Abstraction in Model Checking
Multi-Agent Systems

Francesco Russo

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy
Imperial College London, May 2011
Abstract

This thesis presents existential abstraction techniques for multi-agent systems preserving temporal-epistemic specifications. Multi-agent systems, defined in the interpreted system frameworks, are abstracted by collapsing the local states and actions of each agent. The goal of abstraction is to reduce the state space of the system under investigation in order to cope with the state explosion problem that impedes the verification of very large state space systems. Theoretical results show that the resulting abstract system simulates the concrete one. Preservation and correctness theorems are proved in this thesis. These theorems assure that if a temporal-epistemic formula holds on the abstract system, then the formula also holds on the concrete one. These results permit to verify temporal-epistemic formulas in abstract systems instead of the concrete ones, therefore saving time and space in the verification process.

In order to test the applicability, usefulness, suitability, power and effectiveness of the abstraction method presented, two different implementations are presented: a tool for data-abstraction and one for variable-abstraction. The first technique achieves a state space reduction by collapsing the values of the domains of the system variables. The second technique performs a reduction on the size of the model by collapsing groups of two or more variables. Therefore, the abstract system has a reduced number of variables. Each new variable in the abstract system takes values belonging to a new domain built automatically by the tool. Both implementations perform abstraction in a fully automatic way. They operate on multi agents models specified in a formal language, called ISPL (Interpreted System Programming Language). This is the input language for MCMAS, a model checker for multi-agent systems. The output is an ISPL file as well (with a reduced state space).

This thesis also presents several suitable temporal-epistemic examples to evaluate both tech-
niques. The experiments show good results and point to the attractiveness of the temporal-epistemic abstraction techniques developed in this thesis. In particular, the contributions of the thesis are the following ones:

- We produced correctness and preservation theoretical results for existential abstraction.

- We introduced two algorithms to perform data-abstraction and variable-abstraction on multi-agent systems.

- We developed two software toolkits for automatic abstraction on multi-agent scenarios: one tool performing data-abstraction and the second performing variable-abstraction.

- We evaluated the methodologies introduced in this thesis by running experiments on several multi-agent system examples.
I would like to express my gratitude to:

- Alessio Lomuscio
  For his support, for his helping me in every moment, for his patience, for his suggestions, for his advices, for standing my loud voice (and with an horrible Neapolitan accent) during our meetings.

- Mika Cohen
  For his kindness, for his suggestions, corrections and improvements to my work, for his helping me so kindly, and for sharing ideas with me about abstraction.

- Hongyang Qu
  For his help in understanding MCMAS source code.

- Marco Paladini
  A very good friend of mine with whom I got my degree in Naples, and for helping me so often.

- Ioana Cristina Boureanu
  For standing my jokes and for listening to so many songs about Naples.

- Nicoletta Moccia
  Because thanks to her I have always lived immersed in the light without ever knowing the darkness.
• Daniela Diamare
  For having always been close to me despite the distance, and for being the wonderful person she is.

• Salvatore Russo
  for his support.
DECLARATION

I, Francesco Russo, declare that the research work presented in this thesis is my own.
Dedication

to my mother
“There is no such thing as a free lunch”

Robert Anson Heinlein
## Contents

**Acknowledgements** iii

### 1 Preliminaries 1

1.1 Introduction ............................................. 1

1.1.1 General overview ....................................... 1

1.1.2 Definition of the problem ............................... 5

1.1.3 Contributions ........................................... 7

1.1.4 Applications of the abstraction technique presented in this thesis ........ 8

1.1.5 Structure of the thesis ................................. 8

### 2 Literature Review 11

2.1 Modal logics ............................................... 11

2.1.1 Syntax .................................................. 11

2.1.2 Semantics .............................................. 13

2.1.3 Soundness and completeness of a logic ................. 14

2.2 Reasoning about time ..................................... 15

2.2.1 Transition systems ................................... 16

2.2.2 Computation tree logic (CTL) ........................ 16
2.2.3 Linear temporal logic (LTL) ................. 18

2.3 Reasoning about knowledge ....................... 19
  2.3.1 Knowledge in games ......................... 20
  2.3.2 Agents and multi-agent systems .......... 20
  2.3.3 Interpreted systems ......................... 21

2.4 Multi-modal logics ............................... 25
  2.4.1 The multi-modal logic $L^n$ ................ 25
  2.4.2 The logic $S5^n$ ............................. 26
  2.4.3 Computation tree logic of knowledge (CTLK) ............. 29

2.5 Model checking ................................ 30
  2.5.1 Fix-point characterisation of CTL .......... 32
  2.5.2 Symbolic Model Checking and OBDDs .... 36
  2.5.3 Model checking epistemic properties ..... 39
  2.5.4 Model checkers, MCMAS and ISPL ....... 40

2.6 Abstraction .................................. 52
  2.6.1 Limits of symbolic model checking and state of the art of abstraction ... 52
  2.6.2 Symmetry reduction techniques for CTLK .... 52
  2.6.3 Predicate abstraction ....................... 55
  2.6.4 Epistemic abstraction on Kripke structures . 59
  2.6.5 Existential abstraction on Kripke structures .... 61

3 Theoretical results in abstraction of MAS .......... 69
  3.1 Existential abstraction of interpreted systems .... 69
5.3.1 Simplified Black-jack ............................................. 130
5.3.2 Simplified poker game ........................................ 135

6 Conclusion ................................................................. 143

6.1 Thesis contributions ............................................... 143

6.2 Comparison with related work .................................. 144

6.2.1 Symmetry reduction techniques for CTLK ........ 144
6.2.2 Epistemic abstraction on Kripke structures .... 145
6.2.3 Existential abstraction on Kripke structures .... 147
6.2.4 Predicate abstraction ........................................... 149

6.3 Future work ............................................................. 150

Bibliography ............................................................... 152
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Some state space reduction techniques and their main limitations</td>
<td>68</td>
</tr>
<tr>
<td>4.1</td>
<td>Verification results for the card game</td>
<td>111</td>
</tr>
<tr>
<td>4.2</td>
<td>Verification results for the number transmission problem</td>
<td>111</td>
</tr>
<tr>
<td>5.1</td>
<td>Verification results for the simplified Black-jack</td>
<td>135</td>
</tr>
<tr>
<td>5.2</td>
<td>Verification results for the simplified poker game</td>
<td>138</td>
</tr>
<tr>
<td>6.1</td>
<td>Verification results for the muddy children puzzle, from [CDLQ09a]</td>
<td>146</td>
</tr>
<tr>
<td>6.2</td>
<td>Verification results for the NSPK protocol, from [CDLQ09a]</td>
<td>147</td>
</tr>
<tr>
<td>6.3</td>
<td>Verification results for several benchmarks, from [CGJ^+03]</td>
<td>148</td>
</tr>
<tr>
<td>6.4</td>
<td>Some qualitative comparisons with other related techniques</td>
<td>150</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Some research areas belonging to verification .................................................. 2
1.2 Main areas of research and potential areas of influence .................................. 4
1.3 In existential abstraction each concrete edge is represented at the abstract level 5
1.4 Structure of the thesis ...................................................................................... 8
2.1 An informal definition of agent, from Woo09 ...................................................... 21
2.2 A BDT representing the formula $x_1 \land x_2 \land (x_3 \lor x_4)$ .......................... 36
2.3 A BDD representing the formula in Figure 2.2 ............................................... 36
2.4 The MCMAS internal structure, from LR06a .................................................. 43
2.5 An ISPL-program describing a card game ....................................................... 50
2.6 A Verilog program, from CJK04 ................................................................. 57
2.7 The corresponding abstract program of the program in Figure 2.6 from CJK04 58
2.8 The running system, from ED07 .................................................................... 60
2.9 The abstract running system, from ED07 ....................................................... 60
2.10 Existential abstraction, from CGJ+03 ............................................................. 62
2.11 A Kripke structure and a model that simulates it ........................................... 63
2.12 A simulation (left) and a minimal simulation (right) ....................................... 64
Chapter 1

Preliminaries

1.1 Introduction

1.1.1 General overview

Multi-agent systems are computer systems composed of several “entities” that communicate, interact with each other and perform actions by evaluating the internal information the entities own. These systems can be of many types, such as: railroad crossings, web services, cash machines, games, negotiation protocols, transmission protocols, software development protocols, air traffic protocols, security protocols and so forth. To study such systems, computer scientists build mathematical models that are able to capture and reproduce their distinctive patterns of behaviour. In computer science, it is extremely important to have guarantees about the correct behaviour of the systems under investigation. This is one of the main goals of verification.

Verification is the area of computer science concerned with checking the correctness of systems. Some research areas belonging to verification are depicted in Figure 1.1. The areas in bold font in the picture are those with which this thesis deals. There are many techniques that have been developed to check the soundness of computer systems. Nowadays, the most used technique to check software designs is testing. As the name suggests, this technique involves the checking of a great number of executions in search of undesired behaviours, commonly called bugs. Unfortunately, although testing is widely used, the approach does not give any absolute guarantee of correctness. Even after testing a system deeply and widely, a bug might not be
detected, but it may still be present. A crucial goal in verification is to have a technique that can assure us the system is correct.

*Formal verification* is a promising field of computer science that tries to achieve this objective. In formal verification we can find two main fields: *theorem proving* [Bib74] and *model checking* [CES2]. The first one is a deductive verification technique that makes use of automated theorem provers to establish the correctness of a system. This method is applicable to infinite-state systems, but it requires user skills and experience. In contrast to theorem proving, model checking is fully automatic, but it is typically restricted to “small” finite-state systems.

![Diagram](image)

**Figure 1.1:** Some research areas belonging to verification.

Model checking [CES2, CSO1, CGP99, HR04] techniques have been successfully applied in hardware verification, and they have become an industrial standard software tool for hardware designs [BFPW03, GvdM04, NNP+04, RL07, WFHP02]. However, it is often not possible to check the correctness of systems with a high number of states. This is because the number of states in a system grows exponentially with the number of variables used to describe it. In order to solve this problem, known as the *state explosion problem*, many techniques have been introduced and developed in the last twenty years. A contribution to mitigate the state explosion problem was made with the introduction of ordered binary decision diagrams (OBDDs...
1.1. Introduction

McM92, BCM92) to describe boolean functions. OBDDs are directed graphs with a single root. Each node represents a boolean variable in the boolean function, and it has two successors exactly. OBDDs have no cycles and terminate with just two terminal nodes, called 0-terminal and 1-terminal. These diagrams are compact representations of boolean functions. It was shown by Bryant in Br86 that OBDDs admit a unique representation up to a variable ordering. This result was a fundamental breakthrough in model checking as it reduces the problem of establishing whether a specification holds in a given model to the problem of comparing two OBDDs. Representing state spaces with OBDDs allowed the checking of many systems that were intractable before. However, most real systems still remain “too big” (in terms of state space) to be checked even by using OBDDs. Further attempts to mitigate the state explosion problem have been made by state space reduction techniques. Those techniques mainly include: symmetry reductions CDLQ96, CEFJ96, ET99, ID96, CDLQ09a, ES96, ES97 and abstraction BR00, CGJ00, Das03, Kur94. This research focuses on abstraction. Abstraction is a way of simplifying mathematical descriptions of systems. An abstract model is simpler than the original or concrete one and, at the same time, it must capture all important features of the initial system. The goal of abstraction techniques is to reduce the state space of computer systems. Generally speaking, abstraction techniques remove or simplify details as well as cutting entire components from the original design that are “irrelevant” for the property under investigation. The information loss incurred by simplifying the model however has a price: testing an abstract model potentially leads to incorrect results. Fortunately, we have results stating that under some conditions if a property holds in the abstract model that property is valid in the concrete one too. The converse is not guaranteed. In other words, if a property is not valid in the abstract system it may or may not be valid in the actual one. Therefore, from an abstract model we can have information about the concrete system only if a property holds on the abstract one.

There are several abstraction techniques in the computer science literature. These methods can be arranged in two main groups: predicate abstraction techniques DDP99, DD02, DD01, BR00, Das03 and the existential abstraction ones CGL94, CGJ00, CGJ03.

Predicate abstraction was first introduced by Graf et al. in GS97. In order to apply predicate abstraction we need to describe a computer system by a set of logical formulas. A finite set of local state predicates are selected, and any two local states satisfying exactly the same predicates are collapsed. This method makes use of a surjective function that maps a set of concrete states into a set of abstract ones. Conversely, the “concretisation” is performed via a concretisation
A detailed introduction to predicate abstraction can be found in [Das03]. The second method, called existential abstraction, was introduced by Cousot in 1977 [CC77, CC99] for a simple programming language and developed by Edmund Clarke et al. in [CGL94, CGJ+00, CGJ+03].
for the verification of the universal fragment of temporal specifications on Kripke structures \cite{HR04}. Existential abstraction techniques consist of reducing states by collapsing the data values from which they are built. The “size” of a computer system is reduced by means of a surjective function $h$ (also called abstraction function) that “collapses” concrete states together in such a way as to preserve the peculiar characteristic of concrete system. The term “existential” expresses the fact that each transition (or edge) between states in the concrete system is represented by a transition in the abstract system (see Figure 1.3). Unfortunately, this does not guarantee the preservation of entire paths or executions of the concrete system. This phenomenon can cause spurious paths in the abstract system that have no corresponding paths in the concrete one. This thesis focuses on existential abstraction. Figure 1.2 shows where this thesis is located in the verification field and the potential areas this thesis might influence.

1.1.2 Definition of the problem

Existential abstraction is an approach to reduce the time and memory required by model checking techniques. In particular, the mathematical descriptions of systems in real world applications include so many states, to make detecting undesired behaviours unfeasible. Existential abstraction techniques have been useful to increase the feasibility of formal verification, which is important to prevent malfunctions in industrial systems. Currently, existential abstraction typically requires a considerable amount of expertise, creativity, insight and experience. Therefore, the definition and implementation of fully automatic abstraction techniques for computer
systems appears difficult to obtain especially for multi-agent systems in which the concept of “knowledge” plays a fundamental role in the evolution of the system as the agents act with the respect to the information that they have about the current state of the system in which they “live”. For multi-agent systems, particular attention is required in “manipulating” the knowledge of agents in order to reduce the “size” of information that agents know from the concrete description of the system to the abstract one. Epistemic scenarios are suitable to model several situations such as: security protocols [BCL10, BCL09], games [CDLR09, LQR10] and web services [LQSS07, LQS08b]. Model checking can be used in these scenarios, whose state spaces are very large, but new automatic abstraction techniques are needed.

This thesis intends to develop automatic abstraction techniques for multi-agent systems specified in a special formal language, called ISPL (interpreted system program language) for the verification of temporal-epistemic specifications. The objective of this research is to extend existing existential abstraction techniques developed by Clarke et al. to epistemic scenarios in which the concept of knowledge is involved. In this thesis, an existential abstraction technique for epistemic scenarios is presented and discussed in Chapter 3. This thesis also presents two different implementations of existential abstraction for multi-agent systems against temporal-epistemic scenarios. The first implementation performs data-abstraction (presented in Chapter 4). The second implementation performs variable-abstraction (presented in Chapter 5).

The objectives of this thesis are the following:

- To define and investigate existential abstraction techniques for the verification of multi-agent systems.
- To produce theoretical results that state the abstraction techniques presented in this thesis are correct.
- To implement two software tools performing automatic abstraction of multi-agent systems.
- To apply these tools to the verification of several examples from the multi-agent system literature against temporal-epistemic formulas.
- To evaluate the effectiveness of the tools by analysing the time and memory consumptions required by the verification process of abstract systems compared with the verification of
concrete ones.

1.1.3 Contributions

The main contributions of this research work are listed as follows:

1. Theoretical

- We proved validity preservation Theorem 3.1 (Chapter 3).
- We proved simulation Theorem 3.2.1 (Chapter 3).
- We proved preservation Theorem 3.2 for quotient multi agent systems (Chapter 3).
- We proved correctness Theorem 4.1 (Chapter 4).

2. Methodological

- We introduced a systematic data-abstraction algorithm (Chapter 4).
- We introduced a systematic variable-abstraction algorithm (Chapter 5).

3. Implementation

- We implemented a toolkit for the automatic execution of data-abstraction algorithm.
- We implemented a toolkit for the automatic execution of variable-abstraction algorithm.

4. Conceptual

- We exploited the key idea of formula-guided abstraction. The abstraction process is tailored on the specifications we want to check against the interpreted system under investigation.
- We identified two classes of interpreted systems in which to apply data and variable abstraction to achieve the best performance in terms of reduction of the state space.
1.1.4 Applications of the abstraction technique presented in this thesis

The abstraction techniques presented in this thesis can be applied to study very large and complex scenarios in which some requirements are more naturally expressed using intentional properties such as knowledge. In this sense, multi-agent model checking allows for the verification of certain scenarios where some entities interact by performing actions based on their knowledge.

Automatic tools for model checking have been employed successfully in the formal verification of various scenarios. One of the most common applications of model checking was the verification of hardware circuits. Applications of “traditional” model checking to epistemic logics add power and expressiveness to the analysis of games, communication protocols and software development systems.

1.1.5 Structure of the thesis

The structure of the rest of the thesis is organised as follow (see also Figure 1.4):

- **Chapter 1**: Introduction
- **Chapter 2**: Literature Review
  - Modal Logic and MAS
  - Model Checking
- **Chapter 3**: Theoretical Results in Abstraction of MAS
- **Chapter 4**: Automatic Data-Abstraction of MAS
- **Chapter 5**: Automatic Variable-Abstraction of MAS
- **Chapter 6**: Conclusion

Figure 1.4: Structure of the thesis.
• Chapter 2 summarises some background material on multi-agent system theories, temporal epistemic logic, model checking, interpreted system, abstraction in model checking and the main technique of abstraction refinement.

• Chapter 3 presents an abstraction technique for multi-agent systems expressed in the interpreted system framework and the theoretical results obtained in this thesis.

• Chapter 4 presents an automatic data-abstraction technique to abstract multi-agent systems. Several applications of data-abstraction technique are presented and studied. Moreover, experimental results are showed.

• Chapter 5 presents an automatic variable-abstraction technique to abstract multi-agent systems. Application of variable-abstraction and experimental results are showed as well.

• Chapter 6 describes the results obtained, presents open issues and possible extensions of this research work.
Chapter 2

Literature Review

In this Chapter, a review of relevant literature is presented. The background reported here is useful to follow the research presented in this thesis. The next paragraphs present several modal logics, model checking techniques, and abstraction techniques. The material summarises relevant parts of \[BdRV01, Pop94, Che80, HC98, HR04, FHVM95\].

2.1 Modal logics

2.1.1 Syntax

In this paragraph, basic foundations of modal logics are described. The theoretic material presented here will be used in the next paragraphs to define a temporal epistemic modal logic to reason about time and knowledge. Definitions introduced here are from \[HS07, Gol87, HC98\].

Let \( AF \) be a finite set of atomic formulas. The syntax of propositional modal logic is usually defined by the set \( L_{MF} \) formed by the following well-formed formulas expressed in Backus Naur form (BNF):

\[
\psi := p \mid \psi_1 \land \psi_2 \mid \psi_1 \lor \psi_2 \mid \neg \psi_1 \mid \Box \psi_1 \mid \Diamond \psi_1
\]

The following relations hold: \( \neg \Box \neg \psi = \Diamond \psi \); \( \psi_1 \rightarrow \psi_2 = \neg \psi_1 \lor \psi_2 \); \( \bot = \neg \psi \land \psi \). The formula \( \Box \psi \) can be read as “\( \psi \) is necessarily true” and \( \Diamond \psi \) can be read as “\( \psi \) is possibly true”. A schema
or an *axiom* is defined as set of formulas having the same structure or syntactic form. For instance the schema \( \psi \rightarrow \Diamond \psi \) indicates the following set of well-formed formulas: \( \{ \phi \in L_{MF} \mid \phi \rightarrow \Diamond \phi \} \).

**Definition 2.1** (Modal logic). \([HR04]\)

A modal logic is a set \( \mathcal{L} \subseteq L_{MF} \) such that:

1. All tautologies of propositional logic belong to \( \mathcal{L} \).

2. \( \mathcal{L} \) is closed under Modus Ponens: if \( \phi \in \mathcal{L}, \phi \rightarrow \psi \in \mathcal{L} \) then \( \psi \in \mathcal{L} \).

Usually, a logic is defined from a set of axioms and one or more derivation rules. An element of \( \mathcal{L} \) is called *theorem*. The expression \( \varphi \in \mathcal{L} \) means that \( \varphi \) can be obtained by axioms and rules of \( \mathcal{L} \).

**Definition 2.2** (Normal).

A modal logic is called normal if it contains the following axiom:

\[
K : \Box(\phi \rightarrow \psi) \rightarrow (\Box \phi \rightarrow \Box \psi)
\]

and if the logic is closed under the necessitation rule:

\[
\text{if } \psi \in \mathcal{L} \text{ then } \Box \psi \in \mathcal{L}
\]

The smallest normal modal logic is called \( K \). By the union of one or more axioms with the axiom \( K \), it is possible to build larger normal modal logics. The following axioms are well known in modal logic literature \([HR04]\):

- **T** : \( \Box \phi \rightarrow \phi \), called *reflexive* axiom;
- **B** : \( \phi \rightarrow \Box \Diamond \phi \), called *symmetric* axiom;
- **D** : \( \Box \phi \rightarrow \Diamond \phi \), called *serial* axiom;
- **4** : \( \Box \phi \rightarrow \Box \Box \phi \), called *transitive* axiom;
- **5** : \( \Diamond \phi \rightarrow \Box \Diamond \phi \), called *Euclidean* axiom;
From the combination of some of those axioms, it is possible to define well known logics as: KT, KD, KT4. In this thesis, particular attention is given to the logic S5 defined as follows.

**Definition 2.3 (logic S5).** [HR04]

The logic S5 is the normal modal logic characterised by the axioms K, T, 4, 5.

### 2.1.2 Semantics

Usually, in Propositional Logic atomic propositions are evaluated via a Boolean function \( f : AP \rightarrow \{ \text{true}, \text{false} \} \) that returns a boolean value for a given atomic proposition. Therefore, a model in propositional logic is just an assignment of Boolean values. A model of a modal well formed formula is a complex mathematical structure. The most used structures in the modal logic literature are Kripke structures [HR04].

**Definition 2.4 (Kripke structures).**

A Kripke structure \( K = \langle S, R, V \rangle \) is a triple. \( S \) is a set of states. States are atomic as they have no internal structure. The relation V is an interpretation that associates a set of atomic propositions \( AP \) to each state in \( S \) and \( R \) is a binary relation on \( S \). Relations V and R can be also defined as functions (\( V : S \rightarrow 2^{AP} \) and \( R : S \rightarrow S \)) in an equivalent manner.

A modal formula \( \varphi \), defined in the previous paragraph, can be evaluated on a given Kripke structure by defining the following semantics for Kripke structures.

**Definition 2.5 (Kripke structure semantics).**

\[
\begin{align*}
K, s \models p & \iff p \in V(s), \\
K, s \models \neg \varphi & \iff K, s \not\models \varphi, \\
K, s \models \varphi \land \psi & \iff K, s \models \varphi \text{ and } K, s \models \psi, \\
K, s \models \varphi \lor \psi & \iff K, s \models \varphi \text{ or } K, s \models \psi, \\
K, s \models \Box \varphi & \iff K, t \models \psi \forall t \in S \text{ such that } sRt, \\
K, s \models \Diamond \varphi & \iff K, t \models \psi \exists t \in S \text{ such that } sRt.
\end{align*}
\]
The expression \( K, s \models \varphi \) can be read as “the formula \( \varphi \) holds at the state \( s \in S \) on the Kripke structure \( K = \langle S, R, V \rangle \)”. The expression \( K \models \varphi \) means that the Kripke structure is a model for the formula \( \varphi \) at every states \( s \in S \).

Informally, the formula \( \Box \varphi \) is true in a state \( s \) when all accessible states from \( s \) via the relation \( R \) satisfy the formula \( \varphi \). While the formula \( \Diamond \varphi \) is true in a state \( s \) if there is at least one other state connected via the relation \( R \) that satisfies \( \varphi \).

A model \( K \) can be seen as a valued instance of a more general structure called frame. A frame \( F \) is simply a pair \( F = (S, R) \), where \( S \) is the set of states and \( R \) is a binary relation. Therefore, a model can be seen as the pair \( K = (F, V) \), where \( V \) is the evaluation function defined above. A formula is valid in a frame \( F \) (\( F \models \varphi \)) if \( K \models \varphi \) for all models \( K \) built on \( F \) through any valuation \( V \).

### 2.1.3 Soundness and completeness of a logic

The introduction of frames leads us to two important features of logics: soundness and completeness. As we have just seen a frame validates a formula when the formula is true for all valuations for the frame. If we consider a set of frames satisfying a certain property we can define validity of a modal formula on a set of frames. A formula \( \varphi \) is valid with respect to a class of frames \( \delta \) if \( \forall F \in \delta \ F \models \varphi \). In this case, we write \( \delta \models \varphi \).

**Definition 2.6** (Soundness).

A logic \( \mathcal{L} \) is sound with respect to a class of frames \( \delta \) if \( \forall \varphi \in \mathcal{L} \) we have \( \delta \models \varphi \).

**Definition 2.7** (Completeness).

A logic \( \mathcal{L} \) is complete with respect to a class of frames \( \delta \) if \( \forall \varphi \) such that \( \delta \models \varphi \) implies \( \varphi \in \mathcal{L} \).

Given a Kripke structure \( K = \langle S, R, V \rangle \) it is possible to define classes of frames via the relation \( R \). In this way, a logic can be defined by imposing that \( R \) satisfies certain properties. The most common properties are reflexivity, symmetry, transitivity, seriality.

- Reflexivity holds if \( \forall s \in S \) we have \( sRs \).
2.2 Reasoning about time

If we are interested in reasoning about time, we need some more suitable modal operators to reason about temporal concepts such as temporal intervals. Sometimes we need to express some concept such as: “before that”, “after that”, “until that”, “eventually in the future”, “in the next instant” and so forth. If the underlying relation represents temporal successors, we still could use the □ modal operator to express “in the next instant”, but for all the others we need some extra operators.

This section presents some particular modal logics that are extremely useful to model scenarios where the concept of time is involved. Temporal logics were introduced by Arthur Prior [Pri67] as a result of an interest in the relationship between time and modality attributed to the Greek philosopher of the Megarian school Diodorus Cronus, who lived in the 4th century BC, famous for “the paradox of future contingents” [HR04].

Temporal logics form an interesting part of modal logic defined and built to reason about time. The principal distinction among them concerns whether they model time as linear or branching structures. If the future is determined and all time instants are ordered as in a line from past to future, a suitable logic for describing this situation is a linear time logic. A linear temporal logic is introduced in section 2.2.3. This is used for reasoning about deterministic programs. Nevertheless, a linear temporal logic can be applied to the executions or runs of a system that presents many alternative futures. This can be done since a fixed execution represents a single future. Therefore, the nondeterminism of the system can be taken into account by considering all runs of the systems. Hence, linear temporal logics can also be used to express properties...
on non-deterministic programs. However, when the future is not determined and we need to express the existence of a single execution among many, another type of logic is needed. This kind of logic is called branching time logic or computation tree logic. This logic is introduced in section 2.2.2. Not all structures for time fall into the linear or the branching category (for a further discussion see [Wol87, EH86]) but these certainly are the two most often used in the model checking literature.

2.2.1 Transition systems

A transition system [HR04] can be seen as special version of a Kripke structure in terms of semantics. These systems are very useful for representing scenarios where the concept of time is involved. The usual definition for transition systems is the following one:

Definition 2.8 (Transition system).

A transition system is a triple $T = \langle S, t, V \rangle$.

- $S$ is a set of punctual states or states with an internal structure.
- $t \subseteq S \times S$ is a transition relation representing transitions between states in two different times: given two states $s$ and $s'$ of $S$, $(s, s') \in t$ means that $s'$ is the successor of $s$.
- $V : S \rightarrow 2^{AP}$ is an evaluation function that associates to a given state $s \in S$ the set of formulas that hold at the state $s$.

Definition 2.9 (path).

A path $\pi$ in $T$ is an infinite sequence of states $\pi = s_0, s_1, s_2, \ldots, s_n, \ldots$ such that $(s_i, s_{i+1}) \in t$ for all $i \geq 0$.

The $i$-th state in the path $\pi$ is denoted by $\pi(i)$.

2.2.2 Computation tree logic (CTL)

A branching time logic for reasoning about time is introduced. This logic is called CTL (computational tree logic) [CES2, Eme90, HR04, EH82]. The syntax of CTL is defined by

$$\phi ::= p \mid \neg \phi \mid \phi \lor \psi \mid \text{EX}\phi \mid \text{EG}\phi \mid \text{E}[\phi U \psi]$$
2.2. Reasoning about time

The formula EXφ is read “there exists a path π, starting in π(0), such that at the next state π(1) formula φ holds”. EGφ is read “there exists a path such that formula φ holds at each state belonging to the path π”. E[φUψ] is read “there exists a path such that φ holds until ψ holds”.

CTL modalities are made of a pair of symbols, where the first symbol is a quantifier over paths (E corresponds to the quantifier “∃”), while the second symbol expresses some constraint over paths. The semantics of CTL is given in terms of transition systems and CTL-formulas are interpreted at a state s in a transition system T as follows:

**Definition 2.10 (semantics of CTL).**

T, s |= p iff p ∈ V(s);
T, s |= ¬φ iff T, s \not\models φ;
T, s |= φ₁ ∨ φ₂ iff T, s |= φ₁ or T, s |= φ₂;
T, s |= EXφ iff there exists a path π such that π(0) = s, and T, π(1) |= φ;
T, s |= EGφ iff there exists a path π such that π(0) = s, and T, π(i) |= φ, for all i ≥ 0;
T, s |= E[φUψ] iff there exists a path π such that π(0) = s, and there exists k ≥ 0 such that T, π(k) |= ψ, and T, π(j) |= φ for all 0 ≤ j < k;

The expression T |= ϕ denotes that the formula ϕ is valid in every state s ∈ S.

The following relations hold:

\[
\begin{align*}
AX\psi &= \neg EX\neg \psi; \\
AG\psi &= \neg EF\neg \psi; \\
EF\psi &= E[true \ U\psi]; \\
AF\psi &= A[true \ U\psi] = \neg EG\neg \psi; \\
A[\phi U\psi] &= \neg(E[\neg \psi \ U(\neg \phi \land \neg \psi)]EG\neg \psi).
\end{align*}
\]

AXψ is read “for all paths π, starting in π(0), formula ψ holds in the next state π(1)” . AGψ is read “for all paths, formula ψ globally holds along the path”. Formula EFψ is read “there exists at least one path π in which eventually in the future ψ holds”. Formula AFψ is read “for all paths π, eventually in the future ψ holds”. A[φUψ] is read “for all paths, φ holds until ψ holds”. 
2.2.3 Linear temporal logic (LTL)

In this section, the temporal logic LTL (linear temporal logic) [HR04, Pnu81] is introduced. Given a finite set $AP$ of atomic formulas the syntax of LTL is defined as follows

$$\phi ::= p \mid \neg \phi \mid \phi \lor \psi \mid X\phi \mid G\phi \mid \phi U \psi$$

In this definition, $p \in AP$ is an atomic formula. $X\phi$ is read “for all paths at the next state $\phi$ holds”. $G\phi$ is read “for all paths $\phi$ globally holds”. $\phi U \psi$ is read “for all paths $\phi$ holds until $\psi$ holds”. The semantics of LTL is given on transition systems.

Let $T = (S, t, V)$ be a transition system and $\pi = s_0, s_1, s_2, \ldots, s_n, \ldots$ a path in $T$. Whether $\pi$ satisfies an LTL-formula on $T$ is defined by the relation $|=\ $ in the following way:

**Definition 2.11 (Semantics of LTL).**

$T, \pi |= p$ iff $p \in V(s_0)$,
$T, \pi |= \neg \phi$ iff $T, \pi \not|= \phi$,
$T, \pi |= \phi \lor \psi$ iff $T, \pi |= \phi$ or $T, \pi |= \psi$,
$T, \pi |= X\phi$ iff $T, \pi(1) |= \phi$,
$T, \pi |= G\phi$ iff $T, \pi(i) |= \phi$, for all $i \geq 0$,
$T, \pi |= \phi U \psi$ iff there exists some $k \geq 0$ such that $T, \pi(k) |= \psi$, and $T, \pi(j) |= \phi$, for all $0 \leq j < k$.

$T |= \phi$ denotes that the formula $\phi$ is valid for every path $\pi$.

LTL can express path properties. For example, the formula $FG\phi$ reads “eventually, $\phi$ continuously holds”, while the formula $GF\phi$ reads “infinitely often $\phi$ holds”. However, it is not possible to express the “existence” of paths in LTL as this logic implicitly quantifies universally over paths. Therefore, properties that express the existence of a path cannot be encoded in LTL. This problem can partially be solved by considering the negation of the property. However, properties that mix universal and existential path quantifiers cannot be modeled. These can be expressed in CTL. In fact, CTL allows us to explicitly quantify existentially over paths. Notice that CTL does not extend LTL. There are properties that can be expressed in both logics, like “if $\phi$ then for all futures eventually $\psi$”. This is expressed by $AG(\phi \rightarrow AF \psi)$ [HR04] in CTL, and by $G(\phi \rightarrow F \psi)$ in LTL.
There are formulas, e.g., $F\varphi \rightarrow F\psi$ that belong to $\mathbf{LTL}$, but represent properties not expressible in $\mathbf{CTL}$. In fact, $\mathbf{CTL}$ does not allow us to select a range of paths by describing them with a formula, as $\mathbf{LTL}$ does. There are also properties not expressible in either $\mathbf{LTL}$ or $\mathbf{CTL}$. An extension of both $\mathbf{LTL}$ and $\mathbf{CTL}$ is $\mathbf{CTL}^*$ [EH86], a temporal logic not discussed in this thesis. Temporal logics, just introduced, are useful to check important properties of systems. Below some of these properties are reported succinctly:

- The *reachability* [Pap03] property states that there exist a path in the system under investigation that leads to a given state.

- The *safety* [BCRZ99] property states that a certain undesired situation never occurs in the system.

- The *deadlock* [VHB+03] property states that the system permits reaching a state where no further transitions are possible. Therefore there is at least one execution that leads to a dead end.

- The *fairness* [GPSS80] property states that for every execution in the system, a desired event occurs infinitely often.

### 2.3 Reasoning about knowledge

The concept of knowledge has been studied by philosophers of ancient Greece from around the VII century BC. But the Greek epistemology (from Greek: “episteme” = knowledge, “logos” = discourse or reasoning) was focused only in investigation of a single agent’s knowledge (the concept of “agent” will be discussed in the next section in more detail). Very little attention was paid to the interaction of several entities based on their knowledge of an environment.

Systems of interacting agents that share information and knowledge and choose actions based on these data, have become an active field of research in computer science. These have been used in many scenarios like games, security and communication protocols.
2.3.1 Knowledge in games

Games \cite{LR06b, DHMR06} can be divided in two main groups: partial information games and full information games. In the first, players just have fragments of information of the current configuration of the game. This means there are some data non accessible to all players. In the second, players have a complete information of the entire configuration of the system. Therefore, they know all values of all variables of the model that describes the game. A classic example of a full information game is the chess game. In chess, both players know, in every moment, the current state of the game completely as they can see the position of every piece on the chessboard. These type of games are also called “perfect information” games. Moreover, both players know that the adversary knows about the state of the game. However, there are many other games in which players just have partial information about the current configuration as in card games. In these cases, a player might be able to deduce the cards that other players hold or their strategies by looking, for instance, at the cards they discard during the game or by counting the remaining cards. On the other hand, his adversaries can consider what he knows about their cards and their strategies. In these complex scenarios, players’ knowledge plays a crucial role in order to win the game.

2.3.2 Agents and multi-agent systems

This section introduces the concepts of agents and multi-agent systems. The material reported here is a summary of some parts presented in \cite{Woo09}.

Agents

There is not a formal definition of the term agent in computer science. Several informal definitions have been produced in the literature, but there is a little agreement around this term. Usually, the concept of agent indicates an “entity” with some properties. In an agent, we can associate properties such as autonomy (he/she is able to take decision based on information he/she holds), rationality (he/she applies rational strategies to reach a fixed goal), social characteristics (he/she interacts and shares information with other agents) reactivity (he/she is able to take decisions from the changes of the environment) and pro-activity (the ability of following
2.3. Reasoning about knowledge

"An agent is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its delegated objectives."

Figure 2.1: An informal definition of agent, from [Woo09].

a certain strategy to reach a pre-established goal). However, for the purposes of this thesis it is important to give some kind of definition. In this thesis we will consider the informal definition given in Figure 2.1.

Multi-agent systems

A multi-agent system (MAS) [Woo09] indicates a paradigm composed of autonomous agents communicating and interacting with each other. Multi-agent systems have been showing to be very useful in software engineering, such as: computer grid, semantic web, games, economics and security and communication protocols. Moreover, multi-agent systems can be helpful to study and understand several aspect of human societies and to analyse distributed economic systems. Application of multi-agent systems to social sciences from a computation point of view will probably be the new frontier in computer science of the next years. In this sense, not only can multi-agent systems be successfully employed as a modelling paradigms in computer science, but also in a wider number of scenarios belonging to social and economic science.

Multi-agent systems have been used in bounded model checking [LPW07, LLP02, BCCZ99, JL10]. They have been employed to study epistemic scenarios as: security protocols [LP07a, BCL10, BCL09] and games [LR06b, LQR10, CDLR09, DHMR06].

2.3.3 Interpreted systems

In this section, multi-agent systems are described by using complex structures called interpreted systems. This is an intuitive semantics directly connected with the executions of multi-agent systems; the knowledge of agents is strictly related with the notion of local states.

The formalism of interpreted systems was made popular in [FHVA95] to model a multi-agent system and to study epistemic and temporal properties of agents. In this thesis, interpreted
systems are used as a semantic model for multi-agent systems. In this formalism a system is composed of \( n \) agents and an environment. Each agent and the environment are associated with a set of local states and a set of actions. The local states are private to the agent and the environment. Local protocols define the actions that may be executed at a given local state. The local evolution function of the agent defines the transition relation among the local states. The environment has the same structure as the agents. The formal definition of an interpreted system is given as follows.

**Definition 2.12 (Interpreted system).** \([FHVM95]\)

An interpreted system over a set \( Ag = \{1, \ldots, n\} \) of agents and an environment \( e \) is a tuple

\[
I = (\{L_i\}_{i \in Ag} \cup \{L_e\}, \{ACT_i\}_{i \in Ag} \cup \{ACT_e\}, \{P_i\}_{i \in Ag} \cup \{P_e\}, \{t_i\}_{i \in Ag} \cup \{t_e\}, I_0, V)
\]

where:

- \( L_i \) (\( L_e \), respectively) is a non-empty set of possible local states of agent \( i \) (the environment \( e \), respectively).

  The elements \( l_i \in L_i \) (\( l_e \in L_e \), respectively) contain all the information that the agent \( i \) (the environment \( e \), respectively) has about the current configuration in which he/she is.

  The set of possible global states is defined as the Cartesian product of local states, as follows:

  \[
  W = L_1 \times \cdots \times L_n \times L_e
  \]

  For any global state \( w \in W \), \( l_i \) represents the \( i \)-th component in \( w \), i.e., the local state of agent \( i \) in \( w \). Similarly, \( l_e \) represents the local state of the environment \( e \) in \( w \).

- \( ACT_i \) (\( ACT_e \), respectively) is a non-empty set of actions for the agent \( i \) (the environment \( e \), respectively). Elements \( a_i \in ACT_i \) (\( a_e \in ACT_e \), respectively) denote the actions that an agent \( i \in Ag \) (the environment \( e \), respectively) can perform in a given state.

  The joint action \( Act \) is defined as the Cartesian product of the agents’ actions:

  \[
  Act = ACT_1 \times \cdots \times ACT_n \times ACT_e.
  \]

  Let \( \overline{a} \in Act \) be a joint action and \( a_i \) the action of agent \( i \) in \( \overline{a} \). Joint actions are tuples of actions, one for each agent \( i \) and the environment \( e \).
2.3. Reasoning about knowledge

- The protocol $P_i$ is a function $P_i : L_i \rightarrow 2^{\text{Act}_i}$, that associates a set of actions to each local state of the agent $i$ and $P_e \subseteq L_e \times 2^{\text{Act}_e}$ is the local protocol for the environment. The protocol encodes a set of “rules” that establish which actions can be performed in each local state.

- $t_i$ is the local evolution function. This function returns the set of the next local states given a current local state and all actions for each agent. Formally, the local evolution function is defined as follows:

$$t_i : L_i \times \text{Act} \rightarrow 2^{L_i}$$

$n$ is the number of agents in the system. Similarly, $t_e : L_e \times \text{Act} \rightarrow 2^{L_e}$.

- The term $I_0 \in W$ is used to represent the set of initial global states.

- Finally, $\text{AP}$ is a set of atomic propositions and $V$ is the evaluation function defined as follows:

$$V : W \rightarrow 2^{\text{AP}}$$

This function gives the so called “interpretation” to the global states of the system.

Local states are private. Therefore, a local state of an agent $i$ is not accessible to the other agents. Notice that the set of local states and global states are discrete. This is not a limitation as it is possible to model a continuous state system by using a discrete one to any desired level of accuracy. In contrast to local states, actions are public. In the sense that a local action performed by an agent $i$ can be “seen” by all the other agents. The sets of local states and local actions will always be finite in this thesis. It is possible that an agent $i$ can have multiple actions to perform in a given local state. In this case, the agent $i$ chooses an action in a non-deterministic way.

Local protocols and local evolution functions together determine how the entire interpreted system evolves from a global state to the next one.

1The local protocol $P_i$ can be rewritten as the set $P_i \subseteq L_i \times 2^{\text{Act}_i}$.

2The local evolution function $t_i$ can be rewritten as the set $t_i \subseteq L_i \times \text{Act}_1 \times \cdots \times \text{Act}_n \times 2^{L_e}$. The definition of $t_i$ given here is slightly different from that one given in [FHM95]. In [FHM95], $t_i$ returns a single local state. Here instead, $t_i$ returns a non-empty set of local states.
Definition 2.13 (Global transition relation).

Given an interpreted system $IS$, the global transition relation $T \subseteq W \times ACT \times 2^W$ is such that $(w, \pi, W') \in T$ (where $W' \subseteq W$) if and only if:

$$(\forall i \in Ag: \langle w_i, \pi, L'_i \rangle \in t_i \land \langle w_i, a_i \rangle \in P_i \land \langle w_e, \pi, L'_e \rangle \in t_e \land \langle w_e, a_e \rangle \in P_e)$$

where $L'_i \subseteq L_i$ and $L'_e \subseteq L_e$.

In other words, an interpreted system can move from a global state $w$ to a subset of global state $W' \subseteq W$ in one step if there is a joint action available at $w$ which transforms each local state $w_i$ into a set of local states $L'_i$ and $w_e$ to $L_e$.

Definition 2.14 (Total).

A global transition relation $T$ is total if and only if:

$$\forall w \in W \exists W' \subseteq W (wT W' \land W' \neq \emptyset)$$

In the following, the global transition relation $T$ is assumed to be total.

Definition 2.15 (Path).

A path $\pi$ in an interpreted system $IS$ is an sequence (either finite or infinite) $w^0w^1\ldots$ of global states belonging to $W$ such that every pair of successive states forms a transition, i.e., $w^kT w^{k+1}$ for all $k \geq 0$.

The expression $\pi(k)$ means the $k$-th global state in $\pi$, i.e., $w^k$.

Definition 2.16 (Reachable states).

A global state $w \in W$ is reachable if and only if it can be reached by a finite path $\pi$ starting from an initial state, i.e., $\pi(0) \in I_0$. Let $G$ denote the set of reachable states.

Usually [FHVM95] the knowledge of an agent is defined by means of relations over global states defined as follows.

Definition 2.17 (Epistemic indistinguishability relation).

The epistemic indistinguishability relation for agent $i \in Ag$ in an interpreted system $IS$ is:

$$\sim_i = \{(w, w') \in W \times W \mid l_i = l'_i\}$$
The semantics of interpreted systems is given in section 2.4.3.

2.4 Multi-modal logics

This section introduces multi-modal logics by summarising parts of [HR04].

2.4.1 The multi-modal logic $L^n$

Let $AF$ be a finite set of atomic formulas. The syntax of an multi-modal logic is usually defined by the set $L^n_{MF}$ formed by the following well-formed formulas expressed in BNF:

$$
\psi ::= p \mid \neg \psi \mid \psi \lor \psi \mid \square_1 \psi \mid \square_2 \psi \mid \ldots \mid \square_n \psi.
$$

The other modalities are introduced as in the mono-modal case.

Definition 2.18 (multi-modal logic). [HR04]

A multi-modal logic is a set $L^n \subseteq L^n_{MF}$ such that:

1. $L^n$ includes all tautologies of propositional logic.

2. $L^n$ is closed under Modus Ponens: $\phi \in L, \phi \rightarrow \psi \in L$ then $\psi \in L^n$.

Definition 2.19 (Normal). [HR04]

A multi-modal logic $L^n$ is called normal if it contains the following multi-modal axiom:

$$K_i : \square_i (\phi \rightarrow \psi) \rightarrow (\square_i \phi \rightarrow \square_i \psi)$$

for all $i$ and if the logic is closed under the necessitation rule:

if $\psi \in L^n$ then $\forall i \square_i \psi \in L$

The smallest multi-modal normal logic is represented by $K_n$. The semantics for multi-modal logic is defined via generalised Kripke structures.
Definition 2.20 (Kripke structure for a multi-modal logic).

A Kripke structure for a multi-modal logic is a tuple $K = \langle S, R_1, R_2, \ldots, R_n, V \rangle$, where $S$ is a set of possible worlds, $R_i \subseteq S \times S$ ($i \in \{1, \ldots, n\}$) are $n$ epistemic indistinguishability relations, and $V : S \rightarrow 2^{\text{AF}}$ is an evaluation function that associates a set of atomic formulas holding at a given state $s \in S$.

Definition 2.21 (Semantics). [HR04]

Given a Kripke structure $K = \langle S, R_1, R_2, \ldots, R_n, V \rangle$ and a state $s \in S$, the semantics for multi-modal logic is defined as follows:

- $K, s \models p$ iff $p \in V(s)$,
- $K, s \models \neg \varphi$ iff $K, s \not\models \varphi$,
- $K, s \models \varphi \lor \psi$ iff $K, s \models \varphi$ or $K, s \models \psi$,
- $K, s \models \Box_i \varphi$ iff $K, t \models \psi \forall t \in S$ such that $s R_i t$.

As for the mono-modal case, a Kripke structure $K$ can be seen as a valued instance of the frame $F = (S, R_1, \ldots, R_n)$. Therefore, a model can be seen as the pair $K = (F, V)$. A formula is valid in a frame $F$ ($F \models \varphi$), if $K \models \varphi$ for all models $K$ built on $F$ through any valuation $V$.

2.4.2 The logic $S5^n$

The logic $S5$, introduced in section 2.1.1, can be used to reason about the knowledge of a single agent. However, in multi-agent systems different agents have different information about the “world” in which they interact with each other. In these paradigms, it is crucial to describe what an agent knows about his knowledge as well as what the agent knows about the knowledge of the other agents. This section describes how the logic $S5$ can be generalised to multiple agents. The material presented here summarises parts of [HR04].

The modal operators $\Box_i$ will be rewritten as $K_i$ to emphasise the epistemic use. The formula $K_i \varphi$ can be read “the agent $i$ knows the information formally expressed in the formula $\varphi$”. It is possible to define a language to reason about knowledge given a set of formulas. From the connective operators $\neg, \land, \lor, \rightarrow$ and epistemic operators introduced above for each agent, now it is possible to express some statements like “agent1 knows that agent2 knows $\varphi_1$ and agent3...
knows he knows agent4 knows $\varphi_2$” in this way:

$$K_1(K_2\varphi_1 \land K_3(K_4\varphi_2))$$

The axioms $K$, $T$, $4$, $5$ introduced in section 2.1.1 can be easily extended to the multi-dimensional case and (by substituting each modal operator $\Box$, with the operator $K_i$ to underline the epistemic meaning) those axioms can be interpreted here in the following way:

- The axiom $K$ can be read as “An agent has a knowledge that is closed under logical consequence”. This axiom is sometimes called the omniscience axiom.

- The axiom $T$ ($K_i\varphi \rightarrow \varphi$, $i = 1, \ldots, n$) can be read as “the agent $i$ knows only true facts”. This is also called the truth axiom.

- The axiom $4$ ($K_i\varphi \rightarrow K_iK_i\varphi$, $i = 1, \ldots, n$) can be read as “if the agent $i$ knows something than the agent $i$ knows he/she knows it”. This is also called the positive introspection axiom.

- The axiom $5$ ($\neg K_i\varphi \rightarrow K_i\neg K_i\varphi$, $i = 1, \ldots, n$) can be read as “if the agent $i$ does not knows something than the agent $i$ knows he/she does not know it”. This is also called the negative introspection axiom.

Given a group of agents $Ag = \{1, 2, \ldots, n\}$, let us introduce the following epistemic operators:

- The modality $E_{Ag}$ universally quantifies over all agents $i \in Ag$. For instance, the well formed epistemic formula $E_{Ag}\phi$ means that “everyone in the group $Ag$ knows the information expressed by the modal formula $\phi$”. Hence, the formula $E_{Ag}\phi$ expresses more concisely the following formula:

$$E_{Ag}\phi = K_1\phi \land K_2\phi \land \ldots \land K_n\phi$$

- Sometimes, it is fundamental to express a concept of knowledge “deeper” or “greater” than the concept expressed by the modality $E_{Ag}$. If we want to express the concept “everyone knows that everyone knows that...” infinite times, we can use the common knowledge modality $C_{Ag}$. The epistemic formula $C_{Ag}\phi$ means that “it is a common
knowledge in the group $Ag$ the information encoded by formula $\phi$”. Therefore, formula $C_{Ag}\phi$ expresses succinctly the following infinite conjunction:

$$ C_{Ag}\phi = E_{Ag}\phi \land E_{Ag}E_{Ag}\phi \land E_{Ag}E_{Ag}E_{Ag}\phi \land \ldots $$

- Finally, the modality $D_{Ag}$ is introduced. The formula $D_{Ag}\phi$ expresses the concept that the knowledge of an information encoded by a formula $\phi$ is distributed among the group of agents $Ag$, even though no agent in that group may actually know $\phi$. However, the group of agent could know the information $\phi$ if they shared all the fragments of information they have.

A formula in $S5^n$ is defined by the following BNF syntax:

$$ \phi ::= p \mid \neg \phi \mid \phi \lor \psi \mid K_i\phi \mid E_{Ag}\phi \mid C_{Ag}\phi \mid D_{Ag}\phi $$

The semantics of $S5^n$ is given in terms of Kripke structures.

**Definition 2.22** (Semantics).

*Given a Kripke structure $K = (S, R_1, \ldots, R_n, V)$, a set of agents $Ag = \{1, \ldots, n\}$ and a state $s \in S$:

$K, s \models p$ iff $p \in V(s)$,

$K, s \models \neg \phi$ iff $K, s \not\models \phi$,

$K, s \models \phi \lor \psi$ iff $K, s \models \phi$ or $K, s \models \psi$,

$K, s \models K_i\psi$ iff $K, t \models \psi$ for all $t$ such that $sR_it$,

$K, s \models E_{Ag}\phi$ iff for all $i \in Ag$, $K, s \models K_i\phi$,

$K, s \models C_{Ag}\phi$ iff for each $k \geq 1$ we have $K, s \models E_{Ag}^k\phi$, where $E_{Ag}^k\phi$ means $E_{Ag}E_{Ag} \ldots E_{Ag} - k$ times,

$K, s \models D_{Ag}\phi$ iff for each $y \in S$, $K, y \models \phi$, whenever $sR_iy$ for all $i \in Ag$.*

It can be proved a formula $\varphi$ of $S5^n$ is valid if $\varphi$ is valid in all frames where $R_1, \ldots, R_n$ are equivalence relations.
2.4.3 Computation tree logic of knowledge (CTLK)

In this section interpreted systems are used to give a semantics to \( \text{CTL} \) enriched with epistemic modalities introduced before. This logic is called computation tree logic of knowledge, or more concisely: \( \text{CTLK} \). \( \text{CTLK} \) was defined to reasoning about knowledge and time in [PL03].

The syntax of \( \text{CTLK} \) is defined as follows:

\[
\phi ::= p \mid \neg \phi \mid \phi \lor \phi \mid \text{EX} \phi \mid \text{EG} \phi \mid \text{E}[\phi[U] \psi] \mid K_i \phi
\]

where \( p \in AP \) is an atomic proposition, and the operators \( \text{EX}, \text{EG}, \text{EU} \) are the standard \( \text{CTL} \) operators. The remaining \( \text{CTL} \) operators \( \text{EF}, \text{AX}, \text{AG}, \text{AU} \) and \( \text{AF} \) can be defined in the standard way.

It has been shown in [LR04] that there is a natural correspondence between interpreted systems and Kripke structures and therefore between interpreted systems and transition systems. Given an interpreted system \( IS \), the \( \text{CTL} \)-operators are interpreted via the global transition relation \( T \) defined in Definition 2.13, while the epistemic operators \( K_i \) are interpreted in the interpreted system framework by the epistemic indistinguishability relation \( \sim_i \) introduced in Definition 2.17.

**Definition 2.23 (Semantics).**

Let \( IS \) be an interpreted system defined over a set \( \text{Ag} \) of agents and a set \( AP \) of atomic propositions. Let \( \phi \) be a \( \text{CTLK} \)-formula defined over \( \text{Ag} \) and \( AP \), and let \( w \in W \) be a global state. The semantics of \( \text{CTLK} \) is defined by the following conditions:

\[
(IS, w) \models p \text{ iff } p \in V(w) \text{ with } p \in AP; \\
(IS, w) \models \neg \phi \text{ iff } (IS, w) \not\models \phi; \\
(IS, w) \models \phi_1 \lor \phi_2 \text{ iff } (IS, w) \models \phi_1 \text{ or } (IS, w) \models \phi_2; \\
(IS, w) \models \text{EX} \phi \text{ iff there exists a path } \pi \text{ such that } \pi(0) = w, \text{ and } (IS, \pi(1)) \models \phi; \\
(IS, w) \models \text{EG} \phi \text{ iff there exists a path } \pi \text{ such that } \pi(0) = w, \text{ and } (IS, \pi(i)) \models \phi, \text{ for all } i \geq 0; \\
(IS, w) \models \text{E}[\phi[U] \psi] \text{ iff there exists a path } \pi \text{ such that } \pi(0) = w, \text{ and there exists } k \geq 0 \text{ such that } (IS, \pi(k)) \models \psi, \text{ and } (IS, \pi(j)) \models \phi \text{ for all } 0 \leq j < k; \\
(IS, w) \models K_i \phi \text{ iff } \forall w' \in W \ w \sim_i w' \text{ implies } (IS, w) \models \phi.
\]

A formula \( \phi \) is true in a model, written \( IS \models \phi \), if \( (IS, w) \models \phi \) for all \( w \in W \).
In recent years several extensions of $\text{CTLK}$ have been defined. One of these is $\text{CTLK}^{EDC}$. This logic expresses $\text{CTLK}$-formulas enriched with $\text{E}_{Ag}$, $\text{C}_{Ag}$ and $\text{D}_{Ag}$ modalities already introduced for the logic $\text{S5}^n$. The semantics for these operators is given in terms of the following relations $[\text{PL03}]:$

$$
\sim_{E_{Ag}} = \bigcup_{i \in Ag} \sim_i;
$$

$$
\sim_{D_{Ag}} = \bigcap_{i \in Ag} \sim_i;
$$

$$
\sim_{C_{Ag}} = (\sim_{E_{Ag}})^*.
$$

(i.e., the relation $\sim_{C_{Ag}}$ is defined as the transitive closure of the relation $\sim_{E_{Ag}}$).

The semantics for $\text{E}_{Ag}$, $\text{C}_{Ag}$, $\text{D}_{Ag}$ is given as follows:

$$(IS, w) \models \text{E}_{Ag}\varphi \text{ iff for all } w' \in W \text{ such that } \sim_{E_{Ag}}(w, w') \text{ we have } (IS, w') \models \varphi;$$

$$(IS, w) \models \text{D}_{Ag}\varphi \text{ iff for all } w' \in W \text{ such that } \sim_{D_{Ag}}(w, w') \text{ we have } (IS, w') \models \varphi;$$

$$(IS, w) \models \text{C}_{Ag}\varphi \text{ iff for all } w' \in W, \text{ such that } \sim_{C_{Ag}}(w, w') \text{ we have } (IS, w') \models \varphi.$$

### 2.5 Model checking

The main problem in formal verification is to establish whether or not a given formula $\phi_P$, representing a property $P$ expressed in some logic, is true in a given mathematical model $M_S$ (with either a finite set or an infinite set of states) describing a computer system $S$. In other words, formal verification aims to give an answer to the following question:

$$M_S \models \phi_P$$

Mainly, there are two methods in formal verification for solving this problem: theorem proving $[\text{Bib74}]$ and model checking $[\text{CS01, CGP99, CES86, CJLW99}]$. In this section, only model checking is discussed.

Pioneering work in model checking was made by Clarke et al. in 1982 $[\text{CES2, CES3}]$. Initially, model checking was successfully applied to hardware designs only. However, in recent years model checking has become very useful to test software designs as well.

Model checking techniques are composed of two main steps:
Modelling
In the first step a Kripke structure, or an interpreted system, or a transition system that represents the system $S$ is defined. The property $P$ to check (also called specification) is encoded in a suitable modal logic formula $\phi_P$. Typically, the logics used are either LTL or CLT (or extensions of these logics). The expressivity of a logic is crucial to describe the requirements for the system under investigation.

Checking
In the second step we need to determine whether or not the chosen mathematical structure $M_S$ is a model of the formula $\phi_P$ in an automatic way. For CTL and CTLK this can be done automatically by using a well defined procedure called: the labelling algorithm (introduced in the next section).

The most common properties checked are safety [BCRZ99], reachability [Pap03], deadlock [VHB+03] and fairness [GPSS80] properties presented above.

The labelling algorithm for CTL

This section describes the labelling algorithm [HR04] for CTL, called MCAlgo.

For a given CTL-formula $\varphi$ and model $M$, MCAlgo calculates the set of states $[\varphi]$ in which $\varphi$ is true. Given a transition system $M = (S, t, V)$, in order to check whether $M \models \varphi$, MCAlgo starts labelling the states of $M$ with the sub-formulas of $\varphi$ that are satisfied in those states. The algorithm starts with the smallest sub-formulas and “moves” toward $\varphi$.

The labelling algorithm is described as follows [HR04]:

- If $\phi$ is an atomic formula $p$ then MCAlgo labels the states $s \in S$ with $p$ if $p \in V(s)$.
- If $\phi$ is the conjunction $\phi_1 \land \phi_2$ then MCAlgo labels $s$ with $\phi_1 \land \phi_2$ if $s$ is already labelled with both $\phi_1$, $\phi_2$.
- If $\phi$ is $\neg \psi$ then MCAlgo labels the states $s \in S$ with $\neg \psi$ if $s$ is not already labelled with $\psi$.
- If $\phi$ is $\text{AF} \phi$ then:
  - if any state is labelled with $\phi$ then MCAlgo labels it with $\text{AF} \phi$. 
- MCAlgo repeats the operation of labelling each state \( s \) with \( \text{AF} \phi \) if all successors states of \( s \) are already labelled with \( \text{AF} \phi \), until there is no change.

- If \( \phi \) is \( \text{E}(\phi_1 \cup \phi_2) \) then:
  - if any state \( s \) is labelled with \( \phi_2 \) then MCAlgo labels it with \( \text{E}(\phi_1 \cup \phi_2) \).
  - MCAlgo repeats the operation of labelling each state with \( \text{E}(\phi_1 \cup \phi_2) \) if it is labelled with \( \phi_1 \) and at least one of its successors is labelled with \( \text{E}(\phi_1 \cup \phi_2) \), until there is no change.

- If \( \phi \) is \( \text{EX}(\phi) \) then:
  MCAlgo labels each state \( s \) with \( \text{EX}(\phi) \) if there is at least one of its successors labelled with \( \phi \).

### 2.5.1 Fix-point characterisation of CTL

The purpose of this section is to show that the labelling algorithm, introduced in the previous section, terminates. This can appear obvious when the formula \( \phi \) is made of the connectives \( \neg \), \( \land \) only. Nevertheless, the temporal operators \( \text{EG} \), \( \text{EU} \) need a further discussion. This section shows that the definitions of operators \( \text{EG} \) and \( \text{EU} \) lead to fix-points that assure us the labelling algorithm terminates for \( \text{CTL} \)-formulas. The material presented here summarises standard results from [HR04].

Let \( \phi \) be a \( \text{CTL} \)-formula, let \( T = (S, t, V) \) be a transition system and let \([\phi] \subseteq S\) denote the set of states of \( T \) in which the formula \( \phi \) is true. The labelling algorithm works recursively on the structure of the formula \( \phi \). If \( \phi \) is an atomic formula \( p \) then the set \([\phi]\) is built immediately. If the formula \( \phi \) is the conjunction \( \phi_1 \land \phi_2 \) or the disjunction \( \phi_1 \lor \phi_2 \), then the algorithm calculates the two sets \([\phi_1]\) and \([\phi_2]\) separately and subsequently computes the intersection \([\phi_1] \cap [\phi_2]\) and the union \([\phi_1] \cup [\phi_2]\) respectively. If the formula \( \phi \) is \( \text{EX}\phi \), then the labelling algorithm computes the set of all states which have a transition into the set \([\phi]\). Let call this set \( \text{pre}_3([\phi]) \). Formally, given a set \( X \subseteq S \), the function \( \text{pre}_3(X) : 2^S \rightarrow 2^S \) is defined as follows:

\[
\text{pre}_3(X) = \{ s \in S \mid \exists s'(s' \in X \land t(s, s')) \} = Y
\]

Therefore, this function takes a set of states \( X \subseteq S \) as input and returns the of states \( Y \subseteq S \) from which it is possible to move into a state belonging to \( X \).
Now, if the formula $\phi$ is $\text{EG}\phi$ or $\text{EU}\phi$ the construction of the sets $[\text{EG}\phi]$, $[\text{E} (\phi_1 \text{U} \phi_2)]$ is not immediate as for the previous cases. However, by observing that

$$
\text{EG}\phi \equiv \phi \land \text{EXEG}\phi
$$

$$
\text{E}(\phi \text{U} \psi) \equiv \psi \lor (\phi \land \text{EX}(\phi \psi))
$$

it is possible to rewrite the relations above as follows:

$$
[\text{EG}\phi] \equiv [\phi ] \cap [\text{EXEG}\phi] \quad (2.1)
$$

$$
[\text{E}(\phi \text{U} \psi)] \equiv [\psi ] \cup ([\phi ] \cap [\text{EX}(\phi \psi)]) \quad (2.2)
$$

Moreover, by considering that $[\text{EX}\phi] = \text{pre}_3([\phi ])$ it is possible to rewrite the equivalences $2.1$ and $2.2$ in terms of the function $\text{pre}_3$ in the following way:

$$
[\text{EG}\phi] \equiv [\phi ] \cap \text{pre}_3([\text{EG}\phi]) \quad (2.3)
$$

$$
[\text{E}(\phi \text{U} \psi)] \equiv [\psi ] \cup ([\phi ] \cap \text{pre}_3([\text{E}(\phi \psi)])) \quad (2.4)
$$

The equivalences $2.3$ and $2.4$ show a circularity that might suggest the impossibility of calculating the sets $\text{pre}_3([\text{EG}\phi])$ and $\text{pre}_3([\text{E}(\phi \psi)])$. However, by The Knaster-Tarski Theorem [Tar55, CLT08] the sets above admit fix points, as follows.

**Definition 2.24** (Monotonic).

*Let $Q$ be a set of states and let $\tau$ be a function such that given a subset $X \subseteq Q$ returns a subset $Y \subseteq Q$ (i.e.; $\tau : 2^Q \longrightarrow 2^Q$). A function $\tau$ is said to be a monotonic if the following holds:

$$
X \subseteq Y \Rightarrow \tau(X) \subseteq \tau(Y)
$$

The Knaster-Tarski Theorem states that if a function $\tau$ is monotonic then there exists a greatest and a smallest fix-point for the function $\tau$. This theorem also provides a recipe for computing fixed points. The set $\nu Z.\tau(Z)$ denotes the greatest fix point and $\mu Z.\tau(Z)$ denotes the smallest fix point.

Now, let define $\tau^i(X)$ by induction as follows:
1. base case: $\tau^0(X) = X$

2. inductive step: $\tau^{i+1}(X) = \tau(\tau^i(X))$.

If $Q$ is finite and $\tau$ is monotonic, then there exist two integer numbers $n, m$ such that $\nu Z. \tau(Z) = \cap_i \tau^n(Q)$ and $\mu Z. \tau(Z) = \cup_i \tau^m(\emptyset)$. The monotonicity property of the $\tau$ operator is used with the following relations to define a very useful algorithm for CTL-model checking.

**Algorithm 1** The labelling algorithm MCAlgo, from [HR04].

1. $MCAlgo(\varphi, M)$ {
2. IF $\varphi$ is an atomic formula: return $V(\varphi)$;
3. IF $\varphi$ is $\neg \varphi_1$: return $S \setminus MCAlgo(\varphi, M)$;
4. IF $\varphi$ is $\varphi_1 \lor \varphi_2$: return $MCAlgo(\varphi_1, M) \cup MCAlgo(\varphi_2, M)$;
5. IF $\varphi$ is $\varphi_1 \land \varphi_2$: return $MCAlgo(\varphi_1, M) \cap MCAlgo(\varphi_2, M)$;
6. IF $\varphi$ is $EX\varphi$: return $MCAlgo_{EX}(\varphi, M)$;
7. IF $\varphi$ is $EG\varphi$: return $MCAlgo_{EG}(\varphi, M)$;
8. IF $\varphi$ is $E(\varphi_1 U \varphi_2)$: return $MCAlgo_{EU}(\varphi_1, \varphi_2, M)$;
9. }

**Algorithm 2** The support procedure MCAlgo$_{EX}(\varphi, M)$, from [HR04].

1. $MCAlgo_{EX}(\varphi, M)$ {
2. $X = MCAlgo(\varphi, M)$;
3. $Y = \text{pre } \exists(X)$;
4. return $Y$;
5. }

Let $\tau_{EG, \phi} : 2^S \rightarrow 2^S$ be the operator defined by $\tau_{EG}(X) = [\phi] \cap pre_\exists(X)$, and let $\tau_{EU, \phi, \psi} : 2^S \rightarrow 2^S$ be defined by $\tau_{EU, \phi, \psi}(X) = [\phi] \cup ([\psi] \cap pre_\exists(X))$.

Equations 2.3 and 2.4 imply that $[EG\phi]$ is the fix-point of the operator $\tau_{EG, \phi}$ while $[E(\phi U \psi)]$ is the fix-point of $\tau_{EU, \phi, \psi}$. It is possible to prove that the operators $\tau_{EG, \phi}$ and $\tau_{EU, \phi, \psi}$ are monotonic, and that $[EG\phi]$ is the greatest fix-point of $\tau_{EG, \phi}$ while $[E(\phi U \psi)]$ is the smallest fix-point of $\tau_{EU, \phi, \psi}$ [CGP99, HR04]. Therefore, there exist finite natural numbers $n$ and $m$ such that $[EG\psi] = \tau_{EG, \psi}^n(S)$ and $[E(\psi U \phi)] = \tau_{EU, \psi, \phi}^m(\emptyset)$. 
Algorithm 3 The support procedure MCAlgo\textsubscript{EG}(\varphi, M), from [HR04].

1: \textit{MCAlgo\textsubscript{EG}}(\varphi, M)\
2: \textbf{X} = \textit{MCAlgo}(\varphi, M);\
3: \textbf{Y} = S;\
4: \textbf{Z} = \emptyset;\
5: \textbf{while} (\textbf{Z} \neq \textbf{Y}) \textbf{do}\
6: \textbf{Z} = \textbf{Y};\
7: \textbf{Y} = \textbf{X} \cap \textit{pre\exists}(\textbf{Y});\
8: \textbf{end while}\
9: \textbf{return} \textbf{Y};\
10: \};

Algorithm 4 The support procedure MCAlgo\textsubscript{EU}(\varphi_1, \varphi_2, M), from [HR04].

1: \textit{MCAlgo\textsubscript{EU}}(\varphi_1, \varphi_2, M)\
2: \textbf{X} = \textit{MCAlgo}(\varphi_1, M);\
3: \textbf{Y} = \textit{MCAlgo}(\varphi_2, M);\
4: \textbf{Z} = \emptyset;\
5: \textbf{W} = S;\
6: \textbf{while} (\textbf{Z} \neq \textbf{W}) \textbf{do}\
7: \textbf{W} = \textbf{Z};\
8: \textbf{Z} = \textbf{Y} \cup (\textbf{X} \cap \textit{pre\exists}(\textbf{Z}));\
9: \textbf{end while}\
10: \textbf{return} \textbf{Z};\
11: \};
The labelling algorithm is described by Algorithm 1. The additional procedures \( \text{MCAlgo}_{\text{EX}}(\varphi_1, M), \text{MCAlgo}_{\text{EG}}(\varphi_1, M) \), and \( \text{MCAlgo}_{\text{EU}}(\varphi_1, \varphi_2, M) \) are described by Algorithms 2, 3, 4 respectively.

### 2.5.2 Symbolic Model Checking and OBDDs

This section introduces ordered binary decision diagrams (OBDDs) \cite{Bry86, McM92, BCM92, HR04}. They are extremely useful in automatic verification because they mitigate the state explosion problem by reducing the problem of checking whether or not a given structure \( M \) is a model for a given formula \( \varphi \) to the problem of comparing two OBDDs.

OBDDs represent Boolean functions in a compact way. Boolean functions are defined on Boolean variables. Therefore, Boolean formulas can be seen as Boolean functions. For every

\[
\begin{align*}
\text{Figure 2.2: A BDT representing the formula } x_1 \land x_2 \land (x_3 \lor x_4). \\
\text{Figure 2.3: A BDD representing the formula in Figure 2.2.}
\end{align*}
\]
Boolean function \(f(x_1, \ldots, x_n)\), it is possible to associate a specific diagram, called BDD, by labelling each node of the diagram with a Boolean variable.

**Definition 2.25 (BDD).** [HR04]

A BDD (Binary Decision Diagram) \(D\) is a rooted, directed, acyclic diagram in which each node has exactly two edges to successive nodes. \(D\) describes a unique Boolean function. In \(D\) a fixed assignment to its Boolean variables is given. We start from the root and follow the edge labelled with 1 if the current variable takes the value 1. Similarly, we proceed for value 0. We travel along the tree in a up-bottom fashion until reaching a terminal node that represents the value of the Boolean function.

A binary decision tree (BDT) is a rooted, directed, acyclic tree in which each node in the tree has exactly one predecessor. In a BDD it is possible that more than one node has the same successor. This cannot happen in a BDT.

Figure 2.2 shows a BDT describing the Boolean function \(f(x_1, x_2, x_3, x_4) = x_1 \land x_2 \land (x_3 \lor x_4)\). Notice that the BDT contains many redundancies, while the BDD shown in Figure 2.3 is a more compact representation.

An ordered-BDD (OBDD) is a BDD with a fixed ordering on a list variables \(x_1, \ldots, x_n\) of a Boolean function \(f\).

There are several algorithms to reduce OBDDs. One of those is called the algorithm `reduce` [HR04]. This algorithm traverses the OBDD in input layer by layer starting from the terminal nodes and moving up toward the root. The procedure `reduce` labels each node \(n\) with an integer \(id(n)\) in such a way that two nodes which are the roots of two identical sub-diagrams get the same integer label. Let \(low(n)\) to be the left successor of \(n\) and \(hi(n)\) be the right successor of \(n\). Let us assume that the algorithm `reduce` has already labelled all nodes of a layer \(i + 1\). At the level \(i\), three different situations can occur:

1. If \(id(hi(n)) = id(low(n))\) then `reduce` labels the node \(n\) with \(id(n)\) (i.e., the node \(n\) makes a redundant check).

2. If there is another node \(m\) representing the variable of \(n\) with \(id(low(n)) = id(low(m))\) and \(id(hi(n)) = id(hi(m))\) then `reduce` labels the node \(m\) with \(id(n)\) (i.e., the two nodes \(m, n\) compute the same Boolean function).
3. If the two previous cases do not occur then the algorithm \textit{reduce} labels the node \( n \) with a new integer.

Finally, two sub-diagrams rooted at nodes \( n \) and \( m \) labelled with the same integer are merged together. An OBDD \( D \) is called reduced-OBDD (\textit{ROBDD}) if we have applied the algorithm \textit{reduce} to \( D \). Therefore, in \( D \) no further reduction can be performed.

ROBDDs have been successfully used in model checking. The field of model checking that uses binary decision diagrams is called \textit{symbolic model checking}. The method used in symbolic model checking is the following one:

1. Two ROBDDs are built; one for the specification \( \varphi \) and the other one for the model \( T = (S, t, V) \).

2. By comparing the two diagrams is it possible to establish whether the specification \( \varphi \) holds in the model \( T \).

The technical details of symbolic model checking technique are discussed below [HR04].

**Representing subsets of the set of states.** Given a model \( T = (S, t, V) \), the goal is to use Boolean functions and therefore ROBDDs to represent subsets of the set \( S \). Each state \( s \in S \) is represented by a Boolean vector of values \( x = (x_1, x_2, \ldots, x_n) \), where \( x_i \in \{0, 1\} \), \( \forall i \in \{1, \ldots, n\} \). Each vector \( x \) is described by a Boolean formula \( f \) represented by a conjunction of variables or their negations. Subsets of states are described by a Boolean formula \( F \) made of the disjunction of Boolean formulas \( f \), where each \( f \) represents a single state. For instance: given two states \( s_1 = (1, 0) \) represented by \( f_1 : x_1 \land \neg x_2 \) and \( s_2 = (1, 1) \) represented by \( f_2 : x_1 \land x_2 \), a set \( S = \{s_1, s_2\} \) is represented by the Boolean function \( F : (x_1 \land \neg x_2) \lor (x_1 \land x_2) \).

**Representing the transition relation.** The transition relation \( t \) is a subset of the Cartesian product \( S \times S \). Therefore, a single transition \( t(s, s') \) from \( s \) to \( s' \) can be represented by pair of Boolean vectors \( (x, x') \), where the first vector \( x \) represents the state \( s \), and the second vector \( x' \) represents the state \( s' \). Finally, a Boolean formula \( f : x \land x' \) represents the transition \( t(s, s') \). The entire transition function \( t \) is represented by the disjunction of all formulas \( f \).

**The labelling algorithm and Boolean formulas.** The labelling algorithm operates on the structure of a formula \( \phi \) and builds the set \( [\phi] \) of states at which the formula is satisfied. All the
operations on sets can be represented by Boolean connectives. The union (intersection) of two sets is represented by the connective $\lor$ $(\land)$. Given two set $A, B$, the complementation $A - B$ is represented by the conjunction of the Boolean formula representing $A$ and the Boolean formula representing $\neg B$. The existential quantification of a state $s \in S$ is represented by the Boolean formula $\exists x(f_s)$ where $x$ is the vector representing the state $s$, and $f_s$ is the Boolean formula representing $S$.

Given a Boolean function $f(x_1, \ldots, x_n)$ of $n$ variables and a fixed ordering of its Boolean variables $x_1, \ldots, x_n$, it is shown by Bryant that the reduced OBDD for $f(x_1, \ldots, x_n)$ is unique \cite{Bry86}. Therefore, in order to establish whether a formula $\phi$ holds in a model $T$ we can just compare the structure of the ROBDD for the formula $\phi$ with the structure of the ROBDD for the model $T$.

Symbolic model checking has been also used for the verification of $\text{CTLK}$-specifications in \cite{LP07b}.

### 2.5.3 Model checking epistemic properties

This section presents model checking procedures for the epistemic operators of $\text{CTLK}^{\text{EDC}}$. The material presented here summarises some results in \cite{Rai06}. The procedures for calculating the sets in which the formulas $\mathbf{K}_i \phi$, $\mathbf{E} \phi$, $\mathbf{D} \phi$, $\mathbf{C} \phi$ hold are based on the definitions of the epistemic indistinguishability relations $\sim_i$, $\sim^E_A$, $\sim^D_A$, $\sim^C_A$ respectively presented in section 2.4.3. The labelling algorithm $\text{MCAlgo}_{\text{CTLK}}$ for the temporal fragment of $\text{CTLK}$ is the same as $\text{MCAlgo}_{\text{CTL}}$.

Epistemic modalities are evaluated on reachable states only. This choice will be explained with the following example.

Let $\mathbf{K}_i \phi$ be a formula that is true at a global state $w$. It is possible that a non-reachable (via the transition relation $T$) global state $w' \in W$ might still be accessible from $w \in W$ by means of the relation $\sim_i$. If formula $\phi$ is false at $w'$ then the formula $\mathbf{K}_i \phi$ would be false at state $w$ because of the $\sim_i$-accessibility from $w'$. To avoid this, the set $G$ of reachable global states is calculated before evaluating epistemic modalities.

Given an interpreted system $IS$, the set $G$ is built by iterating the following function $\tau : W \rightarrow$
Chapter 2. Literature Review

W on a given set \( X \subseteq W \) \cite{Rai06}:

\[
\tau(X) = I \cup X \cup \{ w \in S \subseteq W \mid \exists w' ((w' T S) \land (w' \in X)) \}
\]

Where \( T \) is the global transition relation defined in Definition 2.13.

In other words, the set \( \tau(X) \) is calculated by the union of the initial states \( I \), by \( X \) itself, and by all states that are reachable in a step from \( X \) by means of \( T \). Since the set of global states \( W \) is finite, and the operator \( \tau \) is monotonic, by the Knaster-Tarski Theorem there exists an integer number \( n \) and a fix-point which can be calculated by starting from the empty set \( \emptyset \) and by iterating the function \( \tau(\emptyset) \) \( n \)-times.

The Algorithm 5 first computes the set \( X \) of states in which the negation of the formula \( \phi \) is true and then it builds the set \( Y \) of states which have a \( \sim_i \)-transition into the set \( X \). Finally, the set \( [K_i \phi] \) is calculated by intersecting the complement of the set \( Y \) with the set \( G \) of reachable states.

Procedures MCAlgo\(_{K_i} \), MCAlgo\(_{E_{Ag}} \) MCAlgo\(_{D_{Ag}} \) are described by Algorithms 5, 6, 7 respectively. For the operator \( C_{Ag} \), it is shown in \cite{FHVM95} that common knowledge \( C_{Ag} \) can be given as a fix-point, by means of the following equivalence:

\[
C_{Ag}\phi \equiv E_{Ag}(\phi \land C_{Ag}\phi)
\]

it is possible to calculate the fix-point \( [C_{Ag}\phi] \) by starting from the set \( G \) of reachable states and by iterating the function \( \tau(X) = [E_{Ag}(\phi \land (X))] \) a finite number of times.

\begin{algorithm}
\caption{The support procedure \text{MCAlgo}_{\text{CTLK},K}(IS, i, \phi), from \cite{Rai06}.}
\begin{algorithmic}[1]
\State \( X = \text{MCAlgo}_{\text{CTLK}}(IS, \neg \phi) \);
\State \( Y = \{ g \in G \mid \exists g' \in X g \sim_i g' \} \);
\State return \( \neg Y \cap G \);
\end{algorithmic}
\end{algorithm}

2.5.4 Model checkers, MCMAS and ISPL

Given a formula \( \phi \) belonging to a logic \( \mathcal{L} \) and given a model \( M \) specified in a formal language, a model checker is a software tool used to determine whether \( \phi \) is true on model \( M \). Sometimes, a counterexample is generated when the formula is falsified on the model.
### Algorithm 6
The support procedure $\text{MCAlgo}_{\text{CTLK},E}(IS, Ag, \varphi)$, from [Rai06].

1. $X = \text{MCAlgo}_{\text{CTLK}}(IS, \neg \varphi)$;
2. $Y = \{ g \in G \mid \exists g' \in Xg \sim^E Ag g' \}$;
3. return $\neg Y \cap G$;
4. }

### Algorithm 7
The support procedure $\text{MCAlgo}_{\text{CTLK},D}(IS, Ag, \varphi)$, from [Rai06].

1. $X = \text{MCAlgo}_{\text{CTLK}}(IS, \neg \varphi)$;
2. $Y = \{ g \in G \mid \exists g' \in Xg \sim^D Ag g' \}$;
3. return $\neg Y \cap G$;
4. }

Software model checkers that support the verification of $\text{CTLK}$-formulas are MCK [GvdM04], VerICS [KNN08], and MCMAS [LQR09].

### MCK

MCK (model checking knowledge) is a model checker presented by Gammie et al. in [GvdM04] to check $\text{LTL}$ and $\text{CTL}$ specifications and epistemic specifications as well. MCK makes use of a dedicated input language. A multi-agent system is described in its language by specifying an environment agent via a list of shared variables. Every agent is defined by an identifier and can observe only a restricted list of the environment variables. The interaction between temporal and epistemic modalities are implemented in three different ways:

1. An agent’s knowledge can depend on current values of the environment variables only.
2. An agent’s knowledge can depend on current environment variables and on a clock variable.
3. An agent’s knowledge can depend on the history of all of variable values and clock values.

In the first semantics, either $\text{CTL}$ and or $\text{LTL}$ specifications extended with the $K$ modality can be checked. In the second, MCK supports only temporal specifications expressed via the modality $X$ and $K$-specifications for a single agent. In the third semantics, the checker supports linear temporal logic specifications expressed via $X$ and the knowledge operator for a single agent.
Chapter 2. Literature Review

VerICS

VerICS \cite{NNP+04, KNN+08} is a SAT-based model checker for the verification of $\text{CTL}^{E,D,C}$-specifications. The tool accepts input files written in several languages: the ESTELLE language \cite{RT04, CS90}, networks of timed automata \cite{Alu99}, and timed Petri nets \cite{PP06}. This tool provides three different approaches to model checking: bounded model checking \cite{BCC+03}, unbounded model checking \cite{KP03} and on-the-fly model checking \cite{Hol96} that makes use of abstract models. The first approach is restricted to the universal fragment of $\text{CTL}^{E,D,C}$-specifications. The second one can be applied to $\text{CTL}^{E,D,C}$-specifications. The third approach supports $\text{CTL}$-specifications only.

MCMAS

MCMAS (Model Checking for Multi-Agent System) is a temporal epistemic model checker for multi-agent systems defined and implemented by F. Raimondi and H. Qu. \cite{LR06a, RL07, LQR09}. It has been developed in C++ and Java programming language. This tool is currently used for the verification of the $\text{CTL}^{E_D,C}$-formulas on interpreted system semantics. MCMAS has been released under the terms of the GNU General Public License (GPL). This model checker is based on the symbolic method presented in \cite{LR04} and makes use of an external BDD-library called CUDD (Colorado University Decision Diagram) \cite{Som}. The MCMAS interface allows the exhibition of counterexamples when a universal specification is checked and witnesses for existential ones. MCMAS makes use of OBDDs. Therefore multi-agent systems are represented in a very concise way. MCMAS allows the verification of $\text{CTL}^{E_D,C}$-specifications. MCMAS’ high-level architecture (see Figure 2.4) consists of a parser module, a validation module, a support module for the intermediate processing procedure and the verification module. The parser accepts an input file written in a suitable formal language accepted by MCMAS. The grammar rules for the input file are described in the PARSER/ directory. In this directory two files are present: nssis.ll and nssis.yy. The first provides the specification of the lexical analyser while the second provides the description of the parser. After the parsing procedure is computed, MCMAS builds OBDD-representations for the model and the formula to be checked. Each global state is described by an OBDD that contains the representation of each local state for each agent. The execution of the verification with MCMAS is composed of two main steps:
1. MCMAS computes the number of boolean variables needed to encode global states and joint actions.

2. The parser builds OBDDs for evolution functions and the set of initial states.

MCMAS provides many options. It is possible to print witnesses as well as counterexample executions. It provides information of the time (expressed in seconds) required by the whole checking process, the memory consumption (expressed in bytes) and the number of reachable states built before the checking process.

---

**Figure 2.4: The MCMAS internal structure, from [LR06a].**

---

**ISPL**

Multi-agent systems are modeled in MCMAS by interpreted systems described in ISPL (Interpreted Systems Programming Language) [Rai06] [LR04].

An ISPL program $P$ defines local states, actions, protocols, and local transition for agents and environment corresponding to a given interpreted system. Local states for the agents are
defined by means of a finite set \( \text{VAR} = \{v_1, v_2, \ldots, v_m\} \) of variables. Each variable \( v_k \in \text{VAR} \) has an associated finite domain \( D_k \). Let \( \{+,-,\div,\cdot\} \) be the set denoting standard arithmetic operations, and let \( \{<,>,=,\leq,\geq\} \) be the set of binary relation symbols. Arithmetic expressions are built from variables in \( \text{VAR} \), constants in \( D_k \) and arithmetic operations; for instance, \( v_3 - 7 \) is an arithmetic expression. A logic expression \( p \) is built from arithmetic expressions and relation symbols; for instance, \( v_3 - 7 > 9 \) is a logic expression. A Boolean expression \( \psi \) is composed from logic expressions \( p \) by using negation (\( \neg \)), conjunction (\( \land \)) and disjunction (\( \lor \)). Any global state \( d \) can be seen as an evaluation over \( \text{VAR} \), i.e., \( d = (d_1, d_2, \ldots, d_m) \in D = D_1 \times D_2 \times \cdots \times D_m \). Similarly, the local states of an agent can be seen as evaluations over a subset of \( \text{VAR}_i \subseteq \text{VAR} \), named local variables of agent \( i \). The expression \( d \models \phi \) means “the formula obtained by replacing each occurrence of the variable \( v_i \) in \( \phi \) by the constant \( d_i \) evaluates to true”.

Given an interpreted system defined on a set of agents \( Ag \), an ISPL program \( P \) describes each agent \( i \in Ag \) with one program section for each agent. Each of those sections is composed of four subsections containing, respectively:

1. a list of local variable definitions written in the following form: \( v : \{d_1, d_2, \ldots, d_n\} \).
   Where \( v \) is a variable and each \( d_i \) is a value. A variable can be of three types: an integer, a Boolean or an enumeration.

2. a set of actions available to the agent of the form: Actions = \( \{a_1, a_2, \ldots, a_n\} \), where each \( a_i \) is a string of chars denoting the name of an action for the agent.

3. local protocol specifications of the form: protocol-condition: \( \{a_1, \ldots, a_n\} \), where protocol-condition is a boolean expression of the following form: \( v_i = v_j, v_i = d \). \( v_i, v_j \) are local variables and \( d \) is a value of \( v_i \).

4. local evolution function specifications of the form: assignment if PreCondition, where assignment is a conjunction of local equalities, and PreCondition is a boolean formula composed of atoms of the form \( i.\text{Action} = a \), where \( a \) is an action belonging to the agent \( i \in Ag \).

In addition, an ISPL program has a list of specifications expressed in \( \text{CTLK}^{EDC} \) in the Evaluation section, and a list of initial state conditions in the Initial States section. These are built
from equalities of the following form: \( i.v_i = j.v_j, i.v_i = d \), where \( v_i \) and \( v_j \) are local variables of agents \( i \in Ag \) and \( j \in Ag \) respectively.

An ISPL program is composed of five sections:

1. Agents declarations. Agents are defined using a sequence of declarations, each of which has the following syntax:

   \[
   \text{Agent ID} \\
   \quad \langle \text{agent_body} \rangle \\
   \text{end Agent}
   \]

   where \( \langle \text{agent_body} \rangle \) contains the declaration of variables, actions, local protocol, and local evolution function. Each agent contains a set of local variables, defined with their domains. The agent Environment has a special syntax. In fact, the Environment can have “private” variables and observable variables, while a normal agent can have private variables and local observable variables.

   - Local observable variables of an agent \( i \in Ag \) are those variables belonging to the Environment that can be “seen” by the agent \( i \). Therefore, the agent \( i \) knows the values of those variables at every moment and those variables contribute to form the local state of agent \( i \). However, agent \( i \) cannot change the value of a local observable variable. These variables can only be changed by the local evolution function of the Environment.

   - Observable variables have the same characteristics as local observable ones, but they can be seen by all agents. These variables can only be defined for the agent Environment.

For the special agent Environment the following syntax is used:

\[
\langle \text{agent_body} \rangle ::
\quad \text{'Obsvars:'}
\quad \quad \text{ID ':'} \text{ Domain;}
\quad \quad \ldots
\quad \text{'end Obsvars'}
\]
Chapter 2. Literature Review

For all the other agents $i \in Ag$, this syntax is used:

```plaintext
<agent_body> ::

'Vars:'
   ID ':' Domain;
...
'end Vars'

'Lobsvars = {' ID ',', ID ... '};'

'RedStates:'
   ID if <PreCondition> ;
'end RedStates'

'Actions = {' ID ',', ID ... '};'

'Protocol:'
   <ProtocolCondition> ':' {' ID ',', ID ... '};'
...
'end Protocol'

'Evolution:'
   <Assignment> 'if' <PreCondition> ';
...
'end Evolution'
```
'Evolution:'

<Assignment> 'if' <PreCondition> ';'

...

'end Evolution'

where ID is a string-identifier that has a letter as first symbol followed by any finite combination of letters, numbers, or underscore sign. ID is the basic element of ISPL grammar.

ID :: [a-zA-Z][a-zA-Z0-9_]*

while Domain can be

Domain::

  'boolean;' |
  '{' value ',' value ',' ... '}'; |
  value '..' value ';

where value is a string of chars in the first case, and an integer in the last case.

<Assignment> ::

  <Assignment> and <Assignment> |
  ID = value |
  ID = ID

<ProtocolCondition> and <PreCondition> are defined in the following way, respectively:

<ProtocolCondition> ::

  ID = value |
  ID = ID
2. Evaluation function:

```
Evaluation
  <proposition_declaration>
end Evaluation
```

where `<proposition_declaration>` is defined as:

```
<proposition_declaration> ::
  ID 'if' <boolean_expression> ';'
  ...
```

and `<boolean_expression>` is defined as follows:

```
<boolean_expression> ::
  <boolean_expression> 'or' <boolean_expression>
  | <boolean_expression> 'and' <boolean_expression>
  | '!' <boolean_expression>
  | ID '.Lstate = ' value
```

3. Initial states:

```
<InitialStates> ::
  'InitStates'
    <boolean_expression>
  'end InitStates'
```
4. Declaration of groups:

```
<Groups> ::=
  'Groups'
  ID '=' '{' ID ',' ID... '}';'
  ...
  'end Groups'
```

Each line in the Groups section is composed of the name of the group of agents, and the corresponding set of agent names of which the group is composed.

5. List of formulas to check

```
Formulae
  formula ';'
  ...
end Formulae
```

where formula is defined by the following syntax:

```
formula ::=
  ID
  | formula 'and' formula
  | formula 'or' formula
  | '!' formula
  | formula '->' formula
  | 'AG' formula
  | 'EG' formula
  | 'AX' formula
  | 'EX' formula
  | 'AF' formula
  | 'EF' formula
  | 'A (' formula 'U' formula ')
  | 'E (' formula 'U' formula ')'
```
Agent E
Obsvars:
  a: 0..2;
  b: 0..2;
end Obsvars
Vars:
  c11: 1..4;
  c12: 1..4;
c21: 1..4;
c22: 1..4;
end Vars
Actions={watch,null};
Protocol:
a+b<2: {watch};
Other: {null};
end Protocol
Evolution omitted
end Agent

Agent Player1
Lobsvars={c11,c12};
Vars:
n: 1..3;
end Vars
Actions={playcard1, playcard2,null};
Protocol:
n=1: {playcard1};
n=2: {playcard2};
n=3: {null};
end Protocol
Evolution:
n=2 if n=1;
n=3 if n=2;
end Evolution
end Agent

Agent Player2
Lobsvars={c21,c22};
Vars:
n: 1..3;
end Vars
Actions={playcard1, playcard2,null};
Protocol:
n=1: {playcard1};
n=2: {playcard2};
n=3: {null};
end Protocol
Evolution:
n=2 if n=1;
n=3 if n=2;
end Evolution
end Agent

Evaluation
all_bigger_than_4_P1 if E.c11>3 and E.c12>3;
NOT_all_bigger_than_4_P2 if !(E.c21>3 and E.c22>3);
win1 if E.a>E.b and E.a+E.b=2;
win2 if E.b>E.a and E.a+E.b=2;
end Evaluation

Evolution omitted

Agent Player1
Lobsvars={c11,c12};
Vars:
n: 1..3;
end Vars
Actions={playcard1, playcard2,null};
Protocol:
n=1: {playcard1};
n=2: {playcard2};
n=3: {null};
end Protocol
Evolution:
n=2 if n=1;
n=3 if n=2;
end Evaluation
end Agent

Formulæ
AG((all_bigger_than_4_P1 and
NOT_all_bigger_than_4_P2))
->K(Player1,AF win1);
end Formulæ

Figure 2.5: An ISPL-program describing a card game.
where <AgentName> and <GroupName> are ID. The symbol “!” stands for ¬ and “→” stands for →.

A default group is built by including all declared agents if no declaration is specified in Groups-section. The syntax presented above is ISPL/MCMAS 1.0.1 released in October 2010. In Figure 2.5 an ISPL-program for a card game is shown. In this example three agents are defined: the environment E, Player1 and Player2. The agent E has two observable variables that can be seen (but not changed) by all the other agents. In addition, there are four variables representing the cards used in the game. The first two variables (c11, c12) can be seen by the agent Player1 (that means the Player1 knows their values at every state of the system) as they belong to the set Lobsvars defined in the agent Player1. The same holds for the remaining two cards c21, c22. The specifications checked on this system are listed in the section Formulae at the bottom of the ISPL-program.
2.6 Abstraction

2.6.1 Limits of symbolic model checking and state of the art of abstraction

The introduction of OBDDs to represent computer systems brought a fundamental breakthrough in the model checking field. In [Saw06, Vei98], the authors conducted several experiments showing that the use of OBDD’s to model large systems improved the efficiency of symbolic model checking drastically. However, the state explosion problem still remains a huge obstacle that hampers the verification of large systems. This is particularly the case in the verification of multi-agent systems since these are composed of many autonomous and independent agents. Therefore, additional methods are needed to alleviate the state explosion problem. One of the most effective methods for reducing the “size” of a computer model is abstraction. A comprehensive presentation of the most used abstraction techniques is given in [Kan07]. More broadly, three main approaches are prominent in abstraction techniques for temporal logic.

1. The first focuses on exploiting symmetries of the system under investigation [CDLQ09a, ET99, ID96, GPS96, CDLQ09b, CEFJ96].
2. The second is predicate abstraction [DDP99, DD02, DD01, BR00] introduced by Graf et al. in [GS97].
3. Finally, the third approach, called existential abstraction, was introduced by Cousot in 1977 [CC77, CC99] and further developed by Edmund Clarke at al. from 1994 onwards [CGL94, CGJ+00, CGJ+03].

The theoretical results presented in [CGL94, CGJ+00, CGJ+03] are the basis for the existential abstraction techniques presented in this thesis.

2.6.2 Symmetry reduction techniques for CTLK

This section summarises the methodology presented in [CDLQ09b, CDLQ09a]. In these works the authors present methods to reduce the state space of multi-agent systems specified in the
2.6. Abstraction

They extend the results presented in \cite{CEFJ96, ES96, ES97} from \textsc{ctl} to \textsc{ctlk} and define a technique that builds an abstract interpreted system with a reduced state space by exploiting the symmetries in the original interpreted system.

There are two types of symmetries: data symmetries \cite{CDLQ09a} and agent symmetries \cite{CDLQ09b}.

Let $I = \langle \{ L_i \}_{i \in Ag} \cup \{ L_e \}, \{ ACT_i \}_{i \in Ag} \cup \{ ACT_e \}, \{ P_i \}_{i \in Ag} \cup \{ P_e \}, \{ t_i \}_{i \in Ag} \cup \{ t_e \}, I_0, V \rangle$ be an interpreted system over a set $Ag$ of agents defined together with a set $VAR$ of variables and a domain $D_v$ for each variable $v \in VAR$.

Informally, a data symmetry is a domain bijection $\pi : D_v \rightarrow D_v$ that permits the “swap” or “interchange” of two values of a variable $v$ or two local states without altering the executions of the interpreted system.

**Definition 2.26 (Data symmetry).** \cite{CDLQ09a}

A set $X \subseteq W$ of global states is data symmetric iff for all domain bijections $\pi : D_v \rightarrow D_v$ we have $w \in X$ iff $\pi(w) \in X$. A relation $Q \subseteq W \times W$ is data symmetric iff for all domain bijections $\pi$ we have $(w, w') \in Q$ iff $(\pi(w), \pi(w')) \in Q$. The system $I$ is data symmetric iff the global transition relation $R$, the set $I_0$ of initial states, and the evaluation function $V$ are data symmetric.

Two global states $w, w' \in W$ are symmetric, written $g \equiv g'$, iff $\pi(g) = g'$ for some domain bijection $\pi$. Let $[w]$ be the equivalence class of global states $w$ with respect to $\equiv$. An abstract interpreted system $I'$ of a concrete one $I$ is an interpreted system with a set $I'_0 \subseteq I_0$ of initial global states such that $I_0 = \{ [w] | w \in I'_0 \}$.

**Theorem 2.1. (Reduction)**\cite{CDLQ09a}

Let $I$ be an interpreted system and let $I'$ be the corresponding abstract interpreted system. For any \textsc{ctlk}-specification $\phi$ we have:

$$I \models \phi \iff I' \models \phi.$$
the agent by replacing each local state \( l_i \) belonging to a global state \( g \) by the local state of an agent \( j = \rho(i) \in Ag \), as follows:

\[
\rho(g) = (l_{\rho^{-1}(1)}(g), \ldots, l_{\rho^{-1}(n)}(g))
\]

The bijection \( \rho \) is extended to atomic formulas \( \varphi \) in a direct way by replacing every agent name \( i \) in the formula \( \varphi \) with its equivalent name \( \rho(i) \) in the abstract system. The function \( \rho \) is defined by induction on the structure of a general formula belonging to CTLK in a natural way. These new formulas are evaluated on the abstract interpreted system via a counterpart semantics [Lew68]. In counterpart semantics, the bijection \( \rho \) “interchanges” agents across global states. Each agent \( i \) at the global state \( g \) in the concrete system has an “equivalent” agent \( \rho(i) \) at the state \( g' \) in the abstract one. The new semantics is defined in order to create the following connection between global states: if \( g' \models \varphi(\rho(i)) \) then \( g \models \Diamond \varphi(i) \), where \( g \) is a concrete state and \( g' \) is an abstract one.

**Definition 2.27** (equivalent semantics induced by \( \approx^\rho \)). [CDLQ09b]

Let us consider an interpreted system \( I \), a set of global states \( G \) and a set of functions \( \{ \rho_1, \ldots, \rho_n \} \). Two global states \( g, g' \in G \) are said to be epistemically equivalent under the bijection \( \rho : Ag \rightarrow Ag \) to agent \( i \in Ag \) written \( g \approx^\rho_i g' \), if and only if, \( g \sim^i_\rho g' \).

The expression \( g \approx^\rho_i g' \) means that for all the other agents \( i \in Ag \) there is no difference between an agent \( k \in Ag \) at state \( g \) and agent \( \rho(k) \) at state \( g' \).

Multi-agent systems often possess useful symmetries that can be easily used to reduce the system state space. In [CDLQ09b], the authors illustrate the state space symmetry reduction by means of the muddy children puzzle example.

**Muddy children puzzle [FHVM95]**

There are \( n \) children playing in a garden. Some of them, let us say \( q \) children, get mud on their forehead. Then they make a circle in order for them to see the foreheads of all other children. At this point, a man announces to all of them “At least one of you has mud on his forehead!”, and then he asks the question: “Does any one of you know whether you have mud on your forehead?”. The man keeps asking the same question over and over at each round, and each time the children answer simultaneously. All children answer “I do not know” for \( q - 1 \) rounds.
2.6. Abstraction

At the round $q$, all the muddy children simultaneously answer “Yes, I know” as now they know they have mud on their forehead.

For this example, a set of bijections $\Pi = \{\rho_1, \ldots, \rho_n\}$ over agents can be defined such that the resulting interpreted system $I'$ has just one child (agent) with a muddy forehead or two children with muddy forehead and so forth. The reduction is drastic and it exploits the fact that every child is perfectly equivalent to any other one. The starting system $I$ has $2^n$ initial states where $n$ is the number of children. The system $I'$ instead has $n + 1$ initial states only. Therefore, the technique reduced an exponential state space (with respect to the number $n$ of agents) to a polynomial state space. If a formula is valid in the abstract system then the formula is valid in the concrete one. On the other hand, if a formula does not hold in the abstract system then the formula is false in the concrete one as well. Agent symmetry reduction techniques can be applied only to systems in which there are several equivalent agents. In [CDLQ09b] the authors proved the following theorem.

**Theorem 2.2. (Reduction)** [CDLQ09b]

Let $I$ be an interpreted system and let $I'$ the abstract interpreted system created by the set of bijections $\Pi = \{\rho_1, \ldots, \rho_n\}$ over agents. For any $\text{CTLK}$-specification $\phi$ we have:

$$I \models \phi \iff I' \models \phi.$$ 

2.6.3 Predicate abstraction

This section summaries relevant parts of the methodology presented in [CJK04]. In this work the authors developed a predicate abstraction method to abstract hardware systems specified in a formal language, called Verilog [CTVW04]. An example of this language is given in Figure 2.6.

Let $\{r_1, \ldots, r_n\}$ denote the set of variables in a Verilog-program. A concrete state of a system $S$ described by a Verilog-program is identified by vector $\tau = (r_1, \ldots, r_n)$ of values of all variables belonging to the program. Let $C$ be a set of all concrete states of the system $S$. Let $f_i(r_1, \ldots, r_n)$ denote the next-state function. The authors define the transition relation $R$ of the system $S$ in terms of the next-state function as follows:

$$R(\tau, \tau') := \bigwedge_{i=1}^{n} (r'_i \text{ iff } f_i(\tau))$$
In predicate abstraction concrete states of a system are mapped by means of several predicates into Boolean values. Let \( \{ P_1, \ldots, P_n \} \) be the set of predicates defined over the concrete system \( S \). In the abstraction process, a concrete state \( \tau \in C \) is transformed into a vector \( \bar{\tau} \) of Boolean values by means of an abstraction function \( \alpha : C \rightarrow A \), where \( A \) is the set of the abstract states generated by the abstraction method. Predicate abstraction makes use of model generators. These tools are used to automatically generate pairs of states that represent the abstract transitions of the abstract system. One of the provers most often used in predicate abstraction is called *Simplify*, introduced in [DNS05]. In [CJK04] the authors employ the same technique introduced earlier in [CKSY04], where a theorem prover is used to perform abstraction. During the abstraction process, an abstract variable \( a_i \) is associated to each predicate \( P_i \). Each concrete state \( r = \{ r_1, \ldots, r_n \} \) is abstracted into a vector \( a = \{ a_1, \ldots, a_n \} \) in a such a way that \( a_i = P_i(r) \). A transition in the concrete system from a state \( r \) to state \( r' = \{ r'_1, \ldots, r'_n \} \) is represented in the abstract system by a transition from \( a \) to \( a' = \{ a'_1, \ldots, a'_n \} \), where \( a'_i = P_i(r') \).

The transition relation \( \hat{R} \) of the abstract system is built as follows:

\[
\hat{R} = \{ (\bar{a}, \bar{a}') \mid \exists \tau, \tau' \in C : \Gamma(\tau, \tau', \bar{a}, \bar{a}') \} \tag{2.5}
\]

\[
\Gamma(\tau, \tau', \bar{a}, \bar{a}') := \bigwedge_{i=1}^{n} a_i = P_i(\tau) \land R(\tau, \tau') \land \bigwedge_{i=1}^{n} a'_i = P_i(\tau') \tag{2.6}
\]

\( \hat{R} \) is generated by rewriting \( \Gamma(\tau, \tau', \bar{a}, \bar{a}') \) into a conjunctive normal form formula \( \phi \). Subsequently, the formula \( \phi \) so obtained is given as input to a theorem prover, and all values satisfying the formula \( \phi \) are automatically generated by the theorem prover to obtain all the abstract transitions that correspond to the concrete transitions \( (\tau, \tau') \). The theorem prover automatically builds \( \hat{R} \) satisfying the relations (2.5) and (2.6).

In [CJK04], the authors show the following example:

**Example:** The concrete transition relation of the Verilog program in Figure 2.6 is the following one:

\[
R(x, y, x', y') := (x' \iff (x < 100) ? (x + y) : x) \land (y' \iff x) \tag{2.7}
\]

The expression \( x ? y : z \) means that a procedure executes \( y \) if the condition \( x \) holds, otherwise \( z \) is executed. The set of predicates chosen for this system is: \( \{ x < 200, x < 100, x + y < 200 \} \).

The abstraction procedure associates \( a_1, a_2, a_3 \) to each predicate, respectively. The equation (2.7) is written into a formula expressed in conjunction normal form, as follows:

\[(a_1 \iff (x < 200)) \land (a_2 \iff (x < 100)) \land (a_3 \iff (x + y < 200)) \land R(x, y, x', y') \land \]
module main (clk);
input clk;
reg [7:0] x,y;

initial x = 1;
initial y = 0;

always @(posedge clk) begin
y <=x;
if (x<100) x<=y+x;
end
endmodule

Figure 2.6: A Verilog program, from [CJK04].

\[(a'_1 \Leftrightarrow (x' < 200)) \land (a'_2 \Leftrightarrow (x' < 100)) \land (a'_3 \Leftrightarrow (x' + y' < 200))\]

Subsequently, the formula is given to a theorem prover that automatically generates the abstract transition relation \(\hat{R}\) in the TRANS section in Figure 2.7. The output in Figure 2.7 is generated automatically and it represents the abstract system of the system described in Figure 2.6. Notice that the TRANS statement is a disjunction of blocks. For instance, the first block \((a1\&!a2\&!a3\&next(a1)\&!next(a2)\&!next(a3))\) in Figure 2.7 represents the transition from the abstract state in which \(a_1\) is true and \(a_2, a_3\) are false to the same abstract state. The formula checked on this example is \(\mathbf{AG}(x < 100)\). The \(x < 100\) is represented by the Boolean variable \(a2\)in the abstract model as showed in section SPEC at the bottom of Figure 2.7.

The use of model generators in predicate abstraction for the construction of the abstract transition relation \(\hat{R}\) constitutes a main problem in terms of efficiency. Calling theorem provers usually involves a substantial computational cost in performing abstraction. This cost might compromise the benefit of using predicate abstraction. The second problem is related to the discrepancies between the high level description of a system used by predicate abstraction, and low level design used by the great majority of model checkers. In fact, predicate abstraction is only effective if the predicates describing the system involve the relationships among a great number of variable constraints.
MODULE main

VAR a1: boolean; //stands for x<200
VAR a2: boolean; //stands for x<100
VAR a3: boolean; //stands for x+y<200

INIT (a1 & a2 & a3)

TRANS (a1 & !a2 & !a3 & next(a1) & !next(a2) & !next(a3)) |
    (a1 & a2 & !a3 & !next(a1) & !next(a2) & !next(a3)) |
    (a1 & a2 & a3 & next(a1) & next(a3)) |
    (a1 & !a2 & next(a1) & !next(a2) & !next(a3)) |
    (!a1 & !a2 & !next(a1) & !next(a2)) |
    (a1 & a3 & next(a1) & !next(a2) & !next(a3))

SPEC AG(a2)

Figure 2.7: The corresponding abstract program of the program in Figure 2.6 from [CJK04].
2.6.4 Epistemic abstraction on Kripke structures

This section summaries the results presented in [ED07]. In this work, the authors present an abstraction technique that is applied to Kripke structures. Moreover, they produce several preservation results that assure the technique presented is correct. They consider specifications expressed in a special logic, called: $\text{KCTL*P}$. The syntax of this logic is the following one:

$$
\phi ::= p \mid \neg \phi \mid \phi \lor \phi \mid \mathbf{G}\phi \mid [\phi\mathbf{U}\psi] \mid \mathbf{K}_{i}\phi \mid \phi\mathbf{W}\phi \mid \phi\mathbf{Pr}\phi \mid \phi\mathbf{A}\phi \mid \phi\mathbf{O}\phi \mid \phi\mathbf{S}\phi \mid \phi\mathbf{B}\phi \mid \forall \phi \mid \exists \phi \mid \mathbf{P}_{i}\phi
$$

This logic has the usual modal and temporal operators, but in addition: $\mathbf{W}$ that means “waiting for”, $\mathbf{Pr}$ that means “previous”, $\mathbf{A}$ that means “always in the past”, $\mathbf{S}$ that means “since”, $\mathbf{B}$ that means “back to” and the path quantifiers $\forall$, $\exists$. Finally, the formula $\mathbf{P}_{i}\phi$ means “agent $i$ thinks that $\phi$ is possible” (for more details see [ED07]). The semantics chosen for this logic is based on 3-valued multi-agent Kripke structures [Sch91] defined over a set of agents $\text{Ag}$ by the tuple $M = (Q, R, L, \sim_{i})$, where $Q$ is the set of states, $R : Q \times Q \to \{0, 1, \bot\}$ is the transition relation, $L : Q \times \text{AP} \to \{0, 1, \bot\}$ is the labelling function over a set of atomic propositions $\text{AP}$ and $\sim_{i} : Q \times Q \to \{0, 1, \bot\}$ is the epistemic indistinguishability relation for each agent $i \in \text{Ag}$. The extra value $\bot$ is added to describe those situations in which there is an incomplete observation (for more details see [KP06]). The symbol $\bot$ can be read as “unknown”.

The authors present one simple example only, called the running system (see Figure 2.8). This toy-system is made of two agents $H$ and $L$ that are asynchronously composed. Agent $H$ continuously increases a variable $x$ by 2 units per time, and only when $H$ receives a message from agent $L$ it starts increasing $x$ by 1 unit per time. The specification checked on this system expresses the following question: “Is the agent $L$ able to know the parity of the variable $x$?”

The abstract system is built via a surjective function $\rho : Q \to Q$ that creates several equivalence classes representing the abstract states. Those classes form the abstract states of the abstract system. The abstract transition relation $\hat{R}$ is defined in the following way:

$$
\hat{R}([q], [q']) = \begin{cases} 
1 & \text{if } (\forall q_1 \in [q])(\forall q_2 \in [q'])(R(q_1, q_2) = 1); \\
0 & \text{if } (\exists q_1 \in [q])(\exists q_2 \in [q'])(R(q_1, q_2) = 0); \\
\bot & \text{otherwise.}
\end{cases}
$$

for any $q, q' \in Q$, where $[q], [q']$ represent the abstract states, and $q_1, q_2$ represent the concrete
local integers: \( x, y, z \). \( x := 1; \ y := 0; \ z := 0; \)

\[
H ::
\begin{align*}
1. & \text{ while true } \\
2. & \text{ if } z = 0 \text{ then } \\
3. & \quad x := x + 2; \\
4. & \text{ if } z = 1 \text{ then } \\
5. & \quad x := x + 1 \land z := 0; \\
\end{align*}
\]

\[
L ::
\begin{align*}
1. & \text{ while true } \\
2. & \text{ if } z = 0 \text{ then } \\
3. & \quad y := y + 1 \land z := 1; \\
\end{align*}
\]

Figure 2.8: The running system, from [ED07].

Figure 2.9: The abstract running system, from [ED07].

states. The labelling function \( \hat{L} \) is defined as follows:

\[
\hat{L}(\lbrack q \rbrack, p) = \begin{cases} 
1 & \text{if } (\exists q \in \lbrack q \rbrack)(L(q, p) = 1); \\
0 & \text{if } (\forall q \in \lbrack q \rbrack)(L(q, p) = 0); \\
\bot & \text{otherwise.}
\end{cases}
\]

for any \( q \in Q \) and \( p \in AP \).

The system in Figure 2.8 is abstracted via a surjective function \( \rho \). This function collapses two states \( (x, y, z), (x', y', z') \in Q \) together if the following condition holds:

\[
\text{even}(|x - x'|) \land \text{even}(|y - y'|) \land z = z'
\]

The resulting abstract system is depicted in Figure 2.9.

Unfortunately, this method, based on 3-valued multi-agent Kripke structures, has a major problem: it is not computationally grounded (in the sense of [Woo00]). This may impede the application of the technique to any interesting scenario or real application. The models are
simply arbitrary Kripke structures, and even if the original Kripke structure is induced by a multi-agent system, the abstraction technique does not build another system that can be tested by a model checker.

### 2.6.5 Existential abstraction on Kripke structures

This section summaries main results presented in [CGL94, CGJ+00, CGJ+03]. There are several definitions of simulation in the literature. Here, two different definitions of simulation (both given by Clarke et al.) are reported.

**Definition 2.28** (Simulation: first definition). \([CGJ+03]\)

Given two Kripke structures \(K = \langle S, R, V \rangle\) and \(\hat{K} = \langle \hat{S}, \hat{R}, \hat{V} \rangle\) and given two sets \(AP, \hat{AP}\) of atomic propositions of \(K\) and \(\hat{K}\) respectively such that \(\hat{AP} \subseteq AP\), a relation \(\simeq \subseteq S \times \hat{S}\) between \(K\) and \(\hat{K}\) is a simulation relation if and only if for all \(s, \hat{s}\) such that \(s \simeq \hat{s}\), the following conditions hold:

- \(V(s) \cap \hat{AP} = \hat{V}(\hat{s})\).
- If \(sRg\), then \(\hat{s}\hat{R}\hat{g}\) for some \(\hat{g}\) such that \(g \simeq \hat{g}\).

A Kripke structure \(\hat{K}\) simulates \(K\) (denoted by \(\hat{K} \succeq K\)) if there exists a simulation relation \(\simeq\) such that for every state \(s \in S\) of \(K\) there exists a state \(\hat{s} \in \hat{S}\) of \(\hat{K}\) such that \(s \simeq \hat{s}\).

In [CGL94] Clarke et al. defined simulation via a surjective function. A surjection \(h\) collapses states of a Kripke structures into abstract states. Formally, \(h : S \rightarrow \hat{S}\), where \(\hat{S}\) is the set of abstract states. The surjection \(h\) induces an equivalence relation \(\equiv_h\) on the set \(S\) of a Kripke structure \(K\) in the following way: let \(s\) and \(g\) be states in \(S\), then

\[
s \equiv_h g \iff h(s) = h(g).
\]

Therefore, it is possible to describe a simulation either via equivalence relations or via surjections. In Figure 2.10 the system \(K\) represents the original Kripke structure, and \(\hat{K}\) the system that simulates \(K\). The dotted lines in \(K\) show how the global states of \(K\) are grouped together by the equivalence relation \(\equiv_h\).
Definition 2.29 (Simulation: second definition). [CGL94]

Given a Kripke structure \( K = \langle S, R, V \rangle \), with a set \( I \in S \) of initial states, and a Kripke structure \( \hat{K} = \langle \hat{S}, \hat{R}, \hat{V} \rangle \) with a set \( \hat{I} \in \hat{S} \) of initial states, and given a surjection \( h \), a system \( \hat{K} \) simulates \( K \) if:

1. If \( \forall \hat{d} \in \hat{S} \exists d \in S \) such that \( h(d) = \hat{d} \) and \( d \in I \), then \( \hat{d} \in \hat{I} \);
2. If \( \forall \hat{d}_1, \hat{d}_2 \in \hat{S} \exists d_1, d_2 \in S \) such that \( \hat{d}_1 = h(d_1), \hat{d}_2 = h(d_2) \) and \( d_1 R d_2 \), then \( \hat{d}_1 \hat{R} \hat{d}_2 \);
3. \( \hat{V}(\hat{d}) = \bigcup_{h(d)=\hat{d}} V(d) \).

Among all systems \( \hat{K} \) that simulate a system \( K \) there are systems that are the most accurate simulation of \( K \). These simulations are also called minimal simulations and they are defined as follows.

Definition 2.30 (minimal simulation). [CGL94, CGJ+03]

Given a Kripke structure \( K = \langle S, R, V \rangle \), a surjection \( h \) and a set \( I \in S \) of initial states, the minimal simulation of \( K \), is the system \( \hat{K}_{min} = \langle \hat{S}, \hat{R}, \hat{V} \rangle \) with \( \hat{I} \) the set of initial states, such that:
2.6. Abstraction

1. \( \forall \hat{d} \in \hat{S} \exists d \in S \) such that \( h(d) = \hat{d} \) and \( d \in I \) iff \( \hat{d} \in \hat{I} \).

2. \( \forall \hat{d}_1, \hat{d}_2 \in \hat{S} \exists d_1, d_2 \in S \) such that \( d_1 = h(d_1), \hat{d}_2 = h(d_2) \) and \( d_1 R d_2 \) iff \( \hat{d}_1 R \hat{d}_2 \).

3. \( \hat{V}(\hat{d}) = \bigcup_{h(d) = \hat{d}} V(d) \).

![Figure 2.11: A Kripke structure and a model that simulates it.](image)

Notice that every Kripke structure \( \hat{K} = \{ \hat{S}, \hat{R}, \hat{V} \} \) with \( \hat{I} \) set of initial states such that \( \hat{K} \succeq K \), is such that \( \hat{K} \succeq \hat{K}_{\text{min}} \succeq K \). Therefore, \( \hat{K}_{\text{min}} \) is the most accurate approximation among those consistent with \( h \) (i.e., each transition in \( \hat{K}_{\text{min}} \) corresponds to a transition in \( K \)). Moreover, the following conditions hold:

- \( \hat{I}_{\text{min}} \subseteq \hat{I} \)
- \( \hat{R}_{\text{min}} \subseteq \hat{R} \)

This means the minimal simulation has no extra behaviours. In Figure 2.11, a Kripke structure \( K \) (on the bottom of the figure) is abstracted by means of a surjective function \( h \) into a system \( \hat{K} \) (on the top of the figure). Figure 2.12 shows two possible simulations. The model on the right of the figure is a minimal simulation, i.e., all of its transitions are abstract counterparts of concrete transitions. The model on the left of the figure is not a minimal simulation: there
Chapter 2. Literature Review

Figure 2.12: A simulation (left) and a minimal simulation (right).

is a spurious loop on the state $s'_2$ whilst in Figure 2.11 there is no transition from states $s_4$, $s_5$ into either $s_4$ or $s_5$ in the concrete system. Informally, simulations that are “closer” to the minimal one guarantee more accurate analysis.

In [CGJ+03], the authors introduce a constraint on abstraction functions to guarantee that no false positives are possible on the abstract model. A false positive is a formula that holds in the abstract model but does not hold in the concrete one.

**Definition 2.31 (Appropriate).** [CGJ+03]

Given a Kripke model $K = \langle S, R, V \rangle$ and given a set of atomic formulas $A$, an abstraction function $h$ is appropriate for a specification $\phi$ if for all atomic sub-formulas $p \in A$ of $\phi$ and for all states $w_1$ and $w_2$ in the domain $S$ it holds that:

$$\text{if } w_1 \equiv_h w_2 \text{ then } w_1 \models p \iff w_2 \models p$$

A function $h$ is appropriate for a set $F$ of formulas if $h$ is appropriate for all formulas $\phi \in F$.

An abstract interpretation $\hat{V}(\hat{w})$ is consistent, for a given abstract state $\hat{w}$, if all concrete states $w$ corresponding to $\hat{w}$ satisfy all atomic formulas in $\hat{V}(\hat{w})$. The following proposition holds.

**Proposition 2.6.1.** [CGJ+03]

If $h$ is appropriate for $\phi$, then:

1. All concrete states in an equivalence class of $\equiv_h$ satisfy the same atomic formulas.

2. The abstract states satisfy all atomic formulas from each state in the respective equivalence classes.

3. The atomic formulas of the abstract states are consistent.
This proposition simply says that $d \equiv_h d'$ implies $V(d) = V(d')$, $h(d) = \hat{d}$ implies $\hat{V}(\hat{d}) = V(d)$, and $\hat{V}(d)$ is consistent.

The application of model checking techniques to abstract models could lead to incorrect results as abstract systems usually contain less information than the original one. Fortunately, it was proved in [CGL94] that validity of the universal fragment of $\CTL^*$ (ACTL*) transfers from the abstract system to the concrete one.

Let us suppose we have $n$ variables $v_1, \ldots, v_n$ and each variable ranges over a non-empty set $D_i$ ($i \in \{1, \ldots, n\}$), and a given Kripke structure $K = \langle W, R, V \rangle$ where the set of states $W = D_1 \times \cdots \times D_n$ is the Cartesian product of all variable domains; let $C$ be a function that transforms a formula built from atomic propositions belonging to $\hat{AP}$ into a semantically equivalent formula made of atomic propositions belonging to $AP$, and let us suppose to have an atomic formula $\hat{v}_i = \hat{d}_i$ belonging to $\hat{AP}$. If this formula is valid then it means that at the concrete level we have a formula $v_i = d_i$ belonging to $AP$, for some $d_i$ satisfying $h_i(d_i) = \hat{d}_i$. Therefore, $C$ maps the formula $\hat{v}_i = \hat{d}_i$ into the formula:

$$\bigvee \{v_i = d_i \mid h_i(d_i) = \hat{d}_i\},$$

This expression represents the disjunction of all atomic formulas $v_i = d_i$ for which $d_i$ is transformed into $\hat{d}_i$ by the surjective function $h$. For more complex formulas, the mapping is defined recursively on the structure of a formula belonging to the universal fragment of $\CTL^*$.

\textbf{Definition 2.32 (The map $C$). [CGL93]}

The function $C$ is the mapping from formulas describing an abstract interpreted system $\hat{IS}$ to formulas describing the corresponding concrete interpreted system $IS$ that is defined as follows:

- $C(\hat{v} = \hat{d}_i)$ is $\bigvee \{v_i = d_i \mid h_i(d_i) = \hat{d}_i\}$.
- $C(\hat{v}_i \neq \hat{d}_i)$ is $\neg C(\hat{v}_i = \hat{d}_i)$.
- $C(\phi \land \psi) = C(\phi) \land C(\psi)$.

$\CTL^*$ is a temporal logic that extends both LTL and CTL. This logic is not discussed in this thesis. For more information about $\CTL^*$ see [HR04].

$\text{ACTL}^*$ was given in [CGL94] for the logic ACTL*. However, in this thesis will be given for ACTL only.
- $\mathcal{C}(\phi \lor \psi) = \mathcal{C}(\phi) \lor \mathcal{C}(\psi)$.

- $\mathcal{C}(AX\phi) = AX\mathcal{C}(\phi)$.

- $\mathcal{C}(AG\phi) = AG\mathcal{C}(\phi)$.

- $\mathcal{C}(\phi AU\psi) = \mathcal{C}(\phi) AU\mathcal{C}(\psi)$.

Lemma 2.6.1. \cite{CGJ+03}

Assume $\hat{K} \succeq K$ by means of a surjection function $h$. If $\pi$ is a path in $K$, then $h(\pi)$ is a path in $\hat{K}$.

The following theorem, called the Preservation Theorem, was proved in \cite{CGJ+03} for the logic $\textsc{ACTL}^*$. However, in this thesis the proof is presented for $\textsc{ACTL}$ only. We report here the entire proof of Preservation Theorem\cite{2.3} since it will be extended, in the next Chapter (Theorem \ref{3.1}), to $\textsc{ACTLk}$-specifications evaluated on the interpreted system frameworks. Hence, this proof is essential to follow the proof of Theorem \ref{3.1}.

**Theorem 2.3** (Preservation Theorem for $\textsc{ACTL}$). \cite{CGJ+03}

Let $h$ be appropriate for every $\textsc{ACTL}$-formula $\phi$. Then $\hat{K} \succeq K$ and $\hat{K} \models \phi$ implies $K \models \mathcal{C}(\phi)$.

**Proof.**

The proof proceeds by induction on the structure of the formula $\phi$.

Let us suppose that $w = (l_1, \ldots, l_n)$, $h(w) = (\hat{l}_1, \ldots, \hat{l}_n)$ and let $D_i$ be the domain of values of the variable $v_i$.

1. If $\phi = (\hat{v}_i = \hat{d}_i)$ with $d_i \in D_i$, then $h(w) \models \phi$ iff $\hat{l}_i = \hat{d}_i$. Obviously, $w \models (v_i = l_i)$. Since we have $h_i(l_i) = \hat{d}_i$, we can infer that $w$ satisfies the following disjunction:

   $$\bigvee \{v_i = d_i \mid h_i(d_i) = \hat{d}_i\}$$

   But this is just $\mathcal{C}(v_i = d_i)$, and therefore $w \models \mathcal{C}(v_i = d_i)$.

2. The case for $\hat{v}_i \neq \hat{d}_i$ is similar.
3. $h(w) \models \phi \land \psi$ implies $h(w) \models \phi$ and $h(w) \models \psi$. The induction hypothesis implies $w \models C(\phi)$ and $w \models C(\psi)$, so $w \models C(\phi \land \psi)$.

4. The case for $\phi \lor \psi$ is similar.

5. Given a path $\pi$, $h(\pi) \models \text{AX}\phi$ implies $(h(\pi))^1 \models \phi$. Now $(h(\pi))^1 = h(\pi^1)$, and so the induction hypothesis implies $\pi^1 \models C(\phi)$. Therefore, $\pi \models \text{AX}C(\phi)$, and so $\pi \models C(\text{AX}\phi)$.

6. If $h(\pi) \models \text{A}(\phi U \psi)$, then there exists $n \in \mathbb{N}$ such that $(h(\pi))^n \models \psi$ and, for all $i < n$, $(h(\pi))^i \models \phi$. This implies $h(\pi^n) \models \psi$ and $h(\pi^i) \models \phi$ for all $i < n$. Using the inductive hypothesis, we find that $\pi \models C(\text{A}(\phi U \psi))$.

7. If $h(\pi) \models (\text{AG})\psi$ then $(h(\pi))^i \models \psi$ for all $i$. This implies $h(\pi^i) \models \psi$ for all $i$. Using the inductive hypothesis, we find that $\pi \models C(\text{AG}\psi)$.

In this Chapter we summarised the background material that is fundamental to describe and to discuss the research reported in Chapters 3, 4 and 5. We presented the better known techniques in the abstraction and space reduction literature to permit the verification of large state space systems. The summary presented is obviously non exhaustive and only focuses on some important works that are related to this thesis. Hence, the research stressed in this thesis starts from all these results just presented and tries to fill all the gaps left from the works discussed in this Chapter. Those gaps can be summarised in the Table 2.1 below. In the last Chapter these relations will be focused and analysed in more detail by making use of the theoretical results and methodologies presented in Chapters 3, 4 and 5. The next chapter introduces the theoretical results that represent the basis for the methodologies presented in Chapter 4 and 5.
### Table 2.1: Some state space reduction techniques and their main limitations.

<table>
<thead>
<tr>
<th>Other state space reduction techniques</th>
<th>Main limitations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbolic representation (OBDDs) [Bry86]</td>
<td>Not sufficient to solve state explosion problem</td>
</tr>
<tr>
<td>Predicate abstraction [CJK07, CJK04, CTVW04]</td>
<td>Low level designs and usage of model generators</td>
</tr>
<tr>
<td>[CKSY04, DDP99, DD02, Das03, GS97]</td>
<td></td>
</tr>
<tr>
<td>Existential abstraction [CGL94, CGJ⁺00, CGJ⁺03]</td>
<td>No epistemic specifications</td>
</tr>
<tr>
<td>Symmetry reduction [CDLQ09a, CDLQ09b]</td>
<td>Only for interpreted systems with symmetries</td>
</tr>
<tr>
<td>[ES96, ES97, CEFJ96, ID96]</td>
<td></td>
</tr>
<tr>
<td>Epistemic abstraction on Kripke structures [ED07]</td>
<td>Not computationally grounded</td>
</tr>
</tbody>
</table>
Chapter 3

Theoretical results in abstraction of MAS

3.1 Existential abstraction of interpreted systems

As discussed in the previous chapter, several abstraction techniques have been introduced and developed in the last thirty years in order to face the state explosion problem. However, very little effort has been made to introduce abstraction techniques for multi-agent systems.

It is possible to define an abstract multi-agent system by partitioning the system states into equivalence classes. States of the abstract systems are called abstract states, while states of the concrete system are called concrete states. States in the abstract system are equivalence classes of concrete states. An abstract state is an equivalence class of concrete states that satisfy the same set of atomic propositions. Moreover, each transition in the concrete system is represented by a transition in the abstract system. Therefore an abstract system simulates the behaviours of the concrete one.

In order to extend the existential abstraction technique presented in [CGJ03] to interpreted systems the abstraction process must “collapse” local concrete states of each agent 

\[ i \] separately. We do this by partitioning each set of local states \[ L_i \] of agent \[ i \] into several equivalence classes. The resulting equivalence classes will form the set of abstract local states of agent \[ i \] in the abstract interpreted system. In the same way, the set \[ ACT_i \] is partitioned into several equiva-
lence classes. Each of these classes is called an abstract action and they represent the actions available to the agents in the abstract system. The abstraction of local protocols and and local evolution functions is obtained directly by substituting all local states \( l \) and local actions with the corresponding abstract local states \([l]\), and abstract local actions \([a]\) respectively. Therefore, if an action \( a \) is available to an agent \( i \) at a state \( l \) in the concrete model, then an abstract action \([a]\) is available to the same agent \( i \) at the abstract local state \([l]\) in the abstract model. Similarly, given a local state \( l \) and a joint action \( \langle a_1, \ldots, a_n \rangle \) if the local evolution function returns a set of local states \( L' \), then for the corresponding abstract joint action \( \langle [a_1], \ldots, [a_n] \rangle \) and abstract local state \([l]\) the abstract local evolution function returns \([L']\). The set of atomic propositions \( A \) is modified in a such a way that atomic propositions are defined on the corresponding abstract states. Finally, the interpretation function \( V' \) is obtained from \( V \) by deleting all the atomic formulas that distinguish between local states belonging to the same equivalence class (the abstract state). Therefore, the abstract interpreted system is a quotient system of the original one and it is defined in terms of equivalence classes of local states and local actions.

Now, consider an interpreted system \( \mathcal{I} \) defined over the set of agents \( \mathcal{A} \) and over a set of propositions \( A \). For each \( i \in \mathcal{A} \), define two equivalence relations. The first relation is defined for the local states of \( i \): \( \equiv_i \subseteq L_i \times L_i \). The second one is defined for local actions: \( \equiv_i \subseteq \text{ACT}_i \times \text{ACT}_i \). For \( l \in L_i \), the equivalence class of \( l \) with respect to \( \equiv_i \) is represented by \([l]\). Similarly, \([a]\) represents the equivalence class of \( a \in \text{ACT}_i \) with respect to \( \equiv_i \). Let \([g]\) represent the global state \( \langle [g_1], \ldots, [g_n] \rangle \) and \([a]\) represent the joint action \( \langle [a_1], \ldots, [a_n] \rangle \). Let us introduce the following definitions.

**Definition 3.1** (abstract atomic propositions \( A' \)).

Let \( A \) be a set of atomic propositions, the set \( A' \subseteq A \) consist of all propositions of \( A \) that do not distinguish between equivalent local states. Formally:

\[
A' = \{ \alpha \in A | \forall g, g' \in S \forall i \in \mathcal{A} ( ( \alpha \in V(g) \land g_i \equiv_i g'_i ) \implies \alpha \in V(g') ) \}.
\]

**Definition 3.2** (Quotient of interpreted system).

Consider an interpreted system \( \mathcal{I} \) defined over the set of agents \( \mathcal{A} \) and over a set of propositions \( A \). Let \( A' \) be the set of atomic propositions defined in Definition 3.1. The quotient system of \( \mathcal{I} \) is the interpreted system \( \mathcal{I}' \) over the set \( \mathcal{A} \) of agents and the set \( A' \) of proposition such that:
3.1. Existential abstraction of interpreted systems

1. \( L'_i = \{ [l] \mid l \in L_i \} \);

2. \( ACT'_i = \{ [a] \mid a \in ACT_i \} \);

3. \( P'_i = \{ \langle [l], [a] \rangle \mid \langle l, a \rangle \in P_i \} \);

4. \( t'_i = \{ \langle [l], [\overline{a}], [l'] \rangle \mid \langle l, \overline{a}, l' \rangle \in t_i \} \);

5. \( I'_0 = \{ [g] \mid g \in I_0 \} \);

6. \( V'([g]) = V(g) \cap A' \).

3.1.1 Simulation for interpreted systems

As discussed in the previous Chapter, an abstract interpreted system \( \mathcal{I}' \) simulates another system \( \mathcal{I} \) if every behaviour of the \( \mathcal{I} \) is reproduced by \( \mathcal{I}' \). In other words the set of behaviours of \( \mathcal{I} \) is a subset of behaviours of \( \mathcal{I}' \). Since \( \text{ACTL} \) operators quantify over all paths or executions, it follows that if any \( \text{ACTL} \)-formula is verified by the abstract model then the formula is verified by the concrete one. This preservation can be extended to \( \text{ACTLK} \) if any epistemic indistinguishability relation in the original system is represented by an epistemic possibility relation in the abstract system.

**Definition 3.3** (Simulation).

*Given an interpreted system \( \mathcal{I} \) over the set \( A_g \) of agents, a set \( A \) of atomic formulas, an interpreted system \( \mathcal{I}' \) over the same set \( A_g \) of agents, and a subset \( A' \subseteq A \) of propositions, a simulation relation between \( \mathcal{I} \) and \( \mathcal{I}' \) is defined as a relation \( \simeq \subseteq S \times S' \) such that:

1. If \( g \in I_0 \) then \( g' \in I'_0 \) for some \( g' \) such that \( g \simeq g' \)

and whenever \( g \simeq g' \) then:

2. \( V'(g') = V(g) \cap A' \)

3. If \( gR s \) then \( g'R's' \) for some \( s' \) such that \( s \simeq s' \)

4. If \( g \sim_i s \) then \( g' \sim_i s' \) for some \( s' \) such that \( s \simeq s' \)

where \( R \) and \( R' \) are the global transition relations of \( \mathcal{I} \) and \( \mathcal{I}' \) respectively.*
An interpreted system $\mathcal{I}'$ simulates $\mathcal{I}$, we write $\mathcal{I} \simeq \mathcal{I}'$, if there is a simulation relation between $\mathcal{I}$ and $\mathcal{I}'$.

In the definition above, the statement (1) says that each initial state in $\mathcal{I}$ is represented by an initial state in $\mathcal{I}'$. Statement (2) says that states must agree on propositions in $A'$. Statement (3) says that every transition in $\mathcal{I}$ is represented by a transition in $\mathcal{I}'$.

The theorem reported here is an extension to the epistemic modality of Theorem 2.3.

**Theorem 3.1** (Validity Preservation Theorem).

Assume an interpreted system $\mathcal{I}'$ that simulates another interpreted system $\mathcal{I}$. For any $\mathbf{ACTLK}$ formula $\phi$ over $A'$ defined in Definition 3.1, we have:

$$g \simeq g', (\mathcal{I}', g') \models \phi \implies (\mathcal{I}, g) \models \phi$$

(3.1)

**Proof.**

The proof reported here extends the one presented in the Theorem 2.3, but this time it proceeds by induction on the structure of a general $\mathbf{ACTLK}$-specification instead of an $\mathbf{ACTL}$-specification only.

For the modality $K$, we need to prove that: $g \simeq g'$, $g' \models_{\mathcal{I}'} K_i \phi \implies g \models_{\mathcal{I}} K_i \phi$. Let us assume $g' \models_{\mathcal{I}'} K_i \phi$, we need to prove $g \models_{\mathcal{I}} K_i \phi$. This means $\forall s \in G$ such that $s \sim_i g \implies s \models_{\mathcal{I}} \phi$.

Suppose $s \sim_i g$, we need to prove $s \models_{\mathcal{I}} \phi$.

For all $s \in G$ such that $s \sim_i g$ we have, by the simulation requirement 4, that $s' \sim_i g'$. Now, we supposed $g' \models_{\mathcal{I}'} K_i \phi$ and that means $\forall s' \in G'$ such that $s' \sim_i g' \implies s' \models_{\mathcal{I}'} \phi$. So, we have $s' \models_{\mathcal{I}'} \phi$. By the inductive hypothesis $s' \models_{\mathcal{I}'} \phi \implies s \models_{\mathcal{I}} \phi$. Therefore, we obtain $s \models_{\mathcal{I}} \phi$ as required.

**Corollary 3.1.1** (Validity Preservation Corollary).

Given an interpreted system $\mathcal{I}$ and an interpreted system $\mathcal{I}'$ that simulates $\mathcal{I}$. For any $\phi$ in $\mathbf{ACTLK}$ over $A'$ defined in Definition 3.1, we have:

$$\mathcal{I}' \models \phi \implies \mathcal{I} \models \phi.$$
3.2 Preservation theorem for quotient interpreted systems

In this section we explore how a quotient system is related to the simulation relation just introduced. It is possible to show that every behaviour and every epistemic possibility of the original system is represented by a behaviour and an epistemic possibility in the quotient system.

Lemma 3.2.1.

If $\mathcal{I}'$ is a quotient of $\mathcal{I}$, then $\mathcal{I}'$ simulates $\mathcal{I}$.

Proof.

The intent is to show that the relation $\simeq = \{ \langle g, [g] \rangle \mid g \in S \}$ is a simulation between $\mathcal{I}$ and $\mathcal{I}'$. The proof proceeds by showing all the simulation requirements one by one.

- Simulation requirement 1: it follows from requirement 5 in Definition 3.2.
- Simulation requirement 2: it follows from requirement 6 in Definition 3.2.
- Simulation requirement 3: Let us suppose $gRs$. Now, by Definition 2.13, there is a $\overline{a} \in ACT$ such that (for all $i \in Ag$): $\langle g_i, \overline{a}, s_i \rangle \in t_i$ and $\langle g_i, a_i \rangle \in P_i$. By requirements 3 and 4 in Definition 3.2, $\langle [g_i], [\overline{a}], [s_i] \rangle \in t'_i$ and $\langle [g_i], [a_i] \rangle \in P'_i$. Therefore $\langle [g_i], [\overline{a}], [s_i] \rangle \in t'_i$ and $\langle [g_i], [\overline{a}] \rangle \in P'_i$. Thus by Definition 2.13 $[g]R'[s]$. Therefore,

$$gRs \implies [g]R'[s]$$

(3.2)

from which simulation requirement 3 follows.

- Simulation requirement 4: Assume $g \sim_i s$. By Definition 2.17 $g_i = s_i$, i.e., $[g_i] = [s_i]$, i.e.,

$$[g]_i = [s]_i$$

(3.3)

Also by definition 2.17, $g, s \in G$. By 3.2 and simulation requirement 1, $[g], [s] \in G'$. Therefore, the simulation requirement 4 simply follows by (3.3).

$\square$
The following theorem states that whenever a formula belonging to ACTLK is valid in a quotient system $\mathcal{I}'$ of a system $\mathcal{I}$ that formula is also valid in the original system $\mathcal{I}$.

**Theorem 3.2** (Preservation for quotient interpreted systems).

Let $\mathcal{I}'$ be a quotient of interpreted system $\mathcal{I}$. For any ACTLK-specification $\phi$ over $A'$ defined in Definition 3.1, if $\models_{\mathcal{I}'} \phi$, then $\models_{\mathcal{I}} \phi$.

**Proof.**

The proof simply follows by combining Corollary 3.1.1 with Lemma 3.2.1.

Therefore, instead of checking the system $\mathcal{I}$ that usually is very large, it may be convenient to check the quotient system $\mathcal{I}'$. If the model checker reports that the system $\mathcal{I}'$ is a model for the specification $\phi$ then Theorem 3.2 guarantees that the original system $\mathcal{I}$ is a model for the specification $\phi$ as well.

### 3.3 Existential abstraction examples

This section illustrates two epistemic scenarios in which existential abstraction is applied manually. These examples are the *Red-Black Card Game* and the *Transmission Protocol*.

#### 3.3.1 The Red-Black Card Game example

In this section we build a toy system to show how existential abstraction presented before is applied in detail. We call this system *Red-Black Card Game*.

In this game there are two players: $A$ and $B$. The players receive 9 cards each from a deck of 20 cards. The remaining two cards are put aside. The deck is made up of 10 red cards, $r_1, \ldots, r_{10}$, and 10 black cards, $b_1, \ldots, b_{10}$. The game consists of 9 rounds. At each round, both players drop one single card on a table simultaneously. If the players play cards with different colours then the player that played the red card wins the current round. If the cards played have the same colour, then the player that dropped the card with the higher number wins the current round.
3.3. Existential abstraction examples

round. The game continues until the players have played all cards they own. The player who has won the highest number of rounds wins the game.

We model this game by considering an interpreted system with three agents: players $A$ and $B$, and a score keeper $S$. $S$ represents the environment agent.

Let $C$ be the set of cards. For each agent $i \in Ag = \{A, B, S\}$ we associate the corresponding sets $ACT_i$, $L_i$, and functions $P_i$, $t_i$ that represent local actions, local states, local protocol and local evolution function respectively.

A player can either play a card or do nothing ($\epsilon$); the set $ACT_i$ of actions for player $i \in \{A, B\}$ is:

$$ACT_i = \{\text{play } c \mid c \in C\} \cup \{\epsilon\}$$

The score keeper can either evaluate who wins the round or do nothing:

$$ACT_S = \{\text{eval}, \epsilon\}$$

A local state of an agent contains the “data” an agent possesses about the current status of the game. A player $i \in \{A, B\}$ “sees” the cards he holds and “remembers” the moves played so far:

$$L_i = \{\langle h, m \rangle \in 2^C \times ACT^* \mid |h| + |m| = 9\}$$

where $h \subseteq C$ represents the current hand of the agent and $m \in ACT^*$ represents the remembered sequence of joint actions. The score keeper just keeps a record of the score:

$$L_S = \{\langle a, b \rangle \in \{0..9\} \times \{0..9\} \mid a + b \leq 9\}$$

where the record $\langle a, b \rangle$ says that player $A$ has won $a$ number of rounds and player $B$ has won $b$ number of rounds.

Given a local state, the local protocol $P_i$ returns the local actions available to agent $i$ in that state. A player $i \in \{A, B\}$ plays a card non-deterministically from its hand until all cards are played.

$$P_i(\langle h, m \rangle) = \{\text{play } c \mid c \in h\}, \text{ if } h \neq \emptyset$$

$$P_i(\langle h, m \rangle) = \{\epsilon\}, \text{ if } h = \emptyset$$
The score keeper evaluates the score of each round until all cards have been played:

\[ PS((a, b)) = \{eval\}, \text{ if } a + b < 9 \]
\[ PS((a, b)) = \{\epsilon\}, \text{ if } a + b = 9 \]

The local evolution function \( t_A \) for player \( A \), specifying how the local state of player \( A \) is updated by a joint action, contains the following transitions. In the expressions below \( l \xrightarrow{\pi} l' \) is another way to express the triple \( (l, \overline{a}, l') \), and \( \cdot \) represents the usual “append” operation.

\[
\begin{align*}
\langle h, m \rangle &\xrightarrow{\langle playc, playc', eval \rangle} \langle h \setminus \{c\}, m \cdot \langle playc, playc', eval \rangle \rangle \\
\langle h, m \rangle &\xrightarrow{\langle \epsilon, \epsilon, \epsilon \rangle} \langle h, m \rangle
\end{align*}
\]

In the first transition above, the card \( c \) played by \( A \) is removed from the hand \( h \) and the move \( \langle playc, playc', eval \rangle \) is appended to the memory \( m \). In the second transition, the local state remains the same when the agents perform the action \( \epsilon \). The local evolution function \( t_B \) for player \( B \) is defined in the same way as \( t_A \) above, but with the hand \( h \setminus \{c'\} \) in the successor state in the first transition.

The score keeper updates the variables \( a \) and \( b \) that keep the score obtained by players. At each round, either the value of \( a \) or \( b \) is incremented by one unit if Player A or Player B has won that round respectively.

\[
\begin{align*}
\langle a, b \rangle &\xrightarrow{\langle playc, playc', eval \rangle} \langle a + 1, b \rangle, \text{ if } c > c' \quad (3.4) \\
\langle a, b \rangle &\xrightarrow{\langle playc, playc', eval \rangle} \langle a, b + 1 \rangle, \text{ if } c' > c \quad (3.5) \\
\langle a, b \rangle &\xrightarrow{\langle \epsilon, \epsilon, \epsilon \rangle} \langle a, b \rangle \quad (3.6)
\end{align*}
\]

The relation \( > \) is the total ordering of cards. Therefore \( r_i > b_j \), and if \( i > j \) then \( r_i > r_j \) and \( b_i > b_j \). In the first transition above, \( A \) plays the stronger card and so the score keeper increases by one unit to the score of \( A \). The same happens in the second transition.

When the game starts each player holds 9 unique cards, has no moves recorded and has 0 points. Therefore the set of initial global states \( I_0 \) is defined as follows:

\[
I_0 = \{\langle (h, m), (h', m'), (a, b) \rangle \mid |h| = |h'| = 9, h \cap h' = \emptyset, |m| = |m'| = 0, \ a = b = 0\} \]

The set of atomic propositions \( A \) contains formulas: \textit{onlyred}_i (“Player \( i \) holds only red cards.”) and \textit{win}_i (“Player \( i \) has won the game.”), for \( i \in \{A, B\} \). The evaluation function \( V \) is defined
as follows:

\[ \text{onlyred}_A \in V(\langle\langle h, m \rangle, l, l' \rangle) \iff h \cap \{b_1, \ldots, b_{10}\} = \emptyset \]

\[ \text{win}_A \in V(\langle l, l', \langle a, b \rangle \rangle) \iff a > b \text{ and } a + b = 9 \]

The conditions for the propositions \text{onlyred}_B and \text{win}_B are analogous.

The Figure 3.1 shows a part of the global transition relation \( R \) from one particular initial global state. There are approximately \( 9 \cdot 10^6 \) possible initial states. The transitions in the figure represent joint actions. The label \( c, c' \) represents the joint action \( \langle \text{play} c, \text{play} c', \text{eval} \rangle \).

In Figure 3.1 dots represent omitted parts of the global transition relation \( R \). All states represented by cycles in the figure have transitions to other states even if those transitions are not drawn.

It is easy to see from transitions (3.4), (3.5), (3.6) that if a player gets all red cards it is obvious that he will certainly win the game. Therefore, we expect the following specification to be
satisfied.

\[
\text{onlyred}_B \rightarrow K_B(\text{AF} \text{win}_B \land K_A \text{AF} \text{win}_B), \quad (3.7)
\]

\[
(\text{onlyblack}_A \land \text{onlyred}_B) \rightarrow C_{Ag} \text{AF} \text{win}_B, \quad (3.8)
\]

\[
(\text{onlyblack}_A \land \text{onlyred}_B) \rightarrow D_{Ag} \text{AF} \text{win}_B, \quad (3.9)
\]

\[
(\text{onlyblack}_A \land \text{onlyred}_B) \rightarrow E_{Ag} \text{AF} \text{win}_B. \quad (3.10)
\]

Formula (3.7) states that if \(B\) gets only red cards then not only does he know that he will win the game but also he knows that \(A\) knows this. Formulas (3.8), (3.9) state that if \(A\) gets only black cards and \(B\) gets only red cards, then is a common, distributed knowledge that agent \(B\) eventually wins, respectively. Finally, formula (3.10) means “everybody knows \(B\) eventually wins”. Notice that formula 3.8 implies formula 3.10 that implies formula 3.9.

Unfortunately, the state space of the interpreted system defined above for this game is too large. Even symbolic techniques (OBDDs) are not sufficient to represent it on current model checkers. Nevertheless, thanks to Theorem 3.2, instead of checking the system above directly, it is possible to abstract the system to reduce its state space as shown below.

The abstraction chosen for this case collapses all red cards into one, collapses all black cards into one and abstracts away the memory \(m\) of both players. The abstraction process is described via the following surjection \(\rho : C \rightarrow \{\text{red, black}\}\). This function collapses all red cards to the abstract card \(\text{red}\) and all black cards to the abstract card \(\text{black}\):

\[
\rho(r_i) = \text{red} \quad \text{and} \quad \rho(b_i) = \text{black}
\]

For each local state \(\langle h, m \rangle\) of a player \(i \in \{A, B\}\), the memory \(m\) and the card indexes in the hand \(h\) are all abstracted away:

\[
\langle h, m \rangle \equiv_i \langle h', m' \rangle \iff \rho(h) = \rho(h')
\]

where \(\rho(h)\) is the multi-set \(\{\rho(c) \mid c \in h\}\). For example, if \(h = \{b_1, b_2, r_1\}\) then \(\rho(h) = \{\text{black}, \text{black}, \text{red}\}\). Local states of the score keeper are ignored by the abstraction process. Therefore, \(\equiv_S\) is simply identity on \(L_S\).

The equivalence relation \(\equiv_i\) is defined as the smallest equivalence over \(ACT_i\) satisfying the condition (3.11). Therefore, the actions of players are simply abstracted by deleting the indexes from cards.

\[
\text{play} \ c \equiv_i \text{play} \ c' \iff \rho(c) = \rho(c') \quad (3.11)
\]
As before, the actions of the score keeper remain the same: \(\equiv_S\) is identity on \(ACT_S\).

Once equivalences relations \(\equiv_A\), \(\equiv_B\) and \(\equiv_S\) are defined, it is possible to build the quotient of this system with respect to those equivalences relations. The abstract actions \(\text{play}\) \(c\) are represented by \(\text{play}\) \(\rho(c)\), and the abstract local states \([\langle h, m \rangle]\) are represented by the multi-set \(\rho(h)\).

\[
ACT'_i = \{\text{play}\ red, \text{play}\ black, \epsilon\}
\]

\[
L'_i = \{h \mid h \text{ is the multi-set } \{\text{black, red}\} \text{ with } |h| \leq 9\}
\]

\[
P_i(h) = \{\text{play} x \mid x \in h\}, \text{ if } h \neq \emptyset
\]

\[
P_i(h) = \{\epsilon\}, \text{ if } h = \emptyset
\]

for players \(i \in \{A, B\}\), while \(ACT'_S = ACT_S\), \(L'_S = L_S\) and \(P'_S = P_S\).

The abstract local evolution function \(t'_A\) for the player \(A\) contains the following transitions:

\[
h \xrightarrow{(\text{play} x, \text{play} x', \text{eval})} h \setminus \{x\}
\]

\[
h \xrightarrow{(\epsilon, \epsilon, \epsilon)} h
\]

for abstract cards \(x, x' \in \{\text{red, black}\}\), where \(h \setminus \{x\}\) is the result of removing one instance of the abstract card \(x\) from the multi-set \(h\). The abstract local evolution function \(t'_B\) for player \(B\) is defined in the same way, but with the multi-set \(h \setminus \{x'\}\) as the successor state in the first transition.

The abstract local evolution function \(t'_S\) for the score keeper contains the following transitions:

\[
\langle a, b \rangle \xrightarrow{(\text{play} x, \text{play} x, \text{eval})} \langle a + 1, b \rangle
\]

\[
\langle a, b \rangle \xrightarrow{(\text{play} x, \text{play} x, \text{eval})} \langle a, b + 1 \rangle
\]

\[
\langle a, b \rangle \xrightarrow{(\text{play}\ red, \text{play}\ black, \text{eval})} \langle a + 1, b \rangle
\]

\[
\langle a, b \rangle \xrightarrow{(\text{play}\ black, \text{play}\ red, \text{eval})} \langle a + 1, b \rangle
\]

\[
\langle a, b \rangle \xrightarrow{(\epsilon, \epsilon, \epsilon)} \langle a, b \rangle
\]

for abstract card \(x \in \{\text{red, black}\}\). In the first transition above, the score keeper adds a point to the score of \(A\) when players \(A\) and \(B\) play cards of the same colour, while in the second transition, the score keeper adds a point to the score of \(B\) when players \(A\) and \(B\) play cards of the same colour.
Chapter 3. Theoretical results in abstraction of MAS

The set $I_0'$ of abstract initial states consists of all abstract global states $\langle h, h', \langle a, b \rangle \rangle$ such that

$$|h| = |h'| = 9, \ a = b = 0$$

$$|\text{Red}(h)| + |\text{Red}(h')| \leq 10, \ |\text{Black}(h)| + |\text{Black}(h')| \leq 10$$

where $\text{Red}(h)$ is the multi-set of red cards in multi-set $h$, and similarly for $\text{Black}(h)$.

The abstract and concrete systems have the same propositions, $A' = A$, since no proposition in $A$ distinguishes between global states that have been identified. The abstract evaluation function $V'$ is defined by:

$$\text{onlyred}_A \in V'((h, l, l')) \iff \text{Black}(h) = \emptyset$$

$$\text{win}_A \in V'((l, l' \langle a, b \rangle)) \iff a > b \text{ and } a + b = 9$$

and analogously for propositions $\text{onlyred}_B$ and $\text{win}_B$.

This completes the construction of the abstract system $\mathcal{I}'$. The abstract global transition relation $R'$ is sketched in Figure 3.2. The initial states are reduced from $9 \cdot 10^6$ to 36 possible ones in the abstract system.

After having generated the abstract system $\mathcal{I}'$ it is possible to check the ACTLK-formula (3.7) on it in order to know if this specification holds. Notice that now the specification (3.7) uses atomic formulas from $A'$ only. It is possible to check that the formula holds. Since the answer is affirmative it is possible to make use of the Preservation Theorem 3.2 and conclude that the specification holds also in the concrete system $\mathcal{I}$.

This particular abstraction was chosen to check formulas (3.7), (3.8), (3.9), (3.10). For other formulas we may require a different abstraction. For instance, the formula $\text{onlyred}_B \rightarrow \text{AF K} B \text{win}_B$ holds for the concrete system but not for the particular abstract system above.

In this example, the abstraction used has reduced the number of global states of the system $\mathcal{I}$ by a factor of $5 \cdot 10^{11}$: The concrete system has approximately $3 \cdot 10^{18}$ reachable states, while the abstract system has approximately 5 million reachable states “only”. The card game example and the particular surjection function $\rho$ chosen show the potential effectiveness of the abstraction technique presented.
3.3. Existential abstraction examples

Figure 3.2: Sketch of the global transition relation for the abstract card game.
3.3.2 Transmission protocol example

In this section abstraction is applied to a very well known example in the temporal-epistemic logic literature: the bit transmission protocol [FHVM95]. The system is made of three agents: a sender, a receiver and an environment. The sender $S$ tries to send some information to a receiver $R$ via a lossy communication channel. To study the behaviour of the transmission protocol a domain with just two values $\{0, 1\}$ is sufficient. If the protocol works correctly for a domain of just two bits then it would also work for any data domain.

The behaviour of this system for a general domain $D$ works as follows. A sender $S$ and a receiver $R$ communicate via a unreliable channel that can lose the information while this travels at any moment. The goal of the protocol is to transmit a data value $d \in D$ from the sender $S$ to the receiver $R$ in such a way that the sender $S$ will know that the receiver $R$ knows the value $d$. The protocol specifies that $S$ sends the data value to $R$, and continues to send it until $S$ receives an acknowledgment from $R$. For its part, once $R$ receives the data value, $R$ sends an acknowledgment of receipt to $S$, and re-sends it indefinitely. When $|D| = 2$, the transmission protocol is the bit transmission protocol.

The transmission protocol for a non-empty data domain $D$ is defined as an interpreted system $\mathcal{I}^D$ with two agents, the sender $S$ and the receiver $R$. Let us assume the sender $S$ observes his value and whether or not he has received an acknowledgement:

$$L_S = \bigcup_{d \in D} \{d, \langle d, \text{ack} \rangle\}$$

In local state $d$, the sender sees (has) his data value $d$, while in local state $\langle d, \text{ack} \rangle$, the sender sees (has) his value $d$ and the acknowledgement. The receiver, on the other hand, either sees nothing or sees the data value:

$$L_R = \{\lambda\} \cup D$$

In local state $\lambda$, the receiver has not yet seen any value, while in local state $d \in D$, the receiver sees the value $d$.

The sender can send his data value or do nothing ($\epsilon$):

$$\text{ACT}_S = \{\text{send } d \mid d \in D\} \cup \{\epsilon\}$$
while the receiver can send an acknowledgement of receipt or do nothing:

\[ \text{ACT}_R = \{ \text{send} \, d \mid d \in D \} \cup \{ \epsilon \} \]

The protocol for the sender is to keep sending his data value until he receives an acknowledgement:

\[ P_S(d) = \{ \text{send} \, d \} \]
\[ P_S(\langle d, \text{ack} \rangle) = \{ \epsilon \} \]

The receiver should do nothing until it receives a data value, and then keep sending an acknowledgement:

\[ P_R(\lambda) = \{ \epsilon \} \]
\[ P_R(d) = \{ \text{send} \, \text{ack} \, d \} \]

The local evolution function \( t_S \) for the sender contains the following transitions:

\[ d \xrightarrow{\langle \text{send} \, d, \epsilon \rangle} d \]  
(3.12)

\[ d \xrightarrow{\langle \text{send} \, d, \text{send} \, \text{ack} \, d \rangle} d \]  
(3.13)

\[ d \xrightarrow{\langle \text{send} \, d, \text{send} \, \text{ack} \, d \rangle} \langle d, \text{ack} \rangle \]  
(3.14)

\[ \langle d, \text{ack} \rangle \xrightarrow{\langle \epsilon, \text{send} \, \text{ack} \, d \rangle} \langle d, \text{ack} \rangle \]  
(3.15)

In (3.12), the joint action \( \langle \text{send} \, d, \epsilon \rangle \) leaves the local state of \( S \) unchanged: Since the receiver does nothing, the sender obtains no new information. In (3.13), the receiver sends an acknowledgement which is lost on the communication channel, so the local state of \( S \) is unchanged. In (3.14), on the other hand, the acknowledgement reaches the sender, and the local state is updated accordingly. In (3.15), the sender stays in the same local state once the the sender has received the acknowledgement.

Analogously, the local evolution function \( t_R \) for the receiver contains the following transitions:

\[ \lambda \xrightarrow{\langle \text{send} \, d, \epsilon \rangle} \lambda \]

\[ \lambda \xrightarrow{\langle \text{send} \, d, \epsilon \rangle} d \]

\[ d \xrightarrow{\langle \pi, \text{send} \, \text{ack} \, d \rangle} d \]
for $\pi \in ACT_{S}$.

Initially, the sender sees its data value $d$ and the receiver sees nothing:

$$I_0 = \{ \langle d, \lambda \rangle \mid d \in D \}$$

Let the set $A$ of propositions contain the proposition $recack$ (“The sender has received an acknowledgement”) and the propositions $val = d$ (“The value is $d$”), for all $d \in D$. The evaluation function is as expected:

$$V(\langle d, l \rangle) = \{ val = d \}$$

$$V(\langle \langle d, ack \rangle, l \rangle) = \{ val = d, \text{recack} \}$$

for $l \in L_R$. This completes the definition of the interpreted system $I^D$ for the transmission protocol.

One of the specifications often checked on the transmission protocol system is that whenever the sender $S$ has received an acknowledgement, the sender $S$ knows that the receiver $R$ knows that the value is $d$. Formally, for all $d \in D$, if $I^D$ satisfies:

$$AG \left((val = d \land \text{recack}) \rightarrow K_SK_R(val = d)\right) \quad (3.16)$$

By applying Preservation Theorem 3.2, it is possible to show that the specification (3.16) holds in $I^D$ for any chosen data domain $D$, if the specification holds for the bit transmission protocol, i.e., if it holds for $D = \{0, 1\}$. The latter is, of course, feasible for a model checker to determine.

Fix any $d_0 \in D$. The concrete system $I^D$ can be abstracted by identifying all data values $d$ which are distinct from $d_0$. Define an abstraction function $\rho : D \rightarrow \{ d_0, \neg d_0 \}$ by:

$$\rho(d_0) = d_0$$

$$\rho(d) = \neg d_0, \text{ if } d \neq d_0$$

Local states that are identical are identified after applying $\rho$ on the data values inside:

$$d \equiv_S d' \iff \rho(d) = \rho(d')$$

$$d \equiv_R d' \iff \rho(d) = \rho(d')$$

$$\langle d, \text{ack} \rangle \equiv_S \langle d', \text{ack} \rangle \iff \rho(d) = \rho(d')$$
Similarly, actions that are identical are collapsed together after renaming:

\[
\text{send } d \equiv_S \text{send } d' \iff \rho(d) = \rho(d')
\]

\[
\text{send ack } d \equiv_R \text{send ack } d' \iff \rho(d) = \rho(d')
\]

Now, let \( I' \) be the quotient of \( I^D \) with respect to equivalences \( \equiv_S \) and \( \equiv_R \).

The abstract system \( I' \) is just the bit transmission protocol system \( I^{(d_0, \sim d_0)} \), except that \( A' \) contains only the abstract propositions \( \text{reckack} \) and \( \text{val} = d_0 \).

If we check system \( I^{(d_0, \sim d_0)} \) against formula (3.16) we would find the formula to be satisfied. Therefore, it is possible to infer that the specification (3.16) holds for concrete system \( I^D \) as well. Since, the abstract system \( I' \) is just \( I^{(0,1)} \), by assumption, \( I' \) satisfies:

\[
\text{AG }((\text{val} = d_0 \land \text{reckack}) \rightarrow \text{KSKR}(\text{val} = d_0))
\]  

(3.17)

By Preservation Theorem 3.2 it follows that the formula (3.17) is satisfied by the concrete system \( I^D \). Since, \( d_0 \) was chosen arbitrarily from \( D \), it is possible to conclude that the general formula (3.16) holds for all \( d \in D \).

The transmission protocol (together with its several variations) is a well known example employed in the literature to illustrate abstraction for reactive systems. Usually, formulas checked on the transmission protocol describe only control flow. Therefore, those specifications permit us to abstract away all data values. Formula (3.16) instead is related to the knowledge of specific data, and so does not permit us to abstract away all data values. For a surjection function that collapses all the data values (i.e. \( \rho : D \rightarrow \{0\} \)), it is not possible to apply Theorem 3.2 to the formula (3.16) since the set of abstract atomic propositions \( A' \) in the abstract system would contain only the proposition \( \text{reckack} \) according to Definition 3.1 of \( A' \).

In this chapter we developed the theoretical basis for the abstraction techniques that will be presented in the next chapters. These contributions, regarding the construction of quotient interpreted systems, do not refer to the way this partition can be constructed. Hence, in this thesis we identify two ways to partition the state space of interpreted systems. The first concerns the partitioning of the domains of variables. This technique is called \textit{data-abstraction} and will be presented in the next chapter. The second concerns the partitioning of the set of variables for each agent in the interpreted system under investigation. This technique is called \textit{variable-abstraction}.
Both techniques rely on the results presented in this chapter that extend those results presented by Clarke in \cite{CGL94,CGJ00,CGJ03}.
Chapter 4

Automatic data-abstraction for MAS

4.1 Model checking in temporal epistemic logics: a general overview

While a number of abstraction-based techniques have been put forward for plain temporal logic, e.g., [HSGS07, Cha05, CGJ+03, Wan04, CGL94], little attention has gone so far toward developing efficient state-reduction methodologies preserving the validity of temporal-epistemic specifications. Crucially, there is no automatic implementation enabling the user to perform automatic abstraction directly on the program. In this Chapter a data abstraction technique is presented in order to fill this gap. This technique makes use of ISPL, the input language of MCMAS, to describe interpreted systems. Data-abstraction notions can be defined on interpreted systems semantics and automatic reduction can be performed directly on ISPL programs. The technique is applied on two scenarios inspired by popular examples in the MAS literature: a card game [CDLR09], and the transmission problem [FHVM95], which have been introduced in the previous Chapter. Both the scenarios considered have over $10^{10}$ reachable states so are too large to be checked by MCMAS directly, but can be verified effectively by model checking the reduced program. The examples describe reactive systems. Experimental results for those systems are reported in two tables. Both tables show considerable reductions in verification time and memory used.
4.2 Data-abstraction algorithm

Definition 3.2 and Theorem 3.2 do not give a constructive way for building the abstract model. For any implementation purposes, an algorithm for defining appropriate equivalence relations should be given. In the following, such a procedure is described in the case of interpreted systems.

As seen in Chapter 2, an interpreted system \( I = \langle \{ L_i \}_{i \in Ag} \cup \{ L_e \}, \{ ACT_i \}_{i \in Ag} \cup \{ ACT_e \}, \{ P_i \}_{i \in Ag} \cup \{ P_e \}, \{ t_i \}_{i \in Ag} \cup \{ t_e \}, I_0, V \rangle \) over a set \( A \) of atomic propositions, defines local states, actions, protocols, and local transition functions for agents and environment corresponding to a given interpreted system.

Global states are defined by means of a finite set \( VAR = \{ v_1, \ldots, v_m \} \) of variables. Each variable \( v_k \in VAR \) has an associated finite domain \( D_k \), where \( k \in \{ 1, \ldots, m \} \). The set \( \{ +, -, \div, \cdot \} \) denotes standard arithmetic operations and the set \( \{ <, >, =, \leq, \geq \} \) denotes binary relation symbols. We consider three types of expressions:

- **Arithmetic expressions.** These are built from variables in \( VAR \), constants in \( D_k \) and arithmetic operations; for instance, \( v_2 - 5 \) is an arithmetic expression.

- **Logic expressions.** These are built from arithmetic expressions and relation symbols as natural; for instance, \( v_2 - 5 > 4 \) is a logic expression. Logic expressions will play a crucial role in the abstraction process. The abstraction algorithm, defined later in this chapter, is based on the logic expressions.

- **Boolean expressions.** These are composed of logic expressions \( l \) using negation \( \neg \), conjunction \( \wedge \) and disjunction \( \lor \); for instance \( (v_2 - 5 > 4) \lor (v_1 - 2 = 3) \) is a Boolean expression. Boolean expressions are used to define atomic propositions \( p \in AP \). Therefore, atomic propositions are composed of one or more logic expressions.

Let \( D = D_1 \times \cdots \times D_m \) be the Cartesian product of all domains of the variables in \( VAR \). Any global state \( g \) of the interpreted system \( I \) can be seen as an instantiation over the set of variables \( VAR = \{ v_1, \ldots, v_m \} \); i.e., \( g = (d_1, \ldots, d_m) \in D \). Similarly, the local states of an agent can be seen as instantiations over a subset \( VAR_i \) of \( VAR \), named local variables of the agent \( i \in Ag \). The expression \( g \models \phi \), where \( g = (d_1, \ldots, d_m) \in D \), means that the formula obtained
4.2. Data-abstraction algorithm

by replacing each occurrence of the variables $v_i \in \text{VAR}$ in $\phi$ by the constant $d_i$ evaluates to true.

The details of the abstraction procedure on the data of the program will be given by describing a variant of the Card Game Example already introduced in the previous Chapter. Here, the game is defined in a slightly different way in order to show how data-abstraction is performed.

**Card Game Example [LQR10].**

“The system has two agents Player1 and Player2 and an environment $e$. There is a deck of $2N$ cards. Each player receives $N - 1$ cards. Two cards are put aside. Higher index cards beat lower index cards. In each round of the game, each player plays a card from his or her hand. The player playing the stronger card wins the round. The game continues until all cards have been played. The player who won the most number of rounds wins the game”.

A formula that can be checked on this system is the following one:

“If a player gets a certain combination of cards, then does he know that he will win the game eventually?”

We model this game by defining an interpreted system $\mathcal{I} = \langle \{L_i\}_{i \in \text{Ag}} \cup \{L_e\}, \{ACT_i\}_{i \in \text{Ag}} \cup \{ACT_e\}, \{P_i\}_{i \in \text{Ag}} \cup \{P_e\}, \{t_i\}_{i \in \text{Ag}} \cup \{t_e\}, I_0, V \rangle$ over the set $\text{Ag}$ of agents and the set $A$ of propositions. described as follows.

The set of agents is $\text{Ag} = \{1, 2\}$. Let $\mathcal{C} = \{1, \ldots, 2N\}$ represent the deck of $2N$ cards. A player $i \in \{1, 2\}$ can either play a card or do nothing:

$ACT_i = \{\text{playcard } c_i^k \mid c_i^k \in \mathcal{C}\} \cup \{\text{nothing}\}$

The environment $e$ either calculates who wins the current round or does nothing:

$ACT_e = \{\text{eval, nothing}\}$

The local state of an agent describes what cards he holds and how many rounds he has played so far, as well as the outcome of the game:

$L_i = \{(\mathcal{H}_i, k, a, b) \mid |\mathcal{H}_i| = (|\mathcal{C}|/2) - 1\}$

where $\mathcal{H}_i \subset \mathcal{C}$ represents the cards held by the agent $i$ and $k \in \mathcal{N} = \{1, \ldots, N\}$ represents the game round, and $a$ and $b$ encode the number of deals won by player 1 and 2, respectively.
Chapter 4. Automatic data-abstraction for MAS

The environment records the current score in its local state, whose domain is:

\[ L_e = \{(a, b) \mid a + b \leq N - 1\} \]

where \( a \) and \( b \) encode the number of deals won by player 1 and 2, respectively. Each player \( i \) plays one card per round until all cards held by the two players are played. The local protocols are defined as follows:

\[ P_i(H_i, k, a, b) = \begin{cases} \{ \text{playcard}_{c^k_i} \mid c^k_i \in H_i \text{ and } k \in \mathcal{N} \} & \text{if } k < N; \\ \{ \text{nothing} \} & \text{if } k = N; \\ \{ \text{eval} \} & \text{if } a + b < N - 1; \\ \{ \text{nothing} \} & \text{if } a + b = N - 1. \end{cases} \]

The local evolution functions have the form:

\[ t_i((H_i, k, a, b), \langle \text{nothing, nothing, nothing} \rangle) = (H_i, k, a, b) ; \]  
\[ t_i((H_i, k, a, b), \langle \text{playcard}_{c^k_1}, \text{playcard}_{c^k_2}, \text{eval} \rangle) = (H_i, k + 1, a', b') ; \]
\[ t_e((a, b), \langle \text{playcard}_{c^k_1}, \text{playcard}_{c^k_2}, \text{eval} \rangle) = (a + 1, b), \text{ if } c^k_1 > c^k_2 ; \]
\[ t_e((a, b), \langle \text{playcard}_{c^k_1}, \text{playcard}_{c^k_2}, \text{eval} \rangle) = (a, b + 1), \text{ if } c^k_1 < c^k_2 ; \]
\[ t_e((a, b), \langle \text{nothing, nothing, nothing} \rangle) = (a, b) . \]

where \( a' \) and \( b' \) in (4.2) are the new updated values of \( a \) and \( b \), according to (4.3) and (4.4). \( I_0 \) is the set of all global states \( g = (H_1, H_2, k, a, b) \) such that:

\[ k = 0, a = 0, b = 0, H_1 \cap H_2 = \emptyset, |H_1| = |H_2| = |C|/2 - 1 \]

The atomic propositions considered are allred\(_i\) (“Player \( i \) holds only red cards.”), win\(_i\) (“Player \( i \) has won the game.”), topred\(_i\) and lowred\(_i\) for \( i \in \{1, 2\} \). Therefore:

\[ AP = \{ \text{allred}_1, \text{allred}_2, \text{win}_1, \text{win}_2, \text{topred}_1, \text{topred}_2, \text{lowred}_1, \text{lowred}_2 \}. \]

The interpretation \( V : W \rightarrow 2^{AP} \) is defined as follows:

\[ \text{allred}_i \in V \iff H_i \subset \{N + 1, \ldots, 2N\} ; \]
\[ \text{topred}_i \in V \iff H_i = \{N + 2, \ldots, 2N\} ; \]
\[ \text{lowred}_i \in V \iff H_i \subset \{N, \ldots, 2N\} ; \]
\[ \text{win}_1 \in V \iff a > b \text{ and } a + b = N - 1 ; \]
\[ \text{win}_2 \in V \iff b > a \text{ and } a + b = N - 1 . \]
The atomic formula $allred_i$ represents a hand in which the player $i$ gets all red cards (we remind that a card is red if it belongs to the set $\{N+1,\ldots,2N\}$). In this case it is still possible that the his adversary at most gets one red card since $\mathcal{H}_i \subset \{N+1,\ldots,2N\}$ and $|\mathcal{H}_i| = |\{N+1,\ldots,2N\}|-1$. The atomic formula $topred_i$ represents a hand in which the player $i$ gets all red cards and he is sure he has got the highest cards since $\mathcal{H}_i = \{N+2,\ldots,2N\}$. This situation guarantees him to win the game. The atomic formula $lowred_i$ represents a hand in which the player $i$ gets all red cards, but it is still possible that his adversary at most gets two red cards since $\mathcal{H}_i \subset \{N,\ldots,2N\}$ and $|\mathcal{H}_i| = |\{N+1,\ldots,2N\}|-2$. Finally, the atomic formulas $win_1$ and $win_2$ represent the winning conditions for player 1 and player 2, respectively.

For $N = 3$ the deck of cards is $\mathcal{C} = \{1,2,3,4,5,6\}$. The set $\mathcal{H}_1$ is described by the variables $c_{11}$ and $c_{12}$, and $\mathcal{H}_2$ by $c_{21}$ and $c_{22}$. In this case both players get 2 cards each, and 2 cards are put aside. The atomic formula $allred_1$ represents the hand $\mathcal{H}_1 \subset \{4,5,6\}$, with $|\mathcal{H}_1| = 2$. Therefore player 1 has got one of the three following hands: $\{4,5\}$, $\{5,6\}$ or $\{4,6\}$. None of these situations can guarantee him to win the game, since if his adversary has got the remaining red card (card 6, 4, and 5 respectively) then they might end up the game with a draw. The atomic formula $topred_1$ represents the hand $\mathcal{H}_1 = \{5,6\}$. In this case player 1 has got the two highest cards in the deck. This hand does guarantee him the victory.

Figure 4.1 illustrates a sketch of the global transition relation for the concrete Card Game Example with a deck of 6 cards ($N = 3$). The dashed line represents the epistemic relation $\sim_1$. Variable $n$ represents the game round. The set of local states for the environment $e$ is $L_e = \{(a,b) \mid a+b \leq 2\}$, i.e., there are just two rounds. In the first round, the players play the cards $c_{11}$, $c_{21}$; in the second round, they play $c_{21}$, $c_{22}$. The notation “$2,3-1,5$” in Figure 4.1 means that Player1 holds cards $c_{11} = 2$, $c_{12} = 3$, while Player2 holds cards $c_{21} = 1$, $c_{22} = 5$. At the first round, Player1 plays card 2 and Player2 plays card 1. At the second round they play cards 3 and 5, respectively. Figure 4.1 shows four possible executions. In the first two executions on the right side of the figure, the players draw ($a=b=1$). In the two executions on the left side of the, Player2 wins in both cases ($b=2$).

### 4.2.1 Definition of the abstraction function

The data-abstraction algorithm starts by partitioning the domains $D_1,\ldots,D_m$ of variables. This takes an interpreted system $\mathcal{I}$ as input and builds a quotient interpreted system $\mathcal{I}'$ for $\mathcal{I}$.
Figure 4.1: Four executions of the card game with $N = 3$. According to Definition 3.2, the quotient system is generated via a set of abstraction functions. The data-abstraction algorithm defines a set of abstraction functions $\{\rho_1, \ldots, \rho_m\}$, where each $\rho_k$ is defined on $D_k$ ($k \in \{1, \ldots, m\}$), the domain of variable $v_k$. Given a variable $v_k$ of an agent $i \in Ag$ over a domain $D_k$, and two values of $v_k$ $d, e \in D_k$ the component abstraction function $\rho_k$ is defined in a similar way as in \[CGJ+03\] as follows:

$$\rho_k(d) = \rho_k(e) \iff \bigwedge_{p \in LE_k} d \models p \iff e \models p$$  (4.12)

where $LE_k$ is the set of all and only logic expressions $p$ that contain local variables of agent $k$.

The abstraction functions $\rho_1, \ldots, \rho_m$ induce $m$ equivalence relations $\equiv_{\rho_1, \ldots, \equiv_{\rho_m}$ respectively, according to Definition 3.2. Hence, two values $e, d$ of a variable $v$ of the agent $i$ are in the same equivalence class $\equiv_{\rho_k}$ if they cannot be distinguished by the same set of logic expressions. Formally:

$$d \equiv_{\rho_k} e \iff \rho_k(d) = \rho_k(e)$$  (4.13)

Informally, when the above happens two values $d$ and $e$ are said to be “collapsed” by the abstraction function $\rho_k$.

Every logic expression gets an integer $l \in \mathbb{N}$ in order to be identified. If a value $e$ of the concrete interpreted system satisfies the logic expression identified by $l$ then $l$ is added to a set of integers $1$.

\[CGJ+03\] the abstraction functions $\rho_k$ are defined for simple Kripke structures with no agents.
4.2. Data-abstraction algorithm

Given an agent $k \in Ag$, an abstraction function $\rho_k$ and a concrete local state $e$. The abstract local state $\rho_k(e)$ is calculated as follows:

$$\rho_k(e) = \sum_{l \in ID_e} 2^l$$  \hspace{1cm} (4.14)

Notice that if two values $e$ and $d$ satisfy the same logic expressions, we have $ID_e = ID_d$. Therefore, we have $\rho_k(e) = \rho_k(d)$. Hence, formula (4.14) satisfies condition (4.12).

For instance, in the interpreted system for the card game with $N = 3$ in Figure 4.1 we have that variables $c_{11}, c_{12}, c_{21}, c_{22}$ of the agent Environment (where $c_{11}, c_{12}$ represent the cards held by agent Player 1, and $c_{21}, c_{22}$ the cards held by the agent Player 2) have the domains $D_{c_{11}} = D_{c_{12}} = D_{c_{21}} = D_{c_{22}} = \{1, 2, 3, 4, 5, 6\}$. The set of variables of the agent Environment, Player 1, Player 2 are $\text{VAR}_e = \{a, b\}$, $\text{VAR}_1 = \{k, c_{11}, c_{12}\}$, $\text{VAR}_2 = \{k, c_{21}, c_{22}\}$, respectively. For instance, according to the condition (4.12), the concrete values 1 and 2 of the concrete variables $c_{11}, c_{12}, c_{21}, c_{22}$ are collapsed into the abstract value 0 as they do not satisfy any logic expression in both $LE_1 = \{c_{11} > 2, c_{12} > 2, c_{11} > 3, c_{12} > 3, c_{11} > 4, c_{12} > 4\}$ and $LE_2 = \{c_{21} > 2, c_{22} > 2, c_{21} > 3, c_{22} > 3, c_{21} > 4, c_{22} > 4\}$. The concrete value 3 becomes the abstract value 1 = $2^0$ since it satisfies formula $c_{ij} > 2$ and this formula gets the index 0. This index is used in the power $2^0$ to calculate the new value 1 (see definition (4.14)). The concrete value 4 becomes 3 since it satisfies formulas $c_{ij} > 2$ and $c_{ij} > 3$. Those formulas have the index 0 and 1 respectively. Therefore, the abstract value 3 is the result of the calculation $3 = 2^0 + 2^1$. Finally, the abstract value 7 represents the concrete values 5 and 6 that satisfy formulas $c_{ij} > 2, c_{ij} > 3$ and $c_{ij} > 4$. Those formulas have index 0, 1 and 2 respectively, therefore: $2^0 + 2^1 + 2^2 = 7$.

Some instances of variables $c_{11}, c_{12}, c_{21}, c_{22}$ are collapsed into new values (in bold fonts) in the following way: $0 = \{1, 2\}$, $1 = \{3\}$, $3 = \{4\}$, $7 = \{5, 6\}$, according to formula (4.12) since values 1, 2 and 5, 6 satisfy the same set of logic expressions such as the set $\emptyset$ and the set.
\{ c_{ij} > 2, c_{ij} > 3, c_{ij} > 4 \}, respectively.

\begin{align*}
1, 2 & \models \emptyset; \\
3 & \models \{ c_{ij} > 2 \}; \\
4 & \models \{ c_{ij} > 2, c_{ij} > 3 \}; \\
5, 6 & \models \{ c_{ij} > 2, c_{ij} > 3, c_{ij} > 4 \};
\end{align*}

where \( i, j \in \{1, 2\} \). Therefore, according to definition 4.14, abstraction function \( \rho_1 \) for the agent 1 is defined as follows:

\begin{align*}
\rho_1(1) &= \rho_1(2) = 0; \\
\rho_1(3) &= 1; \\
\rho_1(4) &= 3; \\
\rho_1(5) &= \rho_1(6) = 7;
\end{align*}

Function \( \rho_2 \) has the same definition.

![DAG diagram](image-url)

Figure 4.2: Four executions of the abstract card game with \( N = 3 \).

The new values are reported in Figure 4.2. Figure 4.2 shows the abstract interpreted system for \( N = 3 \). Notice how some executions branch (some paths split) in the abstract card game, not existing in the original model (see Figure 4.1). This splitting causes new behaviours in the abstract model.
New domains of variable c11, c12, c21, c22 are: \( \{0, 1, 3, 7\} \) (see Figure 4.2). In this case, the abstraction process has reduced the number of reachable states from \( 6!/2! = 360 \) different initial card combinations to \( 4! = 24 \) in the new system.

Now, we can give the definitions of abstract global states and abstract interpreted system.

**Definition 4.1** (Abstract global states).

Consider a set of agents \( Ag = \{1, \ldots, n\} \) and consider a set of abstraction functions \( \rho_k \), for \( k \in Ag \), defined in (4.14) that satisfy the condition (4.12). Consider a concrete global state \( l = (l_1, l_2, \ldots, l_n) \).

The corresponding abstract global state \( l' \) is defined, by means of the set of abstraction functions \( \rho_k \), as follows.

\[
l' = (\rho_1(l_1), \rho_2(l_2), \ldots, \rho_n(l_n)) \]

**Definition 4.2** (Abstract interpreted system).

Consider an interpreted system \( I \) defined over the set of agents \( Ag = \{1, \ldots, n\} \) and over a set of propositions \( A \). Consider a set of abstraction functions \( \rho_k \), for \( k \in Ag \), defined in (4.14) that satisfy the condition (4.12). Let \( A' \) be the set of atomic propositions defined in Definition 3.1. The abstract interpreted system \( I' \) is defined over the set \( Ag \) of agents and the set \( A' \) of proposition such that:

1. \( L'_k = \{ \rho_k(l) \mid l \in L_k \} \);
2. \( ACT'_k = \{ a \mid a \in ACT_k \} \);
3. \( P'_k = \{ \langle \rho_k(l), a \rangle \mid \langle l, a \rangle \in P_k \} \);
4. \( t'_k = \{ \langle \rho_k(l), \bar{a}, \rho_k(l') \rangle \mid \langle l, \bar{a}, l' \rangle \in t_k \} \);
5. \( I'_0 = \{ l' \mid l' = (\rho_1(l_1), \rho_2(l_2), \ldots, \rho_n(l_n)) \text{ and } (l_1, l_2, \ldots, l_n) \in I_0 \} \);
6. \( V'((\rho_1(l_1), \rho_2(l_2), \ldots, \rho_n(l_n))) = V((l_1, l_2, \ldots, l_n)) \cap A' \).
4.3 Implementation

The algorithm presented above was implemented in C++. The prototype toolkit performing data-abstraction on ISPL programs starts by partitioning the domains $D_1, \ldots, D_m$ of variables. The data-abstraction toolkit takes an ISPL program describing an interpreted system $\mathcal{I}$ as input, and returns an ISPL program describing a quotient interpreted system $\mathcal{I}'$ for $\mathcal{I}$. The toolkit automatically builds the set of abstraction functions $\{\rho_1, \ldots, \rho_m\}$ defined in (4.12). The abstract ISPL program $P'$ is automatically generated by substituting in every logic expression $l$ appearing in ISPL program $P$ the corresponding new value in $D'$.

A release of this toolkit is available from [Rus].

Figure 4.3 shows an ISPL program encoding the card game defined previously. The agent Environment has got two observable variables (called Obsvars), $a$ and $b$. Those variables keep the score of player 1 and 2, respectively. Those variables can be seen by all the other agents. Therefore, there is no need to define them for player 1 and 2 as well. The Agent Environment has got four variables representing the cards. The first two $c_{11}, c_{12}$ represent the cards held by Player1, while variables $c_{21}, c_{22}$ represent the cards held by Player2. This choice differs from the definition of the card game interpreted system given in the previous section for design reasons. In this way, we need to write the Evolution rules just once (in the environment agent only). Therefore, we do not need to write them twice in the definitions of the players.

The Evolution of the environment encodes the transition rules (4.3) and (4.4).

The agent Player1 has got the variable $k$ that keeps the round of the game. Variables $c_{11}, c_{12}$ are Lobsvars. This means that, even though those variables belong to agent Environment, they can be seen by the agent Player1. The Protocol says that at round $k = 1$ the first card is played (playcard1), while at round $k = 2$ the second card is played (playcard2). The Evolution encodes the transition rule (1.2). The agent Player2 is defined in the same way.

In the Evaluation section in Figure 4.3 the atomic formulas defined above are defined as follows:

- lowred1 if $(c_{11} > 2$ and $c_{12} > 2);$ 
- topred1 if $(c_{11} > 4$ and $c_{12} > 4);$ 
- allred1 if $(c_{11} > 3$ and $c_{12} > 3);$
Agent Environment

Obsvars:
  a: 0 .. 2;
  b: 0 .. 2;
end Obsvars

Vars:
  c11: 1 .. 6;
  c12: 1 .. 6;
  c21: 1 .. 6;
  c22: 1 .. 6;
end Vars

Actions: {nothing,eval};
-- Protocol is omitted

Evolution
  a=a+1 if c11>c21 and ... 
  b=b+1 if c11<c21 and ... 
--the rest of the Evolution is omitted
end Agent

Agent Player1

Lobsvars={c11,c12};

Vars:
  k: 1 .. 3;
end Vars

Actions = {nothing,playcard1,playcard2};

Protocol:
  k=1: { playcard1 };
  k=2: { playcard2 };
  k=3: { nothing };
end Protocol

Evolution:
  k = 2 if k = 1; -- to keep the round
  k = 3 if k = 2; -- of the game
end Evolution
end Agent

Agent Player2

Lobsvars={c21,c22};

Vars:
  k: 1 .. 3;
end Vars

Actions = {nothing,playcard1,playcard2};

Protocol:
  k=1: { playcard1 };
  k=2: { playcard2 };
  k=3: { nothing };
end Protocol

Evolution is the same as the agent Player1
end Agent

Evaluation
  lowred1 if (c11>2 and c12>2);
  topre1 if (c11>4 and c12>4);
  allred1 if (c11>3 and c12>3);
  win1 if (a>b and a+b=2);
--The corresponding properties
--for Player2 are omitted
end Evaluation

InitStates
  c11<>c12 and c11<>c21 and c11<>c22...
...

Formulae
  (AG(topred1->K(Player1,(AF win1))));
end Formulae

Figure 4.3: Sketch of an ISPL program for the Card Game with 6 cards (N = 3).
win1 if (a>b and a+b=2);
lowred2 if (c21>2 and c22>2);
topred2 if (c21>4 and c22>4);
allred2 if (c21>3 and c22>3);
win2 if (a<b and a+b=2);

Finally, the section InitStates describes all card combinations that Player1 and Player2 can get according to conditions (4.6), as the condition \( c_{11} \neq c_{12}, c_{11} \neq c_{21}, c_{11} \neq c_{22} \) and so on.

**Definition 4.3** (set of atomic propositions \( AP \) for an ISPL).

The set of atomic propositions \( AP \) for an ISPL \( P \) is defined as the set of all IDs appearing on the left side of all \(<\text{proposition_declaration}>s\) in the Evaluation section of ISPL \( P \).

For the ISPL in Figure 4.3 the set of atomic propositions is \( AP = \{ \text{lowred1, topred1, allred1} \} \).

In an ISPL program \( P \), each atomic proposition \( \alpha \in AP \) is defined by a Boolean expression over variables defining on which global states \( d \in D \) the atomic formula \( \alpha \) holds, thereby implementing the evaluation function \( V : D \rightarrow 2^{AP} \) of the interpreted system \( I \). For instance, the atomic proposition \( \text{lowred1} \) is defined by the Boolean expression \( (c_{21}>2 \text{ and } c_{22}>2) \) in terms of the variables \( c_{11} \) and \( c_{12} \).

Let \( AP \) be the set of atomic propositions defined in Definition 4.3. The procedure for building the set \( \{ \rho_1, \ldots, \rho_m \} \) of abstraction functions consists of three key steps.

**1: Building the set \( LE_i \): Algorithm 8**

For each agent \( i \in Ag \), the algorithm builds a set \( LE_i \) (see lines 1–2 of the Algorithm 8) of all and only logic expressions \( l \) that contain local variables of agent \( i \) (see lines 3–4). At line 5 \( K \) represents the set of variables that are updated by an arithmetic expression. At line 6 \( \text{var} \) is a function that returns the variable contained in the given logic expression (as it will be explained later in this section, we assume that each logic expression contains one variable only). Now, if the variable in the logic expression \( l \) belongs to the current agent \( i \) then \( l \) is inserted in the set \( LE_i \). We remind that an atomic proposition is defined by a Boolean expression in the Evaluation section of the ISPL (see Figure 4.3). A Boolean expression is composed of several
Algorithm 8 Building the set $LE_i$. 

1: for all $i \in Ag$ do 
2: $LE_i \leftarrow \emptyset$; 
3: for all $p \in AP$ do 
4: for all logic expressions $l$ in the Boolean expression defining $p$ do 
5: if \( \text{var}(l) \in i \land \text{var}(l) \notin \text{Obsvars} \land \text{var}(l) \notin K \) then 
6: $LE_i \leftarrow LE_i \cup \{l\}$; 
7: end if 
8: end for 
9: end for 
10: end for

logic expressions. Therefore, in the definition of an atomic formula $p \in AP$ there are one or more logic expressions $l$.

Variables $a$ and $b$ are not inserted in $VAR_1$ and $VAR_2$ by Algorithm 8 (see line 5) since they are global observable variables (Obsvars).

For the example in Figure 4.3 the Algorithm 8 generates the sets $LE_e = \emptyset$, $LE_1 = \{c11 > 2, c12 > 2, c11 > 3, c12 > 3, c11 > 4, c12 > 4\}$ and $LE_2 = \{c21 > 2, c22 > 2, c21 > 3, c22 > 3, c21 > 4, c22 > 4\}$ from the logic expressions $p$ found in the atomic formulas defined in Evaluation section in the ISPL program.

Algorithm 8 needs the following observations.

1. We assume that each logic expression contains one variable only. In terms of implementation, before applying the Algorithm 8, if a logic expression in the Evaluation section of the ISPL contains more than one variable, this expression is automatically rewritten by a support procedure as a Boolean expression where each logic expression contains exactly one variable. For instance, the logic expression $v_1 = v_5$, where the domains are $D_{v_1} = D_{v_5} = \{1, 2, 3\}$, is rewritten as the following semantically equivalent Boolean expression:

\[
(v_1 = 1 \land v_5 = 1) \lor (v_1 = 2 \land v_5 = 2) \lor (v_1 = 3 \land v_5 = 3)
\]
2. Further we assume that no logic expression contains the logical “not” connective. Actually, before applying the Algorithm 8, if a Boolean expression contains the “not” connective, this expression is automatically rewritten by an extra algorithm as an equivalent Boolean expression where the “not” connective does not appear. For instance, the Boolean expression \( \neg(v_1 = v_5) \), where the domains are \( D_{v_1} = D_{v_5} = \{1, 2, 3\} \), is rewritten as the following semantically equivalent Boolean expression:

\[
(v_1 = 1 \land (v_5 = 2 \lor v_5 = 3)) \lor (v_1 = 2 \land (v_5 = 1 \lor v_5 = 3)) \lor (v_1 = 3 \land (v_5 = 1 \lor v_5 = 2))
\]

3. Variables that are updated by an arithmetic expression (e.g. \( x = x + 1 \)) in local Evolution sections are not collapsed by the procedure. This is because there might be transitions that are present in the original model but not present in the abstract one. For example, for the card game, \( a \) and \( b \) cannot collapsed as they are updated in the Evolution by the arithmetic expressions \( a = a + 1 \) and \( b = b + 1 \) respectively (see the Evolution of agent Environment in Figure 4.3). Variables that cannot be collapsed are automatically skipped by the Algorithm 8.

---

**Algorithm 9** The generation of the four-dimensional vector \( Q(i, v, l, d) \).

```plaintext
1: \( Q(i, v, l, d) \leftarrow 0 \);
2: \textbf{for all} \( i \in Ag \) \textbf{do}
3: \hspace{1em} \textbf{for all} local variables \( v \in VAR_i \) \textbf{do}
4: \hspace{2em} \textbf{for all} \( l \in LE_i \) \textbf{do}
5: \hspace{3em} \textbf{for all} \( d \in D_v \) \textbf{do}
6: \hspace{4em} \text{if} \( d \models l \) \text{then}
7: \hspace{5em} \( Q(i, v, l, d) \leftarrow \text{True} \);
8: \hspace{4em} \text{else}
9: \hspace{5em} \( Q(i, v, l, d) \leftarrow \text{False} \);
10: \hspace{4em} \text{end if}
11: \hspace{3em} \text{end for}
12: \hspace{2em} \text{end for}
13: \hspace{1em} \text{end for}
14: \text{end for}
```

2: The generation of the four-dimensional vector \( Q \): Algorithm 9
Algorithm 9 builds a four-dimensional vector $Q$ of Boolean values. The first dimension $i$ of $Q$ indexes the agent; the second dimension $v$ indexes the variables of the agent; $l$ indexes all logic expressions in which the current variable appears; the last dimension $d$ indexes the values $d$ of the current variables. The vector $Q$ encodes whether a logic expression $l$ is evaluated to True when all free occurrences of the current variables are replaced by $d$, denoted by $d \models l$.

**Algorithm 10** The generation of new domains $D'_v$ for the abstract ISPL-file.

Line 9 shows how the new values are calculated.

1: for all $i \in Ag$ do  
2: for all local variables $v \in VAR_i$ do  
3: $D'_v \leftarrow \emptyset$;  
4: for all $d \in D_v$ do  
5: $indx_l \leftarrow 0$;  
6: $newval \leftarrow 0$;  
7: for all $l \in LE_i$ do  
8: if $Q(i, v, l, d) = true$ then  
9: $newval \leftarrow newval + 2^{indx_l}$;  
10: end if  
11: $indx_l \leftarrow indx_l + 1$;  
12: end for  
13: $D'_v \leftarrow D'_v \cup \{newval\}$;  
14: end for  
15: end for  
16: end for

3: The generation of new domains $D'_v$ for the abstract ISPL-file: Algorithm 10

From the vector $Q$ new domains of abstract variables are automatically built by collapsing together the values of every concrete variable that satisfies the same set of logic expressions.

Algorithm 10 runs over all agents $i \in Ag$, local variables $v \in VAR_i$ and corresponding values $d \in D_v$ (see lines 1 2 3 respectively). At line 3 $D'_v$ represents the new domain for the current variable $v$.

New values of the variables for the ISPL program $P'$ representing the abstract model are calculated from the old ones by considering which formulas the old values satisfy. Each formula
is identified by an index \((\text{indx}_l)\). The \(\text{indx}_l\) is initialised at line 5. The first logic expression \(l\) in \(L\mathbf{E}_l\) gets the index 0. This index is incremented at each step of the for loop (line 11). Therefore, the second logic expression gets index 1, the third gets index 2 and so forth. All these indexes are used to calculate an integer number that will be the corresponding new value. Each new value of a variable is stored in the integer \(\text{newval}\) and initialised to 0 (line 6). Line 9 shows how the partitioning of values is calculated. Values that satisfy the same set of logic expressions get the same integer. It is easy to see that, by construction, no two values, satisfying different sets of logic expressions, get the same integer.

As expected, the tool might generate behaviours not present in the original model. For instance, let us analyse the following program lines describing part of the environment’s evolution.

\[
\begin{align*}
a &= a+1 \text{ if } c_{11} > c_{21} \\
b &= b+1 \text{ if } c_{11} < c_{21}
\end{align*}
\]

Those lines are first transformed to the following:

\[
\begin{align*}
a &= a+1 \text{ if } (c_{11}=6 \text{ and } c_{21}=5) \text{ or } (c_{11}=6 \text{ and } c_{21}=4) \text{ or } \ldots \\
b &= b+1 \text{ if } (c_{11}=5 \text{ and } c_{21}=6) \text{ or } (c_{11}=4 \text{ and } c_{21}=6) \text{ or } \ldots
\end{align*}
\]

Following the abstraction process, the ISPL program for the abstract model might includes non-determinism. For instance, the previous two lines are transformed to the following ones below since values 6 and 5 become value 4 in the new system for both variables \(c_{11}\) and \(c_{21}\). Therefore, in the case that both player plays card 4 in the abstract system (see \(c_{11}=4\) and \(c_{21}=4\) below) we have that both players are entitle to win the round. Remind that winning a round produces an increment of either \(a\) or \(b\). In this case, MCMAS chooses to execute one of the two following lines non-deterministically. So, the victory of the round is assigned to one of the two players, non-deterministically.

\[
\begin{align*}
a &= a+1 \text{ if } (c_{11}=4 \text{ and } c_{11}=4) \text{ or } \ldots \\
b &= b+1 \text{ if } (c_{11}=4 \text{ and } c_{21}=4) \text{ or } \ldots
\end{align*}
\]

Still, it can be checked that validity for \texttt{ACTLk}-formulas is preserved under the procedure just presented.
Figure 4.4: Sketch of an ISPL-file for the abstract Card Game with 6 cards.
4.4 Correctness theorem

In this section we present a correctness theorem to show that the procedure defined above builds a quotient interpreted system according to Definition 3.2.

**Theorem 4.1 (Correctness Theorem).**

Given an ISPL-program $P$ describing an interpreted system $\mathcal{I}$ and given a specification $\phi$ of the logic $\text{ACTLK}$, let $P'$ be the ISPL program, generated from $P$ by the procedure just presented. $P'$ describes an interpreted system $\mathcal{I}'$. We have that specification $\phi$ holds in $\mathcal{I}$ if $\phi$ holds in $\mathcal{I}'$, i.e.,

$$\mathcal{I}' \models \phi \implies \mathcal{I} \models \phi.$$ 

**Proof.**

The proof follows the scheme depicted in Figure 4.5. We have to show that abstraction algorithm generates an ISPL-code $P'$ that describes an interpreted system $\mathcal{I}'$ that is a quotient one of the interpreted system $\mathcal{I}$ described by the original ISPL-code $P$.

By Theorem 3.2, we only need to prove that $\mathcal{I}'$ is a quotient of $\mathcal{I}$ according to Definition 3.2.

1. $L_i'$ is generated by the abstraction function $\rho_i$ such that $L_i' = \{[l] \mid l \in L_i\}$, which is a partition of the set $L_i$. Therefore point 1 of the definition 3.2 is proved.

2. $\text{ACT}_i' = \text{ACT}_i$ (i.e., $[a] \equiv a$) because $\rho_i$ does not partition the set of actions.

3. $P' = \{([l], a) \mid (l, a) \in P\}$ because the abstraction technique replaces the old values of a variable with the new ones.
4. Note that \( [l] \neq [l'] \) (\([l], [l'] \in L_i'\)) if and only if there is at least one \( l \in [l] \) and one \( l' \in [l'] \) in \( L_i \) such that \( l \neq l' \).

Now, we have to show that if local states \( l \) and \( l' \) are connected in the concrete system, then the corresponding \([l] \) and \([l'] \) are also connected in the abstract one. Formally, we have to show the following condition holds:

\[
\forall l, l' ( (\rho(l) = [l] \land \rho(l') = [l'] \land l \not R l') \implies [l] \hat{R} [l'] )
\]

where \( R \) and \( \hat{R} \) are the global transition relations of concrete system \( I \) and of abstract one \( I' \), respectively. Suppose \( \rho(l) = [l] \land \rho(l') = [l'] \land l \not R l' \), we show \([l] \hat{R} [l'] \). By assuming \( l \not R l' \), we know \( \exists \bar{a} \in \text{Act} \forall i \in \text{Ag} : \langle l, \bar{a}, l' \rangle \in t_i \) and \( \langle l, a \rangle \in P_i \) by Definition 2.13.

As actions are not modified, we have \( \langle [l], [a] \rangle \in P'_i \), where \( l \in [l] \) and \( a \equiv [a] \). Moreover, since the local evolution function is rebuilt by substituting \( l \), such that \( l \in [l] \), with \([l] \), we have \( \langle [l], \bar{a}, [l'] \rangle \in t'_i \) iff \( \langle l, \bar{a}, l' \rangle \in t_i \). Therefore, we have \( \langle [l], \bar{a}, [l'] \rangle \in t'_i \) and \( \langle [l], [a] \rangle \in P'_i \), but that means \([l] \hat{R} [l'] \).

5. Point 5 and 6 in Definition 3.2 can be proved trivially.

\[\square\]

4.5 The number transmission protocol and evaluation

In this section experimental results are presented and discussed. Specifically, experiments are conducted on the card game example and on variant of the bit transmission problem. These scenarios are to be considered as benchmarks of the abstraction technique presented and not as real applications.

4.5.1 Number transmission protocol

This scenario \([\text{LQR10}]\) is an extension of the well known \textit{bit transmission problem} described in \([\text{FHVM95}]\).
Let \( \text{Ag} = \{S, R\} \) be the set of agents where \( S \), \( R \) represent the sender and the receiver respectively. Moreover, the environment agent is labelled with the letter \( E \). In this scenario an integer is sent from the sender to the receiver via an unreliable channel modelled by the environment.

Notice that in the well known bit transmission problem only one bit is sent. In this scenario the sender sends an integer ranging from 1 to \( N \). The set \( D \) can be any set of integers. For the experiments below we chose that the sender’s variable \( N \) ranges over the set of integer values:

\[
D = \{ 0.5 \cdot 10^4, 10^4, 2 \cdot 10^4, 2.5 \cdot 10^4, 3 \cdot 10^4 \}.
\]

The interpreted system for the protocol just introduced is described below.

The environment is described by the set of local states, as follows:

\[
L_E = \{ S, R, RS, none \}
\]

The local state \( S \) represents the channel reliably sending messages from the sender to the receiver and dropping messages from receiver to sender. Conversely, \( R \) represents a situation where messages only travel from receiver to sender. \( RS \) encodes a situation where the channel is transmitting in both directions, whereas when in \( none \) the channel loses all messages.

The set of local states for the sender \( S \) is

\[
L_S = D \times \{ \text{true}, \text{false} \}
\]

where \( D = \{1, \ldots, N\} \) represents the domain of the integer that can be sent and \( \{\text{true}, \text{false}\} \) is the domain of the variable \( \text{ack} \) keeping track of whether an acknowledgement has been received from the sender. The receiver \( R \) is modelled by the set

\[
L_R = \{ \text{received}, \text{notrec} \}
\]

where \text{notrec} represents a situation where the receiver has not yet received any message, while \text{received} encodes the fact that the receiver got a message. The environment can perform four actions:

\[
ACT_E = \{ S, SR, R, none \}
\]

representing which direction, if any, it is letting messages flow. The sender \( S \) can either send the intended message or do nothing

\[
ACT_S = \{ \text{send}, \text{nothing} \}
\]
Similarly, the receiver can either send an acknowledgement or remain silent:

\[ ACT_R = \{ \text{sendack, nothing} \} \]

As in the original bit transmission problem, the sender keeps sending the same message until he or she receives an acknowledgement; the receiver remains silent before receiving the message from the sender; after that he repeatedly sends acknowledgements back to the sender. Consequently the protocols can be defined as follows.

\[
\begin{align*}
PE(S) &= \{ S \}; \\
PE(SR) &= \{ SR \}; \\
PE(R) &= \{ R \}; \\
PE(\text{none}) &= \{ \text{none} \}; \\
P_S(N, \text{false}) &= \{ \text{send} \}; \\
P_S(N, \text{true}) &= \{ \text{nothing} \}; \\
P_R(\text{notrec}) &= \{ \text{nothing} \}; \\
P_R(\text{received}) &= \{ \text{sendack} \}.
\end{align*}
\]

Local evolution functions are defined as follow:

\[
\begin{align*}
t_E(\text{state}, \langle a_E, a_S, a_R \rangle) &= \text{state'}; \\
t_S((N, \text{false}), \langle SR, a_S, \text{sendack} \rangle) &= (N, \text{true}); \\
t_S((N, \text{false}), \langle R, a_S, \text{sendack} \rangle) &= (N, \text{true}); \\
t_R(\text{notrec}, \langle SR, \text{send}, a_R \rangle) &= \text{received}; \\
t_R(\text{notrec}, \langle S, \text{send}, a_R \rangle) &= \text{received}.
\end{align*}
\]

Where \text{state}, \text{state}' \in L_E \text{ and } a_k \in ACT_k, \text{ for } k \in A_g. \text{ Notice that by } (4.15) \text{ the channel behaves non-deterministically. In (4.16) and (4.17) the sender receives an acknowledgement from the receiver as the channel transmits messages from the receiver to the sender in both cases. Similarly, in (4.18) and (4.19) the receiver receives the number.}

Initially, the sender sees its number \( N \) and the receiver has not received anything yet. Therefore, the set \( I_0 \) of initial global states is defined as follows:

\[
I_0 = \{ (E.\text{state}, S.\text{state}, R.\text{state}) \mid S.\text{state} = (N, \text{false}), R.\text{state} = \text{notrec} \}.
\]
Agent Environment

Vars:
  state : \{S,R,SR,none\};
end Vars

Actions = \{S,SR,R,none\};

Protocol:
  state=S: \{S,SR,R,none\};
  state=R: \{S,SR,R,none\};
  state=SR: \{S,SR,R,none\};
  state=none: \{S,SR,R,none\};
end Protocol

-- Evolution Omitted
end Agent

Agent Sender

Vars:
  number : 0..10000;
  ack : boolean;
end Vars

Actions = \{send,nothing\};

Protocol:
  ack=false : \{send\};
  ack=true : \{nothing\};
end Protocol

Evolution:
  (ack=true) if (ack=false)
  and (Receiver.Action=sendack)
  and ((Environment.Action=SR)
  or (Environment.Action=R))
end Evolution
end Agent

Agent Receiver

Vars:
  state : \{received,notrec\};
end Vars

Actions = \{nothing,sendack\};

Protocol:
  state=notrec : \{nothing\};
  state=received : \{sendack\};
end Protocol

Evolution:
  state=received if (Sender.Action=send)
  and (state=notrec)
  and ((Environment.Action=SR)
  or (Environment.Action=S));
end Evolution
end Agent

Evaluation

recNumber if ( (Receiver.state=received);
  recack if ( ( Sender.ack = true ) );
  N1 if ( (Sender.number=1) );
  N2500 if ( (Sender.number>2500) );
  N5000 if ( (Sender.number>5000) );
  N7500 if ( (Sender.number>7500) );
end Evaluation

InitStates

!Sender.Number=0 and Receiver.state=notrec
and Sender.ack=false and (Environment.state=S
or Environment.state=R or Environment.state=SR
or Environment.state=none);
end InitStates

Figure 4.6: Sketch of an ISPL program for the number transmission protocol for $N = 10000$. 
Finally, the evaluation function $V$ for the set of atomic propositions $AP = \{ N1, N2500, N7500, recack, recNumber \}$ is defined as follows (see Evaluation section in Figure 4.6):

\[
\begin{align*}
N1 \in V(E.state, S.state, R.state) \iff & \quad N = 1; \\
N2500 \in V(E.state, S.state, R.state) \iff & \quad N > 2500; \\
N5000 \in V(E.state, S.state, R.state) \iff & \quad N > 5000; \\
N7500 \in V(E.state, S.state, R.state) \iff & \quad N > 7500; \\
recack \in V(E.state, S.state, R.state) \iff & \quad \text{ack} = \text{true} \land R.state = \text{received}; \\
recNumber \in V(E.state, S.state, R.state) \iff & \quad R.state = \text{received}.
\end{align*}
\]

Figure 4.6 represents a sketch of an ISPL-file corresponding to the interpreted system defined above for $N = 10000$. From Figure 4.6 in the Evaluation section, one might want to know if the number sent was either exactly 1 or it was greater than 2500, 5000 or 7500. By applying the abstraction procedure to this file we obtain an abstract Number Transmission ISPL-file in which the Sender can only send 5 possible digits. The system in Figure 4.6 corresponds to the second one listed in Table 4.2. The new values (in bold font) correspond to the concrete system values in the following way:

\[
\begin{align*}
1 & = \{ 1 \}; \\
2 & = \{ 0, 2, \ldots, 2500 \}; \\
3 & = \{ 2501, \ldots, 5000 \}; \\
4 & = \{ 5001, \ldots, 7500 \}; \\
5 & = \{ 7501, \ldots, 10000 \}.
\end{align*}
\]

In fact, the concrete values of variable $N$ satisfy the following subsets of logic expressions:

\[
\begin{align*}
1 & \models \{ N1 \}; \\
0, 2, \ldots, 2500 & \models \emptyset; \\
2501, \ldots, 10000 & \models \{ N2500 \}; \\
5001, \ldots, 10000 & \models \{ N2500, N5000 \}; \\
7500, \ldots, 10000 & \models \{ N2500, N5000, N7500 \}.
\end{align*}
\]

The atomic proposition $N1$ means $N = 1$, therefore is satisfied for value 1. The atomic proposition $N2500$ means $N > 2500$, therefore is satisfied for all values greater then 2500.
(\(N > 2500\)). The rest of the atomic propositions have the same meaning. Atomic propositions \(N1, N2500, N5000, N7500\) are defined in the Evaluation section (see Figure 4.6) as follows:

\[
\begin{align*}
N1 & \quad \text{if } \text{Sender.number} = 1; \\
N2500 & \quad \text{if } \text{Sender.number} > 2500; \\
N5000 & \quad \text{if } \text{Sender.number} > 5000; \\
N7500 & \quad \text{if } \text{Sender.number} > 7500.
\end{align*}
\]

4.5.2 Experimental results

Both examples above were tested on the abstraction tool-kit paired with MCMAS against some specifications for the protocols. In the case of the card game example the formula verified was

\[
\AG \left( topred_1 \rightarrow K_{player1}(AF\text{win1}) \right)
\]

this formula specifies that if player one has only \(topred\) cards he knows he will win the game.

For the number transmission protocol (specifying that once an ack has been received the sender knows that the receiver knows the value of the number transmitted). The formula tested was

\[
\AG \left( (\text{numberN} \land \text{recack}) \rightarrow (K_S K_R \text{number} = N) \right)
\]

The experiments were executed on a machine running Ubuntu 9.10 on an Intel Core 2 1.86GHz with 1GB memory. The results are reported in Table 4.1 and Table 4.2. Both tables show:

- How the number of states increases by increasing the number of cards for the Card Game, and the maximum integer that can be sent by the Sender for the Number Transmission Protocol.

- The total verification time in seconds needed by both abstraction tool and by MCMAS to check the model.

- The BDD memory usage in Megabytes.

From Table 4.1 it is possible to notice that, as expected, the implementation drastically reduces both global states and time for both abstraction process and verification process. In the case
Table 4.1: Verification results for the card game.

<table>
<thead>
<tr>
<th>Number of cards</th>
<th>With reduction</th>
<th>Without reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>States</td>
<td>Time (s)</td>
</tr>
<tr>
<td>6</td>
<td>138</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>22528</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>135866</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>762812</td>
<td>26</td>
</tr>
<tr>
<td>14</td>
<td>$3.877 \times 10^6$</td>
<td>106</td>
</tr>
</tbody>
</table>

Table 4.2: Verification results for the number transmission problem.

<table>
<thead>
<tr>
<th>Maximum number N</th>
<th>With reduction</th>
<th>Without reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>States</td>
<td>Time (s)</td>
</tr>
<tr>
<td>5000</td>
<td>48</td>
<td>0</td>
</tr>
<tr>
<td>10000</td>
<td>60</td>
<td>0</td>
</tr>
<tr>
<td>15000</td>
<td>84</td>
<td>1</td>
</tr>
<tr>
<td>20000</td>
<td>108</td>
<td>1</td>
</tr>
<tr>
<td>25000</td>
<td>132</td>
<td>1</td>
</tr>
<tr>
<td>30000</td>
<td>156</td>
<td>1</td>
</tr>
</tbody>
</table>

of 12 and 14 cards, MCMAS could not verify the specification in over 24hrs, while the abstract systems could be verified in seconds.

It is perhaps less obvious that, from Table 4.2 in the transmission problem for the case of $N = 10000$ and $N = 15000$ the two systems have the same number of reachable states. This is because MCMAS uses 14 BDD variables to encode both 10000 and 15000 states and MCMAS does not remove redundant states, using $2^{14}$ BDD states in both cases. The same phenomenon occurs for $N = 20000$, $N = 25000$ and $N = 30000$. In this case, MCMAS uses 15 BDD variables. In all experiments in both tables the time for the abstraction process was always negligible (it took less than a second) compared to the time needed for the verification process.
4.6 Evaluation of the technique on a realistic use-case: software development

In this section, we apply the methodology above to a scenario from the web service literature. This is an interesting scenario as it is not easy to predict the behaviour of a system in which there are many agents interacting with each other. Some agents may either fail to provide their services correctly or they can have an unexpected behaviour. This is why the web-services are strictly regulated by contracts.

![Diagram](image_url)

Figure 4.7: Interaction between various partners in the composition, from [LQS08b].

In [LQS08a, LQS08b] the authors present a contract-regulated software development system to describe interactions, violations, penalties, changes in a contract regulated scenario between services. The system is a composition of seven agents: a principal software provider (PSP), a software provider (SP), a testing agency (T), a hardware supplier (H), a software client (SC), an insurance company (I) and a technical expert (E). Interactions among these agents are shown in Figure 4.7.

4.6.1 Contract-regulated software development

This scenario [LQS08a, LQS08b] is described as follows.
### 4.6. Evaluation of the technique on a realistic use-case: software development

<table>
<thead>
<tr>
<th>PSP’s obligations:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Update SP and C twice about the progress of the software.</td>
</tr>
<tr>
<td>2. Integrate the components and send them to T for testing.</td>
</tr>
<tr>
<td>3. If components fail, integrate the revised software and send them for testing.</td>
</tr>
<tr>
<td>4. Make payment to SP after successful deployment of software.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C’s obligations:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Request changes before the second round of updates.</td>
</tr>
<tr>
<td>2. Pay penalty if changes are requested after second round of updates.</td>
</tr>
<tr>
<td>3. Make payment to the PSP after every update.</td>
</tr>
</tbody>
</table>

![Figure 4.8: Some obligations of contract parties, from LQS08](LQS08)

A software client wants a software artifact to be created and installed by a technical expert E on hardware supplied by H. The software is built by two agents, such as: the PSP and PS. The PS agent provides certain components of the software and the integration of those components is carried out by PSP with its own software subsequently. These two agents send messages to each other and to the client C to update all parties about the progress of the software development. The client C is allowed to require some changes before the second round of updates only. Any additional request made by the agent C after the second round is identified as a violation of the contract. In this case, two possibilities are open to the client: either he withdraws the additional request or he is charged to pay a penalty as stated by the contract he signed previously. The two software providers can incur a violation as well if they do not send the software updates as stated in the contract. In this case they are charged a penalty. At every update the client C makes a payment to PSP of just part of the total price for the whole software. Payments to PS are performed by PSP. Agent PSP integrates all the components and sends the final product to the testing agent T. If the testing fails the product is sent back to PSP and PS and revised. The revision can be performed twice. If the testing fails three times then the client cancels the contract with PSP. If the test succeeds, the client asks the agent I to issue an insurance to guarantee the correctness of the software. Then C asks H to provide the hardware and E to get the software installed. At this point, if the software cannot be installed the components are revised twice. As before, if the test fails three times then C is allowed to cancel the contract with PSP and H.
Some obligations for $C$ and $PSP$ are showed in Figure 4.8. The part of the contract describing violation conditions and corresponding penalties is shown in Figure 4.10. The expression “no” in the right column of Figure 4.10 means that the contract was broken by the corresponding agent.

In order to distinguish between contractually correct and incorrect behaviours, in [LQS08b, LQS08a] the authors use the labels “green” and “red” in ISPL code to label global states (see Figure 4.9). Labelling of states is performed in the following way.

- All initial states are labelled as green states.

- A state is labelled with label “green” if the transition that drove the system in that state is one allowed by the contract. Green states are listed in the Evaluation section of the ISPL code.

- All the other states that are not “green” are labelled as “red”. Red states are listed in the Evaluation section of the ISPL code as well.

The $\text{ACTLK}$-formulas checked over this system are (we remind that the symbol “!” represents the negation $\neg$ in the ISPL):

$$\text{AG} \ (\text{PSP\textunderscore Green} \ ! \ K(\text{PSP, AF (PSP\textunderscore End)}));$$
$$\text{K} \ (\text{PSP, A(All\textunderscore Green U Software\textunderscore Delivered)});$$
$$\text{A} \ (\text{HardwareSupplier\textunderscore green U HardwareSupplier\textunderscore end });$$
$$\text{A} \ (\text{HardwareSupplier\textunderscore red U HardwareSupplier\textunderscore end });$$
$$\text{A} \ (\text{TestingCompany\textunderscore green U TestingCompany\textunderscore end });$$
<table>
<thead>
<tr>
<th></th>
<th>PSP</th>
<th>- does not send messages to SP and/or C in the first and/or second run of update.</th>
<th>pay penalty charge</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td>- does not send payment to SP.</td>
<td>no</td>
</tr>
<tr>
<td>3</td>
<td>SP</td>
<td>- does not send update messages to PSP or C.</td>
<td>pay penalty charge</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>- does not send its components to PSP.</td>
<td>no</td>
</tr>
<tr>
<td>5</td>
<td>C</td>
<td>- request changes after second update.</td>
<td>pay penalty charge or withdraw changes</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>- does not send the payment to PSP.</td>
<td>no</td>
</tr>
<tr>
<td>7</td>
<td>T</td>
<td>- does not send the testing report to C, PSP and/or SP.</td>
<td>no</td>
</tr>
<tr>
<td>8</td>
<td>H</td>
<td>- does not deliver the hardware system to C.</td>
<td>no</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>- ignores the deployment failure.</td>
<td>no</td>
</tr>
<tr>
<td>10</td>
<td>E</td>
<td>- does not deploy the software on the hardware system.</td>
<td>no</td>
</tr>
<tr>
<td>11</td>
<td>I</td>
<td>- does not process the claim of C.</td>
<td>no</td>
</tr>
</tbody>
</table>

Figure 4.10: Contractual parties and their violation conditions, from [LQS08b].
Chapter 4. Automatic data-abstraction for MAS

A ( TestingCompany_red U TestingCompany_end );
A ( Expert_green U Expert_end );
A ( Expert_red U Expert_end );
A ( InsuranceCompany_green U InsuranceCompany_end );
A ( InsuranceCompany_red U InsuranceCompany_end );
A ( Client_green U Client_end );
A ( Client_red U Client_end );
A ( ServiceProvider_green U ServiceProvider_end );
A ( ServiceProvider_red U ServiceProvider_end );
A ( PSP_green U PSP_end );
A ( PSP_red U PSP_end ).

The first formula says “whenever the PSP is in compliance (i.e., it is in a green state) it knows that the contract will eventually be successfully fulfilled”. The second formula says “PSP knows that all other parties are in compliance with the contract until the software is delivered”. The third formula says “the hardware supplier is in compliance with the contract until he terminates”. All the other formulas have either a clear or an analogous meaning. We expect that all these formulas are satisfied.

4.6.2 Experimental results

MCMAS spent 21 seconds to check the formulas above on the concrete ISPL. The memory consumption was 14,4 MB. The abstraction process was driven by the formulas above and created an abstract system that took only 2 seconds (including the abstraction process) to be checked by MCMAS. The BDD memory was just 6,2 MB for the abstract ISPL.

We obtained such a drastic reduction in time and memory since the abstraction process collapsed together all red states and green states of every agent. This means that in the abstract system we cannot distinguish between either two green states or red states. We remind that green states represent those behaviours that are in compliance with the contract, while red states represent those ones that do not respect the contract. Therefore, in the abstract system we can only check properties that refer to red states or green states without mentioning which
particular green or red states are. For instance, in the Evaluation section of the concrete software development ISPL program the atomic formula $\text{PSP\_green}$ is defined as follows:

$$\begin{align*}
\text{PSP\_green} \text{ if } \text{PSP\_state} &= \text{PSP\_0 or PSP\_state} = \text{PSP\_1 or} \\
& \quad \text{PSP\_state} = \text{PSP\_2 or PSP\_state} = \text{PSP\_3 or} \\
& \quad \text{PSP\_state} = \text{PSP\_4 or PSP\_state} = \text{PSP\_5 or} \\
& \quad \ldots \\
& \quad \text{PSP\_state} = \text{PSP\_53 or PSP\_state} = \text{PSP\_54;} \\
\end{align*}$$

$$\text{PSP\_red} \text{ if } \text{PSP\_state} = \text{PSP\_55 or PSP\_state} = \text{PSP\_56;}$$

where every $\text{PSP\_i}$ (for $0 \leq i \leq 54$) represents a particular behaviour that is in compliance with the contract. For instance, $\text{PSP\_1}$ says “PSP sends payment on time to SP”, while $\text{PSP\_2}$ says “PSP sends messages to SP and C in the first round”. On the other hand, in the definition of atomic formula $\text{PSP\_red}$ above, the two logic expressions $\text{PSP\_55}$ and $\text{PSP\_56}$ represent violations 1 and 2 reported in Figure 4.10, respectively.

The procedure automatically transforms the type of variable $\text{PSP\_state}$ from string of chars to integer type. Therefore, the value $\text{PSP\_0}$ is transformed to the integer 0, $\text{PSP\_1}$ to 1 and so on. We do this since the abstraction process operates on integers. Finally, the Boolean expression above that defines the atomic formula $\text{PSP\_green}$ is transformed to:

$$\begin{align*}
\text{PSP\_green} \text{ if } \text{PSP\_state} &= <54; \\
\text{PSP\_red} \text{ if } \text{PSP\_state} &= >55; \\
\end{align*}$$

since all green states range from the value 0 to 54. The abstraction procedure assigned index 0 to the logic expression $\text{PSP\_state}=<54$ and index 1 to the logic expression $\text{PSP\_state}>=55$ according to Algorithm 10. Algorithm 10 (see line 9) collapsed all green states into the value $2^0 = 1$, while all red states are collapsed into $2^1 = 2$. Therefore, the atomic formulas above were automatically transformed to the following ones in the abstract ISPL:

$$\text{PSP\_green} \text{ if } \text{PSP\_state} = 1;$$
PSP\_red if PSP\_state=2;

The same abstraction was performed for all the other agents. All green states and all red states were collapsed together in the same fashion for the software provider $SP$, the testing agency $T$, the hardware supplier $H$, the software client $SC$, the insurance company $I$ and the technical expert $E$.

The reduction obtained with this abstraction was drastic, especially in verification time. However, the price paid to obtain this reduction consists in the impossibility to distinguish between two different green states and two different red states in the abstract system. Therefore, this abstraction is useful if we want to check properties that express the compliance or not with the contract, but it is useless if we want to know which part of the contract was not respected. For instance, it would be impossible to know in the abstract system whether the agent PSP incurred into violation 1 rather than 2 (see Figure 4.10). The abstraction process took less than a second to be performed.

In this Chapter we presented a methodology to build quotient interpreted systems by partitioning the data domains of each variable defined in the original interpreted system. We proved Theorem 4.1 to show the correctness of this technique. Since this theorem applies to whatever quotient system regardless of the particular partitioning method used, it can be use as theoretical basis for variable-abstraction technique, presented in the next Chapter, as well.
Chapter 5

Automatic variable-abstraction for MAS

5.1 Variable-abstraction algorithm

In the previous chapter existential data-abstraction was performed by automatically computing a set of abstraction functions $\rho_1, \ldots, \rho_k$ that collapse the values of agent variables. The results produced in a number of examples were attractive both in terms of state space reduction and time reduction. However, there are cases for which no reduction is achieved. This phenomenon especially occurs when there are no atomic formulas in the Evaluation section that permit the “collapse” of variable values.

For instance, let $\text{Agent1.c}$ be a variable defined over the set of integers $D_c = \{1, 2, 3, 4\}$, and consider an Evaluation section that defines the following set of atomic propositions

\[ AP = \{ \text{atomicormula}_1, \text{atomicormula}_2, \text{atomicormula}_3, \text{atomicormula}_4 \} \]

where each atomic formula depends on a single value, and there are defined as many atomic formulas in Evaluation section as values in $D_c$, e.g.,

\[
\begin{align*}
\text{Evaluation} \\
\text{atomicormula}_1 & \text{ if } \text{Agent1.c} = 1; \\
\text{atomicormula}_2 & \text{ if } \text{Agent1.c} = 2;
\end{align*}
\]
atomic_formula_3 if Agent1.c = 3;
atomic_formula_4 if Agent1.c = 4;
End Evaluation

Clearly, using the methodology of the previous Chapter, no reduction of the state space can be obtained with data-abstraction (at least as far as domain $D_c$ is considered) since there are no logic expressions that “overlap”. This means there is no value in $D_c$ verifying more than one logic expression.

$$1 \models \text{Agent1.c}=1,$$
$$2 \models \text{Agent1.c}=2,$$
$$3 \models \text{Agent1.c}=3,$$
$$4 \models \text{Agent1.c}=4.$$  

If we apply data abstraction to this particular case then we obtain a new domain $D'_c = \{1, 2, 4, 8\}$ that has four abstract values (i.e. $|D'_c| = |D_c| = 4$). Therefore, no values are collapsed, and consequently no reduction is obtained.

Moreover, there are systems described via ISPL programs in which the atomic formulas are defined via variables that interact with each other. In other words, in the Evaluation section of the ISPL program there is at least one logic expression in which two variables appear. In order for us to use abstraction for these systems as well, a different methodology is required. The key observation is that where concrete states are expressed by means of several variable domains, instead of collapsing values of variables, it might be more useful and effective to collapse multiple variables into one.

Specifically, if an atomic formula Pair1 is defined in terms of the equality of two variables belonging to the same agent (in this case the Environment), as follows:

$$\text{Pair1 if Environment.c1} = \text{Environment.c2};$$

then a different type of existential abstraction can be applied. This technique is called variable abstraction. In this particular case, variables Environment.c1 and Environment.c2 are collapsed into a single abstract variable.
Variable abstraction algorithm collects all the logic expressions that appear in the definition of atomic formulas. The set of logic expressions is called $LE$. If there is a logic expression in $LE$ that contains two or more variables belonging to different agents then this expression is rewritten in a semantic equivalent Boolean expression in which there is one variable for each logic expression.

For instance, if we have $LE = \{ a_1 = a_2, a_1 + b_1 = 0, a_2 = 1, c_1 = 1, d_1 > e_1 \}$ where $a_1, b_1, c_1, d_1, e_1$ belong to agent 1, $a_2$ belongs to agent 2 and all variables have domain $\{0, 1\}$, then formula $a_1 = a_2$ becomes $(a_1 = 0 \land a_2 = 0) \lor (a_1 = 1 \land a_2 = 1)$ and the new set of logic expressions will be $LE = \{ a_1 = 0, a_2 = 0, a_1 = 1, a_2 = 1, a_1 + b_1 = 0, b_1 > c_1, a_2 = 1, d_1 > e_1 \}$.

From this set, the algorithm builds the two following sets:

$LE_1 = \{ a_1 = 0, a_1 = 1, a_1 + b_1 = 0, c_1 = 1, d_1 > e_1 \}$;

$LE_2 = \{ a_2 = 0, a_2 = 1, a_2 = 1 \}$.

This operation is done in order to separate variables belonging to different agents, since we can only collapse variables belonging to the same agent otherwise the epistemic specifications that are valid in the abstract system do not hold in the concrete system.

In the next step, from $LE_1$ and $LE_2$ we obtain the following sets of formula clusters:

$FC_1 = \{ FC_{11}, FC_{12}, FC_{13} \}$,

$FC_2 = \{ FC_{21} \}$.

Where formula clusters $FC_{11}, FC_{12}, FC_{13}$ are defined as follows:

$FC_{11} = \{ a_1 = 0, a_1 = 1, a_1 + b_1 = 0 \}$,

$FC_{12} = \{ c_1 = 1 \}$,

$FC_{13} = \{ d_1 > e_1 \}$,

$FC_{21} = \{ a_2 = 0, a_2 = 1, a_2 = 1 \}$.

Then the variable abstraction algorithm builds the sets of variable clusters as follows:

$VAR'_1 = \{ \{ a_1, b_1 \}, \{ c_1 \}, \{ d_1, e_1 \} \}$;

$VAR'_2 = \{ \{ a_2 \} \}$.

From $VAR'_1$ and $VAR'_2$, we obtain:

$VAR''_1 = \{ \hat{a}_1, \hat{c}_1, \hat{d}_1 \}$,
$\text{VAR}_2'' = \{ \hat{a}_2 \}$
where $\hat{a}_1$ was built from the cluster $\{ a_1, b_1 \}$, $\hat{d}_1$ from $\{ d_1, e_1 \}$, $\hat{c}_1$ from $\{ e_1 \}$ and $\hat{a}_2$ from $\{ a_2 \}$.

Finally for the concrete variables $a_1, b_1$ the algorithm builds the corresponding Cartesian product $\text{CP}_{\hat{a}_1} = D_{a_1} \times D_{b_1}$.

### 5.1.1 Definition of the abstraction function

In the following we extend the results presented in [CGJ+03] from mono-dimensional Kripke structures to interpreted systems.

Let $P$ be an ISPL program describing an interpreted system $\mathcal{I} = \langle \{ L_i \}_{i \in Ag} \cup \{ L_e \}, \{ ACT_i \}_{i \in Ag} \cup \{ ACT_e \}, \{ P_i \}_{i \in Ag} \cup \{ P_e \}, \{ t_i \}_{i \in Ag} \cup \{ t_e \}, I_0, V \rangle$ over a set $\text{AP}$ of atomic propositions. Let $\text{VAR} = \{ v_1, \ldots, v_n \}$ be the set of variables defined in the ISPL program $P$.

**Definition 5.1** ($\text{var}(p)$).

Given a logic expression $p$, $\text{var}(p)$ is the set of variables appearing in $p$.

For instance, for the logic expression $p : x = z + 1$, $\text{var}(p) = \{ x, z \}$.

The function $\text{var}$ can be extended to a set of logic expression $\text{LE}$ as follows:

$$\text{var}(\text{LE}) = \bigcup_{p \in \text{LE}} \text{var}(p).$$

It is possible to define an equivalence relation on the set of atomic propositions $\text{AP}$, called the *proposition interference relation*.

**Definition 5.2** (proposition interference relation $\equiv_{\text{prop}}$). [CGJ+03]

Given $p_1, p_2 \in \text{LE}$, let $\equiv_{\text{prop}}$ denote the proposition interference relation defined as follows:

$$p_1 \equiv_{\text{prop}} p_2 \iff \text{var}(p_1) \cap \text{var}(p_2) \neq \emptyset.$$ 

The transitive closure $\equiv^*_{\text{prop}}$ of relation $\equiv_{\text{prop}}$ induces several equivalence classes on the set $\text{LE}$, denoted by $[p]$. 

By definition, given two logic expressions \( p_1, p_2 \in LE \), we have
\[
\text{var}(p_1) \cap \text{var}(p_2) \neq \emptyset \implies [p_1] = [p_2].
\]

The equivalence class \([p]\) is also called formula cluster. These clusters induce several equivalence relations on the set of variables of an agent \( i \in Ag \).

**Definition 5.3** (variable equivalence relation \( \equiv_{\text{var}} \)). \([CGJ+03]\)

Given variables \( v_1 \) and \( v_2 \in VAR \), let \( \equiv_{\text{var}} \) denote the variable equivalence relation defined as follows:
\[
v_1 \equiv_{\text{var}} v_2 \iff \exists p_1, p_2 \in AP \text{ such that } v_1 \in \text{var}(p_1) \land v_2 \in \text{var}(p_2) \land \text{var}(p_1) \cap \text{var}(p_2) \neq \emptyset.
\]

Relation \( \equiv_{\text{var}} \) induces equivalence classes \( VC_i \) on the set of variables \( VAR = \{ v_1, \ldots, v_n \} \), called variable clusters. The variable-abstraction procedure, presented in this Chapter, partitions the set \( VAR = \{ v_1, \ldots, v_n \} \) of variables into variable clusters \( VC_1, \ldots, VC_m \) (with \( m \leq n \)). Each variable cluster \( VC_i \) can be seen as a variable with domain defined as follows:
\[
D_{VC_i} = \prod_{v \in VC_i} D_v.
\] (5.1)

Instead of building a set of abstraction functions \( \rho_1, \ldots, \rho_k \) on the domains \( D_1, \ldots, D_k \) of the variables \( v_1, \ldots, v_k \), we could compute a set of surjections on the domains \( D_{VC_i} \) (with \( i \in \{ 1, \ldots, m \} \)).

Let \( \{ [p_1], \ldots, [p_m] \} \) be a set of formula clusters and \( \{ VC_1, \ldots, VC_m \} \) the set of corresponding variable clusters. It is possible to rewrite Definition 4.12 for formula clusters. In this case each \( \rho_i \) is defined on the set \( D_{VC_i} \) defined in (5.1).

Given two tuples \( d = (d_1, \ldots, d_k) \) and \( e = (e_1, \ldots, e_k) \in D_{VC_i} \), the component abstraction functions \( \rho_i \) are defined in the following way \([CGJ+03]\):
\[
\rho_i(d_1, \ldots, d_k) = \rho_i(e_1, \ldots, e_k) \iff \bigwedge_{p \in [p]} (d_1, \ldots, d_k) \models p \iff (e_1, \ldots, e_k) \models p.
\] (5.2)

Informally, \( (d_1, \ldots, d_k) \) and \( (e_1, \ldots, e_k) \) are “collapsed” together by \( \rho_i \) if they cannot be distinguished by the same set of formula cluster \([p]\).
As in the data-abstraction, each logic expression gets an integer \( l \in \mathbb{N} \) in order to be identified. If a value \( e \) of the concrete interpreted system satisfies the logic expression identified by \( l \) then \( l \) is added to a set of integers \( ID_e \). Therefore, \( l \in ID_e \) if and only if \( e \) satisfies the logic expression with index \( l \).

Given an agent \( k \in Ag \), the variable abstraction function \( \rho_k \) is defined in the same way of the data-abstraction function. \[ \forall e = (e_1, \ldots, e_n) \in D_{VC_i} \]

\[ \rho_k(e) = \sum_{l \in ID_e} 2^l \] (5.3)

Definition 5.3 is a generalization of definition 4.14. Therefore, the variable abstraction technique presented in this chapter can be seen as an extension of the data abstraction technique presented in Chapter 4.

For example, let \( FC_{11} = \{ a < b, a = b, b = 2, c = \text{blue}, c = \text{red} \} \) be the set of logic expressions that contain all and only variables for agent 1. Let \( VAR'_1 = \{ \{a, b\}, \{c\} \} \) be the set of variable clusters. Variables \( a, b \) are collapsed into an abstract variable \( \hat{v} \), and each pair of the Cartesian product is identified by an integer \( t \) as follows:

\[ 1' = (0, 0), \ 2' = (0, 1), \ 3' = (1, 0), \ 4' = (0, 2), \ 5' = (2, 0), \ 6' = (1, 1), \ 7' = (1, 2), \ 8' = (2, 1), \ 9' = (2, 2). \]

Subsequently, these new values are partitioned. In the matrix in Figure 5.1 the results of all checks for all pairs against all logic expressions are shown. The letters F and T mean False and True, respectively.

From Figure 5.1 it is possible to notice that:

\[ (1, 0), (2, 0), (2, 1) \models \emptyset, \]
\[ (0, 0), (1, 1) \models \{a = b\}, \]
\[ (0, 2), (1, 2) \models \{a < b, b = 2\}, \]
\[ (0, 1) \models \{a < b\}, \]
\[ (2, 2) \models \{a = b, b = 2\}. \]

Pairs satisfying the same subset of logic expressions are “collapsed”. That means these pairs will be replaced by the same abstract value. For instance, the pairs \((1, 0)\), \((2, 0)\) and \((2, 1)\) are replaced by a unique integer.
5.2 Implementation

The variable abstraction technique just presented was implemented in C++ programming language and integrated with MCMAS. The tool for variable abstraction takes a concrete ISPL

<table>
<thead>
<tr>
<th></th>
<th>(0,0)</th>
<th>(0,1)</th>
<th>(1,0)</th>
<th>(0,2)</th>
<th>(2,0)</th>
<th>(1,1)</th>
<th>(1,2)</th>
<th>(2,1)</th>
<th>(2,2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &lt; b</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>a = b</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>2^1</td>
</tr>
<tr>
<td>b = 2</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>2^2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>0</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.1: The automatic calculations of new values from the concrete ones.

Numbers $2^0$, $2^1$, $2^2$ in the rightmost column of the matrix in Figure 5.1 represent the weights associated to logic expressions $a < b$, $a = b$, $b = 2$, respectively. If a value satisfies a logic expression then the corresponding weight is added, otherwise it is skipped.

New values are calculated by the sum of all weights. For instance, the pairs $4' = (0, 2)$ and $7' = (1, 2)$ are replaced by the integer $5 = 2^0 + 2^2$ since they both satisfy logic expressions $a < b$ and $b = 2$. In the same way, values $3'$, $5'$, $8'$ are replaced by $0$ since they do not satisfy any logic expression. Finally, values $1'$, $6'$ are replaced by $2 = 2^1$. Hence, the domain of the abstract variable $\hat{v}$ is $D_\hat{v} = \{0, 1, 2, 5, 6\}$, where

$$
\begin{align*}
0 &= \{3', 5', 8'\} = \{(1,0), (2,0), (2,1)\}, \\
1 &= \{2'\} = \{(0,1)\}, \\
2 &= \{1', 6'\} = \{(0,0), (1,1)\}, \\
5 &= \{4', 7'\} = \{(0,2), (1,2)\}, \\
6 &= \{9'\} = \{(2,2)\}.
\end{align*}
$$

These abstract values (in bold font) are reported in the last row of the matrix in Figure 5.1.

The definitions of abstract global states and abstract interpreted system are the same as Definition 4.1 and Definition 4.2 respectively.
program as input and returns an abstract ISPL program as output. This tool automatically selects the variables to be collapsed according to Definition 5.3.

The procedure that performs variable abstraction can be subdivided into six main steps.

1 - Building the set of logic expressions \( LE_i \): Algorithm [11]

Algorithm 11 Building the set \( LE_i \).

1: for all \( i \in Ag \) do
2: \( LE_i \leftarrow \emptyset \);
3: for all \( p \in AP \) do
4: for all logic expressions \( l \) in the Boolean expression defining \( p \) do
5: if \( \text{var}(l) \in i \land \text{var}(l) \notin \text{Obsvars} \land \text{var}(l) \notin K \) then
6: \( LE_i \leftarrow LE_i \cup \{l\} \);
7: end if
8: end for
9: end for
10: end for

First of all, the algorithm builds the set of all logic expressions that appear in the Evaluation section of the ISPL-file under examination. Let us call this set \( LE \). From \( LE \) the algorithm builds a set \( LE_i \) of logic expressions for each agent containing all and only logic expressions made of its local variables. If a logic expression contains two or more variables belonging to two or more different agents then that logic expression will be removed from \( LE \) and it will be transformed to an equivalent Boolean expression in which each logic expression will contain just one variable. Then, Algorithm 11 operates on \( LE \) and builds a set of logic expression \( LE_i \) for each agent \( i \) that contains all and only variables belonging to agent \( i \). \( K \) is the set of logic expressions in which a variable is updated by an arithmetic expression.

2 - Building the set of formula clusters for each agent \( i \): Algorithm [12]

In the second step Algorithm 12 defines several sets \( FC_i \) of formula clusters. In particular, Algorithm 12 builds one set \( FC_i = \{ FC_{i1}, \ldots, FC_{in} \} \) for each set \( LE_i \). For all logic expressions \( p \in LE_i \) (see lines 4 and 5) Algorithm 12 checks if there is at least one \( FC_{ij} \) built so far such that \( FC_{ij} \) contains a logic expression that interferes (in the sense of Definition 5.2) with the current expression \( p \). Function \textbf{find} at line 5 tries to find such a \( FC_{ij} \), identified by the index.
Algorithm 12 Building the set of formula clusters for each agent $i$.

1: for all $i \in Ag$ do
2: $FC_i \leftarrow \emptyset$;
3: $FC_{ij} \leftarrow \emptyset$;
4: for all $p \in LE_i$ do
5: find $j : var(FC_{ij}) \cap var(p) \neq \emptyset$;
6: $FC_{ij} \leftarrow FC_{ij} \cup \{p\}$;
7: if $j$ not found then
8: $FC_i \leftarrow FC_i \cup \{p\}$; – the singleton $\{p\}$ is the new $FC_{ij}$
9: end if
10: end for
11: end for

3 - Building the set of variable clusters for each agent $i$: Algorithm 13

In the third step Algorithm 13 builds a set $VAR_i'$ of variable clusters $[v]_{ij}$ for each agent $i$. Variable clusters are built from sets $FC_{ij}$ according to variable equivalence relation $\equiv_{var}$ introduced in Definition 5.3.

Algorithm 13 Building the set of variable clusters $VAR_i'$ for each agent $i$.

1: for all $i \in Ag$ do
2: $VAR_i' \leftarrow \emptyset$;
3: for all $FC_{ij} \in FC_i$ do
4: $VAR_i' \leftarrow VAR_i' \cup \{var(FC_{ij})\}$;
5: end for
6: end for

4 - Building the set of abstract variables: Algorithm 14

For each agent $i \in Ag$ and for each variable cluster $[v]_{ij} \in VAR_i'$, Algorithm 14 builds an abstract variable $\hat{v}_{ij}$ (see line 4). Function create at line 4 defines a new variable $\hat{v}_{ij}$ and a set $CP_{\hat{v}_{ij}}$ that is the Cartesian product of the old domains of those variables that were “collapsed”

5.2. Implementation
Algorithm 14 Building the set of abstract variables $VAR_i''$.

1: for all $i \in Ag$ do
2: \hspace{1em} $VAR_i'' \leftarrow \emptyset$;
3: \hspace{1em} for all $[v]_{ij} \in VAR_i'$ do
4: \hspace{2em} create a new variable $\hat{v}_{ij}$;
5: \hspace{1em} $VAR_i'' \leftarrow VAR_i'' \cup \{\hat{v}_{ij}\}$;
6: \hspace{1em} end for
7: end for

5 - The generation of the Boolean vector $Q(i, \hat{v}, p, t)$: Algorithm 15

Algorithm 15 The generation of the Boolean four-dimension vector $Q(i, \hat{v}, p, t)$.

1: $Q(i, \hat{v}, p, t) \leftarrow 0$;
2: for all $i \in Ag$ do
3: \hspace{1em} for all $\hat{v}$ of agent $i$ do
4: \hspace{2em} for all $p \in LE_i$ do
5: \hspace{3em} for all $t \in CP_{\hat{v}}$ do
6: \hspace{4em} if $t \models p$ then
7: \hspace{5em} $Q(i, \hat{v}, p, t) \leftarrow True$;
8: \hspace{4em} else
9: \hspace{5em} $Q(i, \hat{v}, p, t) \leftarrow False$;
10: \hspace{4em} end if
11: \hspace{3em} end for
12: \hspace{2em} end for
13: \hspace{1em} end for
14: end for

Algorithm 15 builds a four-dimensional vector $Q(i, \hat{v}, p, t)$ in order to store the results of the checks (see lines 7, 9) for all tuples $t$ belonging to $CP_{\hat{v}}$ for all abstract variables $\hat{v}$ and for all agents $i$ of the system.

6 - Building abstract domains: Algorithm 16

Each tuple belonging to an abstract domain is represented by a single integer. These integers
Algorithm 16 The generation of new domains $D'_{\hat{v}_{ij}}$ for the abstract ISPL-file.

1: for all $i \in Ag$ do
2:     for all $\hat{v}_{ij} \in VAR_i^{\mu}$ do
3:         $D'_{\hat{v}_{ij}} \leftarrow \emptyset$;
4:         for all tuple $t \in CP_{\hat{v}_{ij}}$ do
5:             $indx_p \leftarrow 0$;
6:             $newval \leftarrow 0$;
7:             for all $p \in LE_i$ do
8:                 if $Q(i, \hat{v}, p, t) = true$ then
9:                     $newval \leftarrow newval + 2^{indx_p}$;
10:                end if
11:             $indx_p \leftarrow indx_p + 1$;
12:            end for
13:         $D'_{\hat{v}_{ij}} \leftarrow D'_{\hat{v}_{ij}} \cup \{newval\}$;
14:     end for
15: end for
represent the abstract values of the abstract variable. Domain $D'_{\hat{v}_{ij}}$ of abstract variable $\hat{v}_{ij}$ is built from a partition over the set $CP_{\hat{v}_{ij}}$.

Finally, this is linked to a module that modifies all logic expressions containing the variables involved in the abstraction process. As discussed above these expressions are replaced by new Boolean expressions in the original ISPL-file. For instance, following the example above, the logic expression $a = b$ will be replaced by the Boolean expression $x = 2 \lor x = 6$, where $x$ is the abstract variable created by collapsing concrete variables $a$ and $b$ and the abstract value $2$ represents the two pairs: $(0,0), (1,1)$, while $6$ represents the single pair $(2,2)$. A release of this toolkit is available from [Rus].

### 5.3 Evaluation

In this section the variable-abstraction technique just presented will be evaluated by means of two examples.

#### 5.3.1 Simplified Black-jack

In the previous chapter a card game was used to test the efficiency of data-abstraction. This game cannot be used to test the variable-abstraction technique as there are no variables that interact with each other. In fact, the atomic propositions defined in the Evaluation section for this scenario depend on the values of one variable only. Therefore, no reduction can be obtained with the technique presented in this Chapter. In the Black-jack simplified game presented in this section, the atomic propositions depend on certain combinations of values of variables, e.g., the sum of two variables.

Simplified Black-jack. The system has three agents: Dealer, Player and an environment $e$. There is a deck of 20 cards. Dealer and Player receive 2 cards each. The players cannot change any card they get. The sum of the two cards represents the score an agent gets. The maximum score allowed by game rules is 12. If an agent gets more than 12 then he loses. If neither player exceeds 12, then the player with the higher score wins. If the two agents get either the same score (so we end up with a draw) or they both exceed 12, then Dealer wins.
The specification checked was

\[ \text{AG}(\text{BestD} \rightarrow \text{K}_d(\text{AF WinD})). \]

This formula states that if the Dealer receives the best combination of cards (whose sum is 12) then he knows he will eventually win the game.

We model the simplified black-jack game with an interpreted system involving ten-card deck. The definition can be easily extended for any number of cards. We consider a family of games where the number of cards of the deck varies.

Let \( P \) be an ISPL program describing an interpreted system \( I = \langle \{L_i\}_{i \in Ag} \cup \{L_e\}, \{ACT_i\}_{i \in Ag} \cup \{ACT_e\}, \{P_i\}_{i \in Ag} \cup \{P_e\}, \{t_i\}_{i \in Ag} \cup \{t_e\}, I_0, V \rangle \) over a set \( AP \) of atomic propositions, defined in the Evaluation section of the ISPL \( P \). Let \( Ag = \{1, 2, e\} \) be the set of agents. Let \( C = \{1, \ldots, 10\} \) represent the deck of cards. An agent \( i \in \{d, p\} \) can either play or read the cards he holds:

\[ ACT_i = \{\text{play, read}\}. \]

The environment either calculates who wins the current round or does nothing:

\[ ACT_e = \{\text{eval, nothing}\}. \]

The local state of an agent describes what cards he holds:

\[ L_i \subseteq C^2 \times \{\text{play, read}\}. \]

Hence, the elements of \( L_i \) are pairs \( ((c_{i1}, c_{i2}), k) \) where \( (c_{i1}, c_{i2}) \) represents the cards held by agent \( i \) and \( k \) represents the status of a player that can be either reading his own cards or playing.

The environment determines who wins the game,

\[ L_e \subseteq \{\text{und, d, p}\} \times C^4. \]

So, the elements of \( L_e \) are 5-tuples \( (\text{win}, c^d_{11}, c^d_{12}, c^p_{11}, c^p_{12}) \) where \( \text{win} = d \) means “Dealer won”, \( \text{win} = p \) means “Player won” and \( \text{win} = \text{und} \) means “undetermined”, and \( c^d_{11}, c^d_{12}, c^p_{11}, c^p_{12} \) are the cards of the players. The set of global states \( W \) is defined as \( W = L_p \times L_d \times L_e \). The local protocols are defined as follows.

\[
\begin{align*}
P_i(c_{i1}, c_{i2}, k_i) &= \{\text{play}\}, \text{ if } k_i = \text{play}; \\
P_i(c_{i1}, c_{i2}, k_i) &= \{\text{read}\}, \text{ if } k_i = \text{read}; \\
P_e(\text{win}, c^d_{11}, c^d_{12}, c^p_{11}, c^p_{12}) &= \{\text{eval}\}, \text{ if } \text{win} = \text{und}; \\
P_e(\text{win}, c^d_{11}, c^d_{12}, c^p_{11}, c^p_{12}) &= \{\text{nothing}\}, \text{ if } \text{win} = d \text{ or } \text{win} = p.
\end{align*}
\]
The local evolution functions have the form:

\[ t_i(c^1_i, c^2_i, \text{read}, (\text{play}, \text{play}, \text{eval})) = (c^1_i, c^2_i, \text{play}); \]
\[ t_e(\text{und}, c^d_1, c^d_2, c^p_1, c^p_2, (\text{play}, \text{play}, \text{eval})) = (d, c^d_1, c^d_2, c^p_1, c^p_2) \text{ if } (c^d_1 + c^d_2 \leq c^d_1 + c^d_2) \land (c^p_1 + c^p_2 \leq 12) \land \right) \]
\[ t_e(\text{und}, c^d_1, c^d_2, c^p_1, c^p_2, (\text{play}, \text{play}, \text{eval})) = (p, c^d_1, c^d_2, c^p_1, c^p_2) \text{ if } (c^d_1 + c^d_2 > 12) \land (c^d_1 + c^d_2 < 12) \lor (c^d_1 + c^d_2 > 12) \land (c^d_1 + c^d_2 \leq 12). \]

In the first transition the player \( i \) starts playing and move from state \text{read} to state \text{play}. In the second transition the environment local states moves towards a state in which the dealer wins since either the dealer has a higher score than the player or they both have a score higher than 12 (that is the maximum score allowed for \(|C| = 10\)). In the second transition the environment local states moves towards a state in which the player wins since either the player has a higher score than the dealer or the dealer has a not allowed score (higher than 12 for \(|C| = 10\)).

The set of initial states is:

\[ I_0 = \{ \text{win}, c^d_1, c^p_1, k_1, k_2 \} | \text{win} = \text{und} \land | \mathcal{H}_1 | = | \mathcal{H}_2 | = 2 \land k_1 = k_2 = \text{read} \}. \]

Let us define the set of atomic propositions \( AP \), as follows:

\[ AP = \{ \text{Best}_1, \text{Best}_2, \text{Win}_d, \text{Win}_e \}. \]

Atomic proposition \( \text{Best}_i \) can be read as “Agent \( i \) got the best score”, while \( \text{Win}_i \) can be read as “Agent \( i \) has won the game”. Let us define evaluation function \( V: W \rightarrow 2^{AP} \) as follows:

\[ \text{Best}_i \in V(\text{win}, c^d_1, c^d_2, c^p_1, c^p_2, k_1, k_2) \iff c^d_1 + c^d_2 = 12; \]
\[ \text{Win}_d \in V(\text{win}, c^d_1, c^d_2, c^p_1, c^p_2, k_1, k_2) \iff \text{win} = d; \]
\[ \text{Win}_e \in V(\text{win}, c^d_1, c^d_2, c^p_1, c^p_2, k_1, k_2) \iff \text{win} = e. \]

Figure 5.2 describes the ISPL code for the concrete simplified Black-jack interpreted system for \(|C| = 10\). Figure 5.3 was automatically generated by the variable-abstraction toolkit and describes the corresponding abstract simplified Black-jack interpreted system shown in Figure 5.2.

Table 5.1 shows the reduction obtained by applying the tool for variable-abstraction for different ISPL programs with different numbers of deck cards. We reported the number of reachable
Figure 5.2: Sketch of an ISPL program for the simplified black-jack game for $|C| = 10$. 

**Agent Environment**

Obsvars:
- win: \{und, d, p\};

end Obsvars

Vars:
- c11: 1.. 10 ;
- c12: 1.. 10 ;
- c21: 1.. 10 ;
- c22: 1.. 10 ;

end Vars

Actions = \{ eval, nothing \};

Protocol:
- win=und: \{eval\};
- Other: \{nothing\};

end Protocol

Evolution:
- win=d if c21+c22 <= c11+c12 and ...
- win=p if c21+c22 > c11+c12 and ...

end Evolution

end Agent

**Agent Dealer**

Lobsvars={c11,c12};

Vars:
- n: \{read, play\};

end Vars

Actions = \{play, read\};

Protocol:
- n=read: \{read\};
- n=play: \{play\};

end Protocol

Evolution:
- Evaluation

BestD if c11+c12 = 12;
BestP if c21+c22 = 12;
WinD if win=d;
WinP if win=p;

end Evaluation

InitStates

Dealer.n=read and Player.n=read and Environment.win=und (...)

end InitStates

Formulae

AG( (BestD) -> K(Dealer, AF WinD) );

end Formulae
Agent Environment

Obsvars:
  win: \{und, d, p\};
end Obsvars

Vars:
  c1: 0 .. 4 ;
  c2: 0 .. 4 ;
end Vars

Actions = \{ eval, nothing \};

Protocol:
  win=und: \{eval\};
  Other: \{nothing\};
end Protocol

Evolution:
  c1=0 if c1 = 4 and...
  c2=0 if c2 = 4 and...
  ...
  win=d if (((c1<>4 or...
  win=p if (c2<>4 and...
end Evolution

end Agent

Agent Dealer

Lobsvars={c1};

Vars:
  n: \{read, play\};
end Vars

Actions = \{play, read\};

Protocol:
  n=read: \{read\};
  n=play: \{play\};
  ...
end Protocol

Evaluation

BestD if c1 = 3;
BestP if c2 = 3;
WinD if win=d;
WinP if win=p;
end Evaluation

InitStates

Player.n=read and Dealer.n=read and
Environment.win=0 ( ... end InitStates

Formulae

\(\text{AG( (BestD) \rightarrow K(Dealer, AF WinD) )}\);
end Formulae

Figure 5.3: Sketch of an ISPL program for the abstract simplified Black-jack game.
Table 5.1: Verification results for the simplified Black-jack.

| deck cards | Without reduction | | | With reduction | | |
|---|---|---|---|---|---|---|---|
| | States | Time (s) | BDD (MB) | States | Time (s) | BDD (MB) |
| 10 | 1024 | 0 | 4.79 | 60 | 0 | 4.62 |
| 40 | 144408 | 1 | 4.99 | 66 | 0 | 4.69 |
| 52 | 134041 | 1 | 5.61 | 67 | 0 | 4.71 |
| 68 | $2.17 \times 10^6$ | 3 | 6.21 | 99 | 0 | 4.67 |
| 100 | $2.14 \times 10^6$ | 13 | 6.33 | 291 | 1 | 4.75 |

global states, time needed for verification and abstraction process in seconds and BDD memory consumption expressed in MB for evaluation purposes. The Black-jack we modelled refers to the one shown in the first line in Table 5.1. The abstraction process took less than a second for all the experiments shown in Table 5.1. The number of cards in the deck was increased until the abstraction procedure took a reasonable time to run. For numbers of cards bigger than 100 the toolkit was running for more than two hours and the abstraction process was stopped before termination. We can notice the reduction is exponential in time and number of reachable global states, while the reduction in terms of BDD is linear. This phenomenon can be explained by the symbolic nature of MCMAS. During the abstraction process redundant BDDs can be introduced.

5.3.2 Simplified poker game

As a further evaluation exercise we run the toolkit, performing variable abstraction, on several variants (for different numbers of cards) of a simplified version of Poker, called simplified poker game. This game offers the possibility to apply variable-abstraction since its rules depend on certain combinations of cards, e.g., the equality of two variables representing two cards.

Simplified poker game. This system has three agents: Player1, Player2 and Environment. Both players receive a certain number of cards each (the number can be 2, 3, 4 or 5) from a deck of $4N$ cards, where $N \in \mathbb{N}$. There are $N$ cards for each “suit” or “category”. There are just two kind of “hands” or “situations”. The first hand is Pair. A player gets a Pair when he holds two cards of the same kind. The second one is Ace. This situation occurs when the player holds a
Chapter 5. Automatic variable-abstraction for MAS

A special card called ace, where $A \in \{1, N + 1, 2N + 1, 3N + 1\}$. Once both players receive cards, they show their hands. Pair beats Ace and the player with Ace wins if his opponent has neither Pair nor Ace. The game terminates with a draw if both players get the same hand.

The specifications checked on this system were:

$$AG(Pair_1 \rightarrow K_1(AF Win_1));$$ (5.4)

$$AG(Ace_1 \rightarrow K_1(AF Win_1));$$ (5.5)

$$AG(Ace_2 \land Pair_1 \rightarrow (AF Win_1)).$$ (5.6)

Formula (5.4) and formula (5.5) state that if Player1 is served with two cards of the same kind or at least one ace respectively, then he knows he will eventually win the game, while formula (5.6) means that if Player2 is served with an ace and Player1 is served with a pair then he eventually win the game. We expect that formulas (5.4) and (5.5) are false while formula (5.6) is true.

We model simplified poker game with an interpreted system $I = \langle \{L_i\}_{i \in Ag} \cup \{L_e\}, \{ACT_i\}_{i \in Ag} \cup \{ACT_e\}, \{P_i\}_{i \in Ag} \cup \{P_e\}, \{t_i\}_{i \in Ag} \cup \{t_e\}, I_0, V \rangle$ over a set $AP$ of atomic propositions.

Let $Ag = \{1, 2, e\}$ be the set of agents. Let $C = \{1, \ldots, 4N\}$ represent the deck of cards. Let red cards be the subset $\{1, \ldots, N\}$, yellow cards the subset $\{N + 1, \ldots, 2N\}$, green cards the subset $\{2N + 1, \ldots, 3N\}$ and black cards the remaining cards. A player $i \in \{1, 2\}$ can either play the cards she holds or do nothing:

$$ACT_i = \{\text{play, nothing}\}.$$ 

The environment either calculates who wins the current round or does nothing:

$$ACT_e = \{\text{eval, nothing}\}.$$ 

The local state of an agent describes what cards she holds:

$$L_i = \{(H_i, k_i) \in 2^C \times \{\text{read, played}\} : |H_i| = 5\}$$

where $H_i \subseteq C$ represents the cards held by the agent $i$ and $k_i$ represents the status of a player $i$ that can be either read or played. The environment determines who wins the game.

$$L_e \subseteq \{\text{und, p1, p2}\} \times 2^C \times 2^C.$$
Let $W = L_1 \times L_2 \times L_3$ the set of global states. Notice that the two players can get the same cards. Hence, the elements of $L_3$ are triples $(\text{win}, \mathcal{H}_1, \mathcal{H}_2)$ where $\text{win} = p1$ means “Player1 won”, $\text{win} = p2$ means “Player2 won” and $\text{win} = \text{und}$ means “undetermined”. The local protocols are defined as follows.

$$P_i(\mathcal{H}_i, k_i) = \{\text{play}\}, \text{if } k_i = \text{read};$$
$$P_i(\mathcal{H}_i, k_i) = \{\text{nothing}\}, \text{if } k_i = \text{played};$$
$$P_e(\text{win}, \mathcal{H}_1, \mathcal{H}_2) = \{\text{eval}\}, \text{if } \text{win} = \text{und};$$
$$P_e(\text{win}, \mathcal{H}_1, \mathcal{H}_2) = \{\text{nothing}\}, \text{if } \text{win} = p1 \text{ or } \text{win} = p2.$$

Let us define the set of atomic propositions $AP$, as follows:

$$AP = \{\text{Pair}_1, \text{Pair}_2, \text{Ace}_1, \text{Ace}_2, \text{Nothing}_1, \text{Nothing}_2\}$$

Before describing the local evolution functions, let us introduce the evaluation function $V : W \rightarrow 2^{AP}$,

$$Pair_i \in V(\text{win}, \mathcal{H}_1, \mathcal{H}_2, k_1, k_2) \iff \exists x, p \in \mathcal{H}_i : x = p + a, \text{ where } a \in \{N, 2N, 3N\};$$
$$\text{Ace}_i \in V(\text{win}, \mathcal{H}_1, \mathcal{H}_2, k_1, k_2) \iff \exists x \in \mathcal{H}_i : x = 1 \lor x = N + 1 \lor x = 2N + 1 \lor x = 3N + 1;$$
$$\text{Nothing}_i \in V(\text{win}, \mathcal{H}_1, \mathcal{H}_2, k_1, k_2) \iff (\text{win}, \mathcal{H}_1, \mathcal{H}_2, k_1, k_2) \neq \text{Pair}_1 \lor \text{Ace}_i.$$ 

The local evolution functions $t_i : W \times \text{ACT} \rightarrow 2^W$ are

$$t_i((\mathcal{H}_i, \text{read}), \langle\text{play, play, eval}\rangle) = (\mathcal{H}_i, \text{played});$$
$$t_i((\mathcal{H}_i, \text{played}), \langle\text{nothing, nothing, nothing}\rangle) = (\mathcal{H}_i, \text{played});$$
$$t_e(\text{und}, \mathcal{H}_1, \mathcal{H}_2, \langle\text{play, play, eval}\rangle) = (p1, \mathcal{H}_1, \mathcal{H}_2) \text{ iff } (\mathcal{H}_1 = \text{Pair}_1 \land \mathcal{H}_2 = \text{Ace}_2) \lor$$
$$\quad \quad (\mathcal{H}_1 = \text{Pair}_1 \land \mathcal{H}_2 = \text{Nothing}_2) \lor (\mathcal{H}_1 = \text{Ace}_1 \land \mathcal{H}_2 = \text{Nothing}_2);$$
$$t_e(\text{und}, \mathcal{H}_1, \mathcal{H}_2, \langle\text{play, play, eval}\rangle) = (p2, \mathcal{H}_1, \mathcal{H}_2) \text{ iff } (\mathcal{H}_2 = \text{Pair}_2 \land \mathcal{H}_1 = \text{Ace}_1) \lor$$
$$\quad \quad (\mathcal{H}_2 = \text{Pair}_2 \land \mathcal{H}_1 = \text{Nothing}_1) \lor (\mathcal{H}_2 = \text{Ace}_2 \land \mathcal{H}_1 = \text{Nothing}_1);$$
$$t_e(\text{und}, \mathcal{H}_1, \mathcal{H}_2, \langle\text{play, play, eval}\rangle) = (\text{und}, \mathcal{H}_1, \mathcal{H}_2) \text{ iff } (\mathcal{H}_1 = \text{Pair}_1 \land \mathcal{H}_2 = \text{Pair}_2) \lor$$
$$\quad \quad (\mathcal{H}_1 = \text{Nothing}_1 \land \mathcal{H}_2 = \text{Nothing}_2) \lor (\mathcal{H}_1 = \text{Ace}_1 \land \mathcal{H}_2 = \text{Ace}_2);$$
$$t_e(\text{win}, \mathcal{H}_1, \mathcal{H}_2, \langle\text{nothing, nothing, nothing}\rangle) = (\text{win}, \mathcal{H}_1, \mathcal{H}_2).$$

To complete the interpreted system for the poker game the set of initial states is defined as
### Table 5.2: Verification results for the simplified poker game.

<table>
<thead>
<tr>
<th>P-cards/D-cards</th>
<th>Without reduction</th>
<th>With reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>States</td>
<td>Time (s)</td>
</tr>
<tr>
<td>2/8</td>
<td>39</td>
<td>0</td>
</tr>
<tr>
<td>2/12</td>
<td>603</td>
<td>1</td>
</tr>
<tr>
<td>2/52</td>
<td>142340</td>
<td>4</td>
</tr>
<tr>
<td>2/80</td>
<td>$2.207 \times 10^6$</td>
<td>16</td>
</tr>
<tr>
<td>3/28</td>
<td>596480</td>
<td>14</td>
</tr>
<tr>
<td>3/32</td>
<td>592384</td>
<td>31</td>
</tr>
<tr>
<td>4/16</td>
<td>155648</td>
<td>16</td>
</tr>
<tr>
<td>5/12</td>
<td>$2.490 \times 10^6$</td>
<td>11</td>
</tr>
</tbody>
</table>

The game shown in Figure 5.4 is an ISPL description of the poker game for $C = \{1, \ldots, 6\}$ and $|\mathcal{H}_i| = 2$. The agent Environment has got an observable variable called win. This variable keeps the score of the players and can be seen by all the other agents. The value und means undetermined and it represents the initial state of the game. Value p1 means the agent Player1 won, while p2 means the agent Player2 won.

Similarly to the card game example shown in Figure 4.3, the agent Environment has got four variables representing the cards. The first two $c_{11}, c_{12}$ represent the cards held by Player1, while variables $c_{21}, c_{22}$ represent the cards held by Player2. This choice differs from the definition of the Simplified Poker game interpreted system given above for design reasons. In this way, we need to write the Evolution rules just once (in the environment agent only). The Evolution of the environment encodes the transition rules defined above.

The agent Player1 has got the variable n that says either to read the cards held or to play. Variables $c_{11}, c_{12}$ are Lobsvars. This means that, even though those variables belong to agent environment, they can be seen by the agent Player1. The Protocol says that at round n=read the cards are read by the agent, while at round n=play the cards are played. The Evolution of
Player1 encodes the transition rule stated above. The agent Player2 is defined in the same way.

In this example, the variable abstraction tool built the following clusters for the Environment:

\[ \text{VAR}_e = \{ \{c_{11}, c_{12}\}, \{c_{21}, c_{22}\}, \{\text{win}\} \} . \]

In Figure 5.5, the abstract ISPL is shown. Notice that abstract variables are \( c_1 \), \( c_2 \) and \( \text{win} \) that correspond to the variable clusters \( \{c_{11}, c_{12}\} \), \( \{c_{21}, c_{22}\} \), \( \{\text{win}\} \) respectively.

We have that the concrete pairs of values satisfy the following atomic formulas.

\[
\begin{align*}
(2, 3), (3, 2) & \models \{\text{Nothing1}\}; \\
(1, 2), (1, 3), (2, 1), (3, 1) & \models \{\text{Ace1}\}; \\
(2, 2), (3, 3) & \models \{\text{Pair1}\}; \\
(1, 1) & \models \{\text{Pair1}, \text{Ace1}\};
\end{align*}
\]

During the abstraction process the atomic formula Nothing1 gets the index 0, Ace1 gets the index 1 and Pair1 gets the index 2. Therefore, according to definition 4.14 abstraction function \( \rho_1 \) for the agent Player1 is defined as follows:

\[
\begin{align*}
\rho_1(2, 3) = \rho_1(3, 2) &= 1; \\
\rho_1(1, 2) = \rho_1(1, 3) = \rho_1(2, 1) = \rho_1(3, 1) &= 2; \\
\rho_1(2, 2) = \rho_1(3, 3) &= 4; \\
\rho_1(1, 1) &= 6;
\end{align*}
\]

The new domains are \( D_{c1} = D_{c2} = \{1, 2, 4, 6\} \).

Table 5.2 shows the reduction obtained in terms of the number of global reachable states and time needed for the verification and abstraction process for several variants of the poker game. In the first column “P-cards” stands for the number of card held by each player, while “D-cards” stands for the number of cards in the deck. We can notice that reductions are exponential in both global states and time for both the abstraction process and the verification process. Nevertheless, the memory used is not significantly reduced, since the abstraction toolkit generates extra BDDs. In the case of Black-jack we used several variants of this game for different numbers of deck cards. For the poker game, instead, we used different numbers
Agent Environment

Obsvars:
  win: {und,p1,p2};
end Obsvars

Vars:
  c11: 1..3;
  c12: 1..3;
  c21: 1..3;
  c22: 1..3;
end Vars

Actions = {eval, nothing};

Protocol:
  win=und: {eval};
  Other: {nothing};
end Protocol

Evolution:
  win=1 if c11 = c12 and c21 <> c22;
  win=1 if (c11=1 or c12=1) and
           (c21<>1 and c22<>1 and c21<>c22);
...
end Evolution
end Agent

Agent Player1

Lobsvars={c11,c12};

Vars:
  n: {read,play};
end Vars

Actions = {play, read};

Protocol:
  n=read: {read};
  n=play: {play};
end Protocol

Evolution:
  n=play if n=read;
end Evolution
end Agent

Agent Player2

Lobsvars={c21,c22};

Vars:
  n: {read,play};
end Vars

Actions = {play, read};

Protocol:
  n=read: {read};
  n=play: {play};
end Protocol

Evolution:
  n=play if n=read;
end Evolution
end Agent

Evaluation

Win1 if Environment.win=1;
Pair1 if
  Environment.c11=Environment.c12;
Ace1 if Environment.c11=1 or
  Environment.c12=1;
Nothing1 if
  Environment.c11<>Environment.c12
  and Environment.c11<>1 and
  Environment.c12<>1;
-- For Player2 is the same
end Evaluation

InitStates

Dealer.n=read and Player.n=read and
Environment.win=und (... end InitStates

Formulae

AG( (Pair1) -> K(Player1, AF Win1) );
end Formulae

Figure 5.4: Sketch of the ISPL program describing the simplified poker game for $|C_i| = 6$. 
Agent Environment
Obsvars:
  win: \{und,p1,p2\};
end Obsvars
Vars:
  c1: 1..6;
  c2: 1..6;
end Vars
Actions = \{ eval, nothing \};
Protocol:
  win=und: \{eval\};
  Other: \{nothing\};
end Protocol
Evolution:
  win=1 if (c1=4 or c1=6) and (c2=2 or c2=1);
  win=1 if (c1=2 or c1=6) and (c2=1);
...
end Evolution
end Agent

Agent Player1 Lobsvars={c1};
Vars:
  n: \{read,play\};
end Vars
Actions = \{play, read\};
Protocol:
  n=read: \{read\};
  n=play: \{play\};
end Protocol
Evolution:
  n=read if n=play;
end Evolution
end Agent

Agent Player2 Lobsvars={c2};
Vars:
  n: \{read,play\};
end Vars
Actions = \{play, read\};
Protocol:
  n=read: \{read\};
  n=play: \{play\};
end Protocol
Evolution:
  n=read if n=play;
end Evolution
end Agent

Evaluation
Win1 if Environment.win=1;
Pair1 if Environment.c1=4 or
Environment.c1=6;
Ace1 if Environment.c1=2 or
Environment.c1=6;
Nothing1 if Environment.c1=1;
-- For Player2 is the same
end Evaluation

InitStates
Dealer.n=read and Player.n=read and
Environment.win=und (... 
end InitStates

Formulae
AG( (Pair1) -> K(Player1,AF Win1));
end Formulae

Figure 5.5: Sketch of the ISPL program describing the abstract simplified poker game.
of cards for the deck and different numbers of cards held by both players. We could not check other variants of the poker game since for other cases the input ISPL files were so large in terms of MB that the abstraction process was stopped after one hour.

In this Chapter we presented a variable-abstraction technique to reduce the state space of interpreted systems. The correctness of this procedure is shown by Theorem 4.1 since the theorem does not refer to a particular partitioning method but to general quotient interpreted system. The general theoretical basis of this Chapter were shown in Chapter 3. In the next Chapter we will compare both techniques presented in Chapter 4 and 5 with abstraction methodologies summarised in Chapter 2. Comparisons will be qualitative and quantitative when possible.
Chapter 6

Conclusion

6.1 Thesis contributions

This thesis was concerned with the definition, development, and the analysis of two existential abstraction techniques for model checking multi-agent systems against ACTLK-specifications. The main contributions of this work can be summarised as follows:

- **Theoretical contributions.** Preservation and correctness results (Theorems 3.1, 3.2.1, 3.2, 4.1) were shown establishing that if a property is verified by the abstract model, then it is verified by the concrete one. These results are noteworthy since they allow us to verify temporal-epistemic formulas on abstract systems instead on concrete ones, therefore saving time and space in the verification process. As we have seen in Chapter 4 in some cases no verification can be performed on large concrete models, making an abstraction methodology desirable.

- **Development of abstraction toolkits.** Two different abstraction toolkits have been developed: one for data-abstraction and one for variable-abstraction. The first was applied to systems in which the specifications to be checked depend on atomic formulas that are defined on values of single variables only. The second was applied to systems in which the atomic formulas are defined via several variables that interact with each other. The best reduction in terms of state space is performed when the logic expressions appearing in the definitions of atomic formulas in the Evaluation section of the ISPL code,
also occur in the local Evolution sections. In other words, the specification we want to check is made of “pieces” (logic expressions) that are part of the internal description of the system under investigation. For instance, in the card game of Chapter 4, described by the ISPL in Figure 4.3, the logic expression \( c_{11}>2 \) appears both in the Evaluation and Evolution sections (after having transformed expression \( c_{11}>c_{21} \) to a semantically equivalent Boolean expression that contains one variable for each logic expression). In this particular case, Table 4.1 shows a drastic reduction in space and time. This leads to the feasibility of model checking for some very large systems to which it was not possible to apply otherwise. This can be explained by considering how the abstraction process actually works. If a logic expression \( p \) appears both in the Evaluation and Evolution sections then the replacement of \( p \) with its semantically equivalent abstract logic expression \( p' \) is direct and it is not needed a further transformation that might compromise the efficiency of the abstraction process.

In all the other cases (like the number transmission problem in Figure 4.6), the reduction is not so dramatic, but it still shows a substantial improvement.

- **Evaluation.** Several benchmarks were conducted to evaluate the power and effectiveness of the abstraction techniques studied. The experiments have shown good results in terms of state space reduction and time required in the model checking process.

### 6.2 Comparison with related work

In this section we compare the abstraction techniques presented in Chapters 3, 4 and 5 with the methodologies presented in the main related works in the field of abstraction and state space reduction, discussed in Chapter 2.

#### 6.2.1 Symmetry reduction techniques for CTLK

In Section 2.6.2 we summarised the reduction techniques presented in [CDLQ09b](#) [CDLQ09a](#). In these works the authors developed methodologies to reduce the size of interpreted systems by exploiting their symmetries. A symmetry can be related to the domains of variables (data
symmetry [CDLQ09a] or to the agents (agent symmetry [CDLQ09b]). Informally, a symmetry is a function that interchanges values in a variable’s domain or agents across global states.

The main difference between the methodologies presented in Chapter 4 and 5 and the approach presented in [CDLQ09b, CDLQ09a] consists in the automatic construction of abstraction functions from the set of atomic formulas defined in the Evaluation section of an ISPL program describing the interpreted system under investigation. In this thesis these are surjections, while the functions used in [CDLQ09b, CDLQ09a] are bijections. Bijections do not allow either false negative or false positive cases. A set of bijection (symmetries) induce an equivalent interpreted system defined on a reduced state space that is evaluated by means of counterpart semantics (in the sense of Lew68).

Unfortunately, bijections can be only used to reduce systems containing symmetries (data symmetries or agent symmetries). Therefore, their use is restricted to symmetric systems, while the applicability of surjections is wider and it can be use for non-symmetric systems. However, the price to pay for using surjections is the possible presence of false negatives. In contrast, reduction Theorems 2.1 and 2.2 [CDLQ09b, CDLQ09a] preserve validity in both directions (from abstract to concrete systems and vice versa). In [CDLQ09b] and in [CDLQ09a] the authors evaluated their reduction techniques on the muddy children puzzle [FHVM95] and on a standard protocol in the security literature, called the Needham-Schroeder Public Key protocol (NSPK) [NS78], respectively. In Tables 6.1 and 6.2 the time is expressed in seconds and it refers to the sum of abstraction process and verification time spent by the symmetry reduction techniques, while the BDD-memory is expressed in MB.

If we compare the reductions in verification time and BDD-memory consumption shown in Tables 6.1 and 6.2 with the reductions shown in Tables 4.1, 4.2, 5.2 and 5.1 in Chapter 4 and 5 we notice that in all experiments conducted on ISPL programs with a comparable “size” in terms of BDD memory, the reductions obtained are exponential in time and linear in memory for both symmetry reductions and the abstraction methods presented in this thesis.

### 6.2.2 Epistemic abstraction on Kripke structures

In Section 2.6.4 we summarised key concepts from [ED07]. In this work the authors presented an abstraction methodology for Kripke structures. This work is the first attempt toward ab-
Table 6.1: Verification results for the muddy children puzzle, from [CDLQ09].

A key difference between our methodologies and the technique presented in [ED07] consists in the usage of interpreted systems as the underlying semantics. As we have seen in Chapter 4 and Chapter 5 these systems are *computationally grounded* (in the sense of [Woo00]), while plain Kripke structures are not. In fact, the epistemic indistinguishability relation $\sim_i$ for an agent $i$ is defined in Definition 2.17 as the equality of the $i$-local states. In [ED07] the authors evaluate their abstraction technique on one toy-system (see Figure 2.8) only. ISPL programs instead offer the possibility to describe real applications, e.g., the software development system discussed in Chapter 4, in a direct and natural fashion.

It is not possible to make a quantitative comparison between the procedures defined in Chapter 4 and 5 and the technique presented in [ED07] as this provides no algorithm for the automatic construction of Kripke structures, nor a constructive way to define the surjective function performing abstraction.
6.2. Comparison with related work

<table>
<thead>
<tr>
<th>agents</th>
<th>Without reduction</th>
<th>With reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>States</td>
<td>Time</td>
</tr>
<tr>
<td>3</td>
<td>1536</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>11400</td>
<td>28</td>
</tr>
<tr>
<td>5</td>
<td>651600</td>
<td>7716</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>&gt;86400</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>&gt;86400</td>
</tr>
</tbody>
</table>

Table 6.2: Verification results for the NSPK protocol, from [CDLQ09a].

6.2.3 Existential abstraction on Kripke structures

In Section 2.6.5 we discussed the methodology presented in [CGL94, CGJ+00, CGJ+03]. These are the key references and the inspiration for this thesis. In these works the authors defined an algorithm for the automatic abstraction of Kripke structures. They produced preservation results, e.g. Theorem 2.3 [CGJ+03], establishing that if an \( \text{ACTL}^* \)-formula \( \phi \) holds in the abstract Kripke structure, then \( \phi \) holds in the concrete one. In case the property does not hold the procedure exploits the counterexample generated in the verification step by the model checker to refine the original Kripke structure in order to establish whether the counterexample is spurious or it is a concrete behaviour of the original system.

Abstraction techniques presented in [CGL94, CGJ+00, CGJ+03] by Clarke’s et al. are “blind”, while methodologies defined in this thesis permit to “calibrate” the abstraction process. This means that these works do not consider the set of specifications to be checked on the system under investigation. Because of this, there is a risk of obtaining an abstract system in which many extra behaviours might be easily introduced. Consequently, the probability to get false negatives becomes very high. This risk is reduced by iterating several counterexample refinement steps in case the formula to be checked is not satisfied. This leads to additional computational costs due to the refinement steps.

By contrast, the abstraction technique presented in this thesis is guided by the atomic formulas appearing in the formula we want to check. The key difference between this research and [CGL94, CGJ+00, CGJ+03] is to guide the abstraction process toward the most suitable reduction according to the specifications to verify. Hence, state space reduction techniques
presented in Chapter 4 and Chapter 5 perform a *formula-guided-abstraction*. In other words, the abstraction process is suited to the specifications to be checked.

Comparing to this thesis, the main advantages of their methodology consists in the refinement process guided by counterexamples generated during the verification step and on the applicability of *ACTL*-specification. By contrast, our methodologies do not catch full *LTL*, but applies to knowledge specifications.

Table [6.3] shows the experimental results appearing in [CGJ+03] for some well known benchmarks in the model checking literature. In Table 6.3 time is expressed in seconds and the number of states is represented by BDD nodes.

The designs (taken from the model checking literature) tested in Table 6.3 are very different from the ISPL programs used in this thesis to evaluate our abstraction techniques. However, it is still possible to make a rough comparison by considering systems with the same “size” in terms of BDD nodes with ISPL designs with a comparable number of global states shown in Tables 4.1, 4.2, 5.2 and 5.1. It is easy to notice that the reduction is exponential in time in both cases.

<table>
<thead>
<tr>
<th>Design</th>
<th>Without reduction</th>
<th>With reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>BDD nodes</td>
</tr>
<tr>
<td>gigamax</td>
<td>0.3</td>
<td>8346</td>
</tr>
<tr>
<td>guidance</td>
<td>35</td>
<td>140409</td>
</tr>
<tr>
<td>waterpress</td>
<td>273</td>
<td>34838</td>
</tr>
<tr>
<td>PCI bus</td>
<td>2343</td>
<td>121803</td>
</tr>
<tr>
<td>ind1</td>
<td>99</td>
<td>241723</td>
</tr>
<tr>
<td>ind2</td>
<td>486</td>
<td>416597</td>
</tr>
<tr>
<td>ind3</td>
<td>617</td>
<td>584815</td>
</tr>
</tbody>
</table>

Table 6.3: Verification results for several benchmarks, from [CGJ+03].
6.2.4 Predicate abstraction

In Section 2.6.3 we summarised the predicate abstraction techniques presented in [CJK04]. In this work the authors presented procedures to generate abstract descriptions of hardware designs specified in Verilog [CTVW04] language, e.g. the program in Figure 2.6. In predicate abstraction the set of concrete states of systems is mapped into Boolean values by making use of a given set of predicates. Predicate abstraction makes use of theorem provers. These tools are used to automatically generate pairs of states that represent the transitions in the abstract system from a conjunctive normal formula that describes the entire global transition relation $R$ and returns a statement that describes the global transition relations $\hat{R}$ in the abstract model as disjunction of cubes (see Trans statement in Figure 2.7).

Due to the substantial difference of the two approaches it is impossible to make a quantitative comparison between predicate abstraction methods and state reduction techniques presented in this thesis. However, some qualitative comparisons (some of them are reported in Figure 6.4) and general considerations are still possible.

A difficulty in predicate abstraction is the identification of the initial set of predicates from which the abstraction process begins. For instance, the set of predicates, e.g. formulas $x < 100$, $x < 200$, $x + y < 200$ in the example in Figure 2.7 are arbitrarily chosen before applying the abstraction process. Procedures presented in this thesis instead execute abstraction in a constructive and automatic way.

Predicate abstraction suffers from two main problems. The first regards the use of model generators in the construction of the abstract system. Calling the theorem prover usually involves a substantial computational cost in performing abstraction and this may compromise the efficiency of the technique. In the worst case, the number of satisfying assignments generated from Equation 2.6 is exponential in the number of initial predicates (see the list of predicates at the top of Figure 2.7) used to build the abstract system. The second problem is related to the discrepancies between the high level description of a system used by predicate abstraction, and low level design used by the great majority of model checkers. In fact, predicate abstraction is mostly effective if the predicates describing the system involve the relationships among a great number of variable constraints (called “latches” in [CJK04]).

The main advantage of predicate abstraction techniques consists in their wide applicability
Chapter 6. Conclusion

Other state space reduction techniques

<table>
<thead>
<tr>
<th>Other state space reduction techniques</th>
<th>main differences with our methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicate abstraction</td>
<td>arbitrary set of predicates and usage of model generators</td>
</tr>
<tr>
<td>Existing abstraction</td>
<td>counterexample guided refinement and “blind” abstraction</td>
</tr>
<tr>
<td>Symmetry reduction</td>
<td>usage of bijection functions to perform abstraction</td>
</tr>
<tr>
<td>Epistemic abstraction on Kripke structures</td>
<td>Not computationally grounded (3-value Kripke structures)</td>
</tr>
</tbody>
</table>

Table 6.4: Some qualitative comparisons with other related techniques.

from hardware designs to many software systems. Plus, predicate abstraction can be applied to infinite state space system, while our methodologies only work on finite systems.

All the comparisons discussed above are summarised in Table 6.4. The results reported in Tables 4.1, 4.2, 5.2, 5.1, 6.3, 6.1 and 6.2 point to the fact that there is not a best abstraction technique. Nevertheless, the general performance of this thesis techniques is comparable with that one of other procedures discussed in Chapter 2.

6.3 Future work

The benefits of the abstraction techniques presented in this thesis are shown from the experiments conducted in Chapter 4 and Chapter 5. However, abstraction on interpreted systems developed in this research work has generated many open issues not considered in this thesis. Some of them are the following:

- There are many ways to improve techniques introduced in this thesis. Abstraction methods of Chapter 4 and Chapter 5 start from logic expressions that are part of the definitions of atomic formulas belonging to the specifications we want to check (see formula 4.12). Instead of relying on the abstraction method on logic expressions, we could perform abstraction directly on Boolean expressions. This method might permit a more formula-guided-abstraction. The main problem consists in the generation of the new ISPL program
6.3. Future work

Based on the Boolean expressions used in the abstraction process. It is easy to replace a value in a logic expression or to rewrite the whole logic expression in terms of new variables. Unfortunately, to write a Boolean expression into a semantically equivalent one is not trivial, and may require attention.

- Instead of manipulating the syntactic structure of an ISPL program, we might operate on the symbolic representations (OBDDs) of global states, transition functions and of the epistemic relation to get the abstract ISPL program from those graphs directly. The major problem of this would become getting an ISPL program from those OBDDs. The translation from OBBDs to ISPL programs is not unique.

- Refinement techniques, starting from an ISPL program representing an abstract system, can be developed in order to eliminate false negatives generated by spurious paths. First of all, the refinement procedure might start from the identification of a spurious counterexample. Spurious counterexamples are executions that belong to abstract systems, but they do not belong to concrete ones. Therefore, these are part of the extra behaviours introduced during the abstraction process.

Let \( \hat{g} \) be an abstract global state of an interpreted system \( \mathcal{I} \). Let \( \rho \) be the abstraction function. The set of concrete states is given by \( \rho^{-1}(\hat{g}) = \{ g \mid \rho(g) = \hat{g} \} \). Following CGJ+03, it is possible to extend \( \rho^{-1} \) to paths \( \hat{\pi} \) as follows:

\[
\rho^{-1}(\hat{\pi}) = \{ g_1, \ldots, g_n \mid \bigwedge_{i=1}^{n} \rho(g_i) = \hat{g} \wedge I(g_1) \wedge \bigwedge_{i=1}^{n-1} \mathcal{T}(g_i, g_{i+1}) \}. \tag{6.1}
\]

\( I \) is the set of initial global states of \( \mathcal{I} \). The expression \( I(g_1) \) means \( g_1 \in I \). \( \mathcal{T} \) is the global transition function defined in 2.13.

The main problem is to study what happens to the accessibility relation \( \sim_i \) for each agent \( i \) of the system \( \mathcal{I} \). Therefore, for interpreted systems the definition (6.1) is incomplete. We need to add an fourth condition in (6.1) for \( \sim_i \).

A refinement procedure should detect two sets of concrete states \( G_D \) and \( G_B \), called dead-end states and bad states, respectively. In a counterexample, dead-end states are reachable via \( \mathcal{T} \), but there are no outgoing transitions from them to the next state. Bad states are not reachable, but they have outgoing transitions to the next state in the spurious path. These states generate the spurious counterexample. If dead end states and bad states have been collapsed into the same abstract state during the abstraction
process, then they have to be separated in the refinement process in such a way that they do not correspond to the same abstract state. In this way, the spurious counterexample is removed\(^1\). A refinement procedure might follow the scheme depicted in Figure 6.1.

- Abstract interpreted systems might be generated by “collapsing agents”. The main problem consists in how to preserve the epistemic specifications during the abstraction process. For instance, if two agents, let us say 1 and 2, both know an information encoded in a formula \(\phi (K_1 \phi \land K_2 \phi)\), then the new agent \(\hat{1}\) created by collapsing 1 and 2 together should know \(\phi (K_{\hat{1}} \phi)\). The same should happen with negation \(\neg\). Let \(Ag\) be the set of agents of an interpreted system. If we have that at least one agent \(i\) belonging to a group of agents \(\Gamma \subseteq Ag\) does not know \(\phi (\neg K_i \phi)\), then new agent \(\hat{i}\), created by collapsing all agents in \(\Gamma\), should not know \(\phi (\neg K_{\hat{i}} \phi)\).

- It could be useful to implement a script that transforms the specification to be checked into a semantically equivalent one. The new one should be made of logic expressions present in the local Evolution section. This might improve the state space reduction for the reasons mentioned in Section 6.1.

\(^1\)For more details see [CGJ+03].
Bibliography


