AUTOMATIC TRANSFORMATION-BASED MODEL CHECKING OF MULTI-AGENT SYSTEMS

Amine Laarej

A thesis
in
The Department
of
Concordia Institute of Information Systems Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Applied Science in Quality Systems Engineering
Concordia University
Montréal, Québec, Canada

August 2020

Concordia University
School of Graduate Studies

This is to certify that the thesis prepared

By: **Amine Laarej**

Entitled: **Automatic Transformation-Based Model Checking of Multi-agent Systems**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science in Quality Systems Engineering

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

Dr. Abdessamad Ben Hamza _____ Chair

Dr. Juergen Rilling _____ Examiner

Dr. Abdessamad Ben Hamza _____ Examiner

Dr. Jamal Bentahar _____ Supervisor

Dr. Rachida Dssouli _____ Supervisor

Approved _____

Dr. Mohammad Mannan Graduate Program Director

_____ 2020 _____

Dean of Gina Cody School of Engineering and Computer Science

Abstract

Automatic Transformation-Based Model Checking of Multi-agent Systems

Amine Laarej

Multi-Agent Systems (MASs) are highly useful constructs in the context of real-world software applications. Built upon communication and interaction between autonomous agents, these systems are suitable to model and implement intelligent applications. Yet these desirable features are precisely what makes these systems very challenging to design, and their compliance with requirements extremely difficult to verify. This explains the need for the development of techniques and tools to model, understand, and implement interacting MASs. Among the different methods developed, the design-time verification techniques for MASs based on model checking offer the advantage of being formal and fully automated. We can distinguish between two different approaches used in model checking MASs, the direct verification approach, and the transformation-based approach. This thesis focuses on the later that relies on formal reduction techniques to transform the problem of model checking a source logic into that of an equivalent problem of model checking a target logic.

In this thesis, we propose a new transformation framework leveraging the model checking of the computation tree logic (CTL) and its NuSMV model checker to design and implement the process of transformation-based model checking for CTL-extension logics to MASs. The approach provides an integrated system with a rich set of features, designed to support the transformation process while simplifying the most challenging and error-prone tasks. The thesis presents and describes the tool built upon this framework and its different applications. A performance comparison with MCMAS, the model checker of MASs, is also discussed.

Acknowledgments

I would like to warmly thank first of all my thesis supervisor, Dr. Jamal Bentahar, for his invaluable support and patience, for his precious and never lacking enthusiasm for research and for his continuous assistance in preparing, writing papers, and this thesis. I want to thank him for having so strongly believed in me, even when I didn't.

I would also like to express my sincere gratitude to my co-supervisor, Dr. Rachida Dssouli, for the trust she put in me, for the support she offered me, and for her endless wisdom, and kindness.

My gratitude also goes to all my colleagues in the Multi-Agent Systems and Web Services laboratory at Concordia University, especially Nagat Drawel, Mounia Elqortobi, and Laura Marianella for their friendliness, their generosity, and their patience.

I would like to thank Concordia University for the funding and the facilities they put under my disposal to achieve this work.

More especially, I owe a huge debt of gratitude to my beloved parents and all my family members. Thank you for being there for me more times than I care to admit, a psychological, physical, and spiritual safety net. I feel very fortunate that you decided to pick me out of the ethereal. For all that is and has gone well in me, and my life is because of you. I, of course, take full responsibility for all the things that fall somewhere in between.

Last, but surely not least, I would like to thank my friends, Oussama, Amine, Omar, Redouane, Youssef, and Abdeladim, for our ever binding relationship and our countless priceless memories.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, we introduce the context of our research. We then explain the research problem we are trying to address and the motivations behind it, before presenting the different features and contributions of this work. Finally, we conclude by going over the organizational structure of this thesis.

1.1 Context of Research

1.1.1 Multi-Agent Systems

Agents have emerged as a new paradigm for software engineering over the last few decades. Because *interaction* is the main characteristic of complex software systems, and almost all real-world applications need software systems that are distributed into many independent yet dynamically interacting components, the development of tools and techniques to model, understand and implement systems built to interact became a major research topic in computer science.

An agent is a computational entity (e.g., a software program) situated in an environment, upon which it can act autonomously to meet its design objectives. Thus, an agent has full control over its own state and behavior without intervention from other agents or humans [51]. For an agent to be considered ***intelligent***, it has to be [52]:

1. *Reactive*: it can perceive any change in its environment and react to it.

2. *Pro-active*: it is capable of making decisions based on its design goals and acting on them.

3. *Social*: it is capable of interacting with other agents.

4. *Rational*: the agent's actions are always consistent with its goals.

A Multi-Agent System (MAS) can be defined as a "*system in which several interacting, intelligent agents pursue some set of goals or perform some set of tasks*" [48]. Such a system can be either compromised of heterogeneous agents (agents that have different implementations, goals, behavior, and even programming teams and technologies) or homogeneous agents (identical agents both design and implementation-wise, like a swarm of identical robots for example). It's the aggregation of all these properties that makes MASs highly complicated and hard to design, but its also what makes them effective at modeling and solving real-world problems, ranging from commercial to industrial, governmental, and healthcare applications [7, 28, 24, 47].

1.1.2 Agent Communication

It comes as no surprise that in a MAS, interaction and communication is perhaps the most crucial aspect needed in order to define an effective MAS. Goal-oriented coordination, be it within a competitive or cooperative setting, is, therefore, an essential pattern of interaction in the context of such systems. As agents affect each other continuously by taking actions to fulfill their goals (whether cooperating to achieve what otherwise could not, or competing for an outcome where the success of an agent might mean the failure of others), the need for a commonly understood language: *lingua franca* for agents to communicate with each other becomes apparent. Agents need to 'talk' in a single commonly understood language to cooperate, negotiate, or exchange knowledge and information, information that's crucial for the achievement of their goals in the most efficient manner. They also need communication protocols to regulate and structure such communication among participants within dialogs and negotiations.

Two approaches have been proposed to model agents communication in the literature:

- The *mental approach*, which focuses on mental notions such as beliefs, desires, goals, and intentions, tries to strike a balance between these notions to meet the

system's specifications effectively [40]. The biggest downside to this approach is known as the semantics verification problem [50]. It stems from the fact that a receiving agent cannot be sure if the sending agent didn't violate any pre-conditions since there is no way to access an agent's mental state [42]. Such pure mental semantics is impossible to use in systems of heterogeneous agents where sincerity is not guaranteed for one reason or another. Moreover, this semantics prevents ACLs from being general enough for the needs of heterogeneous systems [41]. Because of these limitations, the MAS community shifted the focus to the second approach.

- The *social approach* is based on the observation that in every communication that happens in a MAS between two agents, the system itself can be considered an additional third side in this communication. There are thus three angles we can view the communication from the receiver's, the sender's, and the society's (the system's) point of view. The "social" approach, through this shift, allows us to model notions from a more generalized perspective without having to use an agent's external private state. The impact of such a change on the modeling capabilities of heterogeneous MASs is what makes more advanced notions and frameworks possible [43].

The interacting and communicating aspect of MASs also gives rise to other advanced desirable notions that have been studied extensively in the literature, such as trust, reputation, knowledge, and argumentation [30, 6, 22, 27].

1.1.3 Verification of MASs

Because of the inherent complexity of a MAS, verifying that such a system complies with its expected design goals and requirements is a challenging task. In a system with multiple agents all behaving differently towards different goals, whether in competition or collaboration or both, there should be no room for error as to the final compliance of the system with its objectives, especially if the system is safety-critical. Two different approaches for verification of such systems are used today, operating at two different levels: runtime and design-time. At runtime, the most used verification technique is monitoring, which consists of observing the behavior of the system and its agents during execution, and checking to see if the desired properties of the system

are met [4, 3]. Such an approach, while simple to implement, has its shortcomings. As this technique consists of running multiple executions of the actual system's implementation and observing the behavior it produces to check for defects or undesirable outcomes, it naturally suffers from its incomplete verification process limited by the execution's coverage. The design-time approach, on the other hand, relies on static formal verification to verify the system's properties during design and to catch defects and correct them before going to production. It relies on intelligent exhaustive search algorithms to check all the possible states of the system in a fully automated manner. It can even generate counterexamples for undesired behavior of the system allowing for defect correction. However, these logic-based techniques suffer from the state explosion problem that limits their scalability for large systems. Multiple research teams have worked on these problems and proposed solutions to improve the different techniques, such as symbolic model checking that has been used efficiently to automatically verify different aspects of MASs such as knowledge, trust, social commitment, and fulfillment (**Section 2.4** of **Chapter 2** expands further on model checking).

1.2 Motivations and Contributions

As discussed earlier, design-time verification techniques for MASs based on formal verification in general, and model checking in particular offer the advantage of being fully automated. This is by far, the main reason why these techniques are used as the process of hand-on verification is in no way scalable for large systems.

In order to reduce cost, while preserving the robustness and automation that such techniques offer, two model checking based design-time verification approaches have been put forward to reason about new social properties for MASs: direct and indirect verification techniques (cf. Figure 1.1). The direct verification method is the straightforward one where new dedicated verification algorithms are developed to tackle the problem of verifying the new social modalities. They are then implemented from scratch on top of an existing model checker by augmenting it with the new algorithms, or by creating a new dedicated model checker for the new logics. The idea behind the indirect verification is to transform the problem of model checking the new logic into the problem of model checking an existing logic (that has its own model

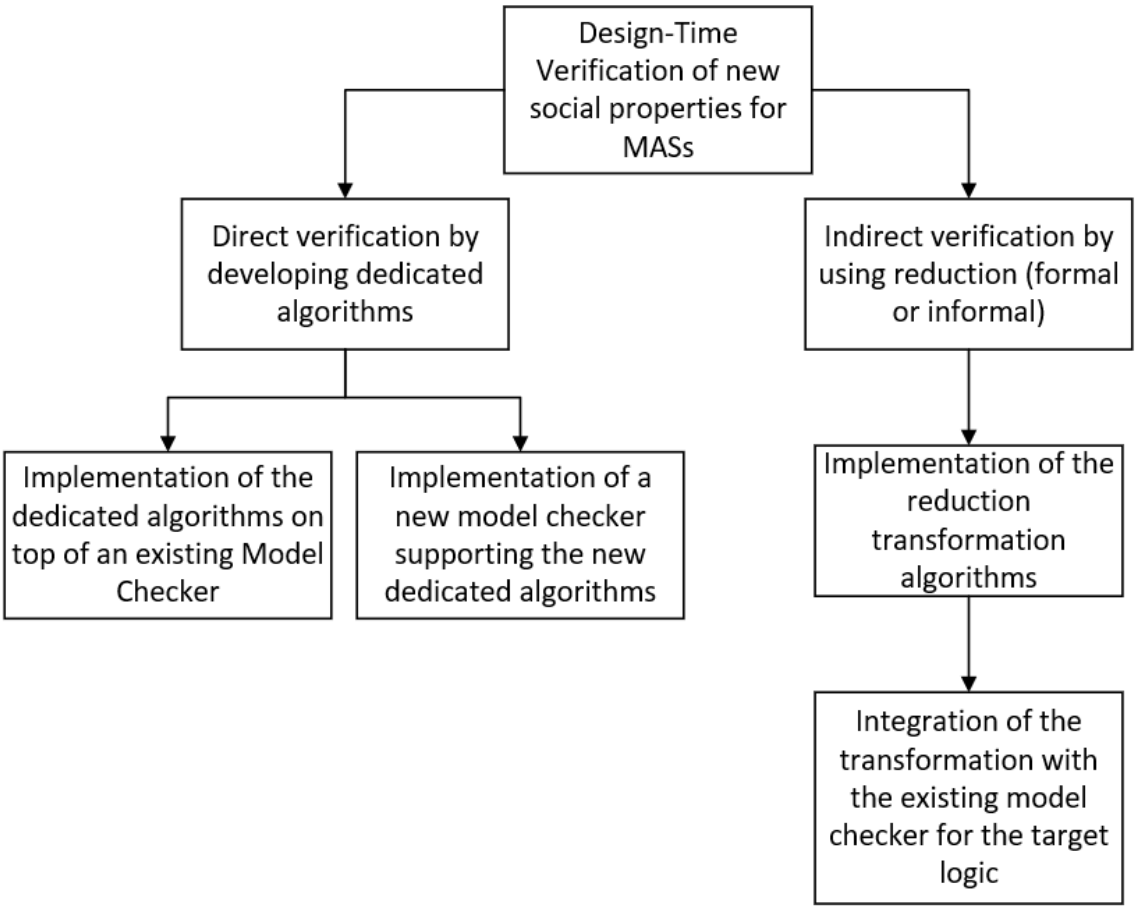

Figure 1.1: A typical design-time verification process for multi-agent systems.

checker) using reduction. This process eliminates the need to create dedicated model checking algorithms automation purposes, and instead allows the more straightforward endeavor of automating the transformations, and then integrating them with the existing model checker.

In practice, the indirect approach - generally referred to as *transformation-based model checking* - offers the benefit of reducing post-development costs related to maintaining the model checker, and requires simple changes in case the core model checker undergoes significant changes that break backward compatibility. In most cases, it also allows the use of reliable model checkers that have been in the field for years, undergoing multiple enhancements, and enjoying the support of a big community.

In this dissertation, we focus on new social properties for MASs expressed based

on temporal logics, as they are the only approaches that allow the introduction of formal semantics. Specifically, we concentrate on the approaches that adopt temporal logics to model new concepts by introducing new modalities to extend Computation Tree Logic (CTL), rather than Linear Time Logic (LTL) or Full Computation Tree Logic (CTL*). The reason we are singling out CTL-based extensions is due to the difference between the model checking algorithms for CTL, LTL, and CTL*. The standard model checking algorithm for both LTL and CTL* is exponential in the length of the formula and linear in the size of the model [39]. The CTL standard model checking algorithm, however, is linear in both the formula and the model [39]. So, even though CTL* is more expressive than CTL, we have no choice but to adapt CTL, although it lacks the capabilities to model interactions and dynamic behavior that intelligent agents presuppose. These capabilities, therefore, have to be achieved through the extension modalities added to CTL.

The implementation of the transformation algorithms from a CTL-extension logic to an existing logic allows us to take full advantage of the automation capabilities that characterize model-checking techniques. While performing this implementation, one should adequately consider the full overhead the transformation mechanism adds to the model-checking process. Otherwise, this process might lead to high memory consumption and limited scalability. Handling these aspects gets harder as the systems under verification grow in scale and complexity. Manually performing this implementation is a tedious, error-prone, and time-consuming process that requires a vast amount of knowledge and technical skills to create parsers from one logic to another, and to allow full integration with the existing model checkers to take full advantage of their capabilities. The current practices generally produce unreliable software resulting in hard to replicate results and difficult to maintain software that's generally based on shallow integration with model checkers at best. In fact, most implementations skip the integration with model checkers all-together and generate an output file for manual use.

To ease the design and implementation of such transformations, we propose in this thesis a new tool for verifying transformation-based model checking algorithms. In this work, we aim to bridge the gap in the available tooling and frameworks to handle these transformations as there are no dedicated tools for these needs. We built this tool on top of the NuSMV model checker, one of the most powerful model checkers

available today for CTL, to consistently produce high-quality software.

To summarize, the key components and features of the proposed tool are as follows:

- The core framework of the tool offers a parser and lexer generator API that simplifies the process of generating parsers for models and transformations considerably.
- The tool uses the ISPL+ formalism used in model checkers dedicated to MASs as input. This support allows the input files to represent MASs easily and intuitively, as opposed to SMV, the input format for NuSMV where it is hard to model agents.
- The tool's API also offers a UI multi-agent system builder based on the ISPL+ formalism, allowing the constructions of agents manually in an FSM-like graphical format.
- The tool offers a second equivalent input format called scalability mode, where the user can input an ISPL+ file instead of UI input, allowing for the processing of large systems where the graphical approach would be too tedious.
- The tool offers a configurable accessibility engine to define accessibility relations easily and automate their calculations during the transformation process.
- The tool offers complete integration with the NuSMV model checker, making all its capabilities available.
- The tool is developed in the Java programming language, making it multi-platform. In fact, the tool was created to be applicable in an industrial setting. Therefore, it has been made without any modification to the NuSMV model checker since it is distributed under the **GNU Lesser General Public License**. This removes the need to release the source code of the components.

The proposed tool has been tested extensively and successfully used in multiple research projects in multiple application domains [14, 33, 15, 18]. It has been used with multiple CTL-extension logics and has produced exceptional results compared to the state-of-the-art implementation approaches.

1.3 Thesis Organization

The rest of this thesis is organized as follows. In Chapter 2, we present the necessary background needed to understand this work. Then, we review a few existing tools that offer some exciting features. In Chapter 3, we go over our published journal paper to see how the tool can be used in a research context and what it offers. In Chapter 4, we dive into the tool's architecture and implementation. Moreover, we go in this chapter over the different performance evaluations and experimental results used in different publications. We then summarize the obtained results and discuss future directions in Chapter 5.

Chapter 2

Background

In this chapter, we briefly present some preliminary knowledge needed for the rest of this thesis. We start by explaining the formalism of interpreted systems as the basis of the input system of the tool both in graphical and file form in Section 2.1. Then, in Section 2.2, we explore the Computational Tree Logic briefly as the basis of all formal semantics used with the tool. Section 2.3 presents all the different extensions to CTL that the tool has been used for and currently supports. Section 2.4 is devoted to providing the proper background to model checking and its tools. After that, in Section 2.5, we discuss relevant related work in the form of the different tools and frameworks available, then we review their features and capabilities and contrast them with the work done in this thesis. Finally, Section 2.6 summarizes the chapter.

2.1 Interpreted Systems

The formalism of Interpreted Systems was first introduced by Fagin et al. [21] as a novel way to model MASs by reasoning about their temporal evolution, allowing for a more intuitive way to capture epistemic and temporal properties.

Let us assume a given MAS is composed of n agents $\mathcal{A} = \{1, ..., n\}$. Each agent $i \in \mathcal{A}$ is described by a set of local states L_i, and a set of local actions Act_i. Each local state of agent i, denoted l_i, such that $l_i \in L_i$, represents the state of agent i at a given moment. A global state $g \in G$ represents the state of the global system at a given moment (a "snapshot" of sorts), it consists of the set of all local states of all agents at that given moment (i.e., $g = (l_1, ..., l_n)$). Consequently, the set of all global

states G is the cartesian product of all local states of the n agents $G = L_1 \times ... \times L_n$.

Moreover, for each agent $i \in \mathcal{A}$, I_i represents an initial state and $\mathcal{P} : L_i \longrightarrow 2^{Act_i}$ the local protocol of agent i, which denotes the possible actions at any given local state. The agents act typically within an environment e that can also be modeled with the same constructs: $\mathcal{P}_e, Act_e$ and L_e. The global transition function τ can be defined as $\tau : G \times ACT \rightarrow G$, where $ACT = Act_1 \times ... \times Act_n$, every component $a \in ACT$ is called a *joint action* (i.e., a tuple of actions corresponding each to an agent).

Bentahar et al. [5] and El-Menshawy et al. [17] extended this formalism to capture the communication construct between interacting agents. For each agent $i \in \mathcal{A}$, they associated a set Var_i of n local Boolean variables ($Var_i = |n|$), that represent communication channels between agent i and all other agents. Each local state l_i is therefore associated with a set of different values, corresponding each to a different value assignment for a variable, let's denote by $l_i^x(g)$ the value of the variable x at local state $l_i(g)$. The intuition behind these added variables is to model a communication channel between two agents, i and j, in the absence of which the two agents cannot communicate. Meaning that a communication channel exists between two agents i and j iff $\exists! x \in Var_i \cap Var_j$, in other words $|Var_i \cap Var_j| = 1$. For a Boolean variable $x \in Var_i \cap Var_j, l_i^x(g) = l_i^x(g')$ means that a communicative act has happened between agent i(in the global state g) and agent j(in the global state g'), that resulted in the two agents sharing the same variable value for x.

2.2 Computation Tree Logic

Computation Tree Logic (CTL, for short) is a branching-time logic where the structure of time is assumed to be branching in a tree-like manner (hence the name) where every moment in time may split into many paths in the future.

Definition 2.1: Syntax of CTL

The syntax of a CTL formula is defined using a BNF grammar as follows:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid EX\varphi \mid EG\varphi \mid E(\varphi \cup \varphi)$$

where $p \in \mathcal{VP}$ is an atomic proposition from the set of atomic propositions $\mathcal{VP}$. "E"

is an existential quantifier over paths. "A" the universal quantifier over paths can be defined as usual in term of the above as: $AX\varphi = \neg EX\neg\varphi$; $AG\varphi = \neg EF\neg\varphi$; and $A(\varphi \cup \psi) = \neg(E(\neg psi \cup (\neg\varphi \wedge \neg\psi)) \vee EG\neg\psi)$.

Definition 2.2: Model of CTL

A transition system $T = (S, R_t, V, I)$ is a tuple where S is a non-empty set of states, $R_t \subseteq S \times S$ is a serial transition relation, $V : S \to s^{\mathcal{VP}}$ is an evaluation function, and $I \subseteq S$ is the set of initial states. A path π in T is an infinite sequence $\pi = (s_0, s_1, ...)$ of states such that $(s_i, s_{i+1}) \in R_t \ \forall i \geq 0$.

Definition 2.3: Semantics of CTL

Given the model M, the satisfaction for a CTL formula φ in a global state s, denoted in the standard notation $(M, s) \models \varphi$, is recursively defined as follows:

- $(M, s) \models p \iff p \in V(s)$;
- $(M, s) \models \neg\varphi \iff (M, s) \not\models \varphi$;
- $(M, s) \models \varphi_1 \vee \varphi_2 \iff (M, s) \models \varphi_1$ or $(M, s) \models \varphi_2$;
- $(M, s) \models EX\varphi \iff$ there exists a path π starting at s such that $(M, \pi(1)) \models \varphi$;
- $(M, s) \models E(\varphi_1 \cup \varphi_2) \iff$ there exists a path π starting at s for some $k \geq 0$, $(M, \pi(k)) \models \varphi_2$ and $\forall\ 0 \leq i < k$, $(M, \pi(i)) \models \varphi_1$;
- $(M, s) \models EG\varphi \iff$ there exists a path π starting at s such that $(M, \pi(k)) \models \varphi, \forall\ k \leq 0$.

2.3 Extensions to CTL

In this section, we present all the different logics (extensions to CTL) that the tool has been used for and currently supports.

2.3.1 CTLKC$^+$

CTLKC$^+$, the logic of knowledge and commitments, has been first presented by Al-Saqqar and al. [1] as a new logic to capture the interaction between knowledge and commitment.

Definition 2.4: Syntax of CTLKC$^+$

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid EX\varphi \mid E(\varphi \cup \varphi) \mid EG\varphi \mid K_i\varphi \mid C_{i\to j}\varphi \mid Fu(C_{i\to j}\varphi).$$

where:

- $p \in \mathcal{VP}$ is an atomic proposition;
- E is the existential quantifier on paths;
- X, G, and U are CTL path modal connectives;
- The modal connective K_i stands for 'knowledge of agent i';
- The modal connective $C_{i\to j}$ stands for 'commitment from agent i to agent j';
- Fu stands for 'fulfillment'.

Definition 2.5: Model of CTLKC$^+$

A Model $\mathcal{M} = (S, I, R_t, \{\approx_i \mid i \in \mathcal{A}\}, \{\approx_{i\to j} \mid (i,j) \in \mathcal{A}^2\}, \mathcal{V})$ that belongs to the set of all models $\mathbb{M}$ is a tuple, where:

- $S \subseteq L_1 \times ... \times L_n$ is the set of reachable global states for the system;
- $I \subseteq S$ is the set of initial global states;
- $R_t \subseteq S \times S$ is the transition relation defined by $(s_1, s_2 \in R_t)$ iff there exists a joint action $a \in ACT$ such that $\tau(s_1, a) = s_2$;
- $\forall i \in \mathcal{A}$,$\approx_i \subseteq S \times S$ is the epistemic accessibility relationship defined by $s \approx_i s'$ iff $l_i(s) = l_i(s')$;
- $\forall (i,j) \in \mathcal{A}^2$,$\approx_{i\to j} \subseteq S \times S$ is the social accessibility relationship defined by $s \approx_{i\to j} s'$ iff $Var_i \cap Var_j \neq \emptyset$, and $\forall x \in Var_i \cap Var_j : l_i^x(s) = l_i^x(s') = l_j^x(s')$;

- $\mathcal{V} : S \to 2^{\mathcal{VP}}$ is the valuation function where $\mathcal{VP}$ is the set of atomic propositions.

The epistemic accessibility relation $\approx_i$ captures the intuition that two states s and s' are equivalent for agent i (in terms of knowledge). Its also worth noting that the epistemic relation is an equivalence relation (i.e., it's reflexive, symmetric, and transitive).

On the other hand, the social accessibility relation $\approx_{i \to j}$ between two global states s and s' is that the two agents have a communication channel (modeled through shared variables), so that agent i sends information(in the form of a message) through the channel in s, and agent j receives said information in s'. This communication between the two said agents results in the shared variables between the two to have the same values. However, this formalism imposes no constraints on the unshared variables as they can be involved for other communications at the same time from other agents (adapted from [1]).

Definition 2.6: Semantics of CTLKC$^+$

The semantics of CTLKC$^+$ is as follows. Given a model $\mathcal{M}$, the satisfaction of a CTLKC$^+$ formula φ in a global state s, denoted by $(\mathcal{M}, s) \models \varphi$ is defined recursively as follows:

- $(\mathcal{M}, s) \models p$ iff $p \in \mathcal{V}$;
- $(\mathcal{M}, s) \models \neg\varphi$ iff $(\mathcal{M}, s) \not\models \varphi$;
- $(\mathcal{M}, s) \models \varphi_1 \vee \varphi_2$ iff $(\mathcal{M}, s) \models \varphi_1$ or $(\mathcal{M}, s) \models \varphi_2$;
- $(\mathcal{M}, s) \models EX\varphi$ iff there exists a path π starting at s such that $(\mathcal{M}, \pi(1)) \models \varphi$;
- $(\mathcal{M}, s) \models E(\varphi_1 \cup \varphi_2)$ iff there exists a path π starting at s for some $k \geq 0$, $(\mathcal{M}, \pi(k)) \models \varphi_2$ and $\forall\ 0 \leq i < k$, $(\mathcal{M}, \pi(i)) \models \varphi_1$;
- $(\mathcal{M}, s) \models EG\varphi$ iff there exists a path π starting at s such that $(\mathcal{M}, \pi(k)) \models \varphi$, $\forall\ k \geq 0$;
- $(\mathcal{M}, s) \models K_i\varphi$ iff $\forall s' \in S$ such that $s \approx_i s'$, we have $(\mathcal{M}, s') \models \varphi$;

- $(\mathcal{M}, s) \models C_{i \to j}\varphi$ iff $\forall s' \in S$ such that $s \approx_{i \to j} s'$, we have $(\mathcal{M}, s') \models K_i\varphi$ and $(\mathcal{M}, s') \models K_j\varphi$;
- $(\mathcal{M}, s) \models Fu(C_{i \to j}\varphi)$ iff $\exists s' \in S$ such that $s' \approx_{i \to j} s$ and $(\mathcal{M}, s') \models C_{i \to j}\varphi$ or $\exists s'' \in S$ and $s'' \approx_i s$ such that $(\mathcal{M}, s'') \models Fu(C_{i \to j}\varphi)$ or $\exists s'' \in S$ and $s'' \approx_j s$ such that $(\mathcal{M}, s'') \models Fu(C_{i \to j}\varphi)$.

2.3.2 RTCTL^{cc}

RTCTL^{cc} [33] is a logic for real-time conditional commitments that extends CTL with conditional commitment modalities, fulfillment modalities, as well as timing constraints. It allows for modeling MASs in environments equipped with the proper mechanisms to react to events happening at precise time instants or time intervals.

Definition 2.7: Syntax of RTCTL^{cc}

- $\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid EX\varphi \mid E(\varphi \cup \varphi) \mid EG\varphi \mid E(\varphi \cup^{[m,n]} \varphi) \mid A(\varphi \cup^{[m,n]} \varphi) \mid CC \mid Fu$;
- $CC ::= WCC(i, j, \varphi, \varphi) \mid SCC(i, j, \varphi, \varphi)$;
- $Fu ::= FuW(i, WCC(i, j, \varphi, \varphi)) \mid FuS(i, SCC(i, j, \varphi, \varphi))$.

where:

- $p \in \mathcal{VP}$ is an atomic proposition;
- E is the existential quantifier on paths;
- X, G and U are CTL path modal connectives;
- $m, n \in \mathbb{N}^+$ denote the bounds of the time interval($n \geq m$);
- $\cup^{[m,n]}$ stands for bounded until;
- The modal connectives WCC, SCC, FuW, and FuS stand for weak and strong conditional commitments, and their fulfillment respectively.

Definition 2.8: Model of RTCTL^{cc}

A Model $\mathcal{M} = (S, I, T, \{\sim_{i\to j} |(i,j) \in \mathcal{A}^2\}, \mathcal{V})$ that belongs to the set of all models $\mathbb{M}$ is a tuple, where:

- $S \subseteq L_1 \times ... \times L_n \times L_e$ is the set of reachable global states for the system;
- $I \subseteq S$ is the set of initial global states;
- $T \subseteq S \times S$ is the transition relation defined by $(s_1, s_2 \in T)$ iff there exists a joint action $a \in ACT$ such that $\tau(s_1, a, a_e) = s_2$;
- $\forall (i,j) \in \mathcal{A}^2, \sim_{i\to j} \subseteq S \times S$ is the social accessibility relationship defined by $s \sim_{i\to j} s'$ iff:
  - $l_i(s) = l_i(s')$;
  - $(s, s') \in T$;
  - $Var_i \cap Var_j \neq \emptyset$, and $\forall x \in (Var_i \cap Var_j) : l_i^x(s) = l_j^x(s')$;
  - $\forall y \in Var_j - Var_i : l_j^y(s) = l_j^y(s')$.
- $\mathcal{V} : S \to 2^{\mathcal{VP}}$ is the labeling function where $\mathcal{VP}$ is the set of atomic propositions.

Definition 2.9: Semantics of RTCTL^{cc}

Given a model $\mathcal{M}$,the satisfaction of a RTCTL^{cc} formula φ in a global state s, denoted by $(\mathcal{M}, s) \models \varphi$ is defined recursively as follows:

- $(\mathcal{M}, s) \models p$ iff $p \in \mathcal{V}$;
- $(\mathcal{M}, s) \models \neg\varphi$ iff $(\mathcal{M}, s) \not\models \varphi$;
- $(\mathcal{M}, s) \models \varphi_1 \vee \varphi_2$ iff $(\mathcal{M}, s) \models \varphi_1$ or $(\mathcal{M}, s) \models \varphi_2$;
- $(\mathcal{M}, s) \models EX\varphi$ iff there exists a path π starting at s such that $(\mathcal{M}, \pi(1)) \models \varphi$;
- $(\mathcal{M}, s) \models E(\varphi_1 \cup \varphi_2)$ iff there exists a path π starting at s for some $k \geq 0$, $(\mathcal{M}, \pi(k)) \models \varphi_2$ and $\forall\ 0 \leq i < k$, $(\mathcal{M}, \pi(i)) \models \varphi_1$;
- $(\mathcal{M}, s) \models EG\varphi$ iff there exists a path π starting at s such that $(\mathcal{M}, \pi(k)) \models \varphi$, $\forall\ k \geq 0$;

- $(\mathcal{M}, s) \models E(\varphi \cup^{[m,n]} \psi)$ iff there exists a pathπ such that $\exists i \in [m, n]$,$(\mathcal{M}, \pi(i)) \models \psi$ and $\forall j \in [m, i[$,$(\mathcal{M}, \pi(j)) \models \varphi$;
- $(\mathcal{M}, s) \models A(\varphi \cup^{[m,n]} \psi)$ iff $\forall \pi$ such that $\exists i \in [m, n], (\mathcal{M}, \pi(i)) \models \psi$ and $\forall j \in [m, i[$,$(\mathcal{M}, \pi(j)) \models \varphi$;
- $(\mathcal{M}, s) \models WCC(i, j, \psi, \varphi)$ iff $\forall s' \in S$ such that $s \sim_{i \to j} s'$ and $(\mathcal{M}, s) \models \psi$, $(\mathcal{M}, s') \models \varphi$;
- $(\mathcal{M}, s) \models SCC(i, j, \psi, \varphi)$ iff $\exists s' \in S$ such that $s \sim_{i \to j} s'$ and $(\mathcal{M}, s) \models \psi$, and $(\mathcal{M}, s) \models WCC(i, j, \psi, \varphi)$;
- $(\mathcal{M}, s) \models FuW(i, WCC(i, j, \psi, \varphi))$ iff $\exists s' \in S$ such that $s' \sim_{i \to j} s$ and $(\mathcal{M}, s') \models WCC(i, j, \psi, \varphi)$ and $(\mathcal{M}, s) \models \varphi \vee \neg WCC(i, j, \psi, \varphi)$;
- $(\mathcal{M}, s) \models FuS(i, SCC(i, j, \psi, \varphi))$ iff $\exists s' \in S$ such that $s' \sim_{i \to j} s$ and $(\mathcal{M}, s') \models WCC(i, j, \psi, \varphi)$ and $(\mathcal{M}, s) \models \psi \vee \neg SCC(i, j, \psi, \varphi)$.

2.3.3 TCTL

Trust Computation Tree Logic (TCTL, for short) [16] is a combination of CTL with trust modalities to reason about trust and time. This logic is an extension of the CTL logic with a new operator for trust, along with its intuitive semantics, to effectively model trust interactions.

Definition 2.10: Syntax of TCTL

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid EX\varphi \mid E(\varphi \cup \varphi) \mid EG\varphi \mid T_p(i, j, \varphi, \varphi)$$

where:

- $p \in \mathcal{VP}$ is an atomic proposition;
- E is the existential quantifier on paths;
- X, G and U are CTL path modal connectives;
- The modal connective $T_p(i, j, \varphi, \psi)$ stands for 'Preconditional Trust' and is read as *"the truster i trusts the trustee j to bring about ψ if the condition φ holds"*.

Definition 2.11: Model of TCTL

A model of trust $\mathcal{M}$ is a tuple $\mathcal{M} = (S, R, I, \{\sim_{i\to j} |(i,j) \in \mathcal{A}^2\}, \mathcal{V})$, where:

- $S \subseteq L_1 \times ... \times L_n \times L_e$ is the set of reachable global states for the system;
- $I \subseteq S$ is the set of initial global states;
- $R \subseteq S \times S$ is the transition relation defined by $(s_1, s_2 \in R)$ iff there exists a joint action $a \in ACT$ such that $\tau(s_1, a, a_e) = s_2$;
- $\forall (i,j) \in \mathcal{A}^2, \sim_{i\to j} \subseteq S \times S$ is the direct trust accessibility relation for each truster-trustee pair of agents defined by $s \sim_{i\to j} s'$ iff:
  - $l_i(s)(v^i(j)) = l_i(s')(v^i(j))$;
  - s' is reachable from s using transitions from R.
- $\mathcal{V} : S \to 2^{\mathcal{VP}}$ is the labeling function where $\mathcal{VP}$ is the set of atomic propositions.

The intuition behind the trust accessibility relation $\sim_{i\to j}$ is that, for agent i to gain trust in agent j, the former (agent i) identifies the states where he is expecting agent j to be trustful.

Definition 2.12: Semantics of TCTL

Given a Model $\mathcal{M}$, the satisfaction of a TCTL formula φ in a global state s, denoted by $(\mathcal{M}, s) \models \varphi$, is defined recursively as follows:

- $(\mathcal{M}, s) \models p$ iff $p \in \mathcal{V}$;
- $(\mathcal{M}, s) \models \neg\varphi$ iff $(\mathcal{M}, s) \not\models \varphi$;
- $(\mathcal{M}, s) \models \varphi_1 \vee \varphi_2$ iff $(\mathcal{M}, s) \models \varphi_1$ or $(\mathcal{M}, s) \models \varphi_2$;
- $(\mathcal{M}, s) \models EX\varphi$ iff there exists a path π starting at s such that $(\mathcal{M}, \pi(1)) \models \varphi$;
- $(\mathcal{M}, s) \models E(\varphi_1 \cup \varphi_2)$ iff there exists a path π starting at s for some $k \geq 0$, $(\mathcal{M}, \pi(k)) \models \varphi_2$ and $\forall\ 0 \leq i < k$, $(\mathcal{M}, \pi(i)) \models \varphi_1$;
- $(\mathcal{M}, s) \models EG\varphi$ iff there exists a path π starting at s such that $(\mathcal{M}, \pi(k)) \models \varphi$, $\forall\ k \geq 0$;

- $(\mathcal{M}, s) \models T_p(i, j, \psi, \varphi)$ iff $(\mathcal{M}, s) \models \psi \wedge \neg\varphi$ and $\exists s' \neq s$ such that $s \sim_{i \to j} s'$, and $\forall s' \neq s$such that $s \sim_{i \to j} s'$, we have :$(\mathcal{M}, s') \models \varphi$.

2.3.4 TCTL^C

TCTL^c is an extension to TCTL with new modalities to express conditional trust [14].

Definition 2.13: Syntax of TCTL^c

The syntax of $TCTL^c$ is the same as that of TCTL in **Definition 2.10**. The only addition is the operator for conditional trust: $T_c(i, j, \psi, \varphi)$, read as *"agent i trusts agent j about the consequent φ when the antecedent ψ holds."*

Definition 2.14: Model of TCTL^c

TCTL^c has the same model of TCTL, described in **Definition 2.11**.

Definition 2.15: Semantics of TCTL^c

The semantics of TCTL^c, for all shared modalities with TCTL, are the same. The semantics for $T_c(i, j, \psi, \varphi)$ however are as follows:

- $(\mathcal{M}, s) \models T_c(i, j, \psi, \varphi)$ iff $\exists s' \neq s$ such that $s \sim_{i \to j} s'$ and $s' \models \psi$,and $\forall s' \neq s$ such that $s \sim_{i \to j} s'$ and $(\mathcal{M}, s') \models \psi$, we have $(\mathcal{M}, s') \models \varphi$.

2.4 Model Checking

Due to their nature, and their innate structure, MASs are actively used to model safety-critical systems. These systems (e.g., medical systems, embedded controllers, avionic systems) usually present characteristics that make them very hard to design correctly (embedded, reactive, concurrent, real-time..) in comparison to "classical" computer systems. Hence the need for powerful and reliable verification methods to check for compliance with specifications and requirements.

Verification includes several techniques that can be used, e.g., inspections, testing, simulation, and formal verification. None of these techniques is absolutely superior to others; they each have their advantages, disadvantages, and domains of applications.

In this dissertation, we focus on formal verification, which, in comparison to other methods, gives the highest assurance of system correctness. There exist two basic approaches to formal verification:

- **Deductive methods** aim to produce mathematical proofs that the system satisfies requirements. Although this process can be partially automated (for simple proofs), it still requires manual proof construction using deductive methods to be performed by experts. This process is time-consuming and challenging to scale for large systems.

- **Automatic methods**: These methods use brute-force to try and test *all* possible behaviors of the system, and validate that they all satisfy the requirement. By taking the brute-force approach, these techniques gain the advantage of being fully automatic. There is, of course, a disadvantage that has to be paid: the high computational requirements for these methods.

Model Checking is one of the most used automatic verification methods. The next section explains the concepts behind it.

2.4.1 The Model Checking Problem

In a nutshell, model checking is the problem of checking if, given a Model $\mathcal{M}$ (representing, for example, a hardware or software model), and a temporal logic formula φ (representing a specification), to check if the Model $\mathcal{M}$ satisfies the logical formula φ. Model checkers typically have three main components [12]:

1. A specification language, based on a temporal logic.

2. A method of encoding the state machine, representing the system under verification.

3. A verification procedure that uses an exhaustive search to determine if the specification is met.

Model checkers, in general, terminate with a "true" answer, or they terminate by providing a counterexample demonstrating inconsistent behavior with the specification. Furthermore, most modern model checkers offer even the possibility to produce witness examples in the case where the specification is valid (cf. Figure 2.1).

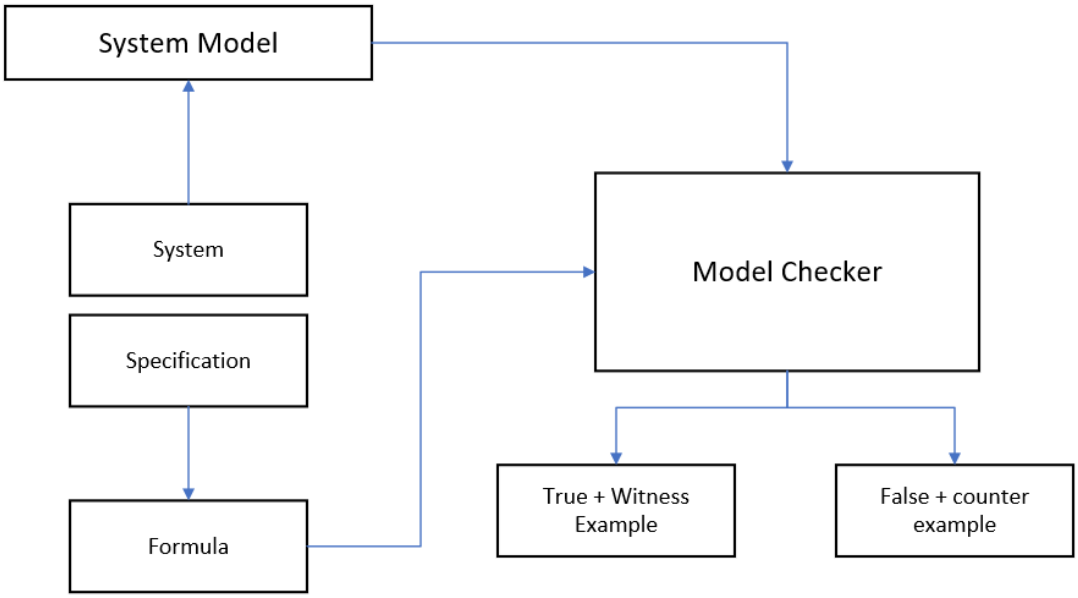

Figure 2.1: A typical model checker with witness and counterexamples.

It is impossible to talk about model checking without mentioning the **state explosion problem**. It is a phenomenon that happens when using model checking based on automata-theoretic techniques. The main issue is that the number of states grows exponentially with the number of variables in the system. In fact, it is the principal driving force behind much of the research in model checking, be it about approaches or new model checkers.

2.4.2 Symbolic Model Checking

Symbolic model checking using OBDDs is one of the solutions to the state explosion problem. The idea is to use data structures called **Binary Decision Diagrams** (BDDs) [32] to represent the state space symbolically (like a sort of canonical form of Boolean formulae); they are finite directed acyclic graphs (DAGs) with some interesting properties. The most important of which is that they require fewer states to represent the state space. A reduced BDD (RBDD) [23] is a BDD that has undergone optimizations repeatedly until reaching a fixed point, and generally results in a quite compact representation of Boolean functions. An ordered BDD (OBDD) is a

BDD with an ordering for some list of variables. What's more interesting is that an ROBDD representing a given Boolean function f is ***unique*** [9].

The main idea behind the usage of these data structures is that they allow the manipulation of entire sets of states at a time, allowing for more efficient operations. These properties are used in model checking CTL by representing sets of states symbolically, namely, those that satisfy the formula being checked. The problem of model checking CTL then becomes the problem of constructing the set of states satisfying a formula φ: $||\varphi||$ in OBDDs, and then comparing it to the set of initial states I represented in OBDDs as well, so that if $I \subseteq ||\varphi||$ then the formula is true.

2.4.3 Model Checking Tools

A huge effort has been, and is being devoted to the development of model checking tools (aka model checkers), by both academic and industrial teams, in order to verify larger models and deal with a big selection of extended frameworks. In this section, we will introduce and summarize two tremendous tools with their capabilities and specification languages.

A NuSMV

NuSMV [25] is a state-of-the-art symbolic model checker written in the C language. Its an effort to rethink the Symbolic Model Verifier, **CMU SMV**(the first model checker based on BDDs). It constitutes both a redesign and an extension of SMV. NuSMV was designed to be robust and close to industrial systems standards [11], but also to be an open, extensible platform for model checking. This aspect has been improved with the recent changes in version 2.6 (the latest major release), where the model checker has been split into two distinct parts: the core engine and the interactive shell. The goal is to allow the core module to be the basis of custom verification tools by allowing for easier integration with existing architectures. NuSMV also offers the capability to perform SAT-based bounded model checking by offering SAT-based techniques in addition to the more classical BDD-based ones [10] (cf. Figure 2.2). The BDD-based model checking functionalities use the CUDD library written in the C language [44]. The currently available SAT solvers for NuSMV are ZCHAFF SAT [45] and MINISAT [20]. It's worth noting that NuSMV is distributed under the LGPL.

NuSMV takes as input files written in its own extension of the SMV language. It

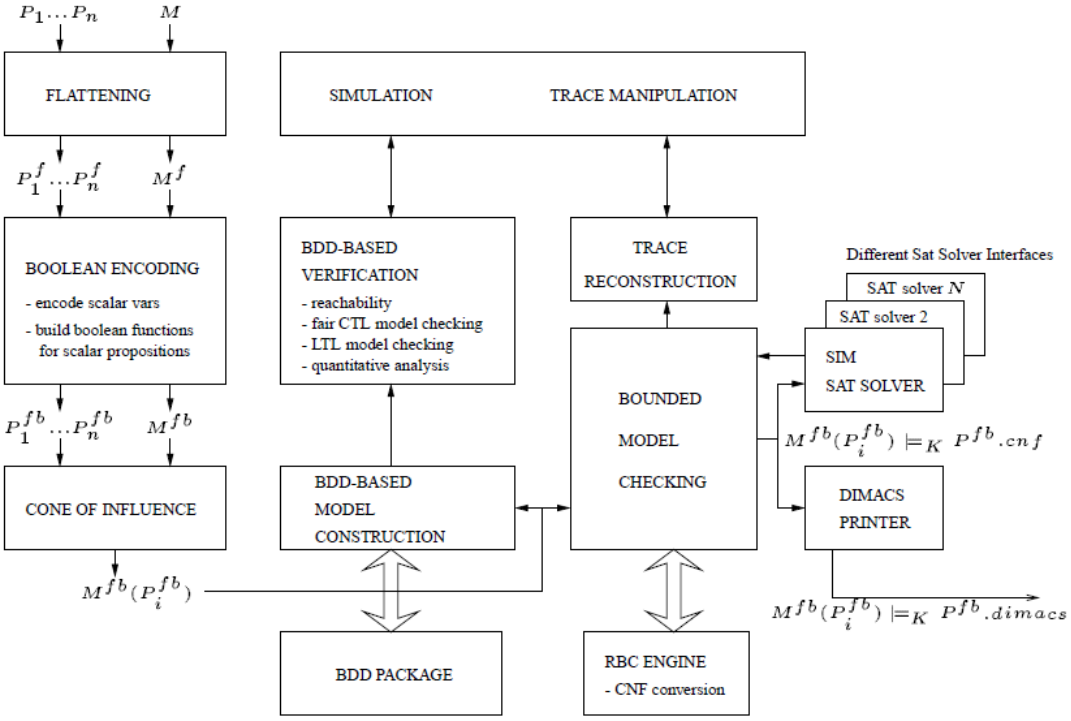

Figure 2.2: The internal structure of NuSMV.

allows the description of modules and processes and the expression of specifications in CTL and LTL logics. The easiest way to use NuSMV is through its interactive shell. A batch mode is also available, but it offers minimal parameters and is the primary method for interacting with outside tools. However, it has the inconvenience of defining a set of operations to be performed in a sequential matter without much possibility for change, making most integrations with NuSMV shallow, unless they actually extend the NuSMV source code. It is also worth mentioning that NuSMV is one of the most powerful model checkers available (NASA has used NuSMV to verify properties of models with over 10^{120} reachable states [49]), and is the result of years of research and improvements. However, the main deterrent from using it in the MAS community is that it does not support any MAS logic. The process of extending NuSMV is not an easy endeavor, making most integrations with it shallow at best.

B MCMAS

The model checker for multi-agent systems (MCMAS for short) [29] is a model checker dedicated to MASs developed at the end of 2005. It's an OBDD-based model checker that can take as input CTL, as well as epistemic, correctness, and cooperation modalities that are specific to MASs. MCMAS uses the Interpreted Systems formalism discussed in **Section 2.1** of the current chapter. The dedicated programming language that MCMAS uses as input is called ISPL (Interpreted Systems Programming Language) and is a text specification describing a system in the interpreted systems formalism. MCMAS is implemented in C++ and uses the CUDD library as well for its BDD-based algorithms.

MCMAS has been extended to handle many logics for MASs. It has been extended into MCMAS+ to deal with social commitments [27]. It has also been used as the core basis for SMC4AC, a model checker recently launched for intelligent agent communication [26]. Moreover, MCMAS has been extended recently into MCMAS-T (Trust-extended MCMAS) [16] to handle the grammar of TCTL (Trust-extended CTL) and its model checking. Further information can be found at **https://vas.doc.ic.ac.uk/software/mcmas/**

2.5 Related Work

In this section, we go over the available tools for different model checkers to review their features and available properties. Since there is no available tooling currently available that's remotely related to what we are trying to achieve with this work, we will simply review features of other tools to allow for better understanding.

2.5.1 gNuSMV

gNuSMV is a basic graphical user interface (GUI from now on) for NuSMV, developed using Python and built on top of the interactive shell. It was built to be a separate process that communicates with NuSMV through the interactive shell [11].

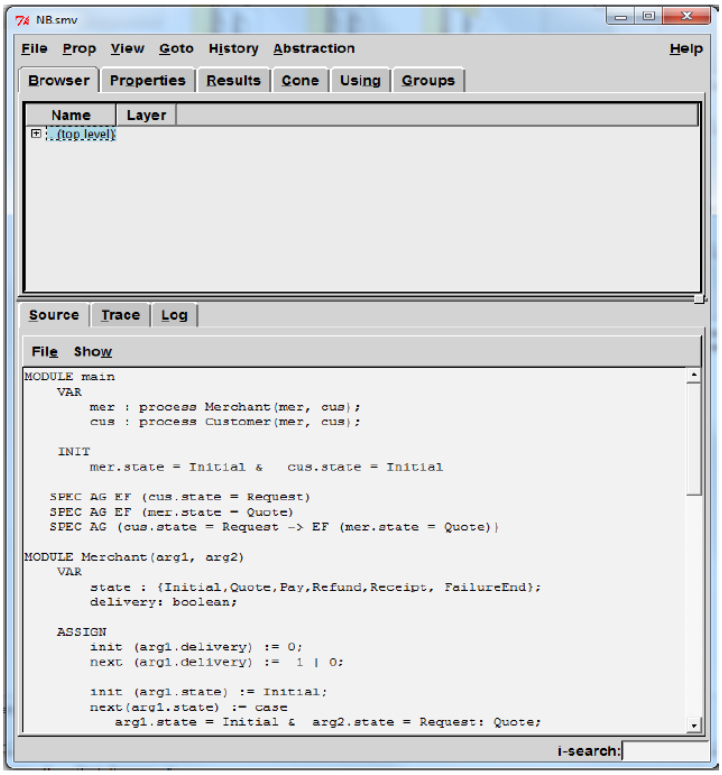

Figure 2.3: gNuSMV model editor window.

The GUI allows the user to edit and modify the file containing the model's description. It also offers several menus to interact with NuSMV in an easier way. This includes:

- An editor window provides basic editing functionalities for the input model file (cf. Figure 2.3). Although it doesn't offer any syntax highlighting or auto-completion, it was the only option offered on the market till a few years ago.
- A formula editor window allowing the user to write new specifications. The available options change in function of the kind of formula to verify. It offers various buttons with the preset temporal modalities suitable for the type of formula created (cf. Figure 2.4).

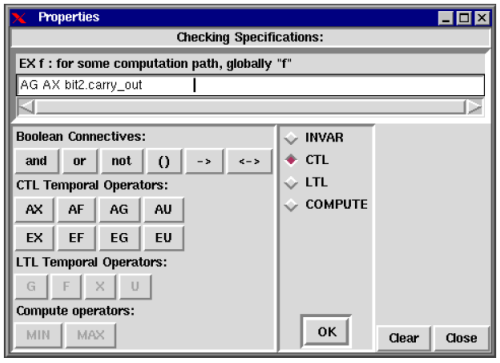

Figure 2.4: gNuSMV formula editor window for CTL.

gNuSMV was developed for NuSMV v1 and has been released for NuSMV v2 as well, but all it still offers is the same basic functions it did since v1, to the point that even the actual snapshots for the two are almost indistinguishable. It also suffers from its design being mostly undocumented and hard to extend despite using a high-level programming language like Python. gNuSMV2 is also behind NuSMV in its development process, as it still has yet to update its settings with the new changes introduced with NuSMV version, including the new heuristics, and the new ordering techniques. It has also dropped ZCHAFF SAT support due to licensing restrictions in order to keep both the Windows and Linux versions aligned.

2.5.2 NuSeen

NuSeen is an eclipse-based environment for NuSMV, that aims of helping NuSMV users by offering a set of useful tools to bridge the gap in available tooling support for NuSMV. The project started in 2013, but it wasn't till the latest NuSMV version

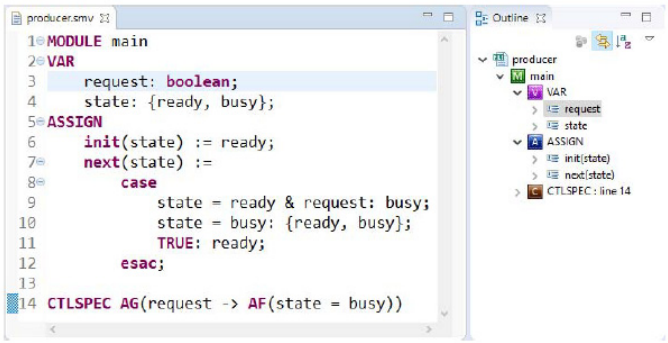

Figure 2.5: NuSeen model editor.

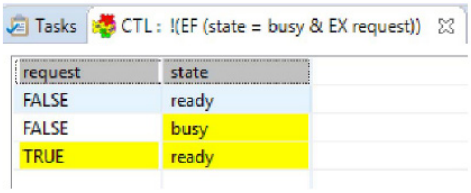

Figure 2.6: NuSeen counterexample tabular format.

(v2.6) and the significant design changes it introduced, that NuSeen was able to finally integrate with NuSMV in order to offer a more extended set of features.

NuSeen mainly focuses on easing the use of NuSMV through the use of graphical elements, menus, text highlighting, and so on. It features [2]:

- An editor (cf. Figure 2.5) that can be used for editing NuSMV models. The editor is based on a grammar (concrete syntax) based on a metamodel for NuSMV input generated from XText in the EMF (Eclipse metamodeling framework) model format. It provides useful features to the user, such as syntax highlighting and context-aware auto-completion based on the outline.

- A counterexample visualizer (cf. Figure 2.6) that automatically detects when a formula is false and intercepts NuSMV output to reformat it into a tabular format.

- A way to run NuSMV from inside eclipse (cf. Figure 2.7) using one of two options: (1) The batch mode of NuSMV to run NuSMV and return its feedback

in an automatic manner; or (2) the interactive mode, that is run from the eclipse terminal. However, the second mode does not offer any extra information or ease of use compared to the regular interactive shell of NuSMV. All actions have to still be done manually using NuSMV shell commands.

- A model advisor that checks a pre-defined set of properties called meta-properties on any model to help inform the user of some usual red flags. The model advisor supports many properties such as consistency, completeness, minimality, etc.
- A test case generator from a given model. The tool currently supports value coverage and decision coverage.

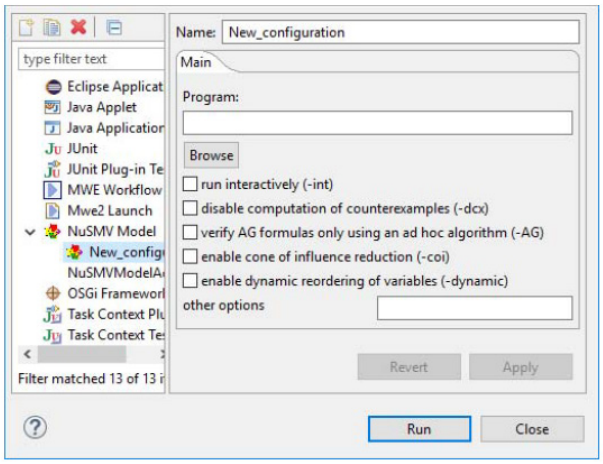

Figure 2.7: NuSeen executor menu.

NuSeen is by far the best NuSMV tool up to the moment this thesis is written. Although it has yet to achieve full integration with NuSMV, the set of tools it offers are highly interesting from both the tooling and validation/verification perspectives.

2.5.3 MCMAS Eclipse Plug-in

MCMAS offers an Eclipse plugin as well. It offers an interesting set of features [29]:

- It is equipped with ISPL program editing through a model editor with syntax highlighting. It allows the user to edit ISPL models for MCMAS from within

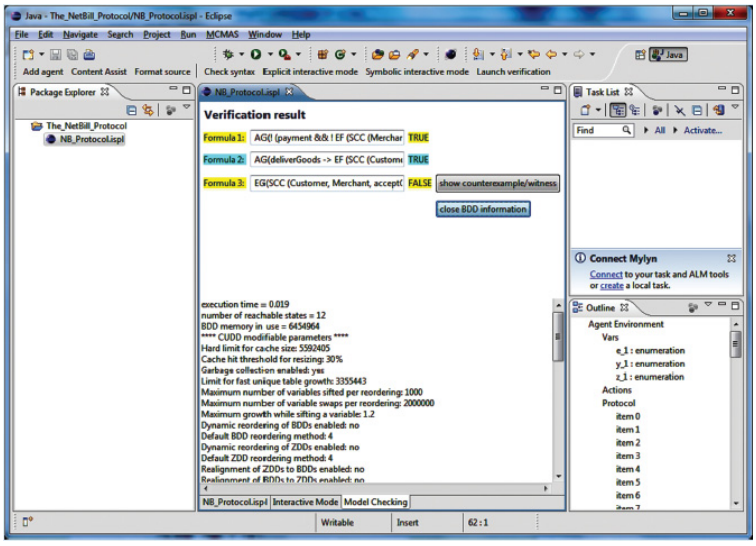

Figure 2.8: MCMAS plug-in verification.

the eclipse editor. It also offers dynamic syntax checking capabilities based using the ANTLR (Another Tool for Language Recognition) framework.

- It offers the ability to run the model in the interactive execution mode of MCMAS (launched with the option "$-s$"). It allows users to execute the model step by step by choosing an initial state and selecting from the set of reachable states at every step (cf. Figure 2.8).

- The GUI offers a display of counter and witness examples for formulas using the "`dot`" utility from the Graphviz graph visualization software (cf. Figure 2.9).

Unlike NuSMV, MCMAS offers its own eclipse plugin developed by the same team, and offers full integration with the model checker, meaning that the simulation and interactive mode is fully controlled by the GUI and allows the user to bypass the regular command-line to perform all possible actions in MCMAS from the plugin.

2.6 Summary

In this chapter, we introduced the background and concepts needed for the rest of this dissertation. We also provided a review of the most relevant related work. In the

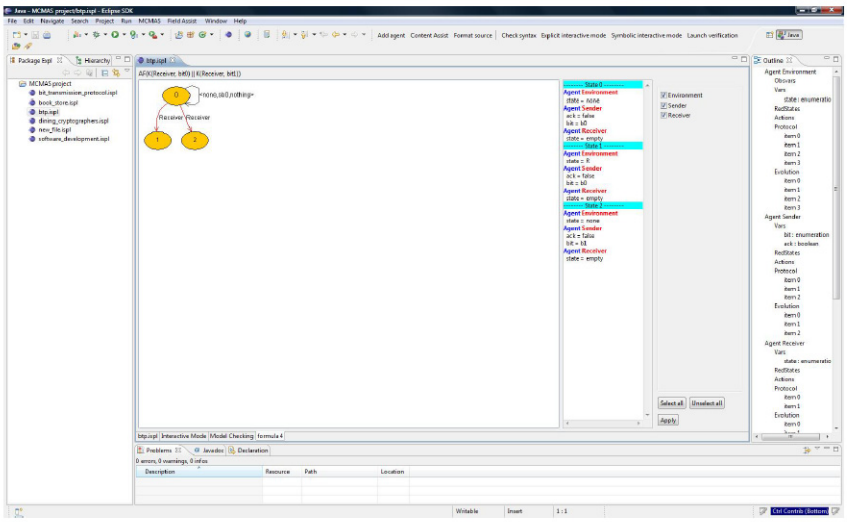

Figure 2.9: MCMAS eclipse plug-in counter-example editor.

next chapter, we present the general approach employed, and present where the tool fits into a typical transformation-based model checking process of a given logic.

Chapter 3

Automated Transformation-based Model Checking MAS

This chapter starts by presenting an overview of the general approach presented in this work to automate transformation-based model checking. Then, an example of such an extension to CTL, namely RTCTL^{cc}, is given with its different transformation algorithms, followed by the model checking results of the entire methodology (published in [33]). Finally, we conclude with a summary of the entire approach.

3.1 An Overview of the General Approach

Figure 3.1 illustrates the overall approach of a transformation-based model checking process. The process consists of three phases. In the first phase, the new logic is defined in terms of model and formulae's semantics, syntax, and structure. Then once the decision to use transformation-based model checking for the new logic is taken, a formal verification technique is created, transforming the problem of model checking the new logic (source logic) into that of model checking an already existing logic (target logic). The choice of the target logic is mainly contingent on the existence of an already established model checker for it. The closeness to the original logic in terms of constructs and modalities is also a desirable treat to have. In the following implementation phase, the different transformation algorithms need to be implemented and then used with an existing model checker for the target logic, such as NuSMV.

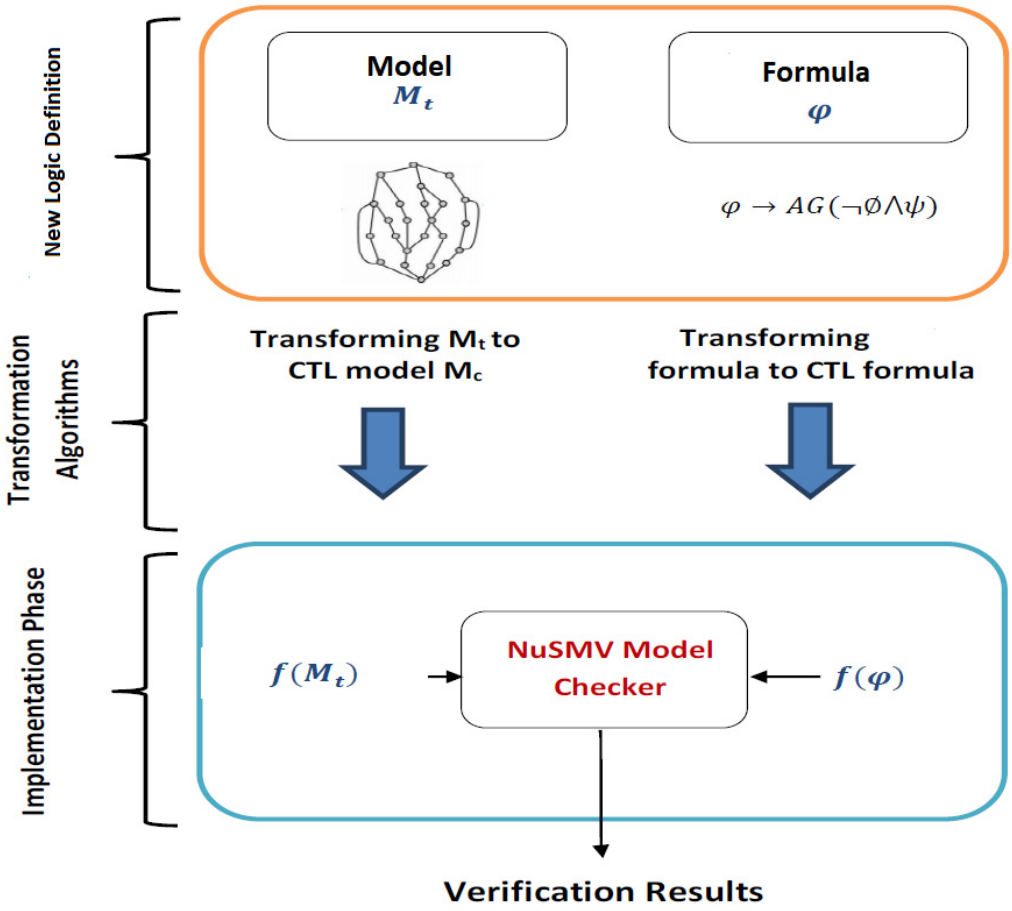

Figure 3.1: A schematic view of a transformation-based model checking.

In a nutshell, given a model $\mathcal{M}$ for a MAS and a formula φ (in the source logic) describing a desirable property, the problem of model checking the source logic can be defined as verifying whether or not φ holds for $\mathcal{M}$, formally denoted $\mathcal{M} \models \varphi$. The transformation-based approach aims to transform the problem of model checking the source logic into that of model checking the target logic, using a set of reduction techniques. The transformation algorithms are developed based on formal reduction methods to provide accurate alignment between source and target logics, while preserving the semantics of the source logic as well as its different model properties.

Technically, after the formal transformation algorithms for both the model and formulae are produced, the implementation phase begins. It consists of creating a parsing system to automate the transformation process of the defined algorithms

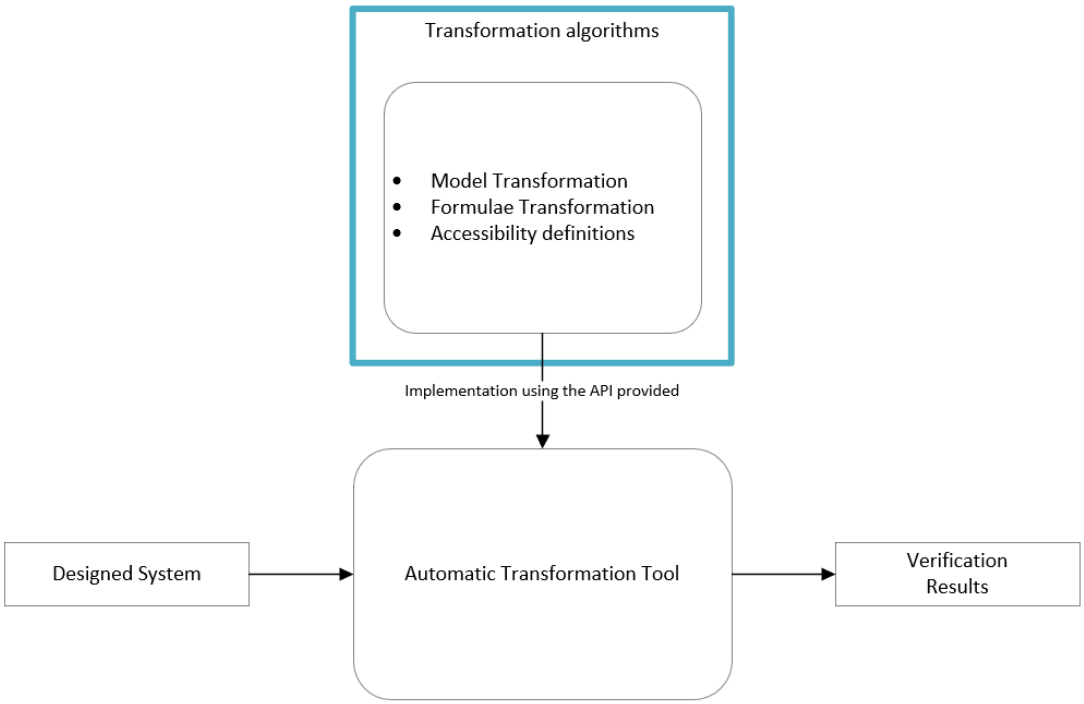

Figure 3.2: A schematic view of the general approach.

for both models and formulae. The process needs to produce a target-valid system (including the model and formulae) that's equivalent to the original system. Finally, the produced system can be manually checked using a model checker that supports the target logic. However, because the design process for MASs is a highly interactive activity, the need to automate the process and integrate the model checker into the workflow becomes of primordial importance. The design of transformation algorithms and their effective implementation, especially when involving integration with a model checker, are highly challenging and prove to be the most time-consuming activities in the entire workflow.

Figure 3.2 summarizes the entire approach proposed in this thesis. The idea is for the transformation and integration tool to offer an entire integrated system to speed-up the implementation phase of the transformation-based methodology. The framework offers an integrated API that helps facilitate the complexity of the entire implementation process considerably. It supports ANTLR generated grammars to simplify further the different tasks for the developer. Once the implementations are

configured into the system, it can be used by the final user to design and check the properties of MASs using the myriad of features the tool offers. Figure 4.4 in Section 4.2.2 goes deeper into the actual implementation activities needed to configure the tool.

3.2 RTCTL^{cc} Example

In this section, we present the problem of model checking the RTCTL^{cc} presented in Section 2.3.2 using the tool. We start by giving the transformation algorithms used, and then we review how the logic is then transformed using the tool.

3.2.1 Transformation Algorithms

To solve the problem of model checking RTCTL^{cc}, we propose to use the transformation-based methodology. This approach offers several advantages. Firstly, it allows the designers to use already existing model checkers, instead of having to create a model checker from scratch and maintain it. Secondly, it constitutes a convenient way of comparing different verification techniques, on the same model checking problem [17]. The approach consists in transforming the RTCTL^{cc} logic into the RTCTL logic [19]. There are two technical reasons behind the choice of RTCTL as the target logic:

- The RTCTL model follows a Kripke structure, the same as an RTCTL^{cc}. Thus, the transformation process can be conducted in logarithmic space.
- The RTCTL logic is already fully backed by the NuSMV model checker, allowing us to take advantage of it.

To transform the model checking problem, we establish two formal transformation algorithms:

- Algorithm 1 automatically transforms an RTCTL^{cc} model $\mathcal{M}$ to an RTCTL model M_t.
- Algorithm 2, on the other hand, recursively transforms an RTCTL^{cc} formula φ into an RTCTL formula $\mathcal{F}(\varphi)$. It is worth mentioning that the transformation of the CTL portion of RTCTL^{cc}, as well as the quantitative formulae (lines 7

and 8), is straightforward. The communicative formulae (lines 9-12), however, are transformed according to the defined semantics.

The proof of the soundness and completeness of the transformation as well as the complexity analysis can be found in [33].

Algorithm 1 An RTCTL^{cc} model $\mathcal{M} = (S, I, T, \{\sim_{i\to j} \mid (i,j) \in \mathcal{A}^2\}, \mathcal{V})$ into an RTCTL Model $M_t = (S_t, I_t, T_t, L_t)$

1: **Input**: the model M

2: **Output**: the model M_t

3: $I_t := I$;

4: $S_t := S$;

5: $L_t : S \to 2^{\mathcal{PV}'}$ where $\mathcal{PV}'$ is defined as the union of the following three sets of atomic propositions (i.e, $\mathcal{PV}' := \mathcal{PV} \cup X \cup Y$):

- The set $PV := \{p, q, ...\}$ of atomic propositions in the model $\mathcal{M}$ to capture the semantics of bounded and unbounded modalities.
- The set $X := \{\alpha^1\alpha^1, \alpha^1\alpha^2, ..., \alpha^n\alpha^n\}$ for the social accessibility relation $\sim_{i\to j}$ to capture the semantics of commitments.
- The set $Y := \{\beta^1\beta^1, \beta^2\beta^1, ..., \beta^n\beta^n\}$ for the symmetric closure of the social accessibility relation $\sim_{i\to j}$ to capture the semantics of fulfillment modalities.

6: The transition relation T_t combines the temporal transition T and asymmetric closures of the accessibility relations under the sequent conditions: for states $s, s' \in S$,

1. If $(s, s') \in T$, then $(s, s') \in T_t$,
2. If $s \sim_{i\to j} s'$, then:
   - If $(s, s') \notin T_t$, then $T_t = T_t \cup \{(s, s')\}$,
   - $L_t(s') := \mathcal{V}(s') \cup \{\alpha^i\alpha^j\}, 1 \le i \le n$ and $1 \le j \le n$, and
   - $L_t(s) := \mathcal{V}(s) \cup \{\beta^i\beta^j\}, 1 \le i \le n$ and $1 \le j \le n$

7: **return** M_t

Algorithm 2 A RTCTL^{cc} formula φ: A RTCTL formula $\mathcal{F}(\varphi)$.

1: $\mathcal{F}(p) = p$ if p is an atomic proposition,
2: $\mathcal{F}(\neg\varphi) = \neg\varphi$,
3: $\mathcal{F}(\varphi_1 \vee \varphi_2) = \mathcal{F}(\varphi_1) \vee \mathcal{F}(\varphi_2)$,
4: $\mathcal{F}(EX\varphi) = EX\mathcal{F}(\varphi)$,
5: $\mathcal{F}(\varphi_1 \cup \varphi_2) = \mathcal{F}(\varphi_1) \cup \mathcal{F}(\varphi_2)$,
6: $\mathcal{F}(EG\varphi) = EG\mathcal{F}(\varphi)$,
7: $\mathcal{F}(E(\varphi \cup^{[m,n]} \psi)) = E(\mathcal{F}(\varphi) \cup^{[m,n]} \mathcal{F}(\psi))$,
8: $\mathcal{F}(A(\varphi \cup^{[m,n]} \psi)) = A(\mathcal{F}(\varphi) \cup^{[m,n]} \mathcal{F}(\psi))$,
9: $\mathcal{F}(WCC(i,j,\psi,\varphi)) = AX((\mathcal{F}(\psi) \wedge \alpha^i\alpha^j) \Rightarrow \mathcal{F}(\varphi))$,
10: $\mathcal{F}(SCC(i,j,\psi,\varphi) = EX(\mathcal{F}(\psi) \wedge \alpha^i\alpha^j) \wedge \mathcal{F}(WCC(i,j,\psi,\varphi))$,
11: $\mathcal{F}(FuW(i, WCC(i,j,\psi,\varphi))) = EX(\beta^i\beta^j \wedge \mathcal{F}(WCC(i,j,\psi,\varphi))) \wedge \mathcal{F}(\varphi) \wedge \neg\mathcal{F}(WCC(i,j,\psi,\varphi))$,
12: $\mathcal{F}(FuS(i, SCC(i,j,\psi,\varphi))) = EX(\beta^i\beta^j \wedge \mathcal{F}(SCC(i,j,\psi,\varphi))) \wedge \mathcal{F}(\psi) \wedge \neg\mathcal{F}(SCC(i,j,\psi,\varphi))$.

3.2.2 Implementation

To simplify the implementation of the algorithms mentioned above, we have used the full capabilities of the tool and the core API. As an example, the following is a step by step description of how we implemented Algorithm 2:

1. Implement a parser for the formulae that generate the intended transformed formulae. The easiest way to do that is by using the provided ANTLR4 integrated capabilities; however, the user is free to choose the most convenient method. The following are the ANTLR4 compatible steps:

   (a) Define an ANTLR grammar file for the formulae. For logics that extend CTL, the tool already provides an extensible CTL ANTLR4 grammar file, and therefore it can be augmented with the definitions of the new modalities.

   (b) Generate ANTLR artifacts (Lexer, Parser, Walker/Visitor) using the ANTLR API.

   (c) Plug-in the generated ANTLR artifacts into the tool.

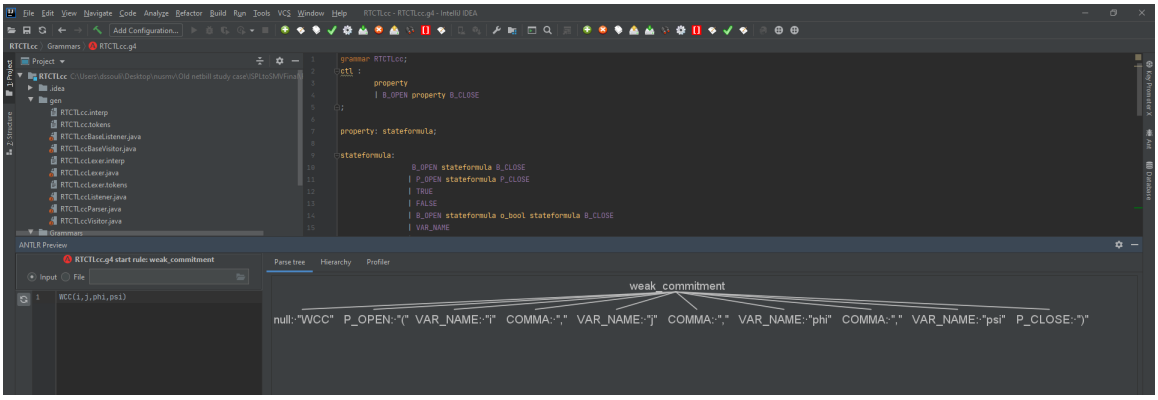

Figure 3.3: ANTLR4 grammar file, parse tree and generated artifacts.

(d) Implement the provided translator API, while making use of the CTL node walking utilities for rules matching the CTL rules.

2. Configure the accessibility engine by implementing the provided interface so that the engine can calculate the accessibility relations automatically.

```
String s = "FuS(j,SCC(i,i,!p, q OR p))";
RTCTLccLexer lexer = new RTCTLccLexer(s);
Stream<Tokens> inputStream =  API.FromANTLR(lexer).createInputStream();
Optional<RTCTLccParser> parser = RTCTLccParser.consume(inputStream);
if (parser.isPresent()){
    parser.get().removeDefaultErrorListeners();
    parser.get().addErrorListener(API.getCTLHelper().makeCTL_ERROR_LISTENER());
    ANTLRParseTree tree = parser.get().generate();
    ANTLRParseTreeWalker walker = new ANTLRParseTreeWalker();
    UtilityTranslator recursive_translator = API.accept(new CustomImplementation());
    CompletableFuture<String> future =  walker.walk(recursive_translator,tree);
    future.thenAccept(System.out::println);
}
```

Figure 3.4: Transformation code snippet.

Figure 3.3 presents an ANTLR4 grammar file, as well as the parse tree created from parsing an RTCTL^{cc}c formula. Notice the 'gen' folder in the project tree on the left, showcasing the different ANTLR generated artifacts. Figure 3.4, on the

other hand, provides a code snippet showcasing how the different described elements described above interact with each other to transform a given formula.

The implementation process of Algorithm 1, on the other hand, is straightforward as both source and target logics follow a Kripke structure. The algorithm then becomes a simple matter of calling the provided Kripke Structure API to initialize the resulting model. Then looping over all the accessibilities calculated from the model and adding the corresponding transition for the symmetric closures if needed.

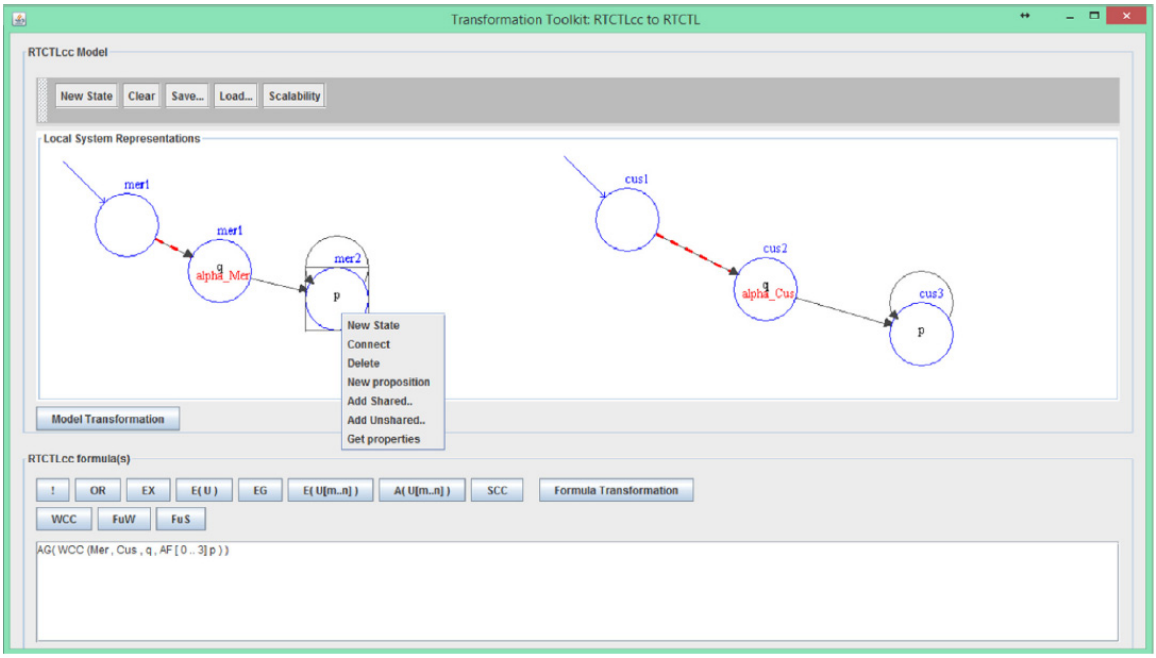

Figure 3.5: The process of specifying a model and a formula using the GUI.

3.2.3 Results

Once both algorithms implemented, the tool is ready to be used for model checking MASs that conform to RTCTL^{cc}. Figure 3.5 depicts the models of two agents, Customer and Merchant, created using the tool's graphical interface (Model Builder v1.0). The dashed red arrows represent the automatically computed accessibility relations between the two agents. Figure 3.6 displays the formula panel and the syntax checker in action. The syntax analyzer uses the parser to check for rule conformance and to suggest solutions to invalid formulae. Once the model and formulae are ready,

we can launch the transformation using the "Model Transformation" button. In Figure 3.7, the transformed system can be seen on the left panel, while the right panel depicts the NuSMV execution results. The execution time for each component is also computed at runtime.

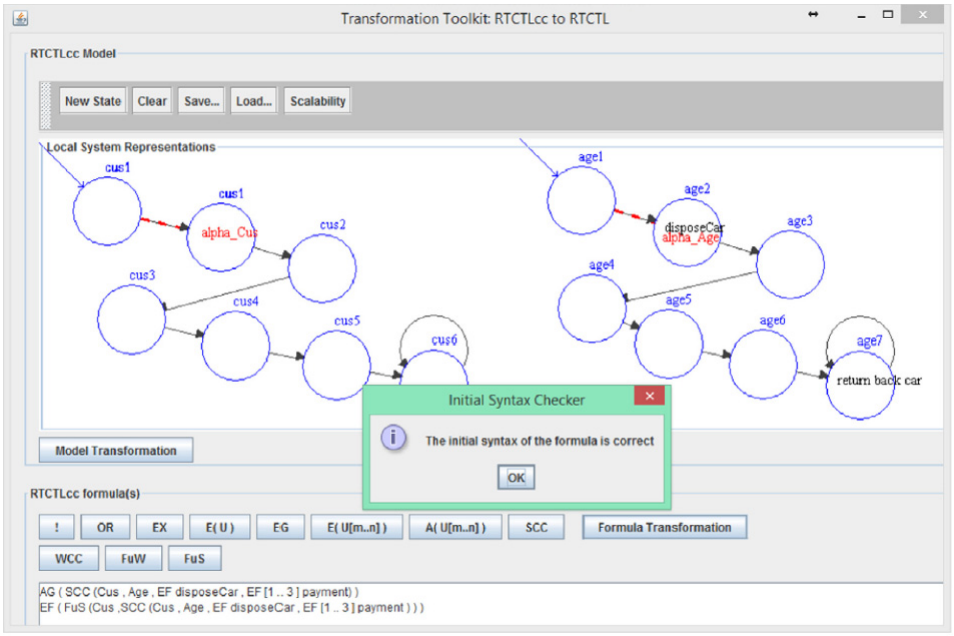

Figure 3.6: Formulae panel during syntax check.

3.3 Summary

In this chapter, we presented the general workflow of the transformation-based model checking activities. Moreover, we situated the involvement of the tool in the life cycle. We then proceeded to present an example of the transformation process of the RTCTL^{cc} logic using the tool. In the next chapter, we will present the implementation of the tool and its general structure.

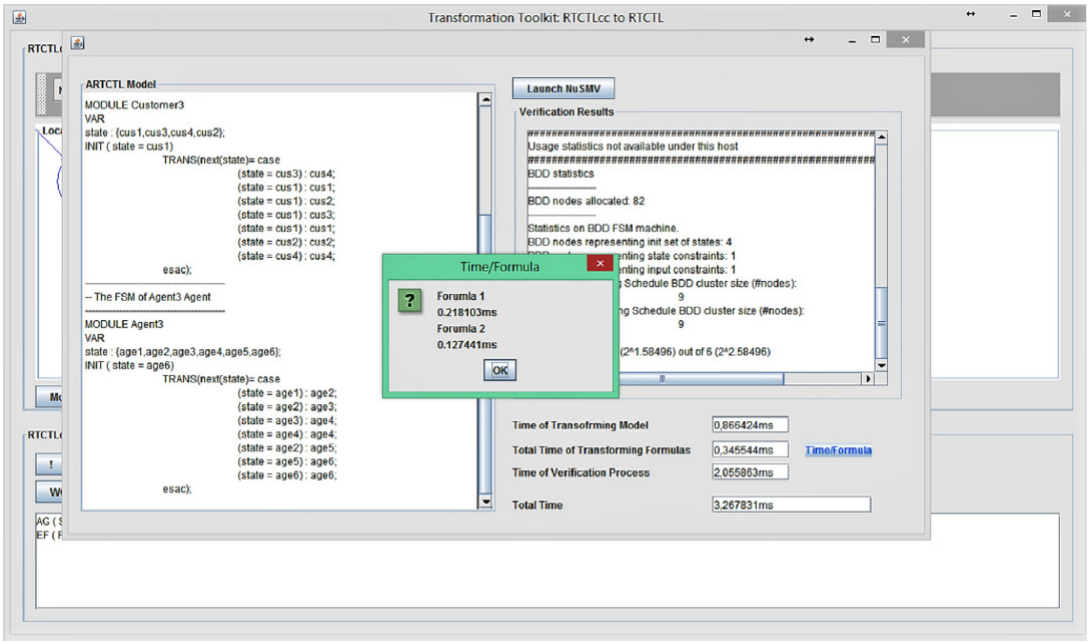

Figure 3.7: Transformation results.

Chapter 4

Implementation

This chapter starts by reviewing the different technologies and libraries used in the implementation of the framework. Then, we present a detailed presentation of the architecture of the tool and its different module components. Then in the last section, we evaluate the efficiency of the tool's performance through the different case studies it was used for, in different publications, and with different logics.

4.1 Technologies Used

This section is an overview of the different technologies and APIs used by the framework.

4.1.1 Java

The Java programming language is the core language for the framework. It's a tried-and-true cross-platform language with years of proven performance, modularity, and flexibility in both industrial and research settings. A hugely active community supports Java and provides a big collection of frameworks and libraries that can deal with most if not every use-case. The recent change to the Java release cycle guarantees that the language is keeping up with the latest trends in programming paradigms, and makes for easier integration with other languages and platforms. The framework has been recently updated to conform to the latest Java Long-term-support version, aka Java 11.

4.1.2 JavaFx

JavaFx, also known as OpenJFX, is an open-source next-generation client application platform for desktop, mobile, and embedded systems built on Java. JavaFX is intended to replace Swing as the standard GUI library for Java SE. It behaves as a GUI library and lends itself to the efficient and rapid development of desktop apps and Rich Internet Applications. JavaFX uses a theater metaphor to address top-level application containers. In FX, the scene graph collects the UI elements, including layouts, controls, shapes, and groups. The elements are referred to as nodes, and each one has automatically available features that the developer can readily access. And FX also has special effects that you can easily add to create blurs, shadows, and other textural touch-ups. FX also offers consistent support for MVC, making the separation of concerns easier and more intuitive, which results in general in modular, reusable code. The earlier versions of the framework implementations were developed using Java's own Swing library (Figure 4.8 shows the old swing interface of the model builder). However, the tool was later upgraded to use JavaFx's components whenever possible(Figure 4.7 shows the new JavaFx interface for the model builder).

4.1.3 Java Native Interface

Java Native Interface, JNI for short, is a foreign function interface that allows code running on the JVM to both call and be called by native applications and libraries, written in C, C++, or assembly [34]. JNI is used in the context of this project to allow for integration with the NuSMV model checker on a source-code level (since it's written in the C language).

Using JNI happens at two different levels:

- On the Java level, native methods and fields as declared using the *native* keyword. Native libraries are then loaded using *System.loadLibrary()* calls (it supports *".so"* files on Linux and *".dll"* files on Windows). The native methods are called just like regular ones, and the mapping between primitive types is straightforward. Native header files are then generated using **javac** by providing the **"-h"** flag with the header output directory path.

- On the native level, the native header files generated from Java need to be implemented in native code to be a perfect match to the generated signatures.

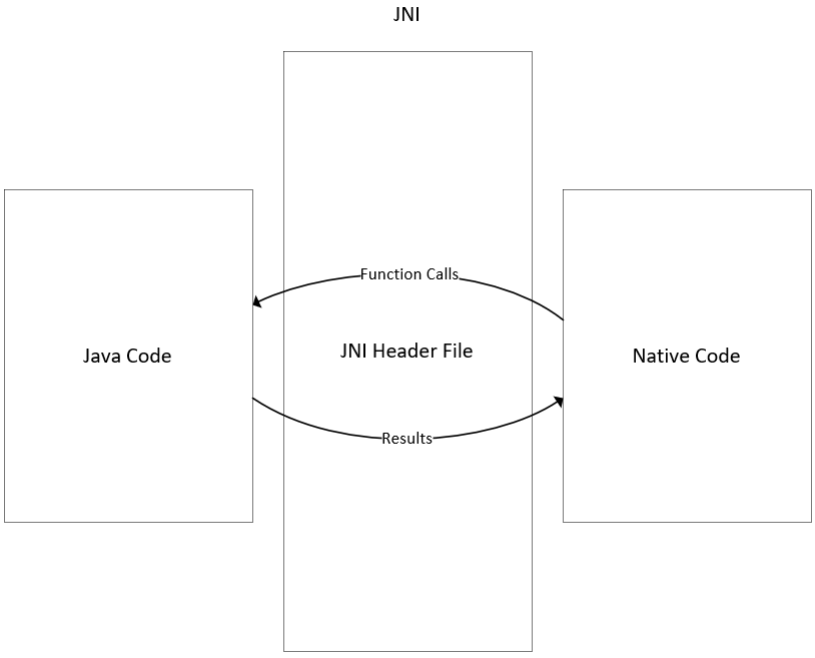

Figure 4.1: JNI general workflow.

JNI provides all necessary methods to get (or convert) Java objects, through the **JNIEnv** pointer that's used in all JNI method calls (in a similar manner to the Java Reflection API).

Most of the work with JNI happens on the native level, and the only additional step needed from the Java side is the generation of header files, that can happen at compile-time (cf. Figure 4.1). The generation task can even be integrated into a makefile that generates the headers, and then compiles the native files and converts them into dynamic libraries.

4.1.4 Java Native Access

Java Native Access, JNA for short, is an open-source library that allows the use of native code using only Java code, without the need to write native code like in the case of JNI.

JNA relies on the idea, that the native code necessary for the use of JNI is in most cases, a straight forward mapping, and thus offers an infrastructure that allows the automation of that process, by generating native code from a more detailed version of Java code than the one needed for the JNI system.

To achieve this, JNA offers a class called 'Structure' that provides most of the needed mechanisms. It also offers a set of interfaces to implement, such as 'Structure.ByValue' which allows us to value the mapping between Java and native code.

JNA thus offers multiple advantages over the use of classic JNI. However, it suffers from performance issues since it needs to dynamically generate native code at runtime (while JNI uses pre-generated stubs). It also suffers from the limitation inherent to the library itself, since it offers a limited set of possible mappings targeting the most common scenarios. So for advanced use-cases, JNI is still the only possible option.

4.1.5 ANother Tool for Language Recognition

ANother Tool for Language Recognition, ANTLR for short, is a powerful parser generator that can process and translate structured text. It's widely used in both academia and industry to build all sorts of languages and frameworks. From Twitter's search query parsing, to Hadoop's Hive and Pig languages, Lex Machina's information extraction from legal texts, to Oracle's SQL developer IDE, ANTLR is known and used everywhere! [35]. It was developed by Terence Parr in Java.

ANTLR takes as input a simple grammar file describing the language and automatically generates a lexer and a parser that can construct parse-trees. ANTLR also offers tree-walkers that use the visitor pattern to visit all nodes and produce application-specific behavior.

ANTLR's grammars are simple to write since they use an intuitive syntax of rules that follow the Backus–Naur Form. The framework then uses the grammar files to construct a parser that would recognize these rules and apply them to the referenced languages. The ANTLR generated lexer is responsible for tokenizing the input stream. The parser then recognizes the defined rules and generates Abstract Syntax Trees, that the Walker can be used to browse one node at a time, to produce the desired application-specific behavior (Figure 4.2 explains the data flow of ANTLR).

We have mainly used ANTLR as a specification in this project, in the sense that the framework offers a set of ANTLR compatible classes that can use an ANTLR generated parser to create tree-walkers automatically, allowing for easier integration with the framework, and allowing the users to ignore most of the difficulties related to creating parsers. (Section 4.2.3 gives a more detailed overview of how ANTLR is used in the context of this project.)

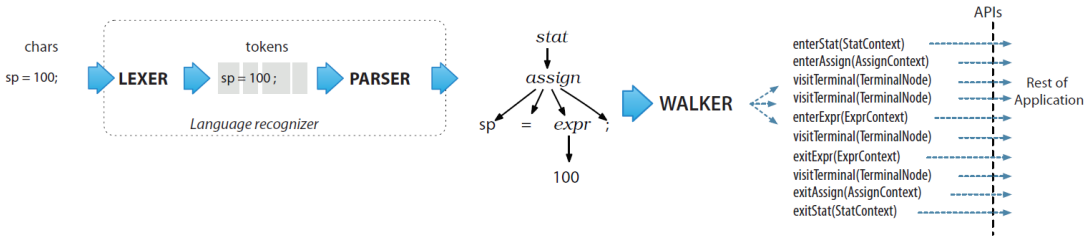

Figure 4.2: ANTLR data flow.

4.2 Tool Modules

This section aims to give an overview of the general architecture of the tools and its different modules. We start with a general overview of the entire architecture, and then we review each module independently by going over its features, its design goals, and a general overview of its architecture.

4.2.1 General Architecture

The tool was designed to follow software architecture's best practices: to be modular, flexible, and to respond to significant future changes. The tool is broken into a set of different modules, each responsible for a task. Each module is entirely independent of the rest to allow for more natural changes in the future, be it to the underlying model checker, to the delivery mechanism (website instead of desktop application), or any other aspect of the tool.

The tool is comprised of two portions, the core framework, and the Model Designer (cf. Figure 4.3). The core framework is the back-end portion of the tool; It has three primary modules:

- The transformation module: is responsible for all the parsing and transformation from any logic extending CTL into a NuSMV compatible logic. It is both responsible for the transformation from one logic to another, but also for generating the SMV output file to be used with NuSMV later. From a functional point of view, it consists of two different libraries: the Transformation API and the Transformation Engine. The API exposes a set of abstract classes and interfaces to allow the user to implement transformations of both the model and

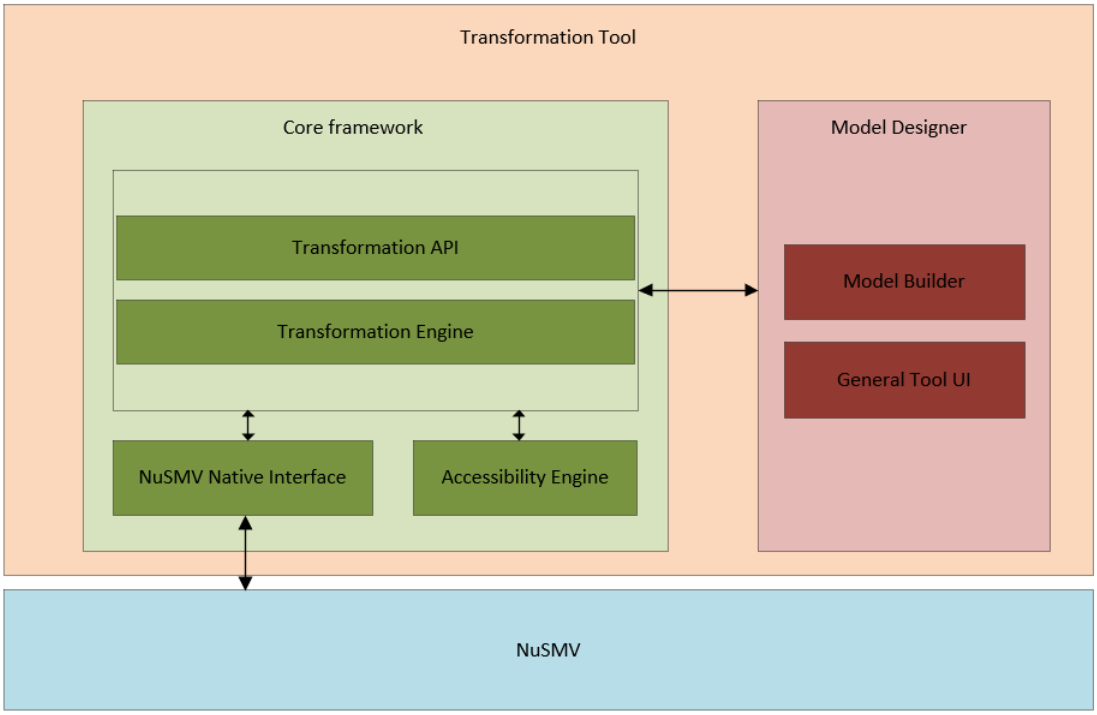

Figure 4.3: General architecture of the tool.

the formulae; it also offers a set of practical utility classes. The engine, on the other hand, is responsible for integrating with the user-defined transformations and is responsible for the actual transformation process.

- The accessibility engine: offers an API for the user to use for defining their accessibility relationships. It then uses the said definitions to calculate accessibility relations automatically.
- The NuSMV native interface: communicates natively with NuSMV and offers the tool the capabilities to interact with NuSMV. It bypasses the NuSMV interactive shell by directly making calls to the core NuSMV API.

The module designer is the graphical user interface of the tool. It can be further divided into two primary packages :

- The model builder: is a GUI utility to create local agent models. It offers an intuitive graphical interface with interactive menus to allow the designer to specify models easily. It can save, load, and edit as many local models (agents) as needed.

- The general tool UI: is the basic set of UI components necessary for the desktop application.

4.2.2 General Usage

From a usage point of view, the tool has two categories of actors (users): **"Researchers"** and **"Designers"**. Researchers are responsible for creating the transformation algorithms from their new CTL-extension logic to NuSMV. They are also responsible for using these algorithms and definitions to configure the different modules of the tool to produce consistently correct results. Designers, on the other hand, are the final intended users of the final product. They use the tool to design multi-agent systems and then exploit the model-checking engine to verify their properties (cf. Figure 4.4).

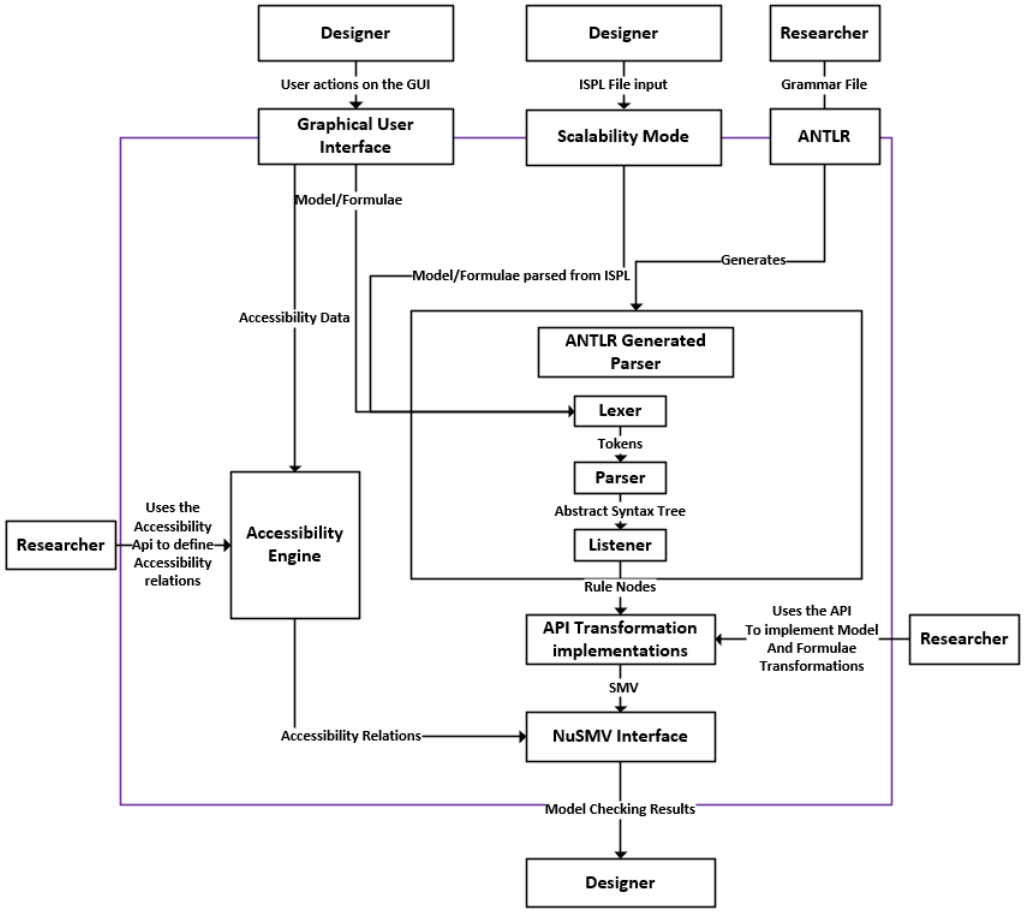

Figure 4.4: Data flow within the tool.

For a designer, the tool offers multiple mechanisms and usage modes to cater to different needs and use cases. It can function in two primary modes: the graphical mode, and the scalability mode. It can also be used by dropping the GUI all-together and integrating the core framework with an interactive shell, or to an existing project. The tool in its graphical state offers two input methods:

- The graphical mode: offers the designer the possibility to design multi-agent systems using the model builder, and then check these designs using model-checking against a set of formulas.
- The scalability mode: offers the designer the capabilities to use an ISPL file description of the design as an input to the tool. This mode allows for the processing of large systems of multiple agents, instead of the graphical mode that becomes unpractical once the number of agents grows.

A researcher, however, needs to configure the tool on multiple levels to produce the final product:

- On the transformation level, he needs to first define the grammars for the model and the formulae of the new logic, and use ANTLR to generate parsers to match the logic. Then, he needs to use the transformation API to define the transformation algorithms to be applied to the matched rules.
- On the accessibility level, if the transformations require accessibility relations, he needs to implement the accessibility API to define each accessibility relation so that the engine can calculate such accessibilities for any given model.

In the following sections, we go deeper into each module and present its general features, its design goals, and a general UML class diagram to explain how each module works. The presented class diagrams aim to explain the general architecture of each module, and only present the most important parts of every module. The original class diagrams are massive in comparison and are therefore almost impossible to present as added figures.

4.2.3 Transformation Module

The transformation module offers the capabilities to support all the transformation-related activities within the core framework. It is entirely independent of all other

modules, and offers a rich set of features:

- For CTL-based logics, the transformation API offers extensible classes that readily support CTL functionalities and transformations; it also offers a rich set of utility classes to ease the transformation implementations for both models and formulae (cf. Figure 4.5).
- For any other logic, a designer needs to manually implement the transformations algorithms by extending the offered abstract classes (cf. Figure 4.5).
- The transformation engine supports ANTLR v4 generated parsers, lexers, and TreeWalkers/ Listeners. We designed the engine to support these artifacts by fully taking into account the ANTLR generation scheme meta-model. This support also offers the added benefit of allowing the engine to support any LL(*) grammar, as well as adaptive LL(*) grammars [36, 37, 38].
- The transformation engine also offers a ready to use ISPL system, allowing the parsing of ISPL MASs models into SMV models. Since the Interpreted Systems formalism is already widely used in the MASs community, this feature on its own allows researchers to use NuSMV with MASs without any change to the conventional processes.

This module is designed with multiple goals in mind. From a feature point of view it aims to:

- Offer opinionated features that allow the user in most common cases to simplify the process (CTL-based logics utilities, ANTLR support, etc.).
- Be flexible enough for users that desire to take charge of particular or all steps manually.

Figure 4.6 offers a general architectural view of the transformation engine in relation to other modules and packages. The Transformation engine is context agnostic since it doesn't have any dependencies on any other modules or packages (except for the ANTLR library). It uses a set of Boundary interfaces to deploy its calls to external packages polymorphically, reversing, thus the dependency on other modules, in line with the Interface segregation principle and the dependency inversion principle [31]. The interactors are responsible for the orchestration of the different sub-processes of

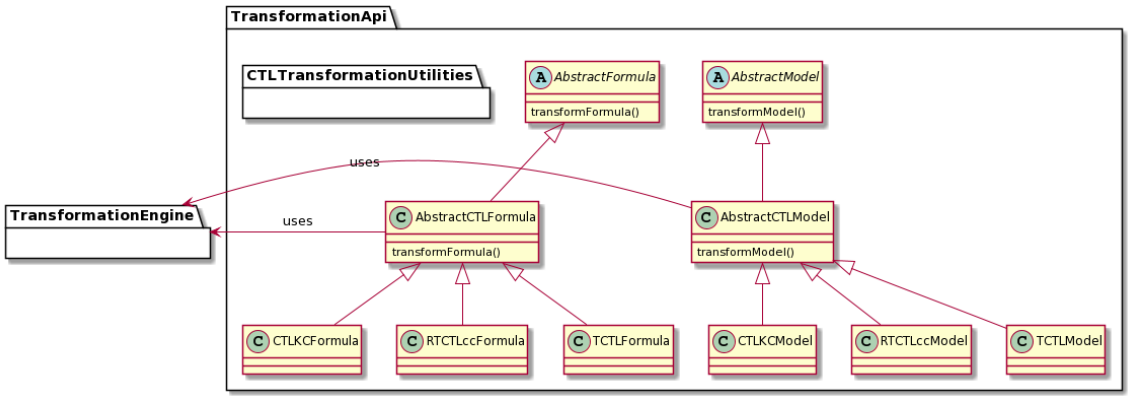

Figure 4.5: General class diagram of transformation API.

the transformation. The Controllers package is the central control unit of the tool; it contains a set of Use-Case handlers, each responsible for a specific scenario. The ISPLEngine package contains the ISPL parsing utility mechanisms, and an interactor responsible for orchestrating the different parts.

4.2.4 The Model Builder Module

The model builder module is in charge of offering the user a complete GUI-driven set of functionalities that allow for an easy and intuitive building of agent models. This module enables designers to model MASs by modeling each agent separately. A designer can use the model builder to draw new local states of each agent. An interactive menu offers a set of context-aware options to create local states, to connect states with edges, and to add different properties to a state. Each local state follows the ISPL+ formalism, and thus offers the Designer the capabilities to add a set of atomic propositions that hold at the given state. Moreover, It allows the Designer to set the values for the different shared and unshared variables on each local state (Figure 4.8). The model builder also computes and draws the accessibility relations for any given local agent model. It displays the said accessibilities as dashed edges in the agent's model (in Figure 4.7, the dashed edge between states s1 and s4, labeled α_i is an accessibility relation edge).

Once the design process of the entire system is complete, the Designer can transform the models automatically into SMV modules, and use the formulae editor to

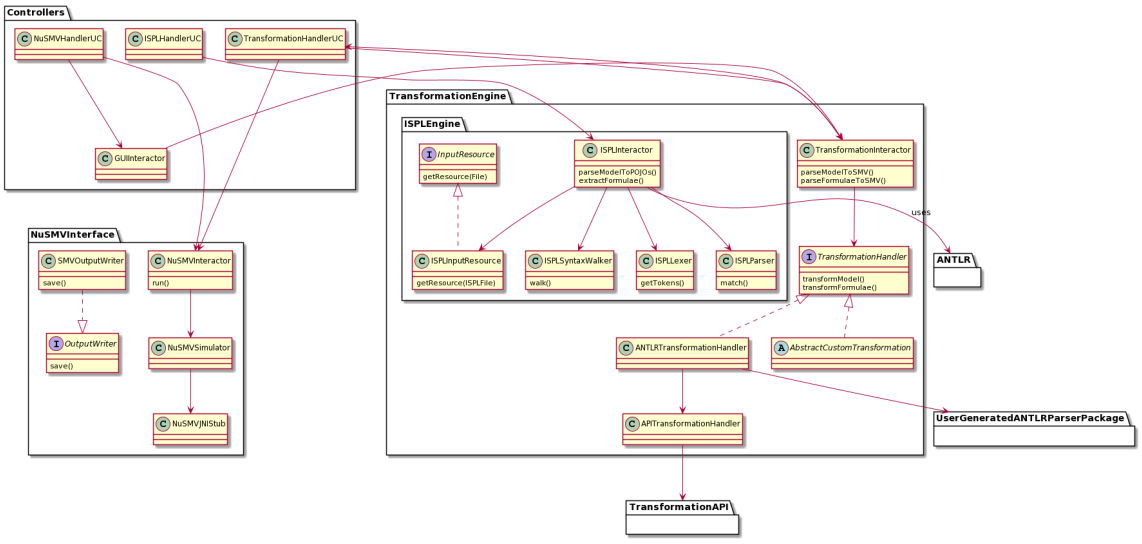

Figure 4.6: General class diagram of the transformation engine.

specify requirements on the MASs. The tool then executes NuSMV in the background and model-checks the system under design against the formulas.

The general architecture of the model builder is depicted in Figure 4.9. It uses a fixed depth **Composite Pattern** since the local model can contain multiple local models of other agents. It also uses a **Strategy Pattern** to allow for future additions of other types of edges.

The communication with the core package happens through a set of Interface Boundaries that decouple both sides, allowing for a more robust design, in line with the **Dependency Inversion Principle**. Both sides use Request and Response classes that get exchanged through the boundary, to allow both sides to depend on each other's behavior, not implementation.

4.3 Experimental Results

The work presented in this dissertation has been used in multiple conferences and journal publications to implement different logics for MASs [14, 33, 15, 18]. In this section, we aim to demonstrate the broad set of possible applications for the tool by presenting the different ranges of case studies it has successfully implemented.

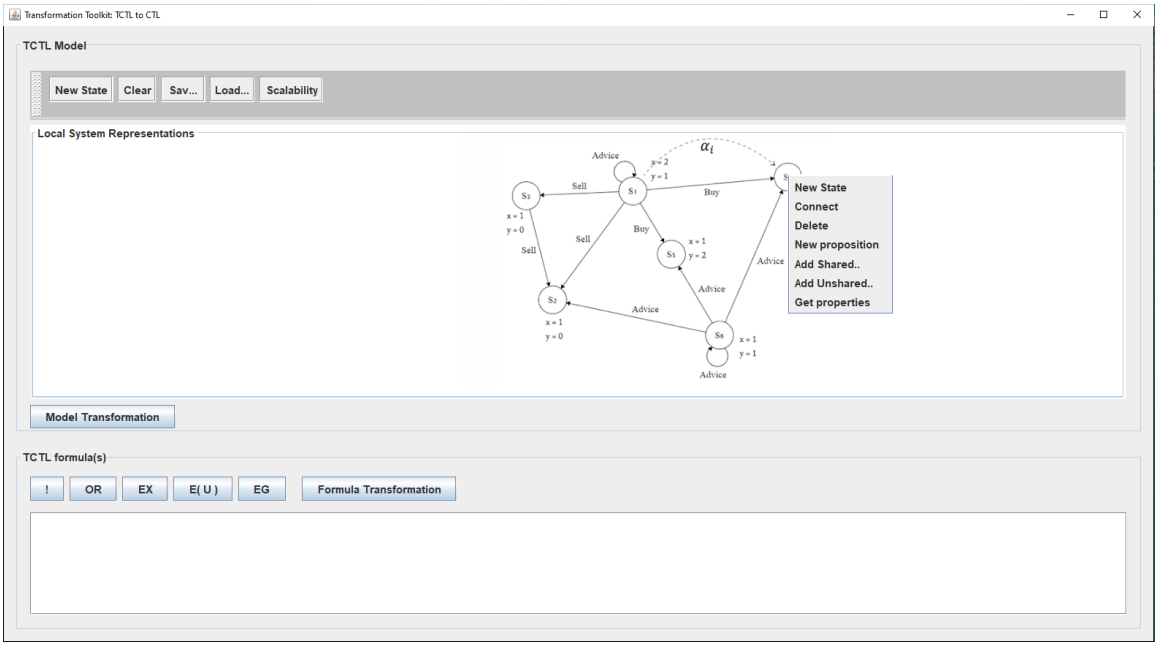

Figure 4.7: Model builder v2.0 new JavaFx interface.

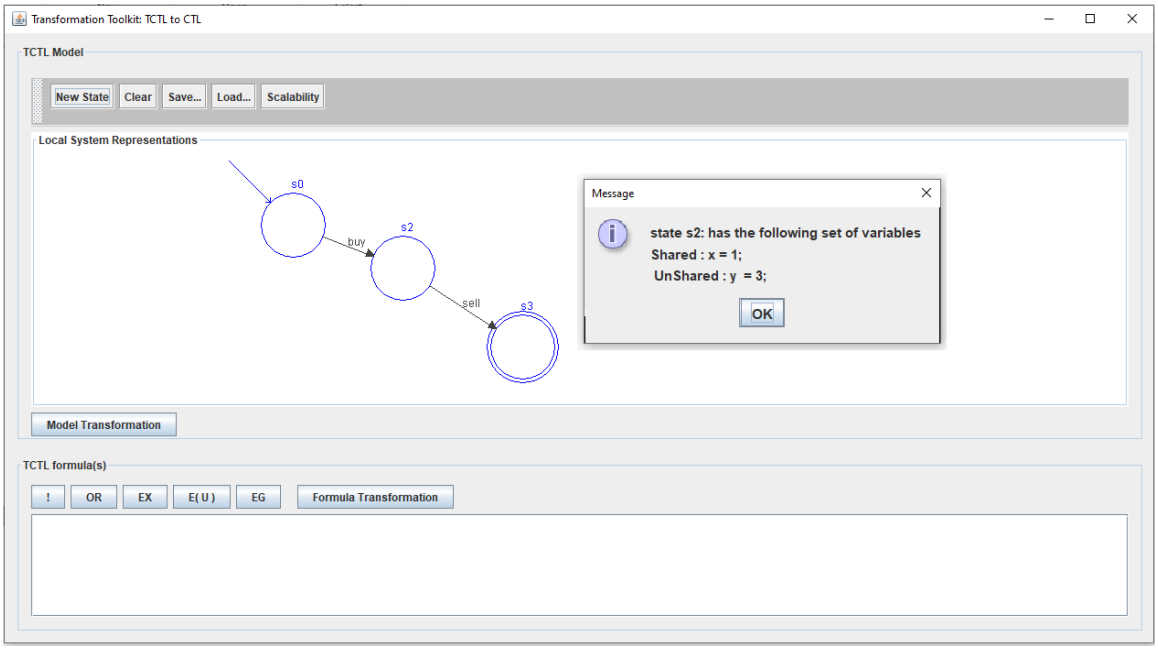

Figure 4.8: Model builder v1.0 swing interface.

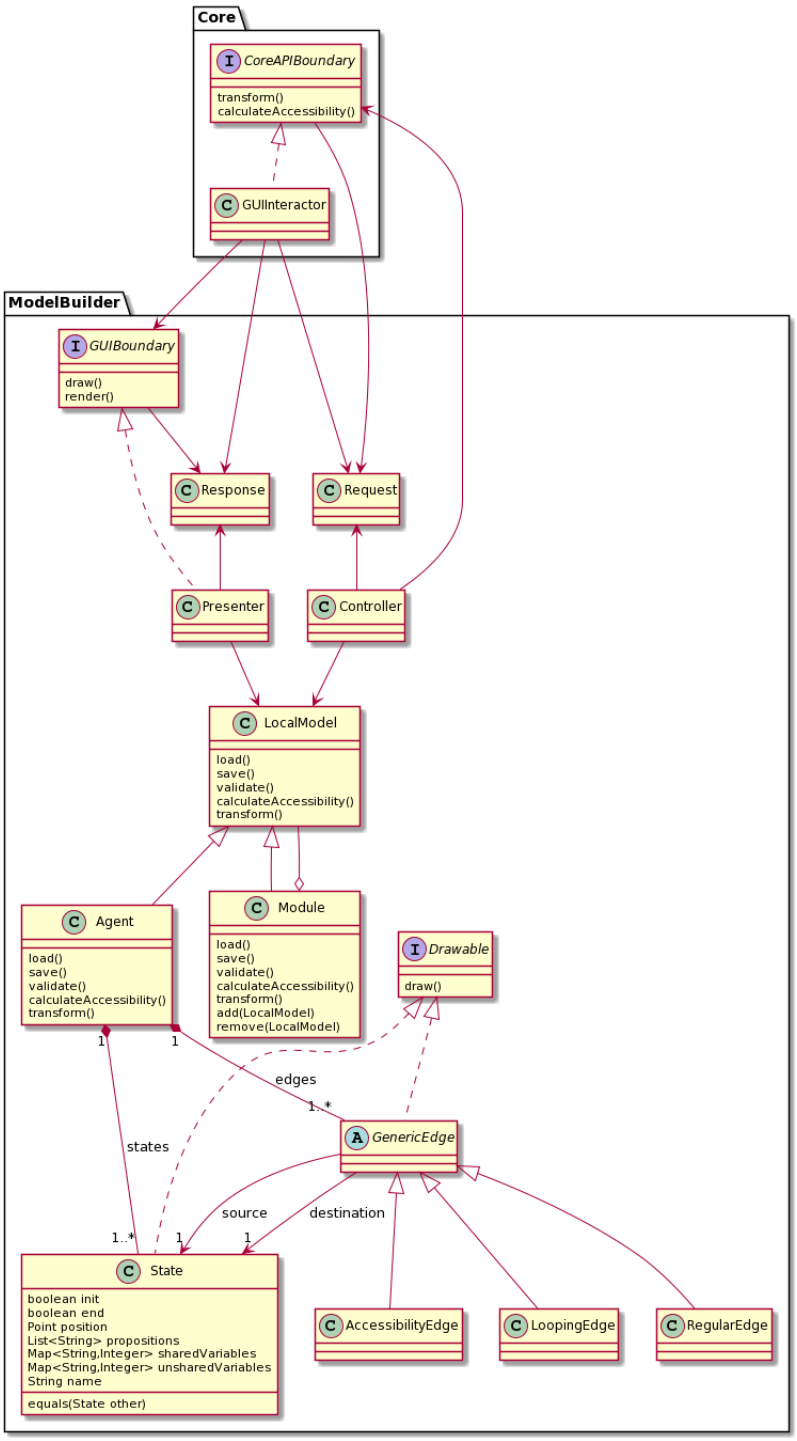

Figure 4.9: General class diagram of the model builder module.

4.3.1 Case Study Results

Multiple new logics have used the tool as their implementation method. In this subsection, we cover the different case studies they used and the experimental results obtained from the tool.

A The buyer-seller ordering protocol

Experiment	no. of agents	no. of reachable states	time of model transformation in ms	average total time in ms
1	3	5	07.00690	9.238688
2	6	25	10.414627	12.7589
3	9	125	12.253960	14.80114
4	12	625	17.273747	19.93857
5	15	3125	18.381369	21.27313
6	18	15625	20.361401	23.67727
7	21	78125	20.997321	24.92944
8	24	390625	26.318603	30.96645
9	30	9.76563e+06	29.235256	34.61843
10	36	2.44141e+08	30.089409	36.64260
11	42	6.10352e+09	31.693837	39.22330
12	66	2.38419e+15	51.885031	117.0266699

Table 4.1: Verification results of the ordering protocol using the toolkit

This ordering protocol, introduced by Desai et al. [13], specifies the rules of interaction between a buyer and a seller, modeled as agents. For example, a buyer interested in certain goods requests a quote from the seller. The seller then responds with an offer. This interaction and many others in this context were used as a case study in [33] to capture the conditional commitments between sellers, buyers, shippers, and other different agents.

Table 4.1 reports the results of the different experiments used in the case study, representing scenarios growing in complexity as the number of agents grows. Analyzing the table, it's clear that the number of reachable states grows exponentially with

the growing number of agents. However, the transformation time increases only polynomially, allowing for the model-checking of complex systems with a massive number of reachable states.

no. agents	no. reachable states	execution time (ms)	memory (MB)
3	17	0.01	6.7
5	28	0.012	7.04
7	50	0.019	7.73
9	94	0.032	8.71
11	182	0.058	9.97
13	358	0.093	11.06
15	710	0.287	13.11
17	1414	0.648	20.21
19	2822	2.139	21.54
21	11,270	5.346	22.19
23	22,534	16.311	27.79
25	45,062	25.269	30.37
27	90,118	56.014	40.41
29	180,230	79.802	31.44
31	360,454	86.345	49.99
33	720,902	112.162	99.85
35	1.4418e+06	192.664	131.28

Table 4.2: Verification results of the landing gear system using the toolkit

B The landing gear system case study

This case study, introduced by Boniol and Wiels [8], is an industrial case study for complex communicative avionics systems. The interaction between the different autonomous components makes it a rich case study in the context of multi-agent systems since each component can be modeled as an autonomous agent. This case study was used in both [18, 33] to model two different logics. In the first, it was used in the process of model-based test generation for safety-critical systems. The second used it to capture the real-time aspects of the logic as a running example.

Table 4.2 presents the results of the tool's usage with the landing gear system. It reports in addition to the different system's number of agents, and reachable states, the memory usage in each operation. It shows how an approach based on model-based transformation can contribute to the design activities and the verification and validation process for safety-critical systems.

Exp.#	Agents#	States#	Time of model transformation (ms)	Time of formulae transformation (ms)	Total time (ms)
1	7	42	15	0.8	20
2	14	468	17	0.9	119
3	21	5586	22	1	1330
4	28	67236	36	1.1	8049
5	35	809682	38	1.2	45051
6	42	9.74E+06	42	1.3	210000
7	49	1.30E+08	48	1.4	390000
8	56	4.49E+12	53	1.5	540000
9	63	2.52E+15	57	1.7	792000

Table 4.3: Verification results of the AGFIL protocol using our tool

C The AGFIL case study

This case-study was introduced by Telang et al. [46]. It's a standard industrial case study about an insurance company in Ireland: AGFIL. This case study captures the rules of interaction between a policyholder and the insurance company. The interaction between the different parties in this scenario made it perfect for capturing the aspects of trust between the different involved parties and was used as such in [14] as a running case study.

Table 4.3 reports the results of the different experiments using the tool with the trust logic. We can observe that the number of reachable states grows exponentially with the number of agents. The transformation time of both the formulae and the model is polynomial in time, however.

(a) - Using our transformation toolkit

Exp	Agents	States	Total time (ms)
1	7	42	20
2	14	468	119
3	21	5586	1330
4	28	67236	8049
5	35	809682	45051
6	42	9.74E+06	210000
7	49	1.30E+08	390000
8	56	4.49E+12	540000
9	63	2.52E+15	792000

(b) - Using the MCMAS-T model checker

Exp	Agents	States	Time (ms)
1	7	42	50
2	14	468	1020
3	21	5586	16340
4	28	67236	99723
5	35	809682	694035
6	42	9.74E+06	3333680

Table 4.4: Comparison of the verification results

4.3.2 Performance Evaluation

To evaluate the performance of the tool and the general approach against the direct implementation approach, we decided to do a benchmark against the MCMAS model checker, since both our tool and MCMAS take an ISPL+ file as input. We used MCMAS-T, an extension of MCMAS with modalities of trust [16], and then we used the implementation of the same trust logic using this tool used in [14]. We then used the same machine that produced the results in Table Y, to run the same AGFIL experiment files using MCMAS-T this time to compare the verification time of the process as the number of agents grows.

Table 4.4 reports the comparison results using the same machine. It's clear that the transformation-based approach provides better results on all metrics. The MCMAS model checker, in general, can't verify models beyond 10+e07 reachable states. MCMAS-T is no exception and suffers the same limitations, stoping at 42 agents and crashing on 49 agents as it takes around 24h without producing any results until the process has to be manually stopped. In addition to that, the metric of time execution makes the tool a more efficient approach, making it a promising and viable methodology in practice.

Figure 4.10 compares the verification results of the tool's trust implementation with the MCMAS-T model checker in the form of a graph. We plot the execution

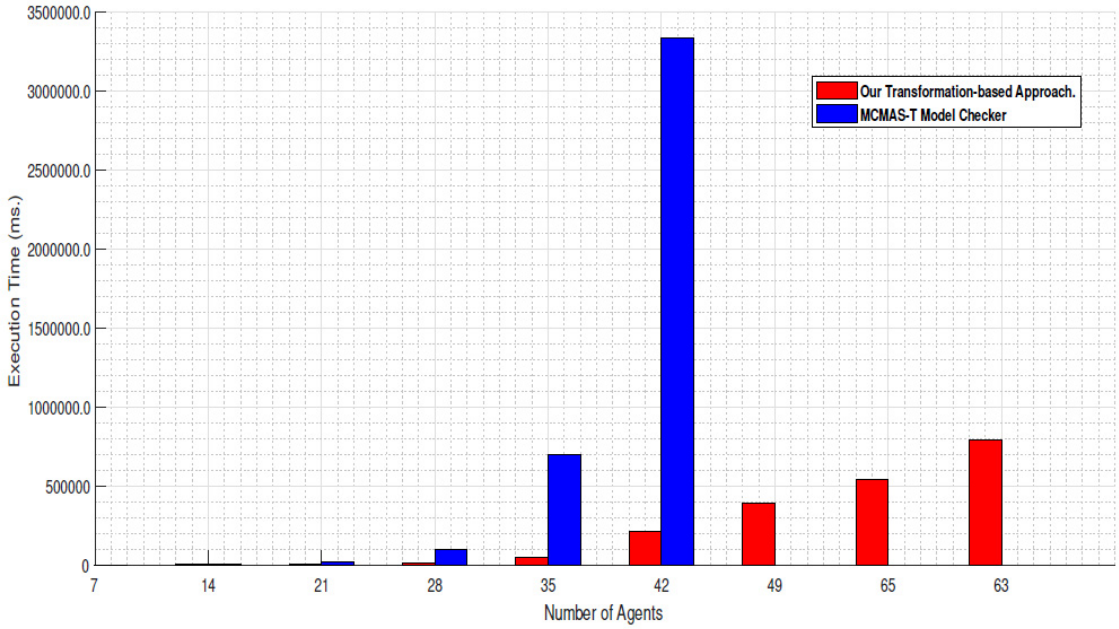

Figure 4.10: Comparison results between our tool and MCMAS-T

time as a function of the number of agents. It gives an evident appreciation of the tool's performance as opposed to the traditional, straightforward implementation methodology.

4.4 Summary

In this chapter, we presented an overview of the tool's implementation. We started by reviewing the different technologies used. Then we went on to expose the architecture and the different design goals of the essential modules. We concluded by a review of the different case studies the tools logics can be used with, to demonstrate the range of possible industrial applications, before presenting a brief performance overview against one of the most famous and used model checkers for MASs.

Chapter 5

Conclusion

5.1 Summary

In this thesis, we proposed a new framework to help automate the transformation-based model checking of MAS logics. We designed the system to offer a rich and powerful API, built to offer advanced features and performance while being modular and flexible to suit different needs. We offered an architectural overview of the framework and its design goals. We then presented a tool built on top of this core framework that we successfully used in multiple research settings with different logics, before concluding with a performance comparison to the MCMAS model checker, one of the most popular model checkers for multi-agent systems.

5.2 Future Directions

This work can be extended in many ways. One can investigate the ways we can manage to support other logics that are not CTL based. Another potential track is to support the integration of model checkers other than NuSMV. One can also try to take this framework's support to a higher level of abstraction to make it independent of the structure of the models and formulae by taking metamodels as input instead.

Bibliography

[1] Faisal Al-Saqqar, Jamal Bentahar, Khalid Sultan, Wei Wan, and Ehsan Khosrowshahi Asl. Model checking temporal knowledge and commitments in multi-agent systems using reduction. *Simulation Modelling Practice and Theory*, 51:45 – 68, 2015.

[2] Paolo Arcaini, Angelo Gargantini, and Paolo Vavassori. Nuseen: an eclipse-based environment for the nusmv model checker. 2013.

[3] Najwa Abu Bakar and Ali Selamat. Runtime verification of multi-agent systems interaction quality. In Ali Selamat, Ngoc Thanh Nguyen, and Habibollah Haron, editors, *Intelligent Information and Database Systems*, pages 435–444, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[4] Ahmed Saleh Bataineh, Jamal Bentahar, Mohamed El Menshawy, and Rachida Dssouli. Specifying and verifying contract-driven service compositions using commitments and model checking. *Expert Syst. Appl.*, 74(C):151–184, May 2017.

[5] Jamal Bentahar, Mohamed El-Menshawy, Hongyang Qu, and Rachida Dssouli. Communicative commitments: Model checking and complexity analysis. *Knowledge-Based Systems*, 35:21 – 34, 2012.

[6] Jamal Bentahar, Babak Khosravifar, Mohamed Adel Serhani, and Mahsa Alishahi. On the analysis of reputation for agent-based web services. *Expert Systems with Applications*, 39(16):12438 – 12450, 2012.

[7] Pratik K. Biswas. Towards an agent-oriented approach to conceptualization. *Applied Soft Computing*, 8(1):127 – 139, 2008.

[8] Frédéric Boniol and Virginie Wiels. The landing gear system case study. In *ABZ*, 2014.

[9] Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug 1986.

[10] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV version 2: An opensource tool for symbolic model checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

[11] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2:410–425, 2000.

[12] Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis. Model checking: Algorithmic verification and debugging. *Commun. ACM*, 52(11):74–84, November 2009.

[13] Nirmit Desai, Ashok U. Mallya, Amit K. Chopra, and Munindar P. Singh. Interaction protocols as design abstractions for business processes. *IEEE Transactions on Software Engineering*, 31:1015–1027, 2005.

[14] Nagat Drawel, Jamal Bentahar, Mohamed El-Menshawy, and Amine Laarej. Verifying temporal trust logic using ctl model checking. In *TRUST @ AAMAS*, pages 62–74, 2018.

[15] Nagat Drawel, Jamal Bentahar, Amine Laarej, and Gaith Rjoub. Formalizing group and propagated trust in multi-agent systems. In *IJCAI*, 2020.

[16] Nagat Drawel, Hongyang Qu, Jamal Bentahar, and Elhadi Shakshuki. Specification and automatic verification of trust-based multi-agent systems. *Future Generation Computer Systems*, 2018.

[17] Mohamed El-menshawy, Jamal Bentahar, and Rachida Dssouli. Symbolic model checking commitment protocols using reduction. In *6619 of LNAI*, pages 185–203. Springer, 2011.

[18] Warda Elkholy, Mohamed El-Menshawy, Jamal Bentahar, Mounia Elqortobi, Amine Laarej, and Rachida Dssouli. Model checking intelligent avionics systems for test cases generation using multi-agent systems. *Expert Syst. Appl.*, 156:113458, 2020.

[19] E. Allen Emerson, Aloysius K. Mok, A. Prasad Sistla, and Jai Srinivasan. Quantitative temporal reasoning. *Real-Time Systems*, 4(4):331–352, 1992.

[20] Niklas Eén and Niklas Sörensson. Minisat. `http://minisat.se/`, 2010.

[21] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning About Knowledge*. MIT Press, Cambridge, MA, USA, 2003.

[22] Joseph Y. Halpern and Leandro C. Rêgo. Reasoning about knowledge of unawareness revisited. *Mathematical Social Sciences*, 70:10 – 22, 2014. Unawareness.

[23] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, USA, 2004.

[24] Nicholas R. Jennings and Michael J. Wooldridge, editors. *Agent Technology: Foundations, Applications, and Markets*. Springer-Verlag, Berlin, Heidelberg, 1998.

[25] Fondazione Bruno Kessler. NuSMV. `http://nusmv.fbk.eu/`, 2019.

[26] Warda El Kholy, Jamal Bentahar, Mohamed El-Menshawy, Hongyang Qu, and Rachida Dssouli. Smc4ac: A new symbolic model checker for intelligent agent communication. *Fundam. Inform.*, 152:223–271, 2017.

[27] Warda El Kholy, Jamal Bentahar, Mohamed EL Menshawy, Hongyang Qu, and Rachida Dssouli. Conditional commitments: Reasoning and model checking. *ACM Trans. Softw. Eng. Methodol.*, December 2014.

[28] Haeng-Kon Kim. Convergence agent model for developing u-healthcare systems. *Future Generation Computer Systems*, 35:39 – 48, 2014. Special Section: Integration of Cloud Computing and Body Sensor Networks; Guest Editors: Giancarlo Fortino and Mukaddim Pathan.

[29] Alessio Lomuscio, Hongyang Qu, and Franco Raimondi. MCMAS: An open-source model checker for the verification of multi-agent systems. *International Journal on Software Tools for Technology Transfer*, 19:9–30, 2015.

[30] Omar Marey, Jamal Bentahar, Rachida Dssouli, and Mohamed Mbarki. Measuring and analyzing agents' uncertainty in argumentation-based negotiation dialogue games. *Expert Systems with Applications*, 41(2):306 – 320, 2014.

[31] Robert C. Martin. Agile software development, principles, patterns, and practices. 2002.

[32] Kenneth Lauchlin McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, USA, 1992. UMI Order No. GAX92-24209.

[33] Mohamed El Menshawy, Jamal Bentahar, Warda El Kholy, and Amine Laarej. Model checking real-time conditional commitment logic using transformation. *Journal of Systems and Software*, 138:189 – 205, 2018.

[34] Oracle. Java native interface specification contents. `https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html`, 2020.

[35] Terence Parr. ANTLR. `https://www.antlr.org/index.html`, 2020.

[36] Terence Parr and Kathleen Fisher. LL(*): The foundation of the ANTLR parser generator. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 425–436. ACM, 2011.

[37] Terence Parr, Sam Harwell, and Kathleen Fisher. Adaptive LL(*) parsing: The power of dynamic analysis. In Andrew P. Black and Todd D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 579–598. ACM, 2014.

[38] Terence John Parr. ANTLR: Another tool for language recognition. 2005.

[39] Philippe Schnoebelen. The complexity of temporal logic model checking. In *Advances in Modal Logic, Vol. 4*, pages 1–44, 2002.

[40] John Searle. *Speech Acts: An Essay in the Philosophy of Language*. Cambridge University Press, 1st edition, 1969.

[41] Munindar P. Singh. Agent communication languages: Rethinking the principles. *Computer*, 31(12):40–47, December 1998.

[42] Munindar P. Singh. *A Social Semantics for Agent Communication Languages*, pages 31–45. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[43] Munindar P. Singh. Formalizing communication protocols for multiagent systems. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence*, pages 1519–1524. Veloso,M.M, 2007.

[44] Fabio Somenzi. CUDD: CU decision diagram package. `https://github.com/ivmai/cudd`, 2018.

[45] D. Tang, Y. Yu, D. Ranjan, and S. Malik. Zchaff sat. `https://www.princeton.edu/~chaff/zchaff.html`, 2007.

[46] Pankaj R. Telang and Munindar P. Singh. Enhancing tropos with commitments. In *Conceptual Modeling: Foundations and Applications*, 2009.

[47] Fenghui Wang, Ming Yang, and Ruqing Yang. Simulation of multi-agent based cybernetic transportation system. *Simulation Modelling Practice and Theory*, 16(10):1606 – 1614, 2008. The Analysis of Complex Systems.

[48] Gerhard Weiss, editor. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, Cambridge, MA, USA, 1999.

[49] Michael W. Whalen, John D. Innis, Steven P. Miller, and Lucas G. Wagner. Adgs-2100 display window manager analysis. *NASA contract report*, November 2005.

[50] Michael Wooldridge. Semantic issues in the verification of agent communication languages. *Autonomous Agents and Multi-Agent Systems*, 3:9–31, 2000.

[51] Michael Wooldridge. *An Introduction to MultiAgent Systems*. Wiley Publishing, 2nd edition, 2009.

[52] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.