LTL} \psi$

For a formula $\psi$ in A-LTL, we write:

$$M_{IDIS} \models^{A-LTL} \psi$$

when all the paths in the IDIS $M_{IDIS}$ satisfy the formula $\psi$.

We refer the interested reader to [Nicola and Vaandrager, 1990] for additional details.
5.2 A Modular Formalisation for Interpreted Systems

5.2.2.2 State-based Branching-Time Temporal-Epistemic Logic

CTLK is a fusion logic [Kurucz, 2006] consisting of the branching-time temporal logic CTL and a multi-modal epistemic logic (Section 3.3.2.4). In comparison to A-LTL, where propositions were based on the global action occurring in a transition, propositions in CTLK are defined over states. For completeness, we present CTLK over IDIS below.

The inductive syntax of CTLK is as follows:

\[ \psi ::= p_i | \neg \psi | \psi \land \psi | AX \psi | A[\psi U \psi] | A[\psi U \ov{\psi}] | K_i \psi \]

where \( i \in Ag \) and \( p_i \in AP_i \).

For a given global state \( g \) and CTLK formula \( \psi \) we define satisfaction (written \( g \models_{\text{CTLK}} \psi \)) below:

- \( g \models_{\text{CTLK}} p_i \) iff \( p_i \in V_i(L_i(g)) \)
- \( g \models_{\text{CTLK}} \neg \psi \) iff it is not the case that \( g \models_{\text{CTLK}} \psi \)
- \( g \models_{\text{CTLK}} \psi \land \varphi \) iff \( g \models_{\text{CTLK}} \psi \) and \( g \models_{\text{CTLK}} \varphi \)
- \( g \models_{\text{CTLK}} \psi \lor \varphi \) iff \( g \models_{\text{CTLK}} \psi \) or \( g \models_{\text{CTLK}} \varphi \)
- \( g \models_{\text{CTLK}} AX \psi \) iff \( \forall \pi \in \Pi(g), \pi(1) \models_{\text{CTLK}} \psi \)
- \( g \models_{\text{CTLK}} A[\psi U \varphi] \) iff \( \forall \pi \in \Pi(g), \exists n \geq 0 \text{ s.t. } \pi(n) \models_{\text{CTLK}} \varphi \) and \( \forall j, 0 \leq j < n, \pi(j) \models_{\text{CTLK}} \psi \)
- \( g \models_{\text{CTLK}} A[\psi U \ov{\varphi}] \) iff \( \forall \pi \in \Pi(g), \forall k \geq 0 \text{ s.t. } \pi(k) \nmodels_{\text{CTLK}} \varphi \) or \( \exists 0 \leq j < k \text{ s.t. } \pi(j) \models_{\text{CTLK}} \psi \)
- \( g \models_{\text{CTLK}} K_i \psi \) iff \( \forall g', g \sim_i g' \rightarrow g' \models_{\text{CTLK}} \psi \)

For a formula \( \varphi \) in CTLK, we write:

\[ \mathcal{M}_{\text{IDIS}} \models_{\text{CTLK}} \varphi \]

when the initial state of the IDIS \( \mathcal{M}_{\text{IDIS}} \) satisfies the formula \( \varphi \).

5.2.3 Behaviours and Languages

We now introduce the notion of languages and behaviours of both individual agents and their compositions.

5.2.3.1 Languages

Definition 5.3. Language of a Component
The *language* of an agent (or a composition) is the set of all possible infinite strings of actions from that agent (or composition).

Given an agent $i$, a string of actions $a \cdot \sigma$ (where $a \in \text{Act}_i$ and $\sigma \in \text{Act}^\omega_i$) is in $\text{lang}_i(s_i) \subseteq \text{Act}^\omega_i$, the language of $i$ starting at the state $s_i \in L_i$, iff:

- $a \in P_i(s_i)$
- $\exists e_i \in E_i$, s.t., $e_i \downarrow 1 = s_i$, $(i, a) \in e_i \downarrow 2$ and $\sigma \in \text{lang}_i(e_i \downarrow 3)$

We abbreviate $\text{lang}_i$ to mean $\text{lang}_i(t^0_i)$ (i.e., $\text{lang}_i$ is the set of all possible action strings for $i$ starting at its initial state).

In addition, given a composition of agents $X$, we write $\text{lang}(X)$ for the agent-wise union of all $\text{lang}_i$, $i \in X$. Given $\text{lang}(X)$, we write $\text{lang}(X) \upharpoonright \text{Act}_i$ for the restriction of $\text{lang}(X)$ to only the actions of $i$.

**Example 5.4. Language of an Agent**

We begin by considering the Agent $i$ with two states ($s_1$ and $s_2$) and two possible actions ($a$ and $b$). The initial state for Agent $i$ is $s_1$. The temporal evolution of Agent $i$ is shown in Figure 5.2.

From Figure 5.2, we can see that:

- $\text{lang}_i(s_1) = (a^\omega) \cup (a^* \cdot b \cdot a^\omega)$
- $\text{lang}_i(s_2) = a^\omega$

As Agent $i$’s initial state is $s_1$, we have that $\text{lang}_i = \text{lang}_i(s_1) = (a^\omega) \cup (a^* \cdot b \cdot a^\omega)$; that is, Agent $i$’s language is either an infinite sequence of $a$s, or a finite (possibly empty) prefix of $a$s, followed by a single $b$ and then an infinite suffix of $a$s.

\[\triangle\]

### 5.2.3.2 Behaviours

The *behaviours* of an agent (or a composition) is the set of all possible infinite strings of local (global) states from that agent (composition).

Given the set $X$ of agents (e.g., $X = \{A_1, A_2\}$), we write $\text{behaviours}(X)$ to specify the set of all possible strings (i.e., in $\left( \prod_{i \in X} L_i \right)^\omega$) of states consistent with the composition
5.2 A Modular Formalisation for Interpreted Systems

We note that $X$ may not be a completely defined interpreted system (e.g., there may exist $e_i \in E_i$, $i \in X$ where $\exists (j, a_j) \in e_i | j \notin X$), but this does not alter the following presentation (i.e., it is defined to work with partially defined compositions). We write $g \xrightarrow{\pi} g'$ when the composition $X$ has a transition between the states $g$ and $g'$ for the (possibly partial) action $\pi$.

**Definition 5.4. Behaviours of a Component**

Given the set $X$ of agents, a path $\pi \in \text{behaviours}(X) \subseteq \left( \prod_{i \in X} L_i \right)^\omega$ iff:

1. $\pi(0) = \left( \prod_{i \in X} l_i^0 \right)$
2. $\forall i > 0, \exists a \in \left( \prod_{j \in X} \text{Act}_j \right)$, s.t., $\pi(i-1) \xrightarrow{a} \pi(i)$

By Clause 2, two states appear in sequence in $\text{behaviours}(X)$ if there exists some consistent joint action $\pi$ between the agents $X$ that transitions them between each pair of states in the path.

We use $\text{behaviours}(X)$ to specify all possible paths of local states for each agent in $X$; as with $\text{lang}$, we write $\text{behaviours}(X) \mid L_i$ for the restriction of $\text{behaviours}(X)$ to the local states of only the agent $i$.

**Example 5.5. Behaviours of an Agent and a Composition**

Consider the two agents shown in Figures 5.3 and 5.4. We label a transition with $n = x$ when that transition is dependent on Agent $n$ performing action $x$; a transition labelled with a single identifier represents that action being performed by the agent at hand. It is assumed that, while the action $c$ does not occur in Figure 5.4, Agent $2$ can potentially perform this action (i.e., $c \in \text{Act}_2$).

![Diagram](image.png)

*Fig. 5.3. The temporal evolution for the exemplary agent Agent$_1$.*

For Figures 5.3 and 5.4, we have the following behaviours:
The temporal evolution for the exemplary agent Agent₂.

\[ \text{Behaviours (Agent}_1 \parallel \text{Agent}_2) = \left( (l_1^1, l_2^1) \cdot (l_1^2, l_2^2) \right) \cup \left( (l_1^1, l_2^1) \cdot (l_1^4, l_2^3) \right) \]

\[ \text{Behaviours (Agent}_1) = \left( l_1^1 \cdot (l_2^1) \right) \cup \left( l_1^1 \cdot (l_3^1) \right) \cup \left( l_1^1 \cdot (l_4^1) \right) \]

We can see that, when considered in isolation, the state \( l_3^1 \) is reachable in the behaviours of Agent₁. However, when Agent₁ is composed with Agent₂, the state \( l_3^1 \) is no longer reachable, as the concrete agent Agent₂ never performs the action \( c \).

\[ \triangle \]

### 5.2.4 Satisfaction Preservation

We now introduce the notion of satisfaction preservation between different compositions for a given component. We use this section to set forth the underlying constructions that can be used in the modular verification of multi-agent systems. In particular, by identifying the suitable fragments of the logics we are concerned with, we formally demonstrate how these fragments can be preserved across different compositions meeting certain criteria.

We note that, while these preliminaries are essential for demonstrating theorems occurring later in the thesis, these statements have not been made previously in the literature, nor have their abilities to preserve identified fragments. For example, while [Cohen et al., 2009c] shows the preservation results for the full interpreted systems when every agent is abstracted, to the best of our knowledge no-one has shown the preservation of \( \forall \text{CTLK} \) and \( \text{A-LTL} \) related to a singly abstracted component (e.g., the environment).

#### 5.2.4.1 Fragments of CTLK and A-LTL

For both CTLK and A-LTL, we consider the subset of formulae to be definable only over a single agent. These formulae are called agent- or \( i \)-local. For CTLK, we also consider two further fragments: the universal fragment (\( \forall \text{CTLK} \)) and the existential fragment (\( \exists \text{CTLK} \)); see Section 3.2.6.2. We note that \( \forall \text{CTLK} \) is also commonly written as
A CTLK formula, $\varphi$, is said to be $i$-local iff the atomic propositions it references are only defined over $i$ (i.e., $\{i \mid p_i \in cl(\varphi)\} = \{i\}$) and the epistemic modalities used are only those for agent $i$ (i.e., $\{i \mid K_i \in cl(\varphi)\} = \{i\}$), where $cl(\varphi)$ is the closure of $\varphi$, containing all sub-formulae of $\varphi$ and itself. Similarly an A-LTL formula $\psi$ is $i$-local iff it only refers to the actions of agent $i$ (i.e., $\{i \mid a_i \in cl(\psi)\} = \{i\}$).

**Example 5.6. Local and Non-local Formulae**

Given two agents $i$ and $j$, such that $AP_i = \{p_i, q_i\}$ and $AP_j = \{p_j, q_j\}$, the formula $AG(p_i \rightarrow K_i q_i)$ would be $i$-local, but the formulae $AG(p_i \rightarrow K_j q_i)$ and $AF(q_j \land p_j \land p_i)$ would be local to neither agent.

**Example 5.7. Universal, Existential and Mixed Formulae**

The formula $AG(p \rightarrow AFq)$ is a universal formula, whilst $EF(p \lor EGq)$ is an existential formula. Furthermore, the formula $A[pUEFq]$ is a mixed formula and is in neither fragment.

### Property Preservation

The logics $\forall$CTLK and A-LTL both present the “upwards preservation property” [Vardi, 1995]. This is where, if a composition with greater behaviours satisfies a specification, then any composition with fewer behaviours (i.e., a composition containing less-abstract components) will also satisfy this property. We will show that if the component $i$ satisfies an $i$-local $\forall$CTLK or A-LTL formula in one composition, then all compositions that introduce more components—or introduce more restrictive components, e.g., with a reduced observable language—will also satisfy that formula. It follows that the composition of two components cannot introduce new behaviours into either of its constituent parts: it can only eliminate behaviours that are inconsistent with composition (c.f., Example 5.5).

We note that it is possible to have two agents that share the same identifier and action set, but which do not exhibit the same behaviour. However, as long as the action set and identifier are the same, we can compose an external agent with either of these other agents without modification. Clearly if the external agent’s behaviour depends on the actions of either of these agents, substituting one agent for the other may change the observable behaviour of the external agent.

**Theorem 5.1. Compositional Restriction**

Given three agent programs $A_1, A_2$ and $A_3$ such that $id_1 = id_2$ and $Act_1 = Act_2$ (i.e., the identifier and actions of $A_1$ and $A_2$ are the same; we denote this set by $Act_O$),
\( \text{lang}(A_1) \subseteq \text{lang}(A_2) \subseteq \text{Act}_O \) then

\[
\text{behaviours}(A_1 \| A_3) \subseteq \text{behaviours}(A_2 \| A_3)
\]

**Proof.** Given that \( \text{Act}_1 = \text{Act}_2 = \text{Act}_O \), we have that \( A_3 \)'s evolution function is \( E_3 : \text{L}_3 \times (\text{id}_1, \text{Act}_O) \times (\text{id}_3, \text{Act}_3) \to \text{L}_3 \). As we have that \( \text{lang}(A_1) \subseteq \text{lang}(A_2) \), it follows that for every sequence of actions in \( \text{lang}(A_1) \), there is an equivalent sequence of actions in \( \text{lang}(A_2) \).

Imagine that there is a behaviour in \( \text{behaviours}(A_1 \| A_3) \) but not in \( \text{behaviours}(A_2 \| A_3) \).

For this to occur, there must be a sequence of actions in \( \text{lang}(A_1) \) that is not in \( \text{lang}(A_2) \) – yet this is a contradiction. As such, we have that \( \text{behaviours}(A_1 \| A_3) \subseteq \text{behaviours}(A_2 \| A_3) \).

\( \Box \)

**Example 5.8. Compositional Restriction**

We now revisit the agents from Example 5.5 (reproduced in Figures 5.5 and 5.6). However, we also consider a further agent \( \text{Agent}_3 \) which is a refinement of \( \text{Agent}_2 \) – this agent is shown in Figure 5.7.

---

**Fig. 5.5.** The temporal evolution for the exemplary agent \( \text{Agent}_1 \).

**Fig. 5.6.** The temporal evolution for the exemplary agent \( \text{Agent}_2 \).
We can see that $\text{Agent}_3$ has no transition labelled with $b$ (or $c$), therefore we have that $\text{lang}(\text{Agent}_3) = a^\omega$, while $\text{lang}(\text{Agent}_2) = (a^\omega) \cup (b^\omega)$. As such, we have that for every string in the language of $\text{Agent}_3$ there is a corresponding string in the language of $\text{Agent}_2$.

When restricted to only the states of $\text{Agent}_1$, we have that:

- $\text{Behaviours}(\text{Agent}_1 \parallel \text{Agent}_3) |_{s_1} = (l_1^1 \cdot (l_2^2)^{a^\omega})$,
- $\text{Behaviours}(\text{Agent}_1 \parallel \text{Agent}_3) |_{s_1} = (l_1^1 \cdot (l_3^3)^{a^\omega})$.

Therefore, we can see that if we have a component with a restricted language of another, that its composition will also be a restriction of the behaviours of the other potential composition.

\[\square\]

**Theorem 5.2. Upwards Preservation of $\forall$CTLK**

Given three agent programs $A_1, A_2$ and $A_3$ such that

\[
\text{behaviours}(A_1 \parallel A_3) \subseteq \text{behaviours}(A_2 \parallel A_3)
\]

then for any formula $\phi$ in $\forall$CTLK, we have:

\[
A_3 \parallel A_2 \models \phi \quad \text{implies} \quad A_1 \parallel A_3 \models \phi.
\]

**Proof.** For the temporal-only fragment of $\forall$CTL, preservation can be shown simply by induction on the length of the formula, so we concentrate on the epistemic fragment and elaborate on the portion of the proof concerning $K_3 \phi$. As such, we assume preservation for the temporal fragment and show proof by induction for the case $K_3 \phi$.

First take an arbitrary state $s_3 \in L_3$. We denote by $G_{13}$ the set of global states for the composition $A_1 \parallel A_3$ and $G_{23}$ for the composition $A_2 \parallel A_3$. Additionally, take $S_1 = \{ g \in G_{13} \mid I_3(g) = s_3 \}$ and $S_2 = \{ g \in G_{23} \mid I_3(g) = s_3 \}$ (i.e., the set $S_1$ is the set of states from the composition of $A_1$ and $A_3$, such that $\forall s \in S_1, s_3 \sim_3 s$).

As $\text{behaviours}(A_2 \parallel A_3) \subseteq \text{behaviours}(A_1 \parallel A_3)$ it follows that the set of outgoing behaviours from any state $s_2 \in S_2$ subsumes or is equal to the set of outgoing behaviours
from any state $s_1 \in S_1$ (i.e., the set of infinite paths in $L_3^{\omega}$ leaving $s_2$ is a—possibly non-strict—superset of those leaving $s_1$).

We write $PI_1(s_1) \subseteq G_{1|3}^{\omega}$ for the set of paths in $A_1 || A_3$ starting at $s_1$, similarly $PI_2(s_2) \subseteq G_{2|3}^{\omega}$ for the set of paths in $A_2 || A_3$ starting at $s_2$. Given that the behaviours outgoing from $s_2$ is greater than from $s_1$, we have that $PI_1(s_1) \subseteq PI_2(s_2)$.

As we assumed that the temporal fragment is preserved by behaviours, it therefore follows that for any 3-local formula $\phi$ in $\forall CTLK$ if $\forall s_2 \in S_2, [A_2] || A_3, s_2 \models \phi$ then we have $\forall s_1 \in S_1, [A_1] || A_3, s_1 \models \phi$.

Consequently for all 3-local formulae $\phi$ in $\forall CTLK$, we have that $A_2 || A_3, s_2 \models \phi$ implies that $A_1 || A_3, s_1 \models \phi$. That is, if we have that all epistemically related states in $G_{2|3}$ satisfy $\phi$, then we also have that all epistemically related states in $G_{1|3}$ satisfy $\phi$. Therefore it follows that $A_2 || A_3, s_2 \models K_3 \phi \implies A_1 || A_3, s_1 \models K_3 \phi$.

It is clear to see that given a formula $\phi$ in $\exists CTL^* K$ (the existential fragment of $CTL^* K$, Section 3.2.6.1) might be satisfied in the composition of $A_2 || A_1$ but not in the composition of $A_1 || A_3$ because $A_2$ admits more behaviours in $A_3$ (i.e., $|lang (A_2)| > |lang (A_1)|$). That is, the behaviour satisfying $\phi \in \exists CTL^* K$ in the model $A_2 || A_3$ may not exist in the more restrictive composition $A_1 || A_3$. Hence our restriction to $ACTLK$.

**Observation 3.** Tree unwindings and behaviours.

Two states, $s_1$ and $s_2$, occur in sequence of a behaviour iff there exists some temporal transition in the composition between $s_1$ and $s_2$. If behaviours were enriched with epistemic transitions (i.e., $s_1$ and $s_2$ would occur in sequence if $s_1 \sim s_2$ for some agent in the composition), then the unwinding of behaviours, where common branches are not duplicated, would be a tree unwinding for the composition (as per Section 4.2).

**Proposition 5.1.** Given three agent programs $A_1$, $A_2$ and $A_3$ such that $id_1 = id_2$ and $Act_1 = Act_2$ and where $lang (A_1) \subseteq lang (A_2)$ then for any $\forall CTLK$ formula $\phi$, $A_2 || A_3 \models \phi$ implies $A_1 || A_3 \models \phi$.

**Proof.** Follows immediately from the Proof of Theorem 5.1 and Theorem 5.2.

That is, as $lang (A_1) \subseteq lang (A_2)$ implies behaviours ($A_1 || A_3 \subseteq$ behaviours ($A_2 || A_3$), and as behaviours ($A_1 || A_3$) $\subseteq$ behaviours ($A_2 || A_3$) implies $A_2 || A_3 \models \phi \Rightarrow A_1 || A_3 \models \phi$, then $lang (A_1) \subseteq lang (A_2)$ implies $A_2 || A_3 \models \phi \Rightarrow A_1 || A_3 \models \phi$.

**Theorem 5.3. Upwards Preservation of A-LTL**

Given three agent programs $A_1, A_2$ and $A_3$ (where $A_1$ and $A_2$ share the same identifier and alphabet) such that $lang (A_1) \subseteq Act_2^{\omega}$, $lang (A_1) \subseteq lang (A_2) \subseteq Act_2^{\omega}$, and $L (\phi)$ $\subseteq$ Act_2^{\omega}$ (i.e., the atomic propositions of the A-LTL formula $\phi$ are over $Act_2^{\omega}$) then:

$$lang (A_2 || A_3) \subseteq L (\phi) \implies lang (A_1 || A_3) \subseteq L (\phi)$$

**Proof.** Trivial. If $A_2$’s language subsumes that of $A_1$, then it follows that $A_1 || A_3$ will also be subsumed by the language of $A_2 || A_3$, for any agent $A_3$. Furthermore, if all of the infinite strings of actions of $A_2 || A_3$ satisfy the A-LTL property $\phi$, then, by the transitivity of the relation $\subseteq$ we have that the language of $A_1 || A_3$ will also satisfy $\phi$. 

□
5.3 Assumption-Based Model Checking

It should be noted that A-LTL is an implicitly quantified $\forall \text{CTL}^*$ property, with its propositions defined over actions. As such, it is not possible to express the existence of a path in A-LTL, which would then remove the possibility of having the upwards preservation property.

5.3 Assumption-Based Model Checking

We now concentrate on a restricted class of multi-agent systems. We consider systems where the interaction set, $\text{int}(i)$, of the agent “under test” (i.e., the agent that the $i$-local formula is defined over) contains only $i$ and $e$ (i.e., no evolution rule in $E_i$ depends directly upon the action $\pi \downarrow j$ for an agent $j \notin \text{int}(i)$). Such a restriction allows us to define the action-based linear-time assumptions over one agent only (i.e., the environment agent).

Under our assumption-based model checking approach, we are interested in verifying whether a) the environment unconditionally satisfies an action-based linear-time formula specifying its interaction with agent $i$ (Section 5.3.1) and b) if agent $i$, under the assumption that the environment adheres to the linear-time assumption, satisfies a state-based branching-time temporal-epistemic guarantee (Section 5.3.2).

By concentrating on the fragments identified in Section 5.2.4.1, and applying the theorems previously presented, we can apply the assumption-based technique in a sound way. After defining an automata-theoretic model checking procedure for the technique (Section 5.4), we demonstrate in Section 5.5 how assumption-based model checking can be used to perform assume-guarantee-style modular reasoning for temporal-epistemic properties.

5.3.1 Action-Based Model Checking of Linear Assumptions

Determining if the environment unconditionally satisfies a linear-time, agent-local, action-based formula can be easily related to the problem of module checking [Kupferman et al., 2001]. To formulate such a setting we consider, at every instance, that every possible combination of actions can be performed for all of the agents that the environment interfaces with. Considering the paths of the environment in such a setting would result in a possible over-approximation of the paths of the environment in a genuine composition, given that every component is unlikely to act chaotically [Roscoe et al., 1996; Sidorova and Steffen, 2001; Leino and Logozzo, 2005] normally.

Given a selection of agents $A_g$ and an agent of particular interest (e.g., the environment $e \in A_g$), we need to construct a valid closure composition to verify if the agent
e unconditionally satisfies its linear-time assumption. To this end, for an agent $A_i$, we consider two possible sets of agents:

- $C_i$—a closure environment for $i$, Definition 5.5—a set of agents such that composing $A_i$ with $C_i$ leads to a well-defined composition;
- $U_i$—a universal environment for $i$, Definition 5.6—a set of agents that acts in the most general, or chaotic, way possible.

**Definition 5.5. Closure Environment**

Given an agent program $A_i$, a closure environment for $A_i$ is a set of agents $C_i = \langle A_{i1}, \ldots, A_{in} \rangle$ such that for all $e_i \in E_i$, for all $(id_j, a_j) \in e_i \downarrow_1$, if $id_j \neq i$ then there exists an $1 \leq m \leq n$ such that $id(A_{im}) = id_j$ and $a_j \in Act(A_{im})$.

A closure environment for an agent is a set of agents such that composition of $A_i$ with $C_i$ forms a well-defined MAS, where all of the named id-action pairs in $i$’s evolution rules exist in $C_i$.

**Example 5.9. Closure Environments**

We now consider the agent $Agent_4$, which contains the following in its evolution function:

\[
\begin{align*}
(\text{state}_n, ((id_1, act_3), (id_2, act_4)) & \rightarrow \text{state}_m) \\
(\text{state}_k, ((id_1, act_1), (id_3, act_5)) & \rightarrow \text{state}_j)
\end{align*}
\]

It follows that a closure environment for $Agent_4$ is $C_4 = \langle Agent_1, Agent_2, Agent_3 \rangle$, such that, at a minimum:

- $id(Agent_1) = id_1$, and where $act_1, act_3 \in acts(Agent_1)$
- $id(Agent_2) = id_2$, and where $act_4 \in acts(Agent_2)$
- $id(Agent_3) = id_3$, and where $act_5 \in acts(Agent_3)$

That is, the agents in the closure environment $C_4$ contain all of the identifiers necessary, as well as all of the agents containing the required respective actions.

**Observation 4. Closure environments and interaction sets.**

For a given agent $A_i$, $|int(A_i)| = |C_i| + 1$. That is, the number of agents in $i$’s interaction set is exactly the cardinality of its closure environment, plus one additional entry for itself.

**Definition 5.6. Universal Environment**

For an agent $A_i$, where $int(i) = \{i_1, \ldots, i_n\}$, its universal environment $U_i = \langle A_{i1}, \ldots, A_{in} \rangle$ is a closure environment such that for all $1 \leq m \leq n$, we have
5.3 Assumption-Based Model Checking

\[ A_{im} = (id_{im}, L_{im}, Act_{im}, P_{im}, E_{im}, \emptyset, \emptyset) \]

defined as follows:

- \( id_{im} \) is a consistent and required identifier to make \( U_i \) a closure environment
- \( L_{im} = \{ * \} \), where \( \{ * \} \) represents a set containing one element
- \( f_{im}^i = \{ * \} \)
- \( Act_{im} = acts_{im} \cup \{ \varepsilon \} \), where \( acts_{im} \) is the set of actions that appear in \( A_i \)’s evolution rules associated with \( id_{im} \)
- \( P_{im} = \{ (\ast, a_{im}) \mid a_{im} \in Act_{im} \} \)
- \( E_{im} = \{ (\ast, \emptyset \rightarrow \ast) \} \)

That is, a universal environment for agent \( A_i \) is a set of agents such that each agent can perform all of the actions referred to by \( A_i \), plus one additional “hidden” action, at all possible instances. Furthermore, we associate no propositions or valuation function on each agent in the universal environment.

**Example 5.10. Universal Environments**

We now reconsider \( Agent_4 \) from Example 5.9. In this example, \( Agent_4 \)’s evolution contained the following agents:

- \( id(Agent_1) = id_1 \), and where \( Act_1, Act_3 \in acts(Agent_1) \)
- \( id(Agent_2) = id_2 \), and where \( Act_4 \in acts(Agent_2) \)
- \( id(Agent_3) = id_3 \), and where \( Act_5 \in acts(Agent_3) \)

As such, it is necessary for \( Agent_4 \)’s universal environment to have agents with the following properties:

- \( lang_1 = (act_1 \mid act_3 \mid \varepsilon)^\omega \)
- \( lang_2 = (act_4 \mid \varepsilon)^\omega \)
- \( lang_3 = (act_4 \mid \varepsilon)^\omega \)

To exemplify, \( Agent_1 \) would be composed of the following:

- \( Act_1 = \{act_1, act_3, \varepsilon\} \)
- \( P_1 = \{\ast \rightarrow \{act_1, act_3, \varepsilon\}\} \)
- \( E_1 = \{\ast, \emptyset \rightarrow \ast\} \)

We can see that such a definition would instantiate an agent with a language of \( (act_1 \mid act_3 \mid \varepsilon)^\omega \). We then also need to instantiate the remaining agents \( Agent_2 \) and \( Agent_3 \) in the same way.

\[ \triangle \]

**Observation 5. Universal environments and languages.**

For all agents \( A_{im} \in U_i \), we have by construction that:

\[ lang(A_i||U_i) \mid Act_{im} \equiv Act_{im}^\omega \]
That is, for each agent in the universal environment for \( i \), its observable language is the set of all possible infinite strings defined over its action set. Furthermore, we have that \( \text{Act}_{im}^{\omega} \models \text{acts}_{im}^{\omega} \).

**Observation 6.** Uniqueness of universal environments.

For a given agent \( A_i \), there are many possible closure environments, but its universal environment is uniquely defined.

**Observation 7.** State-space of a universal environment.

The set of global states of the interaction-defined interpreted system \( A_i \parallel U_i \) will be \( G \subseteq \prod_{r \in \text{arr}(i)} L_r \times L_i \). As \( L_r \) for each universal agent is a singleton, then \( |G| = |L_i| \) (i.e., the composition does not introduce any additional states in the composition).

**Proposition 5.2.** Behaviours and Languages for Closures

Given \( A_i \), and for any \( C_i \) and \( U_i \), we have the following:

- \( \text{lang} (A_i || C_i) \downarrow \text{Act}_i \subseteq \text{lang} (A_i || U_i) \downarrow \text{Act}_i \)
- \( \text{behaviours} (A_i || C_i) \downarrow \subseteq \text{behaviours} (A_i || U_i) \downarrow \)

That is, for any closure, the observable language and behaviours of \( A_i \) are a subset of those possible for its universal environment.

**Proposition 5.3.** Property Preservation for Closures

Given \( A_i \), and for any \( C_i \) and \( U_i \), we have that for any \( i \)-local \( \phi \) in either A-LTL or \( \forall \text{CTLK} \):

\[ A_i || U_i \models \phi \quad \text{implies} \quad A_i || C_i \models \phi. \]

That is, if an agent program \( A_i \) satisfies a given property in a universal environment, then it satisfies that property in all closure environments.

**Proof.** First, consider the case that \( \phi \) is an \( i \)-local A-LTL formula and that the tree unwinding of \( A_i || C_i \) has edges labelled with \( \text{Act}_i \). It can immediately be seen that all action-label branches that occur in \( A_i || C_i \) also appear in \( A_i || U_i \) (i.e., \( \text{lang} (A_i || C_i) \downarrow \text{Act}_i \subseteq \text{lang} (A_i || U_i) \downarrow \text{Act}_i \), as \( \text{lang} (C_i) \subseteq \text{lang} (U_i) \)).

Next, consider that \( \phi \) is an \( i \)-local \( \forall \text{CTLK} \) formula and that tree unwinding of \( A_i || C_i \) has nodes labelled with \( \text{AP}_i \). As with the A-LTL case, it follows that any labelled branch in \( A_i || C_i \) is also in \( A_i || U_i \) (i.e., \( \text{behaviours} (A_i || C_i) \downarrow \text{AP}_i \subseteq \text{behaviours} (A_i || U_i) \downarrow \text{AP}_i \), as \( \text{behaviours} (C_i) \subseteq \text{behaviours} (U_i) \)).

Finally, following Theorem 5.2, Theorem 5.3 and Proposition 5.2, we have that if \( A_i || U_i \) satisfies \( \phi \), any system with fewer behaviours or a smaller language (e.g., \( A_i || C_i \)) also satisfies \( \phi \).
5.3.1.1 Action-Based Model Checking Procedure

Given the deterministic Rabin automaton (DRA) $A_\psi$ for a formula $\psi$, we can construct an agent $A_\psi$ that exhibits the behaviours of $\psi$. We refer to this as a property observer or a property agent; we present property agents in Section 5.3.2.1.

**Property observers.**

Given the DRA, $A_\psi$, for an $i$-local formula $\psi$ in A-LTL, a property observer for $\psi$ (denoted $O_\psi$) is an agent that can be composed with $i$ and watches its actions via $O_\psi$’s evolution function, while taking transitions between states as per the transition relation of $A_\psi$. To this end, $O_\psi$ only has one action ($\tau$).

*Observation 8.* Observer interaction sets.

As a property observer for the $i$-local formula $\psi$ synchronises with $i$, it follows that $\text{int}(O_\psi) = \{i\}$.

5.3.1.2 Property Observer Construction

To construct a property observer $O_\psi$ for the $i$-local formula $\psi$, we perform the following steps:

- Convert $\psi$ to an NBA $B_\psi$ via the classical approach for LTL formulae (Section 3.4.3.1)
- Convert the NBA $B_\psi$ to the DRA $A_\psi$, where $L(B_\psi) \equiv L(A_\psi)$, using Safra’s construction [Safra, 1988; Roggenbach, 2002] (Section 3.4.3.2)
- Encode $A_\psi$ into the property observer $O_\psi$, as per the below

We assume that, given $\psi$, we can obtain the DRA, $A_\psi = (\Sigma, S, s_0, \delta, A)$. Given $A_\psi$, $O_\psi = \langle id, L_\psi, l^0_\psi, Act_\psi, P_\psi, E_\psi, \emptyset, \emptyset \rangle$, where:

- $id$ is a fresh identifier for $O_\psi$
- $L_\psi = S$
- $l^0_\psi = s_0$
- $Act_\psi = \{\tau\}$
- $P_\psi = \{(l, \tau) \mid l \in L_\psi\}$
- $E_\psi = \{(l, (i, a_i), l') \mid l, l' \in L_\psi, a_i \in Act_i, \delta(l, a_i) = l'\}$

It is clear that the property observer $O_\psi$ always selects the $\tau$ action in its protocol, but its evolution lines are only dependent upon the agent $i$ over which $\psi$ is specified.

*Example 5.11.* Property Observer Construction

We begin by considering the LTL formula $\varphi = F G a$, stating that, eventually, the proposition $a$ will always hold. From an action-based interpretation, this specification expresses that eventually the agent will only perform the action $a$. 


The DRA $A_{\varphi}$ for this formula—constructed from the NBA and then determinised using Safra’s construction—is shown in Figure 5.8. The top number in a state is the state number, and the bottom number is the acceptance conditions that state is a member of ($+$) or not ($-$).

The DRA in Figure 5.8 has three states (0, 1 and 2), with the grey state (2) being the initial state, and has only one acceptance pair, i.e., $L_0 = \{0\}, U_0 = \{1, 2\}$. We can see in Figure 5.8 that only a path that gets infinitely trapped in the state 0 will be accepting, corresponding to that path satisfying $FGa$.

The property observer $O_{\varphi}$ for the agent $i$ that corresponds to this DRA is constructed from the following elements:

- $L_{\varphi} = \{0, 1, 2\}$
- $l^0_{\varphi} = 2$
- $E_{\varphi} = \{(1, (id_i, a) \rightarrow 0), (1, (id_i, b) \rightarrow 2), \ldots\}$

We have only presented part of the evolution function, and omitted the action set and protocol function. Furthermore, we assume that the agent $i$ only has two actions $a$ and $b$; this is demonstrated by the transition labelled $\neg a$ from state 1 to 2 being transformed to be conditional on the action $b$. Had $Act_i$ contained more than $a$ and $b$, then the definition of $E_i$ would have had to be suitably extended, to ensure totality, as per Definition 5.1.

\[\triangle\]

5.3.1.3 Model Checking Assumptions

To check if the agent $i$ satisfies $\psi$ (i.e., if $A_i \models ^{LTL}_{\Delta} \psi$), we construct the composition of $A_i$ with $O_{\varphi}$ and $U_i$. We denote by $I_{\{i, \psi\}}$ the IDIS constructed from this composition.
We therefore have the following theorem:

**Theorem 5.4. Model Checking for A-LTL**

\[ A_i \models ^{A\text{-LTL}} \varphi \iff A_i \parallel U_i \models \varphi \] (by definition)

\[ A_i \parallel U_i \parallel O_i \models \varphi \] (as above)

\[ \text{iff There does not exist a path in } I_{(i,\varphi)} \text{ that is rejected w.r.t. the acceptance condition for } A_\varphi \text{ (the DRA for } \varphi). \]

**Proof.** We expand on the final line.

\[ \Rightarrow \] Presume that there exists an infinite path \( \pi \) in \( I_{(i,\varphi)} = A_i \parallel U_i \parallel O_i \) such that \( \pi \) is rejected by \( A_\varphi \). This means that there exists some behaviour of \( A_i \), when composed with \( U_i \), that induced an infinite path in \( O_i \) that was not accepting (i.e., \( \inf(\pi) \models L_\varphi \) is not accepted by the DRA \( A_\varphi \)). This means that \( \pi \models L_\varphi \), and therefore that \( A_i \not\models ^{A\text{-LTL}} \varphi \).

\[ \Leftarrow \] Assume that \( \text{lang}(A_i) \in L(\varphi) \). As such, for any \( \sigma \) in \( \text{lang}(A_i) \), \( \sigma \) is accepted by \( A_\varphi \). Therefore any infinite behaviour of \( O_i \) in \( A_i \parallel U_i \parallel O_i \) is also accepted by \( A_\varphi \). Finally, as \( L_\varphi \) (i.e., the local states of \( O_i \) are finite), this means any infinite behaviour in \( O_i \) must contain a path that is either accepting or rejecting.

It therefore follows that \( A_i \models ^{A\text{-LTL}} \varphi \) iff \( A_i \parallel U_i \parallel O_i \) does not contain any rejecting paths.

Example 5.12. Model Checking for A-LTL

We now revisit the use of the property observer \( O_\varphi \) for the formula \( F G a \) from Example 5.11; we consider \( O_\varphi \) composed with the agent Simple from Figure 5.9. We note that, as Simple does not contain any references to external agents (i.e., its evolution is defined only over its own actions), \( U_{\text{Simple}} = \emptyset \).

![Fig. 5.9. The agent Simple.](image-url)
If we first consider the infinite path \( s_1 \cdot (s_2)^\omega \) in *Simple*, then it is clear that this path will induce the infinite path \( 2 \cdot 1 \cdot (0)^\omega \) in \( O_\varphi \), which is accepting. However, the agent *Simple* also has the behaviour \((s_1)^\omega\), which corresponds to \((0)^\omega\) in \( O_\varphi \) – a path which is not accepting. We can see by inspection that existence of such a path in *Simple*, means that \( s_1 \not\models F G a \).

Therefore, the existence of any path that is not accepting against the DRA \( A_\varphi \), means the agent under observation cannot satisfy the selected assumption.

\[ \triangle \]

**Remark 5.1.** We use the approach defined in the following section to verify if \( I_{\{i, \psi\}} \) contains a rejecting path or not. This is achieved by constructing the product automaton for the formula AG true with \( I_{\{i, \psi\}} \) and then checking if all infinite paths satisfy the acceptance condition for \( \psi \). As per the proof of Theorem 5.4, if there exists a rejecting path (w.r.t., the Rabin condition for the assumption) in \( I_{\{i, \psi\}} \) then there is some behaviour exhibitable by \( A_i \) that invalidates \( \psi \). Using an automata-theoretic approach, checking AG true requires us to check every infinite path in the model for acceptance against \( \psi \).

### 5.3.2 State-Based Model Checking of Branching Guarantees

For assume-guarantee-based compositional model checking, we are interested in checking the satisfaction of an \( i \)-local \( \forall \text{CTL}_K \) guarantee over an individual component, when \( A_i \) is constrained to the paths satisfying an A-LTL assumption. For an arbitrary \( i \)-local \( \forall \text{CTL}_K \) formula \( \varphi \) and an arbitrary A-LTL formula \( \psi \), we write:

\[
A_i \models_{\text{CTL}_K \psi} \varphi
\]

when the agent \( A_i \) satisfies the formula \( \varphi \) if restricted to paths in \( i \) that validate the A-LTL formula \( \psi \).

Following from the previous section, we see that one approach to verifying the agent \( A_i \) against \( \varphi \) could be composing \( A_i \) with \( U_i \) and then performing model checking as normal (e.g., using the approach as presented in Chapter 4). However, when it comes to checking the satisfaction of the \( i \)-local \( \forall \text{CTL}_K \) guarantee, we wish to restrict the paths of \( A_i \) to those that satisfy the \( e \)-local A-LTL assumption \( \psi \). Composing \( A_i \) with \( U_i \) and not taking into consideration the language as specified by \( \psi \) could cause significantly more false-negatives than would occur if we had used \( \psi \).

Given an assumption \( \psi \), the set of assumption-valid paths are the set of paths such that all paths satisfy the \( \psi \). When performing model checking for \( \forall \text{CTL}_K \), quantification of paths takes place over the tree-unwinding of these assumption-valid paths.

It now remains to define a model checking procedure that eliminates paths in \( A_i \) that violate the A-LTL assumption.
5.3 Assumption-Based Model Checking

5.3.2.1 Model Checking Guarantees Against Assumptions

We begin by introducing the constructs required to verify a ∃CTLK guarantee against an A-LTL assumption. To support this, we introduce the notion of property agents and property closure environments. The latter are environments that can be instantiated from a given A-LTL formula, and can be used in the verification of these formulae.

Property agents.

Given the DRA \( A_\varphi \) for an \( e \)-local formula \( \varphi \) in A-LTL, the property agent for \( \varphi \) (denoted \( P_\varphi \)) is an agent that transitions as per the transition relation of the DRA \( A_\varphi \), but has \( e \)'s identifier (i.e., \( id(P_\varphi) \equiv id(A_e) \)) and its action set are those actions appearing in the closure of \( \varphi \), plus the “null” action \( \tau \).


Unlike the property observer \( O_\varphi \) for \( \varphi \), the property agent \( P_i \) performs actions as \( e \) would and does not observe any other actions. That is, while \( int(O_\varphi) = \{ e \} \), \( int(P_\varphi) = \emptyset \). Furthermore, despite differences in \( lang(A_e) \) and \( lang(P_\varphi) \), composing the agent \( A_i \) with either \( A_e \) or \( P_\varphi \) is undetectable to \( i \).

Finally, it holds that:

\[
lang(P_\varphi) = \{ \{ a_e \in Act_e | a_e \in cl(\varphi) \} \cup \{ \tau \} \}^\omega
\]

Property closure environments.

Given two agent programs \( A_i \) and \( A_e \), such that \( int(i) = \{ e \} \) and the \( e \)-local A-LTL formula \( \varphi \), we define the property closure environment \( P_{\{\varphi,i\}} \) for \( i \) and \( \varphi \) to be the extension of the property agent \( P_\varphi \) with the appropriate actions to be a valid closure environment for \( A_i \).

We do not require the protocol of \( P_{\{\varphi,i\}} \) to enable any additional actions in the closure environment; it is simply a syntactic transform of \( Act_{\{\varphi,i\}} \) such that the composition of \( A_i || P_{\{\varphi,i\}} \) is well-defined (Definition 5.1).

As \( int(i) = \{ e \} \), it is simply sufficient to suitably extend the action set of \( P_{\{\varphi,i\}} \) for it to become a closure environment for \( A_i \).

Observation 10. Property agents vs. tableaux.

Neither \( \varphi \), \( P_\varphi \) or \( P_{\{\varphi,i\}} \) are tableaux for \( \varphi \). The reason for this is that the model checking procedure takes into consideration the Rabin condition for the DRA of \( \varphi \). The agent \( P_\varphi \) is not capable of exhibiting only those behaviours that satisfy \( \varphi \).

This is reflected by the fact that

\[
lang(P_{\{\varphi,i\}}) = \{ \{ a_e \in Act_e | \exists e_i \in E_i, (e, a_e) \in e_i \} \cup \{ a_e \in Act_e | a_e \in cl(\varphi) \} \cup \{ \tau \} \}^\omega
\]
As we will see in the coming section, it is the act of performing model checking whilst taking the Rabin condition into consideration that ensures we only quantify over the behaviours of $e$ consistent with $\varphi$.

5.3.2.2 Construction of $P_{\varphi,i}$

For an $e$-local A-LTL formula $\varphi$, we first construct the non-deterministic Büchi automaton $B_\varphi = \langle \text{Act}_e, S, S_0, \delta, A \rangle$, such that $L(B_\varphi)$ are the paths that satisfy $\varphi$. As $\varphi$ is $e$-local, the actions in $cl(\varphi)$ are a subset of $\text{Act}_e$.

We determinise the NBA $B_\varphi$ into the DRA $A_\varphi$ by applying the subset construction and using Safra trees [Safra, 1988; Roggenbach, 2002]. As only one action from $\text{Act}_e$ can be performed per transition, the alphabet of $A_\varphi$ can be defined over $\{a_e \in \text{Act}_e \mid a_e \in cl(\varphi)\} \cup \{\tau\}$ rather than $2^{\text{Act}_e}$, where $\tau$ is used to represent an action occurring that is not specified in $cl(\varphi)$. For an action $a_e \in \text{Act}_e$ such that $a_e \notin cl(\varphi)$, $A_\varphi$ can be stimulated to run over $a_e$ by setting $a_e$ to represent the negation of all the actions that are present in the formula.

Given the DRA $A_\varphi = \langle Q, \Sigma, \delta, Q_0, \text{Acc} \rangle$ as above, we can construct the property closure environment agent $P_{\varphi,i} = \langle id, L_\varphi, l^0_\varphi, \text{Act}_\varphi, P_\varphi, E_\varphi, \emptyset, \emptyset \rangle$, where:

- $id$ is the same as $id(e)$
- $L_\varphi = Q_\varphi$
- $l^0_\varphi = Q^0_\varphi$
- $\text{Act}_\varphi = \{a_e \in \text{Act}_e \mid a_e \in cl(\varphi)\} \cup \{\tau\}$
- $P_\varphi = \{ (l_\varphi \rightarrow a) \mid l_\varphi \in L_\varphi, a \in \text{Act}_\varphi \}$
- $E_\varphi = \{ (l_\varphi, (id, a_\varphi) \rightarrow l'_\varphi) \mid l_\varphi, l'_\varphi \in L_\varphi, a_\varphi \in \text{Act}_\varphi, \delta_\varphi (l_\varphi, a_\varphi) = l'_\varphi \}$

The construction of $\text{Act}_\varphi$ is such that it contains:

- those actions occurring in $\varphi$:

  \[ \{a_e \in \text{Act}_e \mid a_e \in cl(\varphi)\} \]

- those actions occurring in $i$’s evolution protocol for the agent $e$:

  \[ \{a_e \in \text{Act}_e \mid \exists e_i \in E_i, (id(e), a_e) \in e_i, |e_i| \geq 2\} \]

- the additional action $\tau$.

The determinism of $B_\varphi$ is key to the above construction. As the evolution function of each agent should be deterministic, we also require that the transition function of $A_\varphi$ be deterministic, leading to a deterministic evolution function of $P_{\varphi,i}$.

Observation 11. Actions not $cl(\varphi)$. 

We draw attention to the fact that it might seem strange that \( \text{Act}_\varphi \) contains more actions than appear in closure of \( \varphi \). Even if the protocol function enables these actions, and the property closure environments performs them, the agent will still act as per the DRA \( \mathcal{A}_\varphi \). As such, if taking any action outside of \( \text{cl}(\varphi) \) invalidates \( \varphi \), then this will be reflected by \( \mathcal{P}_{\{\varphi,i\}} \) entering an infinite path that would not be accepted by \( \mathcal{A}_\varphi \).

For example, for the formula \( \varphi = G a \), taking any action that is not \( a \) would invalidate \( \mathcal{P}_{\{\varphi,i\}} \) “acting like” \( \varphi \). However, performing an action such as \( b \) would then place \( \mathcal{P}_{\{\varphi,i\}} \) in a state that does not appear on any run that is accepted by \( \mathcal{A}_\varphi \).

**Example 5.13. Property closure environment construction**

We now revisit the DRA from Example 5.11 for the A-LTL formula \( \varphi = F G a \), and show the construction of \( \mathcal{P}_{\{\varphi,i\}} \). While the property \( \varphi \) contains only the single proposition \( a \), we assume that the agent \( i \) (which will be composed with \( \mathcal{P}_{\{\varphi,i\}} \)) also contains the action \( b \) attributed to agent \( e \) in its evolution function.

Consequently, the consistent parts of \( \mathcal{P}_{\{\varphi,i\}} \) are as follows:

- \( L_\varphi = \{0, 1, 2\} \)
- \( l^0_\varphi = 2 \)
- \( \text{Act}_\varphi = \{a, b, \tau\} \)
- \( P_\varphi = \{(0, \{a, \tau\}), (1, \{a, \tau\}), (2, \{a, \tau\})\} \)
- \( E_\varphi = \{(1, (id, a) \rightarrow 0), (1, (id, b) \rightarrow 2), \ldots\} \)

The rest of \( E_\varphi \) would be defined to be consistent with Figure 5.8.

\( \square \)

### 5.3.3 Semantics of \( \models_\phi^{\text{CTLK}} \)

Given \( A_i, \mathcal{P}_{\{\varphi,i\}} \) and an \( i \)-local \( \forall \text{CTLK} \) guarantee \( \psi \), we now need to provide semantics to \( \models_\phi^{\text{CTLK}} \). We remind the reader that \( A^X \) is the automaton \( A \) with an initial state \( X \) (Section 3.4.1). The semantics of \( \models_\phi^{\text{CTLK}} \) are shown in Table 5.1.

To start, as \( \mathcal{P}_{\{\varphi,i\}} \) is a closure environment for \( A_i \), it is straightforward to construct the IDIS representing \( M_{\{\varphi,i\}} = A_i \parallel \mathcal{P}_{\{\varphi,i\}} \). We write \( u \) for a state of the Agent \( i \) and \( X \) for a state in the (agent encoding the) DRA for \( \varphi \). Furthermore, \( V_{\{i,\varphi\}} = V_i \) as \( \psi \) only contains propositions defined over \( AP_i \).
Table 5.1. Satisfaction of a ∀CTLK formula in $M_{(i, \phi)}$

<table>
<thead>
<tr>
<th>Expression</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_{(i, \phi)}, \langle u, X \rangle \models^{\text{CTLK}} \phi$</td>
<td>$p \in V_{(i, \phi)}(u)$</td>
</tr>
<tr>
<td>$M_{(i, \phi)}, \langle u, X \rangle \models^{\text{CTLK}} \neg \phi$</td>
<td>$p \notin V_{(i, \phi)}(u)$</td>
</tr>
<tr>
<td>$M_{(i, \phi)}, \langle u, X \rangle \models^{\text{CTLK}} \psi_1 \land \psi_2$</td>
<td>$M_{(i, \phi)}, \langle u, X \rangle \models^{\text{CTLK}} \psi_1$ and $M_{(i, \phi)}, \langle u, X \rangle \models^{\text{CTLK}} \psi_2$</td>
</tr>
<tr>
<td>$M_{(i, \phi)}, \langle u, X \rangle \models^{\text{CTLK}} \psi_1 \lor \psi_2$</td>
<td>$M_{(i, \phi)}, \langle u, X \rangle \models^{\text{CTLK}} \psi_1$ or $M_{(i, \phi)}, \langle u, X \rangle \models^{\text{CTLK}} \psi_2$</td>
</tr>
<tr>
<td>$M_{(i, \phi)}, \langle u, X \rangle \models^{\text{CTLK}} AX \psi$</td>
<td>$\text{for all paths } \pi = \langle u_0, X_0 \rangle, \ldots \text{ in } M_{(i, \phi)} \text{ s.t. } u_0 = u, X_0 = X, \pi</td>
</tr>
<tr>
<td>$M_{(i, \phi)}, \langle u, X \rangle \models^{\text{CTLK}} A[\psi_1 U \psi_2]$</td>
<td>$\text{for all paths } \pi = \langle u_0, X_0 \rangle, \ldots \text{ in } M_{(i, \phi)} \text{ s.t. } u_0 = u, X_0 = X, \pi</td>
</tr>
<tr>
<td>$M_{(i, \phi)}, \langle u, X \rangle \models^{\text{CTLK}} A[\psi_1 U^{\ell} \psi_2]$</td>
<td>$\text{for all paths } \pi = \langle u_0, X_0 \rangle, \ldots \text{ in } M_{(i, \phi)} \text{ s.t. } u_0 = u, X_0 = X, \pi</td>
</tr>
<tr>
<td>$M_{(i, \phi)}, \langle u, X \rangle \models^{\text{CTLK}} \text{K}_i \psi$</td>
<td>$\text{for all paths } \pi = \langle u_0, X_0 \rangle, \ldots \text{ in } M_{(i, \phi)} \text{ s.t. } u_0 = u, \pi</td>
</tr>
</tbody>
</table>
As we are dealing with agent local properties, which are inherently defined by the current local state of the agent (i.e., the state \( u \) in the above), we do not need to check if there is an outgoing path that is accepted when checking the satisfaction of propositional formulae. By comparison, for formulae of the kind \( K_i \), the satisfaction of a formula at a related state relies on the related state existing on a path that is accepted with respect to the assumption.

### 5.3.3.1 Overall Approach

To conclude this section, we now highlight the overall approach put forward in this section.

Suppose we have a MAS containing two agents \( A_i \) and \( A_e \), such that \( \text{int}(A_i) = \{ A_e \} \). Furthermore, we assume that \( \varphi \) is the A-LTL assumption formula for the language of \( A_e \).

We can perform verification as follows:

- To check if \( A_e \models^{\text{A-LTL}} \varphi \), we compose \( A_e \) with its universal environment \( U_e \) and with \( O_\varphi \), and search for a rejecting path in \( A_e \parallel U_e \parallel O_\varphi \).
- To verify that \( A_i \models^{\text{CTLK}} \varphi \), we compose \( A_i \) with \( P_{\varphi,i} \) and verify the \( i \)-local \( \forall \text{CTLK} \) guarantee \( \psi \), using the Rabin conditions from the DRA for \( \varphi \) to exclude any paths in \( A_i \parallel P_{\varphi,i} \) that do not satisfy \( \varphi \) and are therefore inconsistent with \( A_e \).

As we will demonstrate in the next section, we can define an extension to alternating automata to support checking formulae against the semantics for \( \models^{\text{CTLK}} \).

### 5.4 Libi Epistemic Alternating Automata

We now introduce Libi epistemic alternating automata (LEAA).

These automata are an adaptation of Libi alternating automata by Kupferman and Vardi [Kupferman and Vardi, 1995], suitably modified to support modular verification of \( \forall \text{CTLK} \) properties. We note that for efficiency and simplification purposes, we present LEAAs only for the universal fragment of CTLK. Furthermore, by only considering the universal fragment of CTLK, we need only consider automata with transient or universal states (Section 3.4.2.5); by construction, any infinite branch of a product automata for a \( \forall \text{CTLK} \) formula gets trapped in a universal set.

By extending the weak alternating epistemic automata of Chapter 4 with an additional acceptance condition (matching the Rabin condition of the assumption), we can then translate a \( \forall \text{CTLK} \) formula to a LEAA meeting the semantics shown in Table 5.1.
5.4.1 Automata Structure

We start by introducing an additional formula into the alphabet of LEAA; its alphabet is a strict extension of the alphabet for a WEAA for a CTLK formula, but also includes the additional formula \( \text{Afalse} \). We note that when constructing a LEAA, in a similar way to a WEAA, we include this additional formula in the closure of the formula. As such, any operations working on \( cl(\varphi) \) deal with an enlarged set also including \( \text{Afalse} \).

Informally, a LEAA is an extension of WEAA with two acceptance conditions: one co-Büchi, one Rabin. Formally, a LEAA is an automaton \( A = \langle \Sigma, D, Q, \delta, Q_0, A_t, \alpha, \beta \rangle \), where:

- \( \Sigma, D, Q, \delta, Q_0, A_t \) are as per WEAA
- \( \alpha \subseteq Q \) is the co-Büchi acceptance condition
- \( \beta \subseteq 2^Q \times 2^Q \) is the Rabin acceptance condition

We now define the acceptance of an infinite path, with respect to a LEAA with acceptance conditions of \( \alpha \) and \( \beta \). An infinite path \( \pi \) is accepted by the LEAA \( A = \langle \Sigma, D, Q, \delta, Q_0, A_t, \alpha, \beta \rangle \) iff:

- \( \inf(\pi) \cap \alpha = \emptyset \) (i.e., the path is co-Büchi accepting)
- Or, for all \( (G_j,B_j) \in \beta \) either \( \inf(\pi) \cap G_j = \emptyset \) or \( \inf(\pi) \cap B_j \neq \emptyset \) (i.e., the path is Rabin rejecting).

We make a distinction between standard Libi alternating automata (LAA) as per [Kupferman and Vardi, 2000] and LEAA as presented here: in LAA the first acceptance condition (the co-Büchi condition in LEAA) is a secondary Rabin condition that handles any subformulae in the \( \exists \)CTL fragment. As the presented modular approach does not handle formulae in \( \exists \)CTLK, we can simplify this presentation by adopting a co-Büchi condition to reject any non-accepting branches as per the LEAA for the formula (i.e., infinite branches generated by any of the \( AU \) formulae in \( cl(\varphi) \) and \( \text{Afalse} \), which do not admit accepting branches in the product automaton).

We note that this construction allows for the use of a co-Büchi condition as all other infinite branches are, by construction, accepting. LEAA do not require the Büchi part of the Rabin condition as \( A_t \) and \( K_i \) generate infinite branches that are always accepting. These branches are therefore always accepted by the co-Büchi condition when we are trapped on a infinite branch for these formulae.

5.4.2 Translation from \( \forall \text{CTLK} \) to LEAA

We now introduce a new translation from \( \forall \text{CTLK} \) to LEAA. Compared to the translation from (full) CTLK to WEAA, we embedded the ability to extend any given finite prefix of a run into a (possibly accepting) run by altering the check for validity.
on propositions. Unlike the semantics for checking a formula $\phi$ in the interpreted system $M_{(i, \psi)}$ (where it is the operators of the logic that ensure acceptance against the assumption), we ensure that propositions can be extended to infinite runs. The reason for this is that, when checking an epistemic formula with a propositional subformula, we may “jump” to a state that does not satisfy the proposition but also does not occur on a valid (w.r.t. the assumption) path. As such, if a state does not occur on any valid paths, we wish to ignore that this state does not satisfy the proposition.

The translation from a $\forall$CTLK formula to a LEAA can be seen in Table 5.2.

### Table 5.2. The transition function $\delta$ for a LEAA based on a $\forall$CTLK formula

<table>
<thead>
<tr>
<th>Expression</th>
<th>Truth Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\delta \left( p, \sigma, \bar{k} \right)$</td>
<td>$\text{true}$ if $p \in \sigma$</td>
</tr>
<tr>
<td>$\delta \left( p, \sigma, \bar{k} \right)$</td>
<td>$\text{Afalse}$ if $p \notin \sigma$</td>
</tr>
<tr>
<td>$\delta \left( \neg p, \sigma, \bar{k} \right)$</td>
<td>$\text{true}$ if $p \notin \sigma$</td>
</tr>
<tr>
<td>$\delta \left( \neg p, \sigma, \bar{k} \right)$</td>
<td>$\text{Afalse}$ if $p \in \sigma$</td>
</tr>
<tr>
<td>$\delta \left( \text{Afalse}, \sigma, \bar{k} \right)$</td>
<td>$\bigwedge_{c=0}^{k-1} (c, t, \text{Afalse})$</td>
</tr>
<tr>
<td>$\delta \left( \phi_1 \ast \phi_2, \sigma, \bar{k} \right)$</td>
<td>$\delta \left( \phi_1, \sigma, \bar{k} \right) \ast \delta \left( \phi_2, \sigma, \bar{k} \right)$, for $\ast \in {\land, \lor}$</td>
</tr>
<tr>
<td>$\delta \left( \text{AX} \phi, \sigma, \bar{k} \right)$</td>
<td>$\bigwedge_{c=0}^{k-1} (c, t, \phi)$</td>
</tr>
<tr>
<td>$\delta \left( \text{A} \phi \text{U} \phi_2, \sigma, \bar{k} \right)$</td>
<td>$\bigvee \left( \delta \left( \phi_1, \sigma, \bar{k} \right) \land \bigwedge_{c=0}^{k-1} (c, t, \text{A} \phi \text{U} \phi_2) \right)$</td>
</tr>
<tr>
<td>$\delta \left( \text{A} \phi \text{U} \phi_2, \sigma, \bar{k} \right)$</td>
<td>$\bigwedge_{c=0}^{k-1} (c, t, \text{A} \phi \text{U} \phi_2)$</td>
</tr>
<tr>
<td>$\delta \left( \text{K} \phi, \sigma, \bar{k} \right)$</td>
<td>$\text{true}$ if $\bigwedge_{c=0}^{k-1} (c, i, \phi) \land \bigwedge_{c=0}^{k-1} (c, i, \text{K} \phi)$</td>
</tr>
<tr>
<td>$\delta \left( \text{K} \phi, \sigma, \bar{k} \right)$</td>
<td>$\text{Afalse}$ if not $\bigwedge_{c=0}^{k-1} (c, i, \phi) \land \bigwedge_{c=0}^{k-1} (c, i, \text{K} \phi)$</td>
</tr>
</tbody>
</table>

A propositional check is rejecting iff the formula $\text{Afalse}$ is also rejecting. $\text{Afalse}$ is rejecting iff there exists an infinite path starting at the current state that is accepted with respect to the Rabin acceptance condition of the assumption. In contrast, $\text{Afalse}$ is accepting at a given state iff all paths leaving the current state are rejecting with respect to the Rabin condition for the assumption.

**Observation 12.** Model checking A-LTL formulae using observers.
When performing modular reasoning, it follows that we do not wish to falsify any part of the product automaton if that part of the automaton does not satisfy the assumption. In contrast, accepting a true value from a part of the model that does not adhere does not alter the result: formulae in $\forall\text{CTL}_K$ consider all successors. As such, as long as the paths that satisfy the assumption are accepting, accepting a non-rejecting branch does not affect the overall acceptance of the whole product automaton.

**Example 5.14. Knowledge and Assumptions**

In Figure 5.10, imagine that the only state labelled with $p$ is $w_1$, and, therefore, that $w_2$ and $w_3$ both satisfy $\neg p$. In addition, suppose that $w_2 \models \text{Afalse}$, i.e., that $w_2$ has no outgoing paths that are accepted by the assumption, and that $w_3 \not\models \text{Afalse}$ (i.e., it exists on some accepting path of the assumption).

![Fig. 5.10. Invalid Runs and Epistemic Formulae](image)

Next, imagine that the transition function for $K_i$ did not include $\text{Afalse}$, as per the translation to WEAAs (Table 4.1). To verify if $w_0 \models \text{AX} K_1 p$, it follows that we would need to check if both $w_1$ and $w_2$ satisfy $K_1 p$. This in turn requires the technique to verify if $w_1 \models p$, $w_2 \models p$ and $w_3 \models p$, as $w_1 \sim_1 w_1$, $w_2 \sim_1 w_2$ and $w_2 \sim_1 w_3$.

We immediately see that $w_2 \not\models p$, but following Table 5.2, as $w_2 \not\models \text{Afalse}$, we return true for $w_2 \not\models p$. It also follows that $w_1 \models p$ as $w_1$ is indeed labelled $p$.

However, the interesting case is $w_3 \models p$. In this instance, we have that $w_3 \not\models p$ and $w_3 \not\models \text{Afalse}$. This would mean that, following the translation for $K_i$ in WEAAs (Table 4.1), we should return false here, and therefore we would have $w_0 \not\models \text{AX} K_1 p$.

 Nonetheless, this is not a favourable result as the state $w_2$ does not appear on a path satisfying the assumption (the infinite path $w_0, (w_2)\omega$ is rejecting per the assumption).

Using the translation as specified in Table 5.2, we can see that, when the check $w_2 \models K_1 p$ returns false, we need to then check if $w_2 \models \text{Afalse}$. Given that $w_2 \models \text{Afalse}$ (i.e., it has no outgoing assumption valid paths), we return true for $w_2 \models p$, and so we therefore also return true for $w_2 \models K_1 p$; consequently this means that $w_0 \models \text{AX} K_1 p$ returns true as expected.

$\triangle$
5.4.3 Correctness of LEAA-Based Model Checking

To show correctness of the translation, we begin by considering the operators in $\forall CTLK$ that can induce infinite branches in the product automaton:

- $A U$
- $K_i$
- $AU$
- $A false$

We note that an infinite branch of $A U$ or $K_i$ should be accepting as per WEAAs. However, it is the formulae $AU$ and $A false$ that admit infinite branches that are rejecting in the product automaton.

If we take $\beta$ as the Rabin condition for the assumption $\psi$ and $\alpha$ to be the set of formulae $A U \in cl(\phi) \cup A false$, then it is easy to show a branch of the product automaton where $A U$ appears infinitely often (or $A false$) should be rejecting only if that path is accepting with respect to the Rabin condition of the assumption. These are paths that could appear in any composition satisfying $\psi$, and therefore it is valid for the product automaton to be rejecting.

The converse is more apparent: we do not wish to reject a branch of the product automaton that corresponds to an infeasible path according to the assumption. For example, a $A U$ formula may not hold on a given path that does not satisfy $\psi$. As it does not satisfy $\psi$, such a branch could never appear in a product automaton for an IDIS that does satisfy $\psi$ and therefore the “falsity” of this formula can be ignored.

For the actual model checking procedure, we use a direct result of [Vardi, 1995]:

**Proposition 5.4.** Given an $A$-LTL assumption $\psi$, a $\forall CTLK$ guarantee $\phi$ and the agent $A_i$, we can construct a product $Libi$ alternating automaton $A_{\{A_i, \psi, \phi\}} = M_{\{\sigma\}} \times A_{\phi}$, where $A_{\phi}$ is the LEAA for $\phi$, such that $A_i \models^{CLTK}_\psi \phi$ iff $\mathcal{L}(A_{\{A_i, \psi, \phi\}})$ is non-empty [Vardi, 1995].

**Proof.** Trivial via extension of Theorem 4.2 and Proposition 3.3 in [Kupferman and Vardi, 2000], via a reduction to the fair model checking problem [Kupferman and Vardi, 2000]. This is where the structure of the assumption can be embedded into the structure of the system, and then treated as a fairness condition over the model.

5.5 Compositional Model Checking of Agents

We now build upon the work set forward in the previous two sections to present a sound methodology for the compositional verification of multi-agent systems.

As we are dealing with $i$-local properties, it may not seem intuitive to reason about the knowledge an agent possesses about its own behaviour. Nonetheless, it can be
shown that introducing epistemic modalities, even over local formulae, can alter their satisfaction. We illustrate this in Figure 5.11, showing an IDIS with eight states \((w_0 \text{ to } w_7)\), such that \(w_1 \models p, w_3 \models q, w_0 \sim_1 w_4 \) and \(w_2 \sim_1 w_6\).

\[
\begin{array}{cccccc}
  & w_0 & t & w_1 & t & w_2 & t \\
\downarrow 1 & & w_4 & t & w_5 & t & w_6 & t & w_7 & t \\
\end{array}
\]

Fig. 5.11. The simple interaction-defined interpreted system model \(M\)

Consider the two formulae:

\[
\varphi = AG(p \rightarrow AX K_1 AX q)
\]

\[
\psi = AG(p \rightarrow AX AX q)
\]

The formula \(\psi\) can be obtained from \(\varphi\) by omitting the epistemic modality \(K_1\). However, in the example model above, \(M \models \psi\) but \(M \not\models \varphi\). Although along each temporal trace where \(p\) occurs we do indeed have that \(q\) occurs two transitions later, it is not the case that at every indistinguishable state one transition later that \(p\) that \(q\) holds at the next state. That is, although \(w_1 \models p\), we have that \((w_1, w_2) \in T, w_2 \sim_1 w_6, (w_6, w_7) \in T\) and \(q \notin V(w_7)\).

As another example, we could consider the formulae \(\varphi' = K_1 AF p\) and \(\psi' = AF p\). From Figure 5.11, we can immediately see that \(w_0 \models \psi'\), but as with our previous example, we also have that \(w_0 \not\models \varphi'\).

It therefore follows that for a sound compositional technique, we need to verify epistemic formulae even when verifying components in isolation.

### 5.5.1 Assume-Guarantee Reasoning for Multi-Agent Systems

Our sound, non-circular compositional rule can be seen in Figure 5.12. As per [Cobleigh et al., 2003], we can rephrase these rules into two separate model checking problems, shown in Figure 5.13. We write \([\text{true}]A_i[\psi]\) when \(A_i \models \text{A-LTL} \psi\) and \([\psi]A_i[\varphi]\) when \(A_i \models \text{CTLK} \psi \varphi\).

The first premise of Figure 5.12 states that \(A_e\) unconditionally satisfies the A-LTL formula \(\psi\). The second premise states that, when restricted only to the paths that satisfy \(\psi, A_i\) conditionally satisfies the \(\forall\text{CTLK}\) formula \(\varphi\). The consequent follows immediately, i.e., that given the two premises, the composition of \(A_e\|A_i\) satisfies \(\varphi\).
In the presented rules, we use A-LTL assumptions not to define all valid behaviours but to exclude those behaviours from $A_i$ which are inconsistent with the composition $A_e \parallel A_i$.

**Theorem 5.5. Compositional Preservation of $\forall$CTLK (Soundness)**

The assume-guarantee rule depicted in Figure 5.12 is sound for guarantees in the universal fragment of $\forall$CTLK under the condition that the composition of $A_i$ and $A_e$ is deadlock-free.

**Proof.** Assume that $A_e \models A$-LTL $\psi$ and $A_i \models \forall$CTLK $\phi$, but $A_i \parallel A_e \not\models \forall$CTLK $\phi$. Given that $A_i \parallel A_e$ is deadlock-free, this means that there exists a behaviour of $A_i$ in the composition that was assumed to be invalid as it did not validate $\psi$. For this to be the case, then there must exist a trace $\sigma$ in $A_e$ such that $\sigma \not\models \psi$. This is a contradiction as $A_e \models A$-LTL $\psi$. By reductio ad absurdum, we have that $A_e \not\models A$-LTL $\psi$, and so the application of AGR-MAS cannot guarantee that $A_i \parallel A_e \models \forall$CTLK $\phi$.

Clearly, the inference rules presented are not complete. Should compositional verification state the guarantee is invalid, then this has no consequence upon the validation of the formula on the full model. This is not surprising as our assumption formula only acts to guide the verification along those paths satisfying $\phi$; these still might be a superset of $\text{lang}(A_i)$, and therefore allow more behaviours in $A_i$ than satisfy $\phi$.

We now show the correctness of the whole procedure, which includes utilising the verification procedures presented in Section 5.3 and Section 5.4.

**Proposition 5.5. Correctness of Model Checking under Assumptions**

For two agents $A_e$ and $A_i$, an A-LTL property $\psi$ and an i-local $\forall$CTLK formula $\phi$ such that $A_e \models A$-LTL $\psi$ and $A_i \models \forall$CTLK $\phi$, then for all closure environments $C_e' = C_e \setminus A_i$, $A_e \parallel A_i \parallel C_e' \models \forall$CTLK $\phi$.

**Proof.** We work from top to bottom, using with the assumptions 1) $A_e \models A$-LTL $\psi$; and 2) $A_i \models \forall$CTLK $\phi$.

$A_e \models A$-LTL $\psi$ (Assumption 1)
⇒ We have that \( A_e \parallel U_e \parallel O_\psi \) contains no rejecting paths (Theorem 5.4)

⇒ \( A_e \parallel U_e \models \psi \), by the construction of \( O_\psi \)

⇒ \( A_e \parallel C_e \models \psi \) for all closure environments \( C_e \), given that for any \( C_e \), \( \text{lang}(C_e) \subseteq \text{lang}(U_e) \) by Proposition 5.2

⇒ behaviours \( (A_i || A_e) \subseteq \text{behaviours}(A_i || P_{\{\psi, i\}}) \)

\[ A_i \models_{\psi}^{\text{CTLK}} \varphi \] (Assumption 2)

⇒ \( A_i || P_{\{\psi, i\}} \models \varphi \) (Section 5.3.3)

⇒ As \( \text{lang}(A_i || A_e) \subseteq \text{lang}(A_i || P_{\{\psi, i\}}) \), and behaviours \( (A_i || P_{\{\psi, i\}}) \) satisfy \( \varphi \), we have behaviours \( (A_i || A_e || U_{A_e}) \) satisfy \( \varphi \) (via Assumption 1, Assumption 2 and Proposition 5.1)

⇒ \( A_i || A_e || C'_e \models \varphi \), for any \( C'_e = C'_e \setminus A_i \)

Finally, as the composition of \( A_i \) and \( A_e \) with a closure environment for \( C'_e \) excluding \( A_i \) is a valid closure, we have that \( A_e \parallel A_i \parallel C'_e \models_{\text{CTLK}} \varphi \).

5.5.1.1 Extension to the Multi-Agent Case

An extension of the inference rules from Figure 5.12 when applied to the \( n \)-agent case can be seen in Figure 5.14. This rule states that, for any composition that contains \( A_i \) and \( A_e \) such that \( A_i \) and \( A_e \) validate AGR-MAS (Fig. 5.12), the introduction of additional agents does not invalidate the formula.

The soundness of the rule follows directly from Theorem 5.5, as introducing any new agents to a composition can only further eliminate behaviours from \( i \) and not introduce new ones.

By Theorem 5.1, for any composition \( X \), if \( A_e \models_{\text{LTL}}^{\land} \varphi \) then \( A_e \parallel X \models_{\text{LTL}}^{\land} \varphi \). Therefore if \( \text{int}(i) = e \) and \( [\varphi] A_i [\psi] \), we also have \( A_i \parallel A_e \parallel X \models_{\text{CTLK}} \varphi \).
5.6 Concluding Remarks

In this chapter we have presented a sound technique for assume-guarantee-based compositional reasoning for multi-agent systems. Our approach is such that should the compositional rules yield a satisfiable result, then the formula does indeed hold on the full composition of the components. However, an unsatisfiable result does not give us a definitive result about the system.

We have put forward the use of action-based linear-time assumptions. Our assumptions are used not to constrain the component under test to only those paths that appear in the full composition, but to allow for the specification of an expected set of behaviours that are present along all valid interactions between an agent and its environment. Such a specification can then be used to exclude paths that are known to be inconsistent in the full composition when performing verification of an individual component.

Additionally, we have presented two model checking-based approaches for both parts of the assume-guarantee rules (the assumption and the guarantee) based on alternating automata.

This chapter presented, to the best of our knowledge, the first approach for assume-guarantee reasoning for multi-agent systems when checking formulae in temporal-epistemic logic. While other approaches from the literature use action-based specifications for both parts of the assume-guarantee rule, we note our approach is novel in that it uses action-based specifications for the assumption-only.

By translating the assumption to a deterministic Rabin automaton, and then composing this with the component under test, it is possible to use a natural extension to the automata-theoretic approach of Chapter 4 to verify this component against a temporal-epistemic guarantee.

In the next chapter, we implement the approach for synthesising observers and environments, as well as the overall verification approach. This approach is then evaluated experimentally on a selection of examples.
Chapter 6
Implementation and Evaluation

In the last two chapters, we looked at two novel approaches for the verification of either a multi-agent system as a whole (Chapter 4) or an individual agent (Chapter 5) against a temporal-epistemic formula. We now discuss the implementations of these techniques, and evaluate their effectiveness using a selection of benchmarks from the literature.

6.1 Introduction

In this chapter we present implementations of the theory presented in this thesis, developed to experimentally evaluate both the underlying theory and our overall research hypothesis (Section 1.2.1). Namely, we present four experimental toolkits:

- **etav** – epistemic tree automata verifier – a new, explicit-state model checker designed to verify explicitly-defined interpreted systems against CTLK formulae using WEAs
- **dra2ispl** – deterministic Rabin automata to interpreted systems programming language – an extension to LTL2DSTAR to support the generation of property observers and property agents from A-LTL formulae
- **at-mcmas** – automata-theoretic model checking multi-agent systems – an extension to MCMAS-1.0 to support automata-theoretic verification, *a la etav*
- **agr-mcmas** – assume-guarantee model checking multi-agent systems – an extension to AT-MCMAS implementing the modular approach as presented in Chapter 5

After introducing the specifics of each of the tools, we demonstrate some benchmarks for the correctness and efficiency of the implementations.

We begin with an introduction to **etav** in Section 6.2, a custom model checker developed to demonstrate the theory as presented in Chapter 4. As **etav** employs a explicit-state modelling language (i.e., a complete specification of the underlying...
interpreted system rather than each agent, as per ISPL, c.f., §3.5.3.2), we present results for \texttt{ETAV} separately as part of Section 6.2.2.

We then introduce the three remaining experimental extensions that form the core of the modular approach: \texttt{dra2isp1} in Section 6.3.2 and \texttt{at-mcmas/agr-mcmas} in Section 6.4. Finally, in Section 6.5, we compare \texttt{mcmas-1.0}, \texttt{at-mcmas} and \texttt{agr-mcmas} using existing benchmarks from the literature.

### 6.2 An Epistemic Tree Automata Verifier

We begin by presenting an implementation of the automata-theoretic approach from Chapter 4 and the tool’s experimental evaluation (Section 6.2.2).

#### 6.2.1 Implementation: ETAV

We have implemented the automata-theoretic technique approach as presented Chapter 4 in C++ as a new explicit-state model checker called \texttt{ETAV} (Epistemic Tree Automata Verifier). Currently, \texttt{ETAV} only supports models specified directly as Kripke structures, with all relations explicitly constructed, i.e., the full state space has to be enumerated prior to verification. An open source, GNU GPL-licenced release of \texttt{ETAV} is available [Jones, 2011].

In the following, we use $X_{↓n}$ to represent the $n$-th element of the tuple $X$. Additionally, we use $\bot$ and $\top$ to represent the evaluation of a node in an AND/OR graph to either \texttt{true} or \texttt{false} respectively [Kupferman et al., 2000]. We use $\wp(A)$ to represent the power-set of $A$.

#### 6.2.1.1 Approach

The approach taken by \texttt{ETAV} is to perform depth first construction of the product automaton $A_{P,\psi}$ for the interpreted system, $P$, and the CTLK formula, $\psi$, constructed as an AND/OR graph, interleaved with checking the non-emptiness of the tree. If it can be decided that the tree is accepting (or rejecting) without constructing the full product automaton, \texttt{ETAV} will return this result early and save on unnecessary computation.

The crux of \texttt{ETAV}’s construction of product automaton relies upon the following structures:

- $\textit{visited} : \text{Formula} \times \text{World} \rightarrow \text{Bool}$
- $\textit{eval} : \text{Formula} \times \text{World} \rightarrow \{\top, \bot\}$
- $\textit{path} : (\text{Formula} \times \text{World})^+$
6.2 An Epistemic Tree Automata Verifier

- \( tf : \text{Formula} \rightarrow (\text{Formula})^+ \times \text{Node Type} \times \{d \cup \bigcirc\}, d \in \{\wp(A) \setminus \emptyset\} \cup t \)

For clarity, we write \( tf(\varphi)_1 : (\text{Formula})^+ \) for the subformulae of \( \varphi \); \( tf(\varphi)_2 : \text{Node Type} \) for the node type (to be discussed); and \( tf(\varphi)_3 : \{d \cup \bigcirc\} \) for the “direction” (e.g., either an epistemic direction or a temporal one).

**The visited data structure.**

The data structure \( \text{visited} \), implemented using \texttt{std::multiset} [Plauge et al., 2000], holds a set of nodes of \( \mathcal{A}_\mathcal{P}_\chi \) visited on a certain path. If a newly constructed node in the product automaton is already in \( \text{visited} \), then a cycle has been detected. Once a path in the tree reaches a node that evaluates to either \( \bot \) or \( \top \), or completes a cycle, that node is removed from \( \text{visited} \) and added to \( \text{eval} \) along with the calculated valuation (i.e., \( \top \) or \( \bot \)).

**The eval data structure.**

In a similar way, \( \text{eval} \), implemented with \texttt{std::map} [Plauge et al., 2000], records the evaluation of previously seen nodes. This saves re-evaluating a formula at a given world, or searching for a cycle when one has already been detected. If a node has been previously explored, it will have a definitive value; \( \text{ETAV} \) can simply reuse that value from \( \text{eval} \).

**The path data structure.**

The list \( \text{path} \) records all of the nodes on a path of the tree in the order that they appear. The acceptance of an infinite path-suffix depends upon the non-emptiness of the intersection between the states occurring in the path-suffix and the acceptance condition. The path-suffix can be created by iterating backwards along \( \text{path} \) until the cycle is found.

**The tf data structure.**

Finally, \( tf \) holds the encoding of the transition function \( \delta \). Taking inspiration from [Qian and Nymeyer, 2006], we use a simplified transition relation in which rules are labelled with \( \land, \lor, \top \) or \( \bot \), representing a node-type in the product \( \text{AND/OR} \) graph.

For a given formula \( \varphi \) in CTLK, \( tf(\varphi) \) returns a tuple containing three elements:

1. A selection of subformulae — \( tf(\psi)_1 \)
2. The type of node in the \( \text{AND/OR} \) graph — \( tf(\psi)_2 \)
3. Where to evaluate all of the subformulae — \( tf(\psi)_3 \) — an agent index, e.g., \( i \), in the case of \( K_i \), a temporal index, \( t \), for \( AX/EX \), or \( \bigcirc \)—evaluation at the current state—otherwise
To support multiple directions, i.e., a direction in \( \Gamma \subseteq A \) or \( t \) (used when constructing the transition function for either an epistemic formula or a temporal formula), \( tf \) either returns a member of \( \wp(A) \setminus \emptyset \) (the set of all non-empty groups), \( t \) or \( \top \). We use the first to locate (possibly many) \( R_i \) (e.g., when translating a group formula) and the second to locate \( R_t \). The final entry \( \top \) is used to represent evaluation at the current state (e.g., in the case of atomic propositions, disjunction or conjunction).

The rules for constructing \( tf \) are shown in Table 6.1. We note that propositions (and their negations) are a special case; the node in the AND/OR graph only evaluates to, e.g., \( \top \) for the formula \( p \) iff the current state in the run is labelled with \( p \) (similarly, \( \bot \) for \( p \) if the state is not labelled with \( p \)).

Table 6.1. The rules for constructing the data structure \( tf \) for a formula \( \psi \).

<table>
<thead>
<tr>
<th>( \psi )</th>
<th>( tf(\psi)_{\downarrow 1} )</th>
<th>( tf(\psi)_{\downarrow 2} )</th>
<th>( tf(\psi)_{\downarrow 3} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p )</td>
<td>( \top ) iff ( p \in \sigma )</td>
<td>( \bot ) iff ( p \notin \sigma )</td>
<td>( \top ) if ( p ) is not labelled with ( p )</td>
</tr>
<tr>
<td>( \neg p )</td>
<td>( \bot ) iff ( p \notin \sigma )</td>
<td>( \top ) if ( p ) is not labelled with ( p )</td>
<td>( \bot ) if ( p ) is not labelled with ( p )</td>
</tr>
<tr>
<td>( \varphi_1 \land \varphi_2 )</td>
<td>( { \varphi_1, \varphi_2 } )</td>
<td>( \top )</td>
<td>( \bot )</td>
</tr>
<tr>
<td>( \varphi_1 \lor \varphi_2 )</td>
<td>( { \varphi_1, \varphi_2 } )</td>
<td>( \bot )</td>
<td>( \bot )</td>
</tr>
<tr>
<td>( AX \varphi )</td>
<td>( { \varphi_1 } )</td>
<td>( \land )</td>
<td>( \top )</td>
</tr>
<tr>
<td>( EX \varphi )</td>
<td>( { \varphi_1 } )</td>
<td>( \lor )</td>
<td>( \top )</td>
</tr>
<tr>
<td>( A[\varphi_1 U \varphi_2] )</td>
<td>( { \varphi_2, \varphi_1 \land AX \varphi } )</td>
<td>( \lor )</td>
<td>( \bot )</td>
</tr>
<tr>
<td>( E[\varphi_1 U \varphi_2] )</td>
<td>( { \varphi_2, \varphi_1 \land EX \varphi } )</td>
<td>( \lor )</td>
<td>( \bot )</td>
</tr>
<tr>
<td>( A[\varphi_1 \top \varphi_2] )</td>
<td>( { \varphi_2, \varphi_1 \land AX \varphi } )</td>
<td>( \land )</td>
<td>( \bot )</td>
</tr>
<tr>
<td>( E[\varphi_1 \top \varphi_2] )</td>
<td>( { \varphi_2, \varphi_1 \land EX \varphi } )</td>
<td>( \land )</td>
<td>( \bot )</td>
</tr>
<tr>
<td>( K_i \varphi )</td>
<td>( { \varphi } )</td>
<td>( \land )</td>
<td>( G )</td>
</tr>
<tr>
<td>( \overline{K_i} \varphi )</td>
<td>( { \varphi } )</td>
<td>( \lor )</td>
<td>( G )</td>
</tr>
<tr>
<td>( E_G \varphi )</td>
<td>( { \varphi } )</td>
<td>( \land )</td>
<td>( G )</td>
</tr>
<tr>
<td>( E_G \varphi )</td>
<td>( { \varphi } )</td>
<td>( \lor )</td>
<td>( G )</td>
</tr>
<tr>
<td>( C_G \varphi )</td>
<td>( { \varphi \land C_G \varphi } )</td>
<td>( \land )</td>
<td>( G )</td>
</tr>
<tr>
<td>( \overline{C_G} \varphi )</td>
<td>( { \varphi \lor \overline{C_G} \varphi } )</td>
<td>( \lor )</td>
<td>( G )</td>
</tr>
</tbody>
</table>

Construction of the product automaton.

When \( |tf(\varphi)_{\downarrow 1}| \in \{ \wp(A) \setminus \emptyset \} \cup \{ \top \} \) (i.e., it is not evaluation at the current state), we have \( |tf(\varphi)_{\downarrow 1}| = 1 \), otherwise \( |tf(\varphi)_{\downarrow 1}| = 2 \) (as is the case for non-temporal-epistemic formulae). This is due to the fact that our simplified transition relation only has \( AX \),
6.2 An Epistemic Tree Automata Verifier

EX or an epistemic modality; all of these are unary operators and therefore only have a single subformula. Only an evaluation at the current state (☐) has two operands (e.g., in the case of ∧ or ∨).

If the number of successors in $R_i$ or $R_t$ for a given world is greater than two, the successors are iterated over and the AND/OR node is constructed in the intuitive manner (i.e., a disjunction or conjunction—depending on the node type—is constructed as a tree over all successors). For example, if $φ = E [φUψ]$, then $tf (φ)$ returns the tuple $⟨(ψ, φ ∨ EXφ), ∧, ☐⟩$. This means that the current state must satisfy the conjunction of $ψ$ and $φ ∨ EXφ$. It follows that $tf (Kiφ) ↓3 = i$, $tf (CΓφ) ↓3 = Γ$ ($Γ ⊆ A$) and $tf (AXφ) ↓3 = t$ (c.f., Table 6.1). When $φ$ is an atomic proposition, we have $tf (φ) ↓2 ∈ \{⊤, ⊥\}$.

The depth first construction is called recursively for all elements in $tf(φ) ↓1$ until $tf(φ') ↓2 ∈ \{⊤, ⊥\}$, for $φ' ∈ cl (φ)$. This result is then stored in $eval$ and is also used to label the current node in the AND/OR graph of the product automaton. Eventually, the procedure returns with the root of the AND/OR graph being labelled with $⊤$ or $⊥$.

6.2.1.2 Efficiency

$etav$ builds the product automaton in such a way that it only constructs the parts of the product automaton that are required for deciding the satisfiability of the formula. The $eval$ structure is used to remove the possibility of over computation. By storing the acceptance or rejection of a node in $eval$, $etav$ attempts to save memory by avoiding constructing another part of the product automaton for which it has already calculated the subtree.

As another step, $etav$ will only generate a sibling for a node if the current node is not sufficient to decide the acceptance of the path. For example, if one child of an $∧$-node evaluates to $⊥$, then $etav$ does not check the acceptance of the other child.

A third optimisation step implemented in $etav$ consists of constructing the transition rule for a formula only when it is required, i.e., the transition function $tf$ is not fully instantiated prior to starting the construction of the product automaton. This, in conjunction with the fact that $etav$ only constructs world-formula pairs in the product graph when reached, leads to an “on-the-fly” construction of both $A_D, ψ$ (the WEAA for the formula) and $A_P, ψ$ (the WEAA for the product automaton).

Despite this, the technique cannot be regarded as truly “on-the-fly” as the whole reachable state space for the model is known prior to verification. The indistinguishability relations used when evaluating an epistemic formula are expressed over the whole, reachable state-space. Without having computed the state-space, it is therefore not possible to correctly calculate the satisfaction of a subformula at all indistinguishable states.
6.2.2 Evaluation

We now look to evaluate the effectiveness of \texttt{etav} by considering two common scenarios: a gossip protocol (Section 6.2.2.1) and the faulty train gate controller (Section 6.2.2.2).

We do not draw a comparison between \texttt{etav}, which accepts an explicitly defined state space, and existing symbolic model checkers such as \texttt{mcmas} [Lomuscio et al., 2009]. The verification of concurrent structures, similar to those supported in symbolic model checking, is in a harder complexity class [Lomuscio and Raimondi, 2006], making a direct comparison unjustified. Indeed, symbolic model checkers such as \texttt{mcmas} [Lomuscio et al., 2009] or \texttt{nuSMV} [Cimatti et al., 1999] use implicit declarations for each agent (or, in \texttt{nuSMV}’s case, component) in the system. These component declarations are given programmatically, i.e., in a language closer to a conventional programming language than a finite state machine where each local state is explicitly defined. Under such approaches, the reachable state-space has to be calculated prior to verification. This can be done by composing the implicit component declarations (Section 3.3) and then finding the states reachable under the synchronised transition relation.

By comparison, \texttt{etav} requires the user to provide explicitly the reachable global state-space before verification even begins. It follows that if a model checker does not need to calculate the composition of the whole system and find its reachable states, then its computational task is easier. It should be noted that although the model checker \texttt{mck} [Gammie and van der Meyden, 2004] purportedly supports an explicit-state mode, the input is still given as an implicit declaration. For these reasons, we do not provide an empirical comparison between \texttt{etav} and any other tool.

Nonetheless, both approaches to verification suffer from the state-space explosion problem; it is necessary for both to hold all of the states in memory, irrespective if the states are provided upfront or must be calculated.

6.2.2.1 Gossip Protocol

Gossip, or epidemic, protocols are often used to represent the propagation of messages through large-scale distributed applications, much in the way that “gossip” disseminates through social groups, based only on periodic communication.

The central idea in gossip-based protocols is that the nodes (“participants”) in the system periodically share information (“gossip”) between a small, random subset of other nodes. The propagation of data throughout the system depends heavily upon the peers that a node chooses to communicate with. This is based upon a notion of peer sampling [Jelasity et al., 2007].

We created a rudimentary gossip-based protocol in the input for \texttt{etav}, which is parametric in the number of agents, representing nodes, in the system. Initially, each
agent possesses a unique piece of data. The aim of the protocol is for each agent to propagate its information, possibly indirectly, to every other agent.

In our gossip protocol, the “gossip” is shared between two participants if two participants non-deterministically choose to gossip with each other. If this happens, then the environment acts as the transmission medium and allows for the sharing of data (i.e., the environment is used to encapsulate the sending of data between the two participants).

If more than two agents opt to gossip at any one stage, then the environment non-deterministically chooses which pair will share information. In such an example, two agents may need to gossip many times to ensure that they both receive all the gossip in the system.

For example, if Gossipers 1 and 2 being by sharing their own secrets ($g_1$ and $g_2$), and then Gossipers 1 and 3 “gossip” (sharing $g_1$ and $g_3$), it will then be possible for Gossipers 1 and 2 to gossip again (avoiding 2 and 3 gossiping directly) such that Gossiper 2 can be informed of the gossip, $g_3$, from Gossiper 1. For a large number of gossipers in the system, this can require many iterations of “gossiping”, to facilitate all of the data (“gossip”) to permute through the system.

In our rudimentary gossip protocol (including ancillary transfer variables), the state space for an instantiation with $n$ agents is as follows: 3 agents, 14 states; 4 agents, 259 states; 5 agents, 13,647 states.

The specifications used for verifying the gossip protocol are reported in Table 6.2.

We use $\text{complete}_i$ to represent that agent $i$ holds all the gossip in the system; in what follows we refer to $G_n$ as the $n$-th gossiper (e.g., $G_1$ is the first).

<table>
<thead>
<tr>
<th>Spec #</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>$GP_1$</td>
<td>$EF (\bigwedge_{i \in A} \text{complete}_i)$</td>
</tr>
<tr>
<td>$GP_2$</td>
<td>$K_{G_1} EF (\bigwedge_{i \in A} \text{complete}_i)$</td>
</tr>
<tr>
<td>$GP_3$</td>
<td>$AG (\text{complete}<em>{G_1} \rightarrow K</em>{G_1} AF (\bigwedge_{i \in A} \text{complete}_i))$</td>
</tr>
<tr>
<td>$GP_4$</td>
<td>$AG (\text{complete}<em>{G_1} \rightarrow C</em>{\text{all participants}} AF (\bigwedge_{i \in A} \text{complete}_i))$</td>
</tr>
</tbody>
</table>

The first specification, $GP_1$, represents that there exists an execution of the protocol in which all the agents eventually learn the data. Property $GP_2$ states that the first agent knows $GP_1$ (i.e., that all the agents can learn the data). The next specification, $GP_3$, states that if one agent holds all the data, then that agent knows that all agents eventually learn the data. Finally, $GP_4$ states that if one agent holds all the data, then it is common knowledge between the participants that eventually they will all learn the data.

Specifications $GP_1$ and $GP_2$ are satisfiable on models of all sizes, while $GP_3$ and $GP_4$ are unsatisfiable for models with strictly greater than three agents, as the protocol does not ensure that all of the data will eventually reach all of the agents in larger scenarios.
Table 6.3 shows the results for verifying the gossip protocol with \( \text{etav} \). The final column, Nodes, represents the number of AND/OR nodes in the graph of the product automaton. For \( GP_2 \), we can see that increasing the number of reachable states increases the number of indistinguishable states for agent \( G_1 \). This leads to a greater number of states in the product automaton. When the specifications \( GP_3 \) and \( GP_4 \) are satisfiable (i.e., in a model where \(|A| = 3\)), a comparison shows that evaluating \( C_{\text{all_participants}} \) is more costly than \( K_{G_1} \). This is due to the fact that common knowledge leads to more epistemically indistinguishable states.

We draw attention to the fact that \( GP_2 \) has a large number of nodes for a instantiation containing 5 nodes, in comparison to the other formulae. For \( GP_1 \), this formula can be shown to be satisfied quickly, and the execution be terminated on-the-fly. Similarly, for \( GP_3 \) and \( GP_4 \), these formulae can be shown to be falsified quickly and in an on-the-fly way without constructing a large product automaton. In comparison, the knowledge part of \( GP_2 \) cannot be shown to hold in an on-the-fly manner; therefore the satisfaction of \( EF (\bigwedge_{i \in A} complete_i) \) needs to be checked for each state that is epistemically related for \( G_1 \) from the initial state. This then creates a large number of nodes in the AND/OR graph for this formula on a model containing 5 nodes.

It should be noted that the high execution time for a model with five agents arises from parsing the large, explicitly-declared state space. While constructing the product automaton is relatively quick compared to parsing the model, the larger model still requires more memory to hold the complete product automaton.
6.2.2 Faulty Train-Gate-Controller

The faulty train-gate-controller model [Jones and Lomuscio, 2010] extends the epistemic version [Hoek and Wooldridge, 2002] of the train-gate-controller model by allowing the trains to be faulty, and then get stuck in the tunnel. In the standard model, it considers \( n \) trains attempting to access a shared resource of the tunnel. When the trains do not display faults, and with a correctly functioning controller, it is not possible for more than one train to enter the tunnel at one time. That is, the controller acts as a correct arbiter between the trains, ensuring a mutual exclusion on the tunnel (i.e., there can only be one train in the tunnel at any one time).

An illustration of this model is shown in Figure 6.1.

In the faulty model, trains are extended with a counter variable which represents the number of actions that the train has performed since being last serviced (analogous to a car’s mileage counter). Once the counter exceeds a given threshold, trains can non-deterministically “break”, leading them to be stuck in the tunnel. The counter has an upper limit which, when reached, causes the trains to be serviced, resetting the counter.

The invariant (i.e., that they must hold at every state in the system) specifications in Table 6.4 for the faulty train-gate-controller have the following interpretations: \( TGC_1 \) states that when a train enters the tunnel, there exists a future time when it eventually leaves; \( TGC_2 \) represents a mutual exclusion over the model; \( TGC_3 \) means that when one train is in the tunnel, it knows that the other is not; \( TGC_4 \) states that \( Train_1 \) always knows that there is a mutual exclusion of the tunnel between the trains. All of the specifications are unsatisfiable in a model with broken trains and satisfiable on a model with working
trains; $TGC_5$ expresses that the mutual exclusion over the tunnel is common knowledge between all of the trains in the model for that sized instance.

Table 6.4. Faulty Train-Gate-Controller Specifications

| $TGC_1$ | $AG(train1\_in\_tunnel \rightarrow EF \neg train1\_in\_tunnel)$ |
| $TGC_2$ | $AG(\neg train1\_in\_tunnel \lor \neg train2\_in\_tunnel)$ |
| $TGC_3$ | $AG(train1\_in\_tunnel \rightarrow K_{Train1} \neg train2\_in\_tunnel)$ |
| $TGC_4$ | $AG(K_{Train1} (\neg train1\_in\_tunnel \lor \neg train2\_in\_tunnel))$ |
| $TGC_5$ | $AG(C_{all\_trains} (\neg train1\_in\_tunnel \lor \neg train2\_in\_tunnel))$ |

Table 6.5. Model Checking The Train-Gate-Controller

<table>
<thead>
<tr>
<th>Depth</th>
<th>Formula</th>
<th>Memory (KiB)</th>
<th>Time (s)</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$TGC_1$</td>
<td>12080</td>
<td>1.387</td>
<td>308</td>
</tr>
<tr>
<td></td>
<td>$TGC_2$</td>
<td>12084</td>
<td>1.391</td>
<td>199</td>
</tr>
<tr>
<td></td>
<td>$TGC_3$</td>
<td>12080</td>
<td>1.386</td>
<td>114</td>
</tr>
<tr>
<td></td>
<td>$TGC_4$</td>
<td>30668</td>
<td>1.986</td>
<td>298284</td>
</tr>
<tr>
<td></td>
<td>$TGC_5$</td>
<td>12080</td>
<td>1.381</td>
<td>53</td>
</tr>
<tr>
<td>6</td>
<td>$TGC_1$</td>
<td>7992</td>
<td>0.704</td>
<td>1751</td>
</tr>
<tr>
<td></td>
<td>$TGC_2$</td>
<td>7992</td>
<td>0.715</td>
<td>1118</td>
</tr>
<tr>
<td></td>
<td>$TGC_3$</td>
<td>7992</td>
<td>0.700</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>$TGC_4$</td>
<td>12988</td>
<td>0.852</td>
<td>82098</td>
</tr>
<tr>
<td></td>
<td>$TGC_5$</td>
<td>8124</td>
<td>0.698</td>
<td>901</td>
</tr>
<tr>
<td>W</td>
<td>$TGC_1$</td>
<td>9005</td>
<td>0.650</td>
<td>27822</td>
</tr>
<tr>
<td></td>
<td>$TGC_2$</td>
<td>9007</td>
<td>0.658</td>
<td>27140</td>
</tr>
<tr>
<td></td>
<td>$TGC_3$</td>
<td>9128</td>
<td>0.651</td>
<td>29401</td>
</tr>
<tr>
<td></td>
<td>$TGC_4$</td>
<td>26507</td>
<td>1.113</td>
<td>307169</td>
</tr>
<tr>
<td></td>
<td>$TGC_5$</td>
<td>42884</td>
<td>5.854</td>
<td>563027</td>
</tr>
</tbody>
</table>

In a model with two trains and a maximum counter of seven, varying the breaking counter affects the state space as follows: breaking counter of 1, states 3389; breaking counter of 6, states 2269; and in a working model with no faults, states 1877.

The verification results for this model can be seen in Table 6.5. The column Depth represents the counter value at which a fault can appear; “W” represents a working model, but still with a service counter. These results demonstrate that verifying a
satisfiable invariant formula, even over a smaller state space, requires greater memory than for an unsatisfiable invariant. For satisfiable invariants, the product automaton is required to contain every reachable state.

In a similar way to $GP_2$ for the gossip protocol in the previous section, we can see that $TGC_4$ on a model with the smallest breaking counter, and therefore largest number of states, has the highest number of nodes in the product automaton. Although the formula does not hold on this model, a large product graph needs to be constructed for both the $AG$ and $K_{Train}$ operators in the formula.

For the working model, it follows immediately that verifying a satisfiable formula induces a greater number of nodes in the product graph. There are fewer nodes in the product graph for $TGC_5$ than $TGC_4$ in a broken model, as the greater number of indistinguishable states means that a state invalidating the mutual exclusion can be reached using a shorter epistemic path.

### 6.2.3 Conclusion

While it has not been demonstrated experimentally, we expect the experimental results reported with $etav$ to reflect a worse performance than to those that could be obtained with a symbolic checker, such as $mcmas$ [Lomuscio et al., 2009]. This is due to $etav$ using an explicit-state procedure, rather than the efficient symbolic approach [Burch et al., 1992] that $mcmas$ adopts. We further note that $etav$ is an early-stage prototype, and has not undergone the years of development that $mcmas$ has.

However the presented results validate the correctness of the approach and show promise as $etav$ contains none of the usual optimisations present in other explicit-state model checkers (e.g., state-level compression and bit-state hashing). Most importantly, the automata-theoretic technique was not leveraged on top of other efficient techniques, such as partial order reduction [Lomuscio et al., 2010a], that often have a ground in automata-theoretic approaches. We believe considerable gains can be achieved in this direction but this is out of scope of this thesis.

We also note that automata are usually the basis of conventional “on-the-fly” methodologies [Gerth et al., 1996], but we are not aware of similar approaches for MAS. In a similar way to the technique adopted for bounded model checking of MAS (c.f., [Penczek and Lomuscio, 2003a]), it would be possible to implement an “on-the-fly” model checking procedure for $\exists$CTLK (Section 5.2.4.1) that could explore the state-space while constructing the product graph and considering its non-emptiness. The present work may form a stepping stone in this direction.

Additionally, $etav$ currently only accepts models specified as full Kripke structures, where all the temporal and epistemic successors are explicitly defined. This further makes a comparison complicated as most other model checkers (e.g., $mcmas$ and $nuSMV$) accept declarative and implicit models of the systems to be verified. As such, an
interesting avenue to allow for comparison would be in implementing a parser for ISPL (Section 3.5.3.2) into \textsc{etav}, and then performing a direct comparison on the same input model.

However, all is not lost for the ideology of automata-theoretic model checking of MAS. As we show in Sections 6.5.1 and 6.5.2, an automata-theoretic approach (albeit based on the work of Chapter 5) can out-perform a state-of-the-art symbolic model checker.

6.3 Automatically Constructing Closure Environments

We now introduce our approach for constructing property agents and property environments from A-LTL formulae. We start with the original tool (Section 6.3.1) and our extension (Section 6.3.2); we then demonstrate examples converting an A-LTL formula into both a property agent and a property environment (Section 6.3.3).

6.3.1 About \textsc{LTL2DSTAR}

The tool \textsc{LTL2DSTAR} [Klein and Baier, 2006] was developed as a tool for converting LTL formulae to deterministic automata. The overall approach of \textsc{LTL2DSTAR} is as follows:

1. Take an LTL formula as input
2. Convert the input formula to \textsc{spin} [Holzmann, 2004] syntax and call \textsc{ltl2ba} [Gastin and Oddoux, 2001]
3. \textsc{ltl2ba} generates an NBA in the form of a “never claim” for \textsc{spin}
4. \textsc{ltl2dstar} takes this never claim, performs a selection of optimisations to the NBA, and then uses Safra’s construction [Safra, 1988; Roggenbach, 2002] to convert the (optimised) NBA to a DRA
5. The DRA is then provided to the user in a variety of formats based on a plugin architecture, supporting various output formats

To generate property environments and property agents, we have extended the \textsc{LTL2DSTAR} and created the tool \textsc{dra2ispl}. Furthermore, we have updated the existing \textsc{LTL2DSTAR} software to compile on more modern compilers.

\textsc{LTL2DSTAR} can also generate DSAs (deterministic Street automata) for a given LTL formula. However, as we only require DRAs for our approach we do not focus on this aspect of the tool.
6.3.2 Implementation: DRA2ISPL

We extended LTL2DSTAR with two new modes: `GENERATE_ISPL_AGENT` and `GENERATE_ISPL_OBSERVER`; we refer to the extension of LTL2DSTAR supporting these modes as DRA2ISPL. In the first mode, DRA2ISPL generates a property agent for $\varphi$, which the user can then compose with the agent under test; in the second mode, it generates an observer agent that the user can combine with the environment agent and a definition of a universal environment for that environment, to verify if their environment satisfies a given A-LTL assumption.

To generate a reduced interpreted systems file (i.e., one avoiding spurious transitions), we apply the restriction that only one action can occur per transition when encoding a DRA as an agent. Transitions in the DRA that contain more than one atomic proposition are ignored; empty transitions use the negation of all the actions that occur in the agent. As the transition function for a DRA is defined for all inputs (i.e., for all possible actions, the next state is uniquely defined), we note that this restriction does not affect the encoding (i.e., any transitions referring to one or more positive actions can be safely omitted).

For example, if the DRA contains a transition labelled with "$p \land q$" (i.e., the edge is triggered when the DRA observes a transition labelled with 'p' and 'q' at the same time), we can safely remove this from the translation to ISPL, as there is no way for an agent to perform two actions at once. We note that this is similar to the single action condition of [Manna and Wolper, 1984] and [Pasareanu, 2001]. However, unlike those approaches, we enforce the single action condition during the translation from automata to components, rather than including it as part of the specification (e.g., via expressing that only one action can occur as an invariant part of the assumption).

The source code for DRA2ISPL is available from [Jones, 2014b].

6.3.2.1 Workflow

The approach used to construct either a property observer (Section 5.3.1.1) or a property agent/environment (Section 5.3.2.1) is shown in Figure 6.2.

Depending on the selected mode (i.e., either `GENERATE_ISPL_AGENT` or `GENERATE_ISPL_OBSERVER`), DRA2ISPL will output either a property agent or a property observer. Creating the amalgamation of the current agent with the output of DRA2ISPL/LTL2DSTAR into a complete ISPL file is still a manual task (e.g., using a text editor).

When verifying an agent against an A-LTL assumption (i.e., to ensure if the current environment does indeed satisfy a chosen assumption), we also require the user to manually generate a universal environment for that agent prior to model checking.
When the user is performing verification of a state-based guarantee, and if the “agent under test” observes more actions than specified in the assumption, the action set of the property agent needs to be sufficiently extended (i.e., to be made into a property environment and not a property agent) to include any additional actions. For example, the assumption may state $\text{Fact}_a$, while the agent under test refers to both $\text{act}_a$ and $\text{act}_b$ in its evolution function. Therefore, to make the composition valid, the definition of the property agent must include both actions in its action set.

Lastly, if the identifier of the agent with which we interface is not “Environment”, this also is required to be changed by hand.

We further discuss and exemplify these transformations in Section 6.5.2.2.

### 6.3.3 Example

We now present an example of both $\text{LTL2DSTAR}$ and $\text{DRA2ISPL}$, by translating a given A-LTL formula into both a property agent and a property observer.

We begin by considering the formula

$$G(p \rightarrow FGq)$$

which expresses that, if the action $p$ occurs along any run, then in the future eventually the component will always perform $q$.

In the syntax for $\text{LTL2DSTAR}$, this formula would be expressed in reverse Polish notation (RPN) as

$$G \text{ i } p \text{ F } G \text{ q}$$

where “i” is used for implication. $\text{LTL2DSTAR}$ will then convert this formula into $\text{SPIN}$ syntax:

$$(\text{false}) \text{ V } ((! (p0)) || ((\text{true}) \text{ U } ((\text{false}) \text{ V } (p1))))$$
and invoke `LTL2BA` on it; `LTL2BA` will then generate a `SPIN` “never claim” representing the NBA for the given formula. The never claim output of `LTL2BA` for our chosen formula is shown in Figure 6.3; its corresponding visualisation is shown in Figure 6.4.

```

never {
  accept_init:
    if
      :: (!p0) -> goto accept_init
      :: (1) -> goto T0_S2
      :: (p1) -> goto accept_S3
    fi;
  T0_S2:
    if
      :: (1) -> goto T0_S2
      :: (p1) -> goto accept_S3
    fi;
  accept_S3:
    if
      :: (p1) -> goto accept_S3
    fi;
}
```

Fig. 6.3. The `SPIN` never claim for the NBA of the formula $G(p \rightarrow FGq)$ generated by `LTL2BA`

In Figure 6.3, it can be observed that, for compatibility with `LTL2BA`, `LTL2DSTAR` renames the propositions as follows: $p \Rightarrow p0$ and $q \Rightarrow p1$. It is also apparent that `LTL2BA` generates a “shortcut” in the automaton as the transition from state `init` to `S3` reflects that, even if the automaton has not seen $p$ yet, as long the component always performs $q$ going forward, this satisfies the original formula (i.e., if $y$ always holds, then $x \rightarrow y$ can never be false, and given that $G \phi$ implies $F G \phi$).

Furthermore, we observe that not every run we would expect to be accepting is accepting. In Figure 6.4, we can see that the infinite trace (`accept_init^* T0_S2^ω`) (i.e., a run that visits `accept_init` a finite number of times and then visits `T0_S2` an infinite number of times) would be a valid run for any sequence of actions meeting our assumption, but would also be a rejecting run. This is because `LTL2BA` was designed to generate an automata for a negation of a formula; therefore, adding more rejecting runs does not introduce any problems: the formula is only decided to be false if there exists an accepting run on the negation of the formula. Furthermore, we have that $\sigma \in L(A)$ if there exists a run that accepts it – we are not constrained that all runs must be accepting.
Finally, \texttt{LTL2DSTAR} performs Safra’s construction on the parsed \texttt{SPIN never claim}, and generates a DRA for that formula. The output of \texttt{LTL2DSTAR} in GraphViz [Ellson \textit{et al.}, 2001] \texttt{DOT} format is shown in 6.5.

We can see in Figure 6.5 that the DRA for our formula contains 5 states (labelled 0 through to 4). The two ancillary states (the oval containing “DRA” and the rectangle containing “Safra[NBA = 3]”) are not part of the DRA, and are simply additional information for the user (respectively representing that the output is a DRA and that the NBA input to Safra’s construction contained 3 states).

We now explain Figure 6.5 in more detail. For each state in the DRA, we have that:

- The top number is the state number
- The bottom number is the set of all Rabin conditions that the state appears in (where “+n” means the corresponding state is in $L_n$ and “−n” means that state is in $U_n$; if a Rabin condition is not mentioned, that state appears neither in $U_n$ nor $L_n$).

Finally, a state is grey if it is an initial state of the DRA.

As such, the generated DRA contains two Rabin pairs $\langle (L_0, U_0), (L_1, U_1) \rangle$ such that: State 0 is in $L_1$; State 1 is in $L_0$ and in $U_1$; State 2 is in $U_0$ and $L_1$; and States 3 and 4 are in $U_1$. Consequently, we have that the acceptance condition for the generated DRA:

\[
Acc = \langle \{\{1, 2\}\}, \{\{0\}, \{1, 2, 3, 4\}\} \rangle
\]
6.3 Automatically Constructing Closure Environments

6.3.3.1 Observer Agent

By invoking `ltl2dstar` with the argument

```
--output:plugin:Generate_ISPL_Observer
```

we can get `ltl2dstar` to invoke `dra2ispl`, generating a property observer for our formula. The output of `Generate_ISPL_Observer` is shown in Figure 6.6.

As shown above, the first Rabin pair in the acceptance condition of the generated DRA has an empty \( U_0 \) set; we can see this reflected in the generated ISPL as the definition of \( U_0 \) is:

\[
\mathbb{U}_0 \text{ if } (\text{Environment.state} = s_0) \text{ and } ! (\text{Environment.state} = s_0);
\]

Given this definition, it is clear that \( U_0 \) evaluates to `false` and therefore \( U_0 \) does not hold at any states in the model (i.e., the set of states that satisfy the proposition \( U_0 \), as defined in the ISPL file, is the empty set). As such, calculating the set of states where \( U_0 \) holds is empty, so is in accordance with the generated DRA.

6.3.3.2 Property Agent

Similarly, if we call `ltl2dstar` with the argument

```
--output:plugin:Generate_ISPL_Agent
```

we can get `ltl2dstar` to invoke `dra2ispl`, generating a property observer for our formula. The output of `Generate_ISPL_Agent` is shown in Figure 6.6.
Agent Environment

Vars:
state : { s_0, s_1, s_2, s_3, s_4 }
end Vars
Actions = { nop }
end Protocol
Other: ( nop )
end Protocol
Evolution:
state = s_4 if state = s_0 and !( Env.Action = p or Env.Action = q);
state = s_0 if state = s_0 and Env.Action = p;
state = s_1 if state = s_1 and Env.Action = q;
state = s_4 if state = s_1 and Env.Action = p or Env.Action = q;
state = s_1 if state = s_1 and Env.Action = q;
state = s_4 if state = s_2 and !( Env.Action = p or Env.Action = q);
state = s_0 if state = s_2 and Env.Action = p;
state = s_1 if state = s_2 and Env.Action = p or Env.Action = q;
state = s_1 if state = s_3 and Env.Action = p;
state = s_4 if state = s_3 and Env.Action = q;
state = s_4 if state = s_4 and Env.Action = p;
state = s_0 if state = s_4 and Env.Action = q;
end Evolution
end Agent
Evaluation
L_0 if Environment.state = s_1 or Environment.state = s_2;
U_0 if (Environment.state = s_0) and !(Environment.state = s_0); -- false
L_1 if Environment.state = s_0;
U_1 if Environment.state = s_1 or Environment.state = s_2
or Environment.state = s_3 or Environment.state = s_4;
end Evaluation
InitStates
Environment.state = s_3;
end InitStates

Fig. 6.6. A partial ISPL snippet for a deterministic Rabin observer agent

DRA2ISPL will generate a property agent for our formula. The output of GENERATE _ISPL_AGENT is shown in Figure 6.7.

In Figure 6.7, we can see that although the formula only referred to two atomic actions (p and q), the generated property agent also contains the \( \tau \) action other_action. A rule in the evolution function for the property agent is triggered on “other_action” if no positive propositions occur in a transition (e.g., the transition \( \neg p \land \neg q \) would become an other_action transition in the generated property agent).

6.4 Extensions to MCMAS

To evaluate the assume-guarantee-based approach of Chapter 5, we implemented two extensions to MCMAS-1.0: AT-MCMAS supporting pure automata-theoretic verification in the style of ETAV; and AGM-MCMAS—an extension of AT-MCMAS—to also verify A-LTL assumptions and CTLK formulae against assumptions expressed as DRAs.
Agent Environment

Vars:

state : { s_0, s_1, s_2, s_3, s_4 };
end Vars

Actions = { other_action, p, q };
Protocol:

Other: { other_action, p, q };
end Protocol

Evolution:

state = s_4 if state = s_0 and Action = other_action;
state = s_4 if state = s_0 and Action = p;
state = s_0 if state = s_1 and Action = q;
state = s_1 if state = s_1 and Action = other_action;
state = s_4 if state = s_1 and Action = p;
state = s_1 if state = s_1 and Action = q;
state = s_4 if state = s_2 and Action = other_action;
state = s_4 if state = s_2 and Action = p;
state = s_0 if state = s_2 and Action = q;
state = s_1 if state = s_3 and Action = other_action;
state = s_2 if state = s_3 and Action = p;
state = s_1 if state = s_3 and Action = q;
state = s_4 if state = s_4 and Action = other_action;
state = s_4 if state = s_4 and Action = p;
state = s_0 if state = s_4 and Action = q;
end Evolution
end Agent

Evaluation

L_0 if Environment.state = s_1 or Environment.state = s_2;
U_0 if (Environment.state = s_0) and !(Environment.state = s_0); -- false
L_1 if Environment.state = s_0;
U_1 if Environment.state = s_1 or Environment.state = s_2
or Environment.state = s_3 or Environment.state = s_4;
end Evaluation

InitStates

Environment.state = s_3;
end InitStates

Fig. 6.7. A partial ISPL snippet for a deterministic Rabin property agent

When we refer to the tool “MCMAS-1.0”, this is the version of MCMAS taken from the MCMAS Subversion repository at revision “r883”. Although this release was taken directly from the MCMAS Subversion, this was the development snapshot of the stable and released version of MCMAS-1.0.

The source code for both extensions (i.e., at-MCMAS and agr-MCMAS) is available from [Jones, 2014a].

6.4.1 AT-MCMAS – Automata-Theoretic MCMAS

The method for verifying a formula using the approach presented in Chapter 4 and Section 6.2.1 does not differ between etav and at-MCMAS. However, some of the necessary data structures (e.g., individual states, the transition relation, the calculation of indistinguishable states) can be performed using symbolic data structures (BDDs [Bryant, 1986; McMillan, 1992]) in at-MCMAS.
We are able to utilise directly the encoding \texttt{mcmas-1.0} uses for each agent in the system, and therefore, unlike \texttt{etav}, we can store the set of reachable states symbolically. As such, \texttt{at-mcmas} and \texttt{agr-mcmas} are both “hybrid-state” model checkers; they are neither purely symbolic nor purely explicit-state tools. The system and its agents are both encoded symbolically, while the model checking procedure (i.e., checking if the system satisfies a given requirement) uses an explicit state method (i.e., the WEAA and LEAA for the product automaton are constructed explicitly but using an implicit representation for the states of the system/its components).

In short, \texttt{at-mcmas} is a version of \texttt{mcmas-1.0}, suitably extended to support the depth-first construction of the product automaton between the state-space—calculated in the same way as \texttt{mcmas-1.0}—and the automaton for the formula, as well as for checking the non-emptiness of the product. Following in the same vein as \texttt{etav}, we adopt various approaches to make \texttt{at-mcmas} more efficient (e.g., via the on-the-fly construction of the transition relation for the formula, storing the valuations of formulae at previously seen states, terminating early where possible—for example, if one branch of an “and” node returns false).

For ease of implementation, both \texttt{at-mcmas} and \texttt{agr-mcmas} use a Büchi condition rather than a co-Büchi condition for the acceptance condition of the WEAA/LEAA for a CTLK formula. However, as justified in Chapter 5, adoption of a Büchi or co-Büchi condition does not alter the technique for formulae in \(\forall\)CTLK, as the acceptance condition can be constructed such that a path is co-Büchi accepting iff it is Büchi accepting.

### 6.4.2 Acceptance of Runs

As with \texttt{etav}, constructing the product automaton for an \(E \mathcal{U}, A \mathcal{U}\) or \(K_i\) subformula requires us to ensure that all infinite loops are accepting. Using the \texttt{path} construct as discussed in Section 6.2.1 in combination with a depth-first search we continue down a given path in the product automaton until we hit a state that has appeared previously along the current \texttt{path}. Once we have detected a loop, we can then check if it is accepting, against both the guarantee and the assumption.

While we do not explicitly implement such a procedure, searching for a loop using a stack-like structure (i.e., \texttt{path}) follows closely with the standard “nested depth-first search” as used in LTL model checking [Holzmann \textit{et al.}, 1997]. That is, after the first time we have reached a state \(s\) (via depth-first search), we then perform a second depth-first search to reach \(s\) again [Visser and Barringer, 2000; Holzmann \textit{et al.}, 1997]. Rather than have two procedures for the first and second depth-first search, we implicitly use the data structure \texttt{path}.

As previously introduced for \texttt{etav}, each entry in \texttt{path} consists of a \texttt{Formula} and a \texttt{World}. When performing modular verification, each world is further composed of a pair
6.4 Extensions to MCMAS

Algorithm 2 CHECK_PATH

Input: \( path: (state, \phi)^+ \)  
\# A path containing a loop

Input: \( F_{\text{CTLK}}: (\phi)^- \)  
\# Acceptance condition for the CTLK guarantee

Input: \( F_{\text{A-LTL}}: ((L, U))^+ \)  
\# Acceptance condition for the A-LTL assumption

Output: \{Accept, Reject, Invalid\}  
\# Acceptance of path

1: current \( \leftarrow \) path\( \rightarrow \)last : (state, \phi)
2: last \( \leftarrow \) current\( \rightarrow \)previous : (state, \phi)
3: inf\_states \( \leftarrow \) \emptyset : BDD
4: inf\_form \( \leftarrow \) \emptyset : set
5: while current \( \neq \) last do
6: inf\_states \( \leftarrow \) inf\_states \& current\( \rightarrow \)state
7: inf\_form \( \leftarrow \) inf\_form \& current\( \rightarrow \)form
8: last \( \leftarrow \) last\( \rightarrow \)previous
9: end while
10: if inf\_forms \( \cap \) \( F_{\text{CTLK}} \neq \emptyset \) then  
\# \( F_{\text{CTLK}} \) is a Büchi condition
11: return Accept
12: else
13: invalid \( \leftarrow \) true
14: for \( (L, U) \in F_{\text{A-LTL}} \) do  
\# \( F_{\text{A-LTL}} \) is a Rabin condition
15: good \( \leftarrow \) L\( \rightarrow \)encode : BDD
16: bad \( \leftarrow \) U\( \rightarrow \)encode : BDD
17: if (good \& inf\_states) \&\& (bad \& inf\_states) then
18: invalid \( \leftarrow \) false  
\# inf (path) is not rejecting against the assumption
19: end if
20: end for
21: if invalid = true then
22: return Invalid
23: else
24: return Reject
25: end if
26: end if
of states: one for the agent under test and one for the property agent. When checking the acceptance against the guarantee, we use the projection of the path down to the Formula part of the path; when checking the acceptance against the assumption, we use the projection down the local states of the property agent (i.e., the states of the DRA).

For an assumption, we iterate over the Rabin pairs declared in the ISPL evaluation block and detect if there exists an $i$ such that the $i$-th Rabin pair is accepting against the current path when projected down to the local states for the property agent.

The algorithm for deciding if a path in the product automaton is accepting or not is shown in Algorithm 2. The CHECK_PATH algorithm expects to be called with a path that is already known to contain a loop, as well as the acceptance conditions for the guarantee and the assumption. For AT-MCMAS, the CHECK_PATH algorithm will only consider the Büchi condition for the formula, and does not attempt to calculate the acceptance of a Rabin pair.

As the Rabin condition is encoded using propositional atoms in the model, this means that each entry in a Rabin acceptance condition is a pair of propositions, each of which is defined against the local states of the property agent. Given that each proposition is defined only over the local states of the property agent, this means that evaluating a proposition at a given global state implicitly takes the projection only including the property agent. Furthermore, the BDD encoding for a given part of a Rabin pair (i.e., $U_i$ or $L_i$), will only refer to BDD variables that are used to encode the local states of the DRA.

We now focus on AGR-MCMAS, and on the specific modifications required from AT-MCMAS.

### 6.4.3 Detecting Invalid Runs using Three Valued Semantics

As part of AGR-MCMAS, we require the ability to detect if a run of the product automaton is accepting or rejecting against the A-LTL assumption. To do this efficiently—as well as being able to detect if there are no accepting runs at all—we adopted a novel semantics.

As shown in Table 5.2, when checking certain formulae, we need to be able to determine if we are currently attempting to validate a formula at a state which does not have any outgoing successors that are accepting against the A-LTL assumption (i.e., determining if the state satisfies $\text{Afalse}$). It is clear that, if no out-going runs from a given state are accepting against the assumption, such a state will never occur in the full composition. As we require the model to be deadlock free, if a state has no accepting successors, this means that it is not accepting, as we disallow finite paths.

To handle this in the most user-friendly way possible, we adopt a three-valued logic to support states that do not have any outgoing assumption valid paths.
We allow for three possible values for the acceptance of a branch of the product automaton:

- \textbf{1}, if the path is accepting against the guarantee
- \textbf{0}, if the path is rejecting against the guarantee, but accepting against the assumption
- \textbf{⊥}, if the path is rejecting against the guarantee and rejecting against the assumption

These three values represent the three possible return values from the \texttt{CHECK\_PATH} algorithm (\textit{Accept}, \textit{Reject} and \textit{Invalid}) as shown in Algorithm 2.

The definition of the Boolean operators, which are used in the construction of the AND/OR graph for the product automaton, are shown in Table 6.6.

<table>
<thead>
<tr>
<th>(\land)</th>
<th>\textbf{1}</th>
<th>\textbf{⊥}</th>
<th>\textbf{0}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{1}</td>
<td>\textbf{1}</td>
<td>\textbf{0}</td>
<td>\textbf{0}</td>
</tr>
<tr>
<td>\textbf{⊥}</td>
<td>\textbf{1}</td>
<td>\textbf{1}</td>
<td>\textbf{0}</td>
</tr>
<tr>
<td>\textbf{0}</td>
<td>\textbf{0}</td>
<td>\textbf{0}</td>
<td>\textbf{0}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(\lor)</th>
<th>\textbf{1}</th>
<th>\textbf{⊥}</th>
<th>\textbf{0}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{1}</td>
<td>\textbf{1}</td>
<td>\textbf{1}</td>
<td>\textbf{1}</td>
</tr>
<tr>
<td>\textbf{⊥}</td>
<td>\textbf{1}</td>
<td>\textbf{1}</td>
<td>\textbf{0}</td>
</tr>
<tr>
<td>\textbf{0}</td>
<td>\textbf{1}</td>
<td>\textbf{0}</td>
<td>\textbf{0}</td>
</tr>
</tbody>
</table>

In Table 6.6b, we notice that the disjunction of \textbf{0} with \textbf{⊥}, results in \textbf{⊥}. As per Table 5.2, we only return invalid if the state has no outgoing successors. While the definition of the three-value semantics may appear strange in the above, taking the disjunction of \textbf{⊥} with \textbf{0} never occurs as part of constructing and calculating the acceptance of the product automaton. This can clearly be seen from the clause for \(\text{AX}_1 \text{U} \text{AX}_2\); we could never have the case where, e.g., \(\text{AX}_2\) returns \textbf{0}, while \(\text{AX}_1\) is \textbf{⊥} at the current state and all of the next states. If one clause of a disjunction returns invalid, it is never the case that any other part of the disjunct would be valid. Therefore such a result is not required in the implementation; nonetheless, we provide the result here for completeness.

We do not handle negation in this three valued logic, as negation can only occur against an atomic proposition. A negated atom may evaluate to invalid if and only if the current state has no outgoing paths that are accepted. However, in this instance, the whole subformula (i.e., both the atom and its negation) return invalid, rather than trying to negate an invalid return value. This is again highlighted in Table 5.2.

We now exemplify. Consider the case where we are checking the formula \(\text{AX} \, p\) at the state \(w_0\) in Figure 6.8.

If we imagine that \(w_1\) satisfies \(p\), then we have \(\textbf{1} \land \textbf{⊥} \land \textbf{⊥}\), which again we would like to be \textbf{1}. Furthermore, if we assume that \(w_1\) does not satisfy \(p\), then we have \(\textbf{0} \land \textbf{⊥} \land \textbf{⊥}\), which we would like to be \textbf{0}. That is, if \(w_1 \models p\), then \(w_0 \models \text{AX} \, p\) should hold irrespective as to the valuation of \(p\) at \(w_2\) and \(w_3\) (as long as those states satisfy \textit{Afalse}).
As we will demonstrate in the subsequent sections, we require a logic that does not allow the invalidation of a property in a non-accepting part of the model to impact the satisfaction of the formula in an accepting part of the model.

A simple justification of this is as follows: universal properties are all decomposed into formulae that must hold in a conjunction over the next states. As such, when this conjunction branches into an unfair part of the model, we need to “ignore” this invalid result.

### 6.4.4 State-based Verification of Guarantees

We now discuss how we have implemented the checking of state-based, agent-local $\forallCTLK$ guarantees against A-LTL assumptions.

The cases where we need to ensure we are checking a current path for validity against a given assumption are as follows:

- Formulae of the form $Au$ or $A\neg u$ (potentially) admit infinite paths, so these are easy to check using the approach of the previous sections (i.e., using Algorithm 2)
- As per the semantics of $\models_{\psi}^{\CTLK}$, formulae of the form $p$ and $\neg p$ need to be checked to ensure that the state where $p$ or $\neg p$ is evaluated has a valid extension before returning
- Any subformula $\phi$ that is either a $Ki$ formula, or exists a subformula under the modality $Ki$, needs to be checked to ensure that the state after the epistemic relation exists on an assumption valid path

While the formula $Ki\phi$ does allow for an infinite branch in the product automaton (as the transition function for $Ki$ in a WEA/LEAA includes the formula itself), we note that these paths are not temporal (i.e., it is a chain of states related by the indistinguishability relation for $i$). However, as $Ki\phi$ will appear in the acceptance condition for the whole formula, this means that we will never opt to reject an infinite branch of $Ki$. Therefore, following Algorithm 2, we will never return invalid for a $Ki$ formula (for $Ki\phi$ to return
false, this would mean that the current state is related to some other state that both falsifies $\varphi$ and exists on some assumption-valid path).

We summarise the acceptance criteria from Algorithm 2 in Table 6.7. It is once again immediate that we only reject the product automaton if the path is rejecting against the guarantee but accepting against the assumption. A path that is rejecting against both is classed as invalid.

<table>
<thead>
<tr>
<th>Guarantee Acceptance</th>
<th>Assumption Acceptance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCEPT</td>
<td>ACCEPT</td>
</tr>
<tr>
<td>REJECT</td>
<td>ACCEPT</td>
</tr>
<tr>
<td>ACCEPT</td>
<td>REJECT</td>
</tr>
<tr>
<td>REJECT</td>
<td>INVALID</td>
</tr>
</tbody>
</table>

We now consider the formula $AX\varphi$ at a state $g$, such that $g$ does not have any assumption-valid successors. It is clear that as $g$ has no assumption-valid successors, the state itself is invalid. However, any subformula of $AX\varphi$ would have eventually been checked for assumption validity. Therefore, when using the three valued semantics as presented in Section 6.4.3, the conjunction rules used to encode $AX$ would have either returned invalid (if all successors are invalid) or true (if, e.g., $\varphi$, or one of its subformulae, was $K_i\psi$ and $g$ was related to a state $g'$ that was accepting).

Example 6.1. Invalid Runs and Until Formulae

If we take the formula $\varphi = A[\varphi_1 U \varphi_2]$, such that $\varphi_1$ and $\varphi_2$ are not $K_i$ or $A U$ formulae, it then follows that the acceptance condition for $\varphi$ is the empty set. Consequently, an infinite run that does not eventually satisfy $\varphi_2$ will either be invalid (i.e., it enters an infinite run that is not admitted by the acceptance condition for the assumption) or is rejecting because the branch is accepting with respect to the assumption but rejecting with respect to the acceptance condition of the formula (i.e., the infinite set of formula on the current branch had an empty intersection with the Büchi acceptance condition).

In other words, $\varphi$ is only accepted on finite runs as $\varphi_2$ must eventually be satisfied at some state, and therefore it is implicit that $\varphi$ can never be accepting on infinite runs.

△
6.4.4.1 Detecting invalid runs

To decide if a current state occurs on an assumption valid path, we check the formula $EG \text{true}$ using the standard automata-theoretic approach based on the WEAA encoding.

In Chapter 5, we used the formula $A\text{false}$ to detect invalid paths. However, to allow for an easier approach, we check $EG \text{true}$, which returns \text{true} if and only if the current state has at least one assumption-valid path leaving it. $EG \text{true}$ has the advantage that it also supports the check for seriality, as well as the check for validity; $EG \text{true}$ is a $E\UU$ formula, so it requires at least one state to exist, unlike an $A\UU$, which may hold at a state that has no successors. The encoding of $E\UU$ formulae follows the presentation of Chapter 4, but with the checks presented so far in this section to support checking against the validity of the assumption (i.e., with the additional checks for assumption validity for any subformulae of the form $K_i$, $p$ or $\neg p$).

A check of an $EG \text{true}$ at a given state will return \text{Invalid} if none of its successors have any assumption-valid paths (and therefore the current state is also has no assumption-valid paths leaving it).

Using the previously defined acceptance criteria, checking $EG \text{true}$ resolves to finding a loop in the model that meets the acceptance condition of the assumption. However, unlike the \text{CHECK\_PATH} algorithm presented so far, we remove the need to check against the acceptance condition for the guarantee (as $EG \text{true}$ is always true on deadlock-free models), and we are only concerned with acceptance against the assumption.

We note that the case of \text{rejecting} would never be met: the formula $EG \text{true}$ cannot be false on transition systems with a serial relation (i.e., the model has no deadlock). As such, checking $EG \text{true}$ either returns \text{accepting} (if the state has at least one path that is accepting starting from it) or \text{invalid} (if there are no accepting paths starting at the state).

6.4.4.2 Verification Approach for $\forall$CTLK guarantees

To summarise, inside \texttt{agr-mcmas}, we implement the following high-level approach when checking a guarantee:

1. Construct the full reachable state-space for the composition of the agent under test and the property environment for the assumption
2. Build the product automaton on-the-fly between the system, the assumption and the guarantee, while also checking for acceptance
3. If an infinite loop is found in the product automaton:
   * Check against acceptance condition for the formula (only $K_i$ and $A\UU$ branches are allowed)
   * Check against the DRA if rejected against the formula
4. If the current subformula is $p$, $\neg p$ or the parent formula is $K_i$, spawn a check for $EG \text{true}$ that tries to find at least one accepted path outwards. If there are no such successors, and the subformula returns $reject$, then return $invalid$.

### 6.4.5 Action-based Verification of Assumptions

We now need to present a methodology for checking if a given agent satisfies an A-LTL assumption. Following the standard automata-theoretic approach to verifying LTL, we know that the following are equivalent:

$$A \models ^{A\text{-LTL}} \phi \iff \exists \pi \in \Pi, \neg \text{NBA}_\phi(\pi) \text{ is accepted}$$

$$iff \forall \pi \in \Pi, \text{DRA}_\phi(\pi) \text{ is accepted}$$

$$iff \text{not the case that } \exists \pi \in \Pi, \text{DRA}_\phi(\pi) \text{ is rejected}$$

where $\Pi$ is the set of paths in the system for $A \parallel U_A$.

To verify if a given agent $i$ satisfies an A-LTL assumption $\phi$, we compose the agent with the property observer for $O_\phi$ and its universal environment $U_i$, and then attempt to find a path that is rejected by the observer.

#### 6.4.5.1 Hunting for Invalid Paths

Similar to checking for assumption valid paths using $EG \text{true}$, we use the formula $AG \text{true}$ to ensure that all paths in the product of $A_i \parallel O_\phi \parallel U_i$ are accepted by $A_\phi$.

Checking the formula $AG \text{true}$ ensures that we check all paths through the model. We start with $AG \text{true} = A [true \parallel true]$, and therefore the transition function for $AG \text{true}$ expands to:

$$true \land (false \lor AX AG \text{true})$$

We therefore see that verifying $AG \text{true}$ requires us to check if all infinite paths in the model are accepting against the Rabin condition for the automaton, as it is never possible to satisfy $false$. For the check for $EG \text{true}$, we use a second variant of Algorithm 2 that does not consider the acceptance $AG \text{true}$ (as it is always accepting) and returns $Reject$ if we find a loop in the model for which no Rabin pair in the assumption accepts.

#### 6.4.5.2 Verification approach for A-LTL assumptions

To summarise, inside $agr\text{-mcmas}$, we implement the following high-level approach when checking an A-LTL assumption:
1. Compose the agent we wish to verify with the property observer agent $O_\varphi$
2. “Complete” the model by also composing $A_i \| O_\varphi$ with $U_i$
3. Using the formula $AG \text{true}$, search for any infinite paths in the model that does not meet the acceptance condition for DRA of $\varphi$:
   - As soon as a path is found that is rejecting, return $\text{Reject}$ and terminate without continuing to construct the remaining parts of the product automaton

We reinforce that, while the construction of the product automaton is on-the-fly, the construction of the state-space is not. One possible extension when only checking temporal formulae (e.g., as is the case when checking $AG \text{true}$ during A-LTL checking), would be to adopt a purely on-the-fly verification approach, which would construct the set of reachable states while building the product automaton.

### 6.4.6 Implementation: AGR-MCMAS

In combination with Figure 6.2, Figure 6.9 shows the workflow of using AGR-MCMAS.

![Fig. 6.9. Workflow using agr-mcmas](image)

In can be seen in Figure 6.9 that the user takes two ISPL files, one defining the “agent under test” and the other the output from $\text{dra2ispl}$, and amalgamates these by hand into the input AGR-MCMAS. Following the modular presentation of IDIS, and the syntax of ISPL, this is a simple task, and could easily be automated.

In the implementation, AGR-MCMAS operates in two modes:

- **Guarantee Mode.** This follows the approach presented in Section 6.4.4 to check if an agent satisfies a given guarantee with respect to a translated assumption
- **Assumption mode.** This follows the approach of the previous section (Section 6.4.5) for verifying if a given agent satisfies an A-LTL assumption

In the current implementation, the first mode is the default (and requires no arguments), while checking an A-LTL assumption requires passing the “--ltl 1” flag on
the command-line. Depending on the mode provided, \texttt{AGR-MCMAS} will either follow Section 6.4.4 or Section 6.4.5.

Finally, \texttt{AGR-MCMAS} has three return values:

- \textbf{ACCEPT} – if either $A \models_{\text{CTLK}} \varphi$ or $A \models_{\text{A-LTL}} \psi$

- \textbf{CTLK-REJECT} – if it is not the case that $A \models_{\text{CTLK}} \varphi$

- \textbf{A-LTL-REJECT} – if it is not the case that $A \models_{\text{A-LTL}} \psi$

Now that we have presented the necessary preliminaries behind the implementation of the techniques we compare \texttt{AT-MCMAS}, \texttt{AGR-MCMAS} and \texttt{MCMAS-1.0} in the following section.

6.5 Evaluating Modular MCMAS

We now evaluate the effectiveness of the modular implementation \texttt{AGR-MCMAS}, when compared to both \texttt{MCMAS-1.0} and \texttt{AT-MCMAS}.

Where possible, we compare a number of metrics. However, one of the deciding factors will be the number of reachable states that need to be calculated and stored for each technique. Additionally, we also wish to compare (where possible) the time and memory required for each technique while analysing the same scenario.

For the automata-based approaches, we can also compare the number of nodes in the product automaton between the modular and the monolithic techniques. Such a comparison is not possible for \texttt{MCMAS-1.0}, as it uses the set-based approach to construct the set of states satisfying the formula, rather than building a product automaton between the formula and the model.

We evaluate the techniques on two examples: the not-so-faulty train-gate-controller (Section 6.5.1) and an industrial-focused software development protocol (Section 6.5.2).

6.5.1 Not-so-faulty Train-Gate-Controller

We begin by revisiting the train-gate-controller example from Section 6.2.2.2. As previously, the controller acts as an arbiter, attempting to ensure that there is a mutual exclusion over the tunnel (i.e., that only one train is in the tunnel at any one time).

We illustrate the interactions between a model with two trains and one controller in Figure 6.10. This figure illustrates that \textit{Train}_1 communicates only with the \textit{Controller}, and similarly that \textit{Train}_2 communicates only with the \textit{Controller}. As such, we consider the modular verification of a train against its controller “environment” (i.e., the agents inside the shaded yellow rectangle in Figure 6.10).
As with the evaluation for $\texttt{etaV}$ in Section 6.2.2.2, we consider trains that are enriched with a counter. In the faulty model, this counter is used to store an “error threshold” value that, if exceeded, allows the trains to exhibit a non-deterministic fault. However, as the technique of Chapter 5 is incomplete (i.e., if the guarantee check returns a false, we cannot derive a definitive answer about the monolithic check), using broken trains does not permit a useful evaluation. Consequently, we still consider trains with counters, but with the exception that the faults can never occur.

In Figure 6.11, we provide a sample ISPL agent representing a non-faulty train with a maximum counter of 10 (i.e., the variable $\texttt{counter}$ defined in $\texttt{Train1}$’s local state ranges from 0 to 10). It can be observed from this figure that, once $\texttt{Train1}$ is in the tunnel, it stays there unless the tunnel performs the action $\texttt{leave1}$. That is, not only does the controller (in Figure 6.11 this agent is the $\texttt{Environment}$) decide if the train should enter the tunnel, but also when it should leave the tunnel. Nonetheless, it could be possible to create a train that could stay in the tunnel indefinitely, should its designer choose.

6.5.1.1 Propositions and Specifications

In the evaluation that follows, we select a train-local temporal-epistemic formula to verify. As the train-gate-controller model is effectively a mutual exclusion problem, it is sensible to consider only the following proposition:

\[ \phi = AG (\texttt{train1 in tunnel} \rightarrow AX (K_{\texttt{Train1}} (AX \neg \texttt{train1 in tunnel}))) \]

which has the reading: “if the train is in the tunnel, at the next state it knows at the following state it is out of the tunnel”.

Being able to verify if each train satisfies $\phi$ in isolation is beneficial as it allows us to give guarantees that each train has been designed correctly. It is argued that the faulty
trains of Section 6.2.2.2 are indeed faulty; their presence can lead to the possibility of the mutual exclusion on the tunnel being invalidated.

### 6.5.1.2 Modular Verification of Trains

We now move on to performing modular verification of a selected train.

To perform modular verification, we first identify the assumption that we will show the environment satisfies (i.e., where $A_e = \text{Controller}$). The assumption selected is the A-LTL formula:

$$\psi = G (\text{train}_i \text{enter} \rightarrow \text{Xtrain}_i \text{leave})$$

This formula specifies that, if the environment signals to the train to enter the tunnel, at the next state it signals to the train to leave the tunnel.
We note that, unlike approaches discussed in Section 2.1.2, generating such an assumption is still a manual process.

Figure 6.12 shows the dra2isp3l-generated property environment for \( Train_1 \) for the assumption \( \psi \). Following the presentation in Section 6.3.2, the single action condition is enforced via the translation of the deterministic Rabin automaton for the assumption into an agent.

In the context of the assumption \( \psi \), and for \( A_i = Train_1 \), we look to verify that:

\[
A_i \models_{\psi}^{\text{CTLK}} AG\left(\text{train1_in_tunnel} \rightarrow AX\left(K_{\text{Train1}}\left(AX\neg\text{train1_in_tunnel}\right)\right)\right)
\]

using \( \text{AGR-MCMAS} \). The A-LTL assumption \( \psi \) is used to constrain the behaviour of \( A_i \) to disallow the infinite path under which \( A_i \) idles permanently in the tunnel.

As such, in our evaluation, we need to perform two checks using the modular approach:

1. \( A_e \models^{\text{A-LTL}} \psi \)
2. \( A_i \models_{\psi}^{\text{CTLK}} AG\left(\text{train1_in_tunnel} \rightarrow AX\left(K_{\text{Train1}}\left(AX\neg\text{train1_in_tunnel}\right)\right)\right)\)

In what follows, we refer to the first check as the assumption check and the second as the guarantee check.

```
1 Agent Environment
2 Vars:
3 state : { s_0, s_1, s_2, s_3 };  
4 end Vars
5 Actions = { other_action, enter_1, leave_1 };  
6 Protocol:
7 Other: { other_action, enter_1, leave_1 };  
8 end Protocol
9 Evolution:
10 state = s_0 if state = s_0 and Action = other_action;
11 state = s_1 if state = s_0 and Action = enter_1;
12 state = s_0 if state = s_0 and Action = leave_1;
13 state = s_2 if state = s_1 and Action = other_action;
14 state = s_0 if state = s_1 and Action = enter_1;
15 state = s_1 if state = s_1 and Action = leave_1;
16 state = s_2 if state = s_2; -- self loop
17 state = s_0 if state = s_2 and Action = other_action;
18 state = s_1 if state = s_3 and Action = enter_1;
19 state = s_0 if state = s_3 and Action = leave_1;
20 end Evolution
21 end Agent
22 -- Eval:
23 -- L_0 if Environment.state = s_0 or Environment.state = s_1;
24 -- U_0 if Environment.state = s_2;
25 -- Init:
26 -- Environment.state = s_3;
```

Fig. 6.12. Property environment for a train
6.5 Evaluating Modular MCMAS

Comparison 1.

We begin with an exaggerated example: we assume that each train has a maximum counter of 10,000. Furthermore, we consider an environment suitably extended with its own “timeout counter” of 4, and that it can arbitrate between a total of 6 trains. This timeout counter is used to ensure that, if the environment selects a train to enter the tunnel, it will try up to \( n \) times to allow the train to enter. If after the \( n \)-th time the train does not enter, the controller will assume that the train has “timed out” and selects another train to enter the tunnel.

We begin with a straight comparison between MCMAS-1.0, AT-MCMAS and AGR-MCMAS. We refer to the checks by MCMAS-1.0 and AT-MCMAS as “monolithic verification”, as they consider the system as a whole. The memory usage and time taken for verifying the above model using the three tools is shown in Table 6.8.

### Table 6.8. Train-Gate-Controller: Comparison 1 – Memory and Time

<table>
<thead>
<tr>
<th>Mode</th>
<th>Time (s)</th>
<th>Memory (KiB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monolithic Verification DNF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Guarantee Check</td>
<td>99.987</td>
<td>489,612</td>
</tr>
<tr>
<td>Assumption Check</td>
<td>0.289</td>
<td>27,688</td>
</tr>
</tbody>
</table>

For MCMAS-1.0 or AT-MCMAS, neither tool completed monolithic verification within two hours. Furthermore, even by reducing each train’s counter from 10,000 to 200, this did not assist either tool in completing the verification task within two hours.

In light of the failures of the two monolithic tools, we can see the clear benefit of AGR-MCMAS when decomposing the verification task. As is to be expected, the guarantee check—with a counter of 10,000—generates a much larger model than the assumption check (where the controller only has a counter of four), and therefore it is the guarantee check that has the greater resource requirement (i.e., in memory and time).

As both monolithic approaches did not complete within our designated time limit, we attempt to evaluate the effectiveness of AGR-MCMAS in reducing the state-space by considering the number of BDD variables required to encode the model. This comparison is shown in Table 6.9.

From Table 6.9, we can also observe that, interestingly, checking the assumption requires one extra action variable. When performing the verification of the environment against the assumption, we are required to compose the Environment-under-test with the observer automaton for the property (i.e., the agent who can only perform the null-action \( \text{nop} \), which is unobservable by all other agents in the system), as well as the minimum “skeleton” for each additional agent that the environment communicates with. As such,
Table 6.9. Train-Gate-Controller: Comparison 1 – BDD Variables and Reachable States

<table>
<thead>
<tr>
<th>Mode</th>
<th>State Variables</th>
<th>Action Variables</th>
<th>Total Variables</th>
<th>Max. # States</th>
<th>Reachable # States</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monolithic Verification</td>
<td>100</td>
<td>17</td>
<td>217</td>
<td>$\approx 1.268 \times 10^{30}$</td>
<td>unknown</td>
</tr>
<tr>
<td>Guarantee Check</td>
<td>18</td>
<td>4</td>
<td>40</td>
<td>262,144</td>
<td>65,538</td>
</tr>
<tr>
<td>Assumption Check</td>
<td>22</td>
<td>18</td>
<td>62</td>
<td>4,194,304</td>
<td>1,665</td>
</tr>
</tbody>
</table>

it is therefore necessary to create the composition of universal agents for each other component in the model, plus the property observer with a single action. For each other component, the number of BDD variables required to encode the actions for each concrete agent vs. each skeleton is the same. However, it is necessary to allocate the null action in the observer its own BDD variable. This results in the assumption checking requiring one additional BDD action variable in comparison to the monolithic check.

The column “Max. # States” is equal to the total number of states representable for a given number of BDD variables; for example, with 10 variables, there are $2^{10} = 1024 = 1.024 \times 10^3$ representable states.

As we can see, comparing the representable state-spaces alone shows the potential reductions that assume-guarantee can bring. The maximum number of instantaneous states that need to be considered in either parts of the assume-guarantee check is 4,194,304 states; by comparison, for the monolithic approach, these techniques have to cater for the potential of analysing a state-space of that is twenty four orders of magnitude larger.

Comparison 2.

Given the inability of either monolithic tool to complete verification within the allocated time, we now consider a smaller example, which both tools can complete within our two hour time limit. We evaluate the tools on a model with three trains, each with a maximum counter of 5, an environment with a timeout of 1, and consider the same formula as previously.

Table 6.10 shows the number of BDD variables and number of reachable states considered in this second model. As both AT-MCMAS and MCMAS-1.0 construct the same set of reachable states—it is just their verification approach that differs—it is of no surprise that both tools generate the same number of states.

However, it can be immediately seen that AGR-MCMAS has to explore significantly fewer states: we see a reduction of three orders of magnitude between the modular and monolithic approaches.
Table 6.10. Train-Gate-Controller: Comparison 2 – BDD Variables and Reachable States

<table>
<thead>
<tr>
<th>Mode</th>
<th>State Variables</th>
<th>Action Variables</th>
<th>Total States</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monolithic Verification</td>
<td>23</td>
<td>10</td>
<td>40,961</td>
</tr>
<tr>
<td>Assumption Check</td>
<td>13</td>
<td>11</td>
<td>65</td>
</tr>
<tr>
<td>Guarantee Check</td>
<td>7</td>
<td>4</td>
<td>57</td>
</tr>
</tbody>
</table>

Table 6.11. Train-Gate-Controller: Comparison 2 – Memory and Time

<table>
<thead>
<tr>
<th>Mode</th>
<th>Nodes in product graph</th>
<th>Memory (KiB)</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>WEAA Monolithic (at-mcmas)</td>
<td>825,563</td>
<td>217,504</td>
<td>344.364</td>
</tr>
<tr>
<td>BDD Monolithic (mcmas-1.0)</td>
<td>N/A</td>
<td>28,100</td>
<td>0.147</td>
</tr>
<tr>
<td>Assumption Check</td>
<td>396</td>
<td>25,292</td>
<td>0.101</td>
</tr>
<tr>
<td>Guarantee Check</td>
<td>57</td>
<td>24,112</td>
<td>0.023</td>
</tr>
</tbody>
</table>

We now consider the memory and time used across the three tools (Table 6.11). Unlike Table 6.10, the memory and time is different between mcmas-1.0 and agr-mcmas. As is immediately obvious, at-mcmas’s “hybrid state” approach for verifying formulae is clearly not effective. It requires an order of magnitude more memory, as well as taking three orders of magnitude more time. It is also not surprising that, as agr-mcmas has to consider fewer states, the number of nodes in the product graph is also smaller compared to that of the monolithic check with at-mcmas.

However, although the automata-theoretic approach does not seem favourable in the monolithic sense, its extension to support modular verification is clearly favourable – taking both less memory and less cumulative time. While these results are promising, they are not quite as dramatic as we would have hoped. As we will show in the next section, the model selected for allowing a direct comparison between the three tools does not permit us to demonstrate the true potential of agr-mcmas. By focusing on larger models that are outside the abilities of at-mcmas, and therefore excluding it from our comparison, we can demonstrate the expected benefits of the modular technique.

Comments

While the results shown in Section 6.5.1 highlight the potential benefits of compositional verification, we note that this might be seen as a slightly synthetic benchmark as, apart from the environment, the model is comprised entirely of homogeneous agents. We argue that, while techniques such as parametric verification [Kouvaros and Lomuscio,
2013a; Kouvaros and Lomuscio, 2013b] or symmetry reduction [Cohen \textit{et al.}, 2009a; Cohen \textit{et al.}, 2009b] would be applicable in the case of models containing many homogeneous agents, these would still require construction of the composition of at least the environment and one agent. By comparison, in the approach presented here, we only check the composition between the assumption and the component, and the environment and the assumption.

6.5.2 \textit{Software Development Protocol}

We now look at an example that, unlike the train example of the previous section, does not consist of $n$ homogeneous agents composed with a single environment.

We consider the “software development protocol” example from [Lomuscio \textit{et al.}, 2008; Lomuscio \textit{et al.}, 2012]. In these works, the authors present a scenario focused upon the composition of services providers, where their interactions are governed by a contract that specifies what it means to be in conformance or violation of the agreement. It contains seven agents: a principal software provider, a (non-principal) software provider, a client, an insurance company, a testing provider, a hardware provider and a technical expert.

The setting is as follows. The client desires a piece of software to be developed and deployed by the technical expert on a piece of hardware supplied by the hardware provider. There are two parties that provide the software: the principal software provider and (non-principal) software provider. The principal software provider performs software integration of its own software with the software from the other software provider, when a deliverable is made. This integrated software is then sent to the testing provider for testing. If the software passes testing it is given to the insurance company for the provision of software insurance. The software is finally handed over to the technical expert, who deploys it on the hardware provided by the hardware provider.

Importantly, both software providers and the client have open dialogue during the development. Depending on the instantiation of the protocol, both software providers need to update the other on their progress up to $n$ times; after the $n$-th update, the software is then released to the client. Up until the $n$-th update, the client is able to request changes to the developed software at no additional cost. After the $n$-th update, the software provider either has to rescind his request for a modification to the software, or has an additional cost to pay.

Similarly, the other components (e.g., the testing provider and hardware provider) also are able to repeat certain steps $n$ times. For example, the testing provider and the principal software provider can iterate up to $n$ times if the software fails its testing. If it fails on the $n$-th time, then client can withdraw its software tender from the principal software provider.
To exemplify, we reproduce the violation criterion from [Lomuscio et al., 2012] for the insurance company in Table 6.12. This criterion states that, if the client attempts to process a claim and the insurance company rejects or does not process this claim, then the insurer is in violation of its agreed contractual obligations.

<table>
<thead>
<tr>
<th>Agent</th>
<th>Violation</th>
<th>Possibility of Recovery</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insurance Company</td>
<td>Does not process the claim of the client</td>
<td>no</td>
</tr>
</tbody>
</table>

While this protocol may seem contrived, the authors [Lomuscio et al., 2008; Lomuscio et al., 2012] note that the example is derived and representative of a software procurement workflow from the IT industry.

We note that in the original protocol [Lomuscio et al., 2008; Lomuscio et al., 2012], the scenario has a fixed number of rounds (2). However, by taking the modification as presented above, this allows us to instantiate varying sizes of examples, allowing for a scalable example. In what follows, we therefore verify examples containing a ranging number of iterations, where a higher number of iterations leads to a greater number of reachable states.

We illustrate the interactions between each of the parties in Figure 6.13.

Fig. 6.13. Interactions in the Software Development Protocol
As can be seen from Figure 6.13, the Insurance agent only interacts with the Client agent, while every other agent in the system interacts with every other agent. It is immediately obvious to see that $\text{int}(\text{Insurance Company}) = \{\text{Client}\}$. As the insurance company only interacts with the client, we consider the modular verification of the Insurance Company, reformulated such that the Client represents the agent’s environment (i.e., in a modular setting, $A_e = \text{Client}$).

Furthermore, we make an additional adaptation to the original protocol: after making a request for compensation, within three transitions of the system, the client agrees to settle the claim with the insurer. It is therefore possible to specify the behaviour of the environment as the following assumption:

$$G(\text{Client}_\text{askCompensation} \rightarrow ((XG\neg \text{Client}_\text{askCompensation})$$

$$\land (FG\text{Client}_\text{Settle}))$$

This expresses that, once the Client asks for compensation, it never asks for compensation again, and eventually it always agrees to settle. On the modification where the client always chooses to settle, and cannot ask for compensation twice, it is clear that this should be satisfied by the prescribed behaviour of the client.

### 6.5.2.1 Propositions and Specifications

To perform verification of the software development protocol, we first need to define a set of atomic propositions for the insurance company that our formulae can be built upon:

- **green** – a local state of the insurance company satisfies “green” if it has not violated its contract
- **red** – a local state of the insurance company satisfies “red” if it has violated its contract
- **end** – the insurance company satisfies “end” if the protocol is completed (either successfully or unsuccessfully)
- **Client_Request** – if the client requests compensation (i.e., it has performed the action $\text{Client}_\text{askCompensation}$) and the insurance company observes this action, then the proposition holds
- **Client_Settle** – similar to above, if the client performs $\text{Client}_\text{Settle}$, then the insurance company records this in its local state and the proposition holds

Based on the above propositions, when performing modular verification of the insurance company, we consider the following Insurance Company-local specifications:

$$\phi_1 = A [(green) U (end)]$$
6.5 Evaluating Modular MCMAS

◦ Reading: the insurance company stays in compliance until it reaches the end of its protocol.

\[ \phi_2 = AG(\text{Client}_{\text{Request}} \rightarrow AX K_{\text{Insurance Company}} AF_{\text{Client}_{\text{Settle}}}) \]

◦ Reading: if the Client requests compensation, at the next state the Insurance Company knows eventually the client and the Insurance Company can always come to an agreement.

While we do not exemplify the formula further, we note that the specification:

\[ \phi_3 = AG(\neg \text{Client}_{\text{Request}}) \]

does not hold on any sized instantiation of the example. That is, it is not the case that the proposition Client_{Request} never holds. This is important in the context of \(\phi_2\), as it ensures that this specification does not suffer from antecedent failure [Beer et al., 2001]. If this were the case, then it would lead to the specification being trivially valid because of the pre-condition of the implication is never satisfied in the model.

Nonetheless, as the specification is always false—and given the lack of completeness of the compositional technique—we conclude that its verification does not aid in further evaluation of the technique.

6.5.2.2 Encoding the Model and Assumptions in ISPL

An excerpt of the ISPL code for the insurance company can be seen in Figure 6.14. From this ISPL code, it can be observed that the only other agent that the InsuranceCompany agent is dependent on is the Client agent. This can be seen by looking at those entries in the InsuranceCompany’s evolution definition that contain Agent.Action = Val. Such an observation is inline with Figure 6.13.

As Figure 6.14 only shows the ISPL for the insurance company, we have included snippets of the model level specifics (e.g., the evaluation and initial states block) as part of the figure.
Agent InsuranceCompany

Vars:


end Vars

Actions = { InsuranceCompany_fromClient, InsuranceCompany_fromClient1, InsuranceCompany_Pick1__0, InsuranceCompany_Pick1__1, InsuranceCompany_toClient, InsuranceCompany_Invoke1, nothing};

Protocol:


Evolution:

state = InsuranceCompany_1 if state = InsuranceCompany_0 and Action = InsuranceCompany_fromClient and Client.Action = Client_buyInsurance;

state = InsuranceCompany_2 if state = InsuranceCompany_1 and Action = InsuranceCompany_fromClient1 and Client.Action = Client_askCompensation;

state = InsuranceCompany_3 if state = InsuranceCompany_2 and Action = InsuranceCompany_Pick1__0;

state = InsuranceCompany_4 if state = InsuranceCompany_2 and Action = InsuranceCompany_Pick1__1;

state = InsuranceCompany_5 if state = InsuranceCompany_4 and ((Action = InsuranceCompany_Invoke1 and Client.Action = Client_pickResponse) or (Client.Action = Client_settle));

state = InsuranceCompany_6 if state = InsuranceCompany_3 and ((Action = InsuranceCompany_toClient and Client.Action = Client_pickResponse) or (Client.Action = Client_settle));

end Evolution

end Agent

- Eval:
  - InsuranceCompany_request if InsuranceCompany.state = InsuranceCompany_2;
  - InsuranceCompany_settle if InsuranceCompany.state = InsuranceCompany_3 or InsuranceCompany.state = InsuranceCompany_6;

- Init:
  - InsuranceCompany.state = InsuranceCompany_0;

Fig. 6.14. ISPL for the Insurance Company
Agent Client

Vars:
state : { s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7 }

end Vars
Actions = { other_action, Client_askCompensation, Client_settle, Client_buyInsurance, Client_Pick46__1, Client_Pick46__0 }

Protocol:
Other: { other_action, Client_askCompensation, Client_settle, Client_buyInsurance, Client_Pick46__1, Client_Pick46__0 }

end Protocol
Evolution:
state = s_7 if state = s_0 and !(Action = Client_askCompensation or Action = Client_settle);
state = s_0 if state = s_0 and Action = Client_settle;
state = s_1 if state = s_1 and !(Action = Client_askCompensation or Action = Client_settle);
state = s_1 if state = s_1 and Action = Client_settle;
state = s_7 if state = s_2 and !(Action = Client_askCompensation or Action = Client_settle);
state = s_0 if state = s_2 and Action = Client_settle;
state = s_7 if state = s_3 and !(Action = Client_askCompensation or Action = Client_settle);
state = s_4 if state = s_3 and Action = Client_askCompensation;
state = s_6 if state = s_3 and Action = Client_settle;
state = s_4 if state = s_4; -- self loop
state = s_1 if state = s_5 and !(Action = Client_askCompensation or Action = Client_settle);
state = s_3 if state = s_5 and Action = Client_askCompensation;
state = s_1 if state = s_5 and Action = Client_settle;
state = s_7 if state = s_6 and !(Action = Client_askCompensation or Action = Client_settle);
state = s_4 if state = s_6 and Action = Client_askCompensation;
state = s_6 if state = s_7 and Action = Client_askCompensation;
state = s_6 if state = s_7 and Action = Client_settle;
state = s_6 if state = s_7 and Action = Client_settle;

end Evolution

-- Eval:
-- L_0 if Client.state = s_1 or Client.state = s_2 or Client.state = s_3;
-- U_0 if Client.state = s_4;
-- L_1 if Client.state = s_0;
-- U_1 if Client.state = s_1 or Client.state = s_2 or Client.state = s_3 or
  Client.state = s_4 or Client.state = s_5 or Client.state = s_6 or Client.state = s_7;

-- Init:
-- Client.state = s_0;

Fig. 6.15. Modified Property Environment for the Insurance Company Assumption
The property agent Client for our chosen assumption is shown in Figure 6.15. We note that certain elements of the output have been modified when compared to the standard output of dra2ispl. We make note of these modifications below.

Firstly, the A-LTL assumption

\[
\psi = G(Client_{-}askCompensation \rightarrow (G \neg Client_{-}askCompensation) \land (G Client_{-}Settle))
\]

only contains the atomic propositions Client_{-}askCompensation and Client_{-}Settle. However in Figure 6.14, we can see that the InsuranceCompany agent also has observability on the actions Client_{-}buyInsurance, Client_{-}Pick46_{-}1 and Client_{-}Pick46_{-}0.

As such, we need to modify the property agent \(P_{\psi}\) generated by dra2ispl into the property environment \(P_{\{(\psi, \text{Client})\}}\), such that the composition of Client_{-}∥P_{\{(\psi, \text{Client})\}}\) is well defined. This is done by including those omitted actions into the action set of the agent in Figure 6.14.

Secondly, the agent that is automatically generated out of dra2ispl has the name Environment; this is clearly incorrect in the context of Figure 6.14, so we have changed the property environment’s name to be Client.

### 6.5.2.3 Results

We now consider a scaled version of the protocol: this is where the number of “retries” in the protocol (i.e., the parameter \(n\) as discussed previously) can be modified compared to the original protocol [Lomuscio et al., 2008; Lomuscio et al., 2012], where the number of retries is fixed at two. We therefore verify models with a varying number of retries.

<table>
<thead>
<tr>
<th>Check</th>
<th>Guarantee</th>
<th>SAT/UNSAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modular</td>
<td>(\phi_1)</td>
<td>UNSAT</td>
</tr>
<tr>
<td></td>
<td>(\phi_2)</td>
<td>SAT</td>
</tr>
<tr>
<td>Monolithic</td>
<td>(\phi_1)</td>
<td>UNSAT</td>
</tr>
<tr>
<td></td>
<td>(\phi_2)</td>
<td>SAT</td>
</tr>
</tbody>
</table>

While we demonstrate this experimentally, we note that the potential satisfaction of our two formulae (i.e., \(\phi_1\) and \(\phi_2\)) is shown in Table 6.13. The first formula \(\phi_1\) is never satisfied in either the modular verification approach or the monolithic check. This is not unsurprising as it is indeed possible for the insurance company to violate the protocol,
by not handling the client’s claim. As such, it follows that there does exist a path where green ceases to hold prior to end holding, thus invalidating $\phi_1$ in both instances.

By comparison, our chosen assumption is strong enough to ensure that formula $\phi_2$ is satisfied during the modular check. Given the soundness of the technique, it then immediately follows that the monolithic approach must return the same result. However, we note that under a weaker assumption, it is not the case that the insurance company always satisfies $\phi_2$.

For example, consider the assumption:

$$F \text{Client}_\text{askCompensation}$$

which states that along every run, the client requests compensation. Without the requirement that the client will eventually settle, it can be seen in Figure 6.14 that the agent will get “stuck” in the state InsuranceCompany_5 or InsuranceCompany_6. Consequently, although Client_Request holds, it is not the case that all future states eventually satisfy Client_Settle, which therefore invalidates the specification $\phi_2$.

**Guarantee check.**

As per the description of the protocol, and as shown in Figure 6.14, the insurance company on its own is not scalable (i.e., there is no parameter on the number of “retries” between the insurance company and the client). As such, in Table 6.14, we present the statistics for checking a single instantiation of the insurance company against our two specifications using agr-mcmas in the context of the property environment shown in Figure 6.15.

<table>
<thead>
<tr>
<th>Formula</th>
<th># Reachable states</th>
<th>Time (s)</th>
<th>Memory (KiB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\phi_1$</td>
<td>28</td>
<td>0.009</td>
<td>24,140</td>
</tr>
<tr>
<td>$\phi_2$</td>
<td></td>
<td>0.012</td>
<td>24,140</td>
</tr>
</tbody>
</table>

As we will show in the subsequent section, we do not compare against mcmas-1.0 for these two checks as it is the assumption check that is scalable and therefore is the one we wish to draw comparisons with.

Additionally, as will become apparent, the ability to verify the insurance company in isolation can bring immediate benefits: the state-space of the component itself (albeit when composed with a property closure environment) is only 28 states. This is a drastic
reduction from the $10^6$ reachable states that are experimentally considered in the original investigation [Lomuscio et al., 2012].

Furthermore, we point out that the memory usage for verifying these states seems surprisingly high, with almost 1 MiB per reachable state. However, we note that mcmas-1.0 demonstrates approximately a 20 MiB footprint purely during initialisation (e.g., in parsing the model, starting its dependent libraries such as Cudd, etc.), without actually performing any verification. This, in actuality, makes the memory usage not surprising.

**Assumption check.**

We now compare our various notions of “tractability” (memory, states, time) between our assumption check over the concrete client using agr-mcmas and the monolithic check using standard mcmas-1.0. We consider models where the number of repeated steps in the protocol are varied (e.g., the different number of “retries” that the client and the software providers tolerate). We denote by “# Retries”, this parameter in the model.

In the following results, we consider a time limit of 25,000 seconds (approximately 7 hours). Results that failed to complete within this time are marked with DNF.

Table 6.15 shows the increase in time for verification between the assumption check and full monolithic check. As is immediately obvious, the assumption check massively out-performs the monolithic check. It can be seen that the time taken to verify a full model with a retry counter of 10, is the same amount of time as performing an assumption check with a retry counter of 1,000.

The corresponding results for our other notions of tractability are shown in Table 6.16 (memory) and in Table 6.17 (number of reachable states).

In Table 6.16, which shows the memory usage in KiB, we can see there is a trade-off for tractability in the large examples. While, in the monolithic case, an example containing a retry counter of 20,000 did not complete within 7 hours (the closest was a counter of 200: two orders of magnitude less), the assumption check did complete (in under 30 minutes) but it required almost 3 GiB of RAM.
Table 6.15. Software Development Protocol: Assumption Check – Time

<table>
<thead>
<tr>
<th># Retries</th>
<th>Monolithic Time (s)</th>
<th>Assumption Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>40.345</td>
<td>0.304</td>
</tr>
<tr>
<td>20</td>
<td>269.949</td>
<td>0.510</td>
</tr>
<tr>
<td>50</td>
<td>644.057</td>
<td>1.039</td>
</tr>
<tr>
<td>100</td>
<td>1,229.524</td>
<td>2.180</td>
</tr>
<tr>
<td>150</td>
<td>2,749.592</td>
<td>4.340</td>
</tr>
<tr>
<td>175</td>
<td>DNF</td>
<td>4.529</td>
</tr>
<tr>
<td>200</td>
<td>6,572.624</td>
<td>4.676</td>
</tr>
<tr>
<td>250</td>
<td>DNF</td>
<td>4.860</td>
</tr>
<tr>
<td>500</td>
<td>DNF</td>
<td>10.470</td>
</tr>
<tr>
<td>1,000</td>
<td>–</td>
<td>39.722</td>
</tr>
<tr>
<td>2,000</td>
<td>–</td>
<td>122.412</td>
</tr>
<tr>
<td>4,000</td>
<td>–</td>
<td>315.568</td>
</tr>
<tr>
<td>10,000</td>
<td>–</td>
<td>1,137.865</td>
</tr>
<tr>
<td>20,000</td>
<td>–</td>
<td>1,750.653</td>
</tr>
</tbody>
</table>
Table 6.16. Software Development Protocol: Assumption Check – Memory

<table>
<thead>
<tr>
<th># Retries</th>
<th>Monolithic (KiB)</th>
<th>Assumption (KiB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>58,540</td>
<td>27,852</td>
</tr>
<tr>
<td>20</td>
<td>61,092</td>
<td>30,860</td>
</tr>
<tr>
<td>50</td>
<td>59,632</td>
<td>34,688</td>
</tr>
<tr>
<td>100</td>
<td>63,208</td>
<td>44,620</td>
</tr>
<tr>
<td>150</td>
<td>69,932</td>
<td>54,784</td>
</tr>
<tr>
<td>175</td>
<td>148,636*</td>
<td>56,332</td>
</tr>
<tr>
<td>200</td>
<td>139,972</td>
<td>59,168</td>
</tr>
<tr>
<td>250</td>
<td>260,152*</td>
<td>74,644</td>
</tr>
<tr>
<td>500</td>
<td>349,236*</td>
<td>113,788</td>
</tr>
<tr>
<td>1,000</td>
<td>–</td>
<td>167,080</td>
</tr>
<tr>
<td>2,000</td>
<td>–</td>
<td>277,868</td>
</tr>
<tr>
<td>4,000</td>
<td>–</td>
<td>515,896</td>
</tr>
<tr>
<td>10,000</td>
<td>–</td>
<td>1,486,052</td>
</tr>
<tr>
<td>20,000</td>
<td>–</td>
<td>3,093,700</td>
</tr>
</tbody>
</table>

* These results are categorised as DNF, and memory usage displayed is the instantaneous memory usage at 25,000 seconds.
### Table 6.17. Software Development Protocol: Assumption Check – Reachable States

<table>
<thead>
<tr>
<th># Retries</th>
<th>Monolithic</th>
<th>Assumption</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>$1.13 \times 10^6$</td>
<td>363</td>
</tr>
<tr>
<td>20</td>
<td>$4.92 \times 10^6$</td>
<td>747</td>
</tr>
<tr>
<td>50</td>
<td>$1.33 \times 10^7$</td>
<td>1,515</td>
</tr>
<tr>
<td>100</td>
<td>$5.49 \times 10^7$</td>
<td>3,051</td>
</tr>
<tr>
<td>150</td>
<td>$3.66 \times 10^8$</td>
<td>6,123*</td>
</tr>
<tr>
<td>175</td>
<td>DNF</td>
<td>6,123*</td>
</tr>
<tr>
<td>200</td>
<td>$2.23 \times 10^8$</td>
<td>6,123*</td>
</tr>
<tr>
<td>250</td>
<td>DNF</td>
<td>6,123*</td>
</tr>
<tr>
<td>500</td>
<td>DNF</td>
<td>12,261</td>
</tr>
<tr>
<td>1,000</td>
<td>–</td>
<td>24,555</td>
</tr>
<tr>
<td>2,000</td>
<td>–</td>
<td>49,155</td>
</tr>
<tr>
<td>4,000</td>
<td>–</td>
<td>98,283</td>
</tr>
<tr>
<td>10,000</td>
<td>–</td>
<td>393,195</td>
</tr>
<tr>
<td>20,000</td>
<td>–</td>
<td>786,411</td>
</tr>
</tbody>
</table>

* The state-spaces reported by MCMAS-1.0 are potentially anomalous; see accompanying text.
We note that in Table 6.17 there are models that appear to generate state-spaces with the same cardinality (i.e., during the assumption check with a counter of 150, 175, 200 and 250, \textsc{agr-mcmas} reports a state-space of 6,123 states). These results are anomalous and are indicative of a bug in \textsc{mcmas-1.0}, which \textsc{agr-mcmas} is built upon. As will be discussed later in Table 6.18, these examples all require the same number of BDD variables.

If we compare memory usage against the number of reachable states, we can see that \textsc{agr-mcmas}’s use of an explicit-state methodology starts to show. To store and verify $2.23 \times 10^8$ states using BDDs \textsc{mcmas-1.0} required 136.7 MiB; by comparison, to store and verify $7.86 \times 10^5$ states \textsc{at-mcmas} required 2.95 GiB (i.e., to perform verification on a model containing three orders of magnitude fewer states, \textsc{agr-mcmas} required an order of magnitude more memory). We note that it is not necessarily the states that cause the memory used in \textsc{agr-mcmas}’s case: it is the fact that the product graph is stored explicitly in memory. However, as the size of the product graph is directly proportional to the number of reachable states, the above is still a valid comparison.

Nonetheless, all is not lost for the modular technique: our results do not show any instances where the monolithic check used less memory than the assumption check (i.e., on a verification instance of comparable size the modular approach is never less efficient).

In Table 6.18 we show the number of BDD variables used in both approaches, as well as the number of potentially representable unique states. The number of BDD variables directly corresponds to the number of bits required to encode a single global state in the model (e.g., for a model containing two agents each with a local state composed of three potential assignments, we require four bits—and therefore four BDD variables—to represent any possible global state). As is immediately obvious, the number of representable states is not necessarily the number of reachable states in the system. However, as a gauge for assessing the plausibility of a technique to alleviate the state-space explosion problem, it is useful.

While it is a potentially exaggerated metric to use, it can be seen that, e.g., on a model with 200 retries, the modular assumption check needs to potentially check eighteen orders of magnitude fewer states. Clearly, this is a significant reduction. However, in the realistic case, we see that, in actuality, the reduction is only four orders of magnitude (i.e., $10^8$ states for monolithic vs. $10^4$ states for the assumption check, as seen in Table 6.17).

6.5.2.4 Comments

We note that, unlike the train-gate-controller model of the previous section, the software development example here is not well suited to parametric verification [Kouvaros and Lomuscio, 2013a; Kouvaros and Lomuscio, 2013b]. This is because, again in contrast to
the train-gate-controller model, the model is constructed of many, large heterogeneous subsystems (i.e., there are no agents in the system that are structurally the same as any other subsystem). Similarly, techniques such as agent-based symmetry reduction [Cohen et al., 2009b] could also not be applied here.

As such, and for this example, compositional reasoning is exactly well suited. Given the fact that the Insurance Company can reasonably expect the Client to act in the prescribed way, this allows us to break up the verification tasks into two parts as required for assume-guarantee.

By comparison, parametric verification would not be able to perform such a state-space reduction; it could only verify an infinite family of Insurance Company against one—suitably parametric—Client.

### 6.5.3 Comparison to Other Techniques

In this section, we demonstrated experimental results for two benchmarks to compare the modular technique against two monolithic approaches (one based on symbolic data structures, the other on the work of Chapter 4). The results show that potential savings are achievable, in instances where the model can be decomposed into separate checks.
By means of a comparison, we note that related works for investigating state-space reduction techniques in a multi-agent arena have demonstrated state-space reductions—in the best case and on representative examples—between one and four orders of magnitude.

For example, partial order reduction [Lomuscio et al., 2010a] reports a reduction from $1.51 \times 10^7$ states to $9.00 \times 10^5$ states (two orders of magnitude) when verifying the dining cryptographers example and an LTLK formula. Symmetry reduction [Cohen et al., 2009b] reports a reduction of $7.37 \times 10^5$ states to $9.83 \times 10^4$ states (one order of magnitude) when verifying the same problem but against a CTLK formula. Finally, in [Russo, 2011], Russo applies existential abstraction to reduce the state-space of a card game from $2.17 \times 10^9$ states to $1.35 \times 10^5$ states (four orders of magnitude).

For comparison, we demonstrate achievable reductions from $3.66 \times 10^8$ states to $6.12 \times 10^3$ states (five orders of magnitude) in the concrete case, and a potential reduction of twenty eight orders of magnitude when comparing BDD variable usage alone, on a representative example from industry.

### 6.6 Concluding Remarks

In this chapter, we presented the implementation and evaluation of the theory presented in Chapters 4 and 5. The two extensions to mcmas-1.0 are both interesting as they are “hybrid state”: while the model checking procedure itself is an explicit state procedure, the agents and the reachable states of the model are encoded implicitly using an efficient data-structure (BDDs).

Using three benchmarks from the literature, we evaluated the efficacy of the two approaches. The results for etav demonstrate it is an effective model checker in its own right. However, a potential extension to the tool is required to allow us to draw direct comparisons to similar tools, as nearly all other model checkers take implicitly defined structures, rather than explicitly defined multi-modal Kripke structures.

We demonstrated the use and efficacy of the dra2ispl tool in constructing property closure environments and property observer agents for assumptions specified using the A-LTL logic. For more complex examples, we exemplified that some of the output from dra2ispl may need to be customised to form a well-defined closure environment for the given agent-under-test.

The results for agr-mcmas are, as hoped, promising. On an industrial-focussed example, the technique shows dramatic improvements compared to the standard monolithic approaches. Where the results were presented, we also drew comparisons between comparable state of the art. We noted that agr-mcmas demonstrated greater gains than existing techniques have been able to to-date. However, as agr-mcmas suffers from incompleteness and only supports a restricted class of formulae (agent-local and in the universal fragment), it is not a “silver bullet” for all verification problems.
In this chapter we have experimentally evaluated the verification approaches defined in Chapters 4 and 5. These experimental analyses have demonstrated the expected benefits and disadvantages of the techniques. In particular, while the modular technique can present dramatic reductions in the size of the state-space to be analysed, the underlying use of explicit-state model checking procedures still has the expected downfalls.

In Chapter 7, we evaluate the effectiveness of the approaches put forward in addressing the research hypothesis presented in Chapter 1. Furthermore, we draw comparisons with the related work, discuss potential extensions to the theory presented, and conclude.
Chapter 7
Conclusions

7.1 Overview and Summary

In this thesis, we investigated the applicability of compositional verification to multi-agent systems. As an interim step towards this goal, we introduced an automata-theoretic approach for the verification of general branching-time temporal-epistemic properties. The method proposed for modular reasoning is, we believe, the first application of true assume-guarantee reasoning for multi-agent systems in the context of mentalistic logics.

To this end, the main contributions can be summarised as follows:

• Theoretical contributions
  We provided correctness results for the automata-theoretic approach, as well as correctness and preservation results for the modular approach.
  These results are noteworthy as they demonstrate the first realisable automata-theoretic verification approach for temporal-epistemic logic, and the support for modular verification of temporal-epistemic formulae allows for potential state-space reductions in the verification of complex systems.
  As we have shown in Chapter 6, both the automata-theoretic approach and the modular approach can give effective results. As expected, the modular approach also provides significant benefits when applied to the identified classes of multi-agent systems.

• Development of Toolkits
  We developed various toolkits supporting the theory presented. The first tool, ETAV, is an entirely new model checker for explicitly-defined interpreted systems using the purely automata-theoretic approach.
  The extensions AT-MCMAS and AGR-MCMAS build on top of the open-source model checker MCMAS, and implement the automata-theoretic approach and the modular approach respectively. We note these extensions are “hybrid state”: while the reachable states are stored symbolically using efficient and implicit data-structures,
the model checking procedure itself is still an explicit state procedure based on the structure of the model and of the formula.

Finally, DRAZISPL can automatically "synthesise" agents for checking formulae (named “property observer agents”), as well as environments that directly satisfy a given formula when performing model checking (titled “property closure environments”). We note that our property closure environments are distinct from a standard tableau of a formula and, as such, their construct does not rely on solving the satisfiability problem for linear temporal logic.

- **Evaluation**

We have investigated the use of these tools on various benchmarks from the literature and showed that they can be effective in their approach. Furthermore, while testing the hypothesis of this thesis, we have demonstrated that modular verification can indeed be more tractable than monolithic verification in the expected scenarios, compared to either automata-theoretic verification or well-established symbolic model checking-based approaches.

- **Empirical**

As part of our evaluation, we provided evidence that practical modular model checking approach can be of use. In 1995, the following was postulated:

> In view of these discouraging results, is there hope for modular model checking? One should keep in mind that the bounds [...] are worst-case bounds. In practice, the automaton $A_\phi$ [for the assumption $\phi$] need not be exponential in the size of $\phi$, and the subset construction need not yield an exponential blowup in the size of $A_\phi$ [...]. If the size of the linear assumption $\phi$ is not too large and the doubly exponential blowup is avoided, then our algorithm might not be always impractical.

– Moshe Y. Vardi [Vardi, 1995]

We feel our results indicate a positive and empirical justification of the above statement. This justification provides evidence that, despite the possibilities of higher complexity, modular verification can still be practical in instances arising in real life.

### 7.1.1 Strengths and Weaknesses

To the best of our knowledge, the presented modular approach introduces three significant differences with the current state of the art: support for temporal-epistemic guarantees; the use of mixed semantics for assumptions and guarantees; and the ability to verify properties beyond safety/reachability only.
Furthermore, we argue that other benefits of the modular technique include:

- It supports more efficient and tractable MAS-based reasoning, via partitioning of the state-space into only parts of the behaviour that are relevant to the verification task at hand.
- The ability to verify a given agent earlier in the agent-development life-cycle and without the requirement to have the other agents in the system completed, allowing for a partial system verification.
- Using dra2ispl, we have provided a user-friendly way of generating interface environments without the need to manually implement the desired behaviour of the interaction set for a given agent.

While significant benefits of the technique have been demonstrated (Section 6.5), we identify that the underlying methodology has two distinct disadvantages. The first is the incompleteness of the technique. When performing modular reasoning, it is possible for the technique to return a false negative: that is, under a given assumption, the model checking procedure may state that the agent does not satisfy the expected property. This could either be a genuine error in the agent, or it could simply be that the chosen assumption was too weak to ensure that the agent adheres to its desired specification. To somewhat ameliorate this issue, it would be possible to perform either manual or automated analysis of the counterexample for the failure to verify if the behaviour of the failure is indeed a failure of the real system.

The second issue is in the absence of circularity in the given verification approach. As shown in the inference rules for Chapter 5, ensuring that an environment satisfies its environment is demonstrated through the further use of another assume-guarantee triple: by taking the ancillary assumption true, can we show that the environment satisfies the selected assumption? The assumption true situates the environment with a set of completely chaotic [Roscoe et al., 1996; Sidorova and Steffen, 2001; Leino and Logozzo, 2005]—or, as we have referred to them, universal—agents; it therefore may not be possible to demonstrate that the environment satisfies the presumed assumption. Nonetheless, in a concrete composition, it may have been possible that the environment does indeed satisfy this assumption.

Adopting a circular reasoning technique would allow us to assume a behaviour over the components that the environment interacts with, which then may allow us to demonstrate that the environment does satisfy the assumption previously selected. However, circularity in compositional reasoning is not straightforward and can easily lead to unsoundness [Namjoshi and Trefler, 2010].
7.1.2 Summary of Contributions

To summarise, this thesis presents two novel approaches to the verification of multi-agent systems. The first approach based on automata (Chapter 4) opens the door for a myriad of new techniques that would be normally unsupported by set-theoretic model checking. For example, approaches based on on-the-fly model checking and partial order reduction [Peled, 1996; Alur et al., 2005b] are often deeply ingrained with automata-theoretic approaches. Without the corresponding automata-theoretic approaches for temporal-epistemic logic, it is difficult to directly transfer the state-of-the-art in the verification of temporal-only properties. The presented approach is therefore a major stepping-stone towards direct utilisation of efficient verification approaches from the purely formal verification school of research to that of the intelligent systems arena.

The second approach of Chapter 5 introduced the first technique for compositional reasoning against mentalistic properties of multi-agent systems. Unlike existing approaches, we note that our approach is strong enough to support assumptions in full LTL and guarantees in the universal fragment of CTLK. We have argued that, when performing modular agent-based verification, it is reasonable to express properties in an agent-local way, such that the specifications only concern the agent under verification. For systems that can be suitably decomposed and decoupled, and when verifying the relevant specifications, modular verification has been demonstrated as a significantly more tractable approach (in time, memory and states) against approaches using monolithic verification (either based on symbolic approaches or that of Chapter 4).

7.1.3 Comparison Against the State-of-the-Art

While comparisons have been stated in other parts of this thesis, we now draw attention to the differences at a much higher level.

For the automata-theoretic approach, this approach is wholly novel when compared to existing BDD or SAT-based approaches for solving the same problem. Unlike the set-based manipulations of the aforementioned techniques, we reduce the problem of verification to that of language non-emptiness, more akin to local model checking. By comparison, set-theoretic approaches calculate the set of states where a formula is satisfied, and verification is then in checking if a given set of states is a member of all the states satisfying the formula.

Our modular semantics is close to that of [Josko, 1989], and takes direct inspiration from [Vardi, 1995], where an assumption is used to specify the “assumption valid” paths over which path quantification should appear when checking a \(\forall\)CTL path operator. However, we note that the approach of [Josko, 1989] neither supports epistemic logic, nor does it support assumptions specified in full LTL. Similar works containing
branching-time logics with path quantification, e.g., [Kupferman and Vardi, 2006; Niebert et al., 2008], all support a similar methodology, but again do not support epistemic logic. Additionally, these latter works also do not consider the use of these path criteria in the context of modular model checking. Furthermore, we draw attention to the fact that—again, to the best of our knowledge—none of these techniques have ever been implemented.

Approaches based on reactive modules [Henzinger et al., 1998; Alur and Henzinger, 1999] and tree-containment [Henzinger et al., 2002] are similar to our approach (i.e., in the use of Behaviours and Traces). However, in the reactive modules approach—and as is implemented in MoCHA—users are required to explicitly provide an implementation of the chosen assumption, and then tree-based containment can be applied to demonstrate if the genuine component is a “refinement” of the assumption. In contrast, we use a linear-time assumption to define the behaviours of environment and then, using a specific model checking approach, verify that the environment satisfies this assumption against the semantics of LTL. When verifying a branching-time guarantee, we discount paths in the model of the agent under test that do not satisfy this assumption. By comparison, when performing refinement-based verification, the component is composed with the interface abstraction as normal, and then verification can take place using the standard semantics for CTL, without the need to consider assumption validity of paths.

Recent automated approaches for assume-guarantee reasoning [Cobleigh et al., 2003; Alur et al., 2005a; Nam and Alur, 2006; Cobleigh et al., 2008; Nam et al., 2008; Pasareanu et al., 2008] only cater for safety/reachability guarantees, and learn a deterministic finite automaton representing the environment. This is a significantly weaker set of properties that we are able to verify, but has the advantage that it is not required of the user to manually derive and specify an LTL assumption. However, this approach is again tied to a specific and fixed environment; it cannot assist in the verification of agents against incomplete or partial systems, nor for checking redeployable agents against “generic” environments.

We are aware of a handful of implementations that perform CTL model checking using types of alternating automata: an extension of cwb-nc [Bhat et al., 2001] that uses alternating tableau automata for checking an extension to CTL* supporting state and action-based propositions ([Bentahar et al., 2010] considered the use of this work in an agent-based area); alt-mc [Visser et al., 1997] that uses a game-based construction for the non-emptiness game of alternating automata; and an extension to nuSMV [Qian and Nymeyer, 2006] that uses symbolic data-structures to decide the language non-emptiness of the product automaton. However, we note that none of these approaches support epistemic logic and, with the exception of cwb-nc [Bhat et al., 2001] that is implemented in Standard ML, neither alt-mc or [Qian and Nymeyer, 2006]’s extension to nuSMV are publicly available. In addition, the nuSMV extension [Qian and Nymeyer, 2006] considers an approach based on the reduction of general CTL model checking to reachability only. This is something that was not considered inside the scope of Chapter 4, and we note that
this approach shows a significant overhead as a result of adopting this reduction. We also draw a distinction between our alternating automata for a CTLK formula and the use of tableau automata [Bhat et al., 2001] as they implement an entirely different semantics for the construction of the formula automata. Furthermore, it is worthwhile highlighting that extensions to MCMAS are clearly favourable for development temporal-epistemic verification approaches as ISPL supports a computationally grounded semantics for multi-agent systems [Wooldridge, 2000; van der Hoek and Wooldridge, 2003], when compared to the reactive systems-style formalism that NuSMV adopts.

7.2 Future Work

In the thesis, we have only “opened the door” for a variety of extensions and possibilities based on modular verification of multi-agent systems. We now highlight some of the more interesting possibilities.

7.2.1 “Many Agent Systems”

It would be of immediate relevance to extend the assume-guarantee rules to reason about assumptions defined over multiple components (i.e., not just the environment).

The inference rules from Chapter 5 can be intuitively extended to support multiple assumptions, with one assumption for each component that the agent under test interacts with.

Figure 7.1 shows the extended inference rules that deal with assumptions from multiple components; it follows that each agent in \( \text{int}(i) \) discharges its own assumption, which can then be used to constrain the behaviour of \( A_i \). Figure 7.2 shows the reformulation of Figure 7.1 as multiple model checking queries.

\[
\begin{align*}
\text{AGR-MAS-MULTI-ASSUM} \\
[\text{true}] A_{j_0} \langle \psi_1 \rangle \\
\vdots \\
[\text{true}] A_{j_m} \langle \psi_2 \rangle \\
\{A_{j_1}, \ldots, A_{j_m}\} = \text{int}(i) \\
\psi_1 \land \cdots \land \psi_m \Rightarrow \text{A}_i \langle \phi \rangle \\
[\text{true}] A_i \parallel A_{j_1} \parallel \cdots \parallel A_{j_m} \langle \phi \rangle \\
\end{align*}
\]

**Fig. 7.1.** Inference rules for \( n \)-agent, multi-assumption linear-branching assume-guarantee reasoning
The correctness of these rules follows intuitively from the framework presented in Chapter 5. It is clear that if the agents 1 to \( n \) satisfy their assumptions unconditionally in a universal environment, then when placed in a composition, these agents will remain satisfying their assumptions.

### 7.2.1.1 Model Checking Procedure

The automata-based assumption-checking framework of Chapter 5 will need to be extended to deal with \( n = |\text{int}(i)| \) assumptions.

We identify two tangential approaches:

1. Construct the Büchi automaton for the conjunction of each of the assumptions, i.e., \( \varphi = \psi_1 \land \cdots \land \psi_n \). This will be a single automaton, which encapsulates the assumed behaviour of all the agents in the interaction set. However, unlike the approach presented in Section 6.3.2, care would need to be taken to encode the single action condition into this automaton.

2. Construct \( n \) Büchi automaton, one for each agent. The composition of these automaton and the agent under test can then be constructed in the standard way. As each assumption is encoded as its own automaton, the underlying Rabin automaton only has to observe its own action from the global set of actions that occurred.

However, it is not immediately clear if either approach would be more or less efficient than the other. The first approach has the disadvantage that constructing an automaton based on the conjunction of all assumptions may be very computationally expensive, as it is well established that for large LTL formulae, constructing the Büchi automaton is non-trivial.
In contrast, for the second approach, there may be a combinatorial explosion caused by composing \( n \) automata, each of which have been determinised by the subset construction (thus already dramatically increasing the number of states per automaton). Furthermore, to support the “many automata” approach, Algorithm 2 would need to be extended to ensure that a given path is accepted by all the assumptions. This would require iterating over each assumption and each Rabin pair in that assumption to ensure that the path is accepting.

### 7.2.2 Application of Automata Learning Frameworks

For a valid action-based assumption \( \phi_e \) on the environment \( A_e \), we have shown in Chapter 5 that the observable action-based language of the environment is a subset of the language of the assumption. That is,

\[
\mathcal{L}(A_e) \subseteq \mathcal{L}(\phi_e)
\]

However, rather than explicitly specifying an A-LTL assumption prior to verification, an automaton \( A'_e \) could be identified such that

\[
\mathcal{L}(A_e) \subseteq \mathcal{L}(A'_e)
\]

The inference rules presented so far do not depend directly upon the use of A-LTL assumptions, but rather upon the language accepted by the assumption. As such, directly specifying an automaton that subsumes the language of the component it abstracts supports compositional preservation in exactly the same way as an A-LTL assumption does.

Consequently, we can apply a generic automata learning algorithm that iteratively learns a “candidate assumption” \( \hat{A}_e \) where its language converges towards the language of the component it is learning. For the \( n^{th} \) candidate assumption \( \hat{A}^n_e \), the learning framework will discharge a set of candidate assumptions as follows:

\[
\mathcal{L}(A_e) \subseteq \cdots \subseteq \mathcal{L}(\hat{A}^n_e) \subseteq \mathcal{L}(\hat{A}^{n-1}_e) \subseteq \cdots \subseteq \mathcal{L}(\hat{A}_e^0)
\]

Such a learning-based approach can be utilised in an automatic fashion as follows:

1. Begin by using the \( n^{th} \) candidate assumption in compositional model checking, as per the approach specified in Chapter 5.
2. If all the premises are satisfied (i.e., both the assumption and the guarantee check are successful), then the conclusion holds, so we can deduce that the formula holds on the full model.
3. If the guarantee check fails, then the counterexample is either a valid behaviour of the abstracted environment (so it can be shown that the formula could never hold in the full model) or the counterexample is spurious and therefore the assumption $\hat{A}_e$ was too weak to constrain $A_i$ correctly.

4. If the counterexample is spurious, this invalid behaviour can be used within the learning framework to produce a new candidate assumption $\hat{A}_{e+1}$ that removes this invalid behaviour, and the process can restart.

5. Eventually, the procedure will either show the formula is satisfied or falsified (via a valid counterexample). Importantly, we have that, if the formula is not falsified, the learning algorithm will eventually emit a candidate assumption that is exactly the language of the component it is learning (i.e., $\mathcal{L}(\hat{A}_e) \equiv \mathcal{L}(A_e)$, so $|\hat{A}_e| \equiv |A_e|$). Should the approach reach this eventuality, the size of the state-space under the assumption is the same as verifying the full composition, so the approach will always terminate.

We note that this is a similar approach to that presented in [Bobaru et al., 2008] for “automated assume-guarantee reasoning by abstraction refinement”.

Fortunately, for a falsifiable $\forall$CTLK formula, it is always possible to generate a counterexample [Clarke et al., 2002]—these counterexamples are said to be “tree-like”. As such, given a counterexample for the composition of $A_i || \hat{A}_e$, it is possible to run the counterexample “tree” over the concrete component $A_e$ and verify if the behaviour of the assumption as witnessed in the counterexample is indeed a behaviour of the genuine component. Special care will need to be applied when dealing with the epistemic fragment, as standard counterexamples for $\forall$CTLK will not contain enough information to demonstrate the behaviour of the other components in the system pertaining to the reachability of the epistemically related state [Jones and Lomuscio, 2011].

7.2.3 Truly “Modular” Interpreted Systems

Our presentation of interaction-defined interpreted systems in Chapter 5 is closely related to the modular interpreted systems formalisation of [Jamroga and Ågotnes, 2007; Jamroga et al., 2013]. Both modular interpreted systems and interaction-defined interpreted systems subsume interpreted systems. One of the intents of the modular interpreted systems formalism—which is similar to that of interaction-defined interpreted systems—was attempting to introduce an aspect of modularity and “openness” into interpreted systems. To this end, the authors [Jamroga and Ågotnes, 2007; Jamroga et al., 2013] introduce observable “interaction tokens”. Rather than observing a fixed-cardinality global action (as in interpreted systems), in modular interpreted systems the agents observe sets of named tokens, and update their local state on the ob-
servation of these named tokens. By ignoring named tokens that the agent is uninterested in, this allows for a higher degree of modularity and openness.

Therefore, it follows that modular interpreted systems may be an ideal candidate to refocus the modular verification approach. Assumptions for the agent can be specified over these interaction tokens, rather than over the actions of the agents in the agent’s interaction set.

7.2.4 Richer Assumptions

There are a number of prospects related to extending the specification language of assumptions to allow for the specification of a richer set of properties. We highlight three possibilities below.

7.2.4.1 “Reactive” Properties

Currently, assumptions are specified as purely “inward-looking” specifications against the observable actions of the component they are specified over. For example the property

\[ \text{Ass}_i = G(\text{Act}_x \rightarrow X \text{Act}_y) \]

expresses that whenever \( A_i \) does \( \text{Act}_x \) at the next step it should do \( \text{Act}_y \).

However, it would be of benefit to specify how the environment of a component reacts to the agent under test. For example an assumption such as

\[ \text{Ass}_E = G(\text{Agent}_n \text{act} \text{m} \rightarrow X \text{Environment} \text{act} x) \]

would specify that when agent \( n \) does \( \text{act} \text{m} \), at the next step, the environment should do \( \text{act} \text{x} \). This would be a suitable assumption for agent \( n \) as it is a stronger specification on how the environment interacts with it.

While this has not been investigated in the context of this thesis, the general framework for property closure environments and property observer agents would still apply to this class of specifications. For the property observer \( O_\varphi \), it would be necessary to perform the check \( \text{true} | \text{A}_e [\varphi] \) against a universal agent for \( A_i \), such that \( O_\varphi \) can observe both what \( A_e \) and \( A_i \) do. For the guarantee check, \( \mathcal{P}_{\varphi} \) would also have to observe the actions that \( i \) does while transitioning as \( A_e \).

Such an extension could be simply implemented as part of \texttt{DRA2ISPL}; however, similar to Section 7.2.1.1, care will have to be taken in handling the single action condition, as \( A_i \) and \( A_e \) are composed synchronously and therefore two actions can occur at once.
7.2 Future Work

7.2.4.2 Fluent Properties

Another candidate would be in extending assumptions to also include fluent propositions that are enabled by actions. Fluent propositions are atomic propositions that are enabled, and subsequently disabled, when certain actions are performed. The idea of fluents enabled by actions in (a variant of) A-LTL has previously been addressed in [Giannakopoulou and Magee, 2003].

For example, we might wish to express

\[
G(\text{Environment}_{act,y} \rightarrow X ((\text{prop} \lor \text{Environment}_{act,z})) \lor G(\text{prop} \land \neg \text{Environment}_{act,z}))
\]

which states that when the environment performs the action \(act,y\) then from the next state, either the proposition \(prop\) holds until the environment performs the action \(act,y\) or the environment never performs \(act,y\) and therefore \(prop\) never ceases to hold. In the framework of [Giannakopoulou and Magee, 2003], the proposition \(prop\) would not be embedded in the assumption itself, but would be as an ancillary proposition enabled by \(act,y\) and disabled by \(act,z\).

The use of these propositions would then support checking properties outside of the “introspective” class. In such a setting, the fluents defined over the environment could be used to encapsulate a local variable of the environment. As shown previously, for local propositions, \(p_i \leftrightarrow K_i p_i\); therefore, such propositions can be used to encapsulate that the environment holds certain knowledge about its own state. As such, it could further be verified that the agent under test knows that the other agent possesses this knowledge after the fluent-enabling action has been performed.

For example, if we have the assumption \(GF \text{prop}_e\), then it would be possible to verify (as an extremely simple illustration):

\[
[G F \text{prop}_e]A_i \langle AF K_i \text{prop}_e \rangle
\]

That is, if it can be assumed that the environment will eventually always assert \(\text{prop}_e\), then it should be possible to demonstrate using assumption-based model checking that the agent \(i\) will eventually know this proposition.

Under a less extreme example, it would be possible to formulate an assumption as follows (based on the alternating bit transmission problem from [Lomuscio and Sergot, 2004]):

\[
[G (send\_even \rightarrow G even)]A_i \langle AG (seen\_even \rightarrow (K_i even \land K_e even)) \rangle
\]

The above asserts the following: if it can be assumed that when the environment performs the action \(send\_even\) then it holds forever that the proposition \(even\) holds, we can then prove that once the agent has observed the action \(send\_even\) that the agent both knows \(even\) and it also knows that the environment knows \(even\). That latter follows
trivially as shown in Chapter 5 where an agent always knows a local proposition defined over its own local states (i.e., for $p_i \in AP_i$, $g \models p_i \iff g \models K_i p_i$).

7.2.4.3 Assumptions Over Observable Variables

ISPL supports the concepts of $\text{Obsvars}$ and $\text{Lobsvars}$. These are variables that are declared as part of the environment, but where each agent has visibility of them ("globally observable" to all agents, or "locally observable" to only one agent, respectively). When verifying epistemic properties, the interpretation of the knowledge modality has to be suitably extended to include the observation of these variables when calculating epistemic indistinguishability.

One such use of these declaration blocks is to support a similar variant of "shared variable" concurrency, similar to $\text{nusmv}$. That is, rather than support communication via the observation of observable actions, the environment can communicate with a given agent by manipulating variables occurring in that agent’s $\text{Obsvars}$/Lobsvars declaration.

As such, it would be of interest to extend the modular approach presented to work on these shared variables, either exclusively over shared variables or a mixed presentation for both actions and state assignments. This would be in a similar vein to [Nam et al., 2008] where they investigate learning-based assume-guarantee in the context of shared variable concurrency in $\text{nusmv}$.

Furthermore, this would nicely dovetail with fluent-based properties, as these could be specified in a “state-event” logic [Chaki et al., 2005], where assumptions would contain propositions both over actions and (locally) observable variables.

7.2.5 Symbolic Rabin Fairness

There exist extensions [Burch et al., 1992] to the standard symbolic CTL model checking approach based on fixed points that support Büchi-style fairness. For example, when checking the formula $\text{EG} \phi$, it is possible to restrict the set of states considered to only those states that occur on a run where the property fair is satisfied infinitely often.

If the property fair corresponds to the acceptance condition $F$ in a Büchi automaton, then fair model checking can be used to restrict the model to only those states that occur on a path that would be accepted by the corresponding Büchi automaton.

However, there is not a similar correspondence for Rabin fairness. The development of a fair model checking algorithm that supports a Rabin-style fairness would allow us to apply symbolic model checking when checking the parts of an assume-guarantee triple. This would allow us to move from our hybrid-state model checking approach based
on Libi epistemic alternating automata (Chapter 5) to then use the standard symbolic fixed-point approach to verifying CTLK.

However, for the assumption check to determine if $E \models \Delta_{\mathcal{LT}} \phi$, it would be necessary to change the approach from verifying $E \parallel O \phi \models AG true$ to instead verify $E \parallel O \neg \phi \models EG true$. That is, rather than ensuring that all paths are accepting against $A \phi$, we would use symbolic model checking to see if there exists one infinite, fair path in the model that is accepted against $A \phi$. If such a path exists, this means there exists a behaviour of $E$ that is accepted against the Rabin condition for the negation of the formula – this clearly entails that $E$ does not satisfy $\phi$.

This change would be necessary as fair model checking will simply exclude any paths that are not valid against the fairness condition; therefore using the acceptance condition from $A \phi$ would remove any paths from the composition that were not accepting, which would be undesirable. This approach would be in line with the module checking approach of [Kupferman and Vardi, 1997a].

### 7.2.6 Symbolic Encoding of the Product Automaton

Currently, at-mcmas and agr-mcmas implement a hybrid-state approach to model checking: the state-space is stored implicitly using BDDs, but the product automaton is constructed explicitly.

Via an extension of [Bernholtz and Grumberg, 1993] and [Qian and Nymeyer, 2006], it would be of interest to encode the transition relation of a WEAA as a BDD and then construct the product between the reachable states and the automaton. This would require an encoding of the “next state” function for indistinguishability relation for each agent, as currently mcmas uses “variable quantification” within the BDDs to find the set of global states that are indistinguishable from a starting set of global states.

Given a BDD encoding of the transition function for the automaton, this could be continually applied to the reachable states until a fixed point is reached; this fixed point would represent the states of the product automaton. The non-emptiness check can then either use an approach such as [Qian and Nymeyer, 2006], or of the approach from the previous section for performing symbolic model checking over a Rabin-style fairness condition.

### 7.3 Closing Remarks

Verification by model checking is an important technique for identifying and isolating errors present in a multi-agent system. However, current techniques suffer from issues of scalability, given the exponential nature of composing many components. Modular
reasoning—focused on splitting the verification task into many sub-units—is a plausible technique for ameliorating the state-space explosion problem.

In this thesis, we have presented a generic and flexible approach of modular reasoning for multi-agent systems based on temporal-epistemic logic. While we believe that this technique will assist in designers and verifiers of multi-agent systems via the use of a more scalable technique, it is hoped that the contributions put forward in this thesis will enable future research in the area of scalable verification for multi-agent systems.
References


References


References


References


