Chapter 3
Model Checking Agent Communication

J. Bentahar, J.-J. Ch. Meyer, and W. Wan

Abstract Model checking is a formal and automatic technique used to verify computational systems (e.g. communication protocols) against given properties. The purpose of this chapter is to describe a model checking algorithm to verify communication protocols used by autonomous agents interacting using dialogue games, which are governed by a set of logical rules. We use a variant of Extended Computation Tree Logic CTL* for specifying these dialogue games and the properties to be checked. This logic, called ACTL*, extends CTL* by allowing formulae to constrain actions as well as states. The verification method uses an on-the-fly efficient algorithm. It is based on translating formulae into a variant of alternating tree automata called Alternating Büchi Tableau Automata (ABTA). We present a tableau-based version of this algorithm and provide the soundness, completeness, termination and complexity results. Two case studies are discussed along with their respective implementations to illustrate the proposed approach. The first one is about an agent-based negotiation protocol and the second one considers a modified version of the NetBill protocol.

J. Bentahar, W. Wan
Concordia University, Concordia Institute for Information Systems Engineering, Canada e-mail: bentahar@ciise.concordia.ca

J.-J. Ch. Meyer
Utrecht University, Department of Computer Science, The Netherlands e-mail: jj@cs.uu.nl

M. Dastani et al. (eds.), *Specification and Verification of Multi-agent Systems*,
DOI 10.1007/978-1-4419-6984-2_3, © Springer Science+Business Media, LLC 2010

3.1 Introduction

Model checking is a formal verification method widely used to check complex systems involving concurrency and communication protocols by verifying some desirable properties. *Deadlock-freedom* (it is false that two or more processes are each waiting for another to release a resource), *safety* (some bad situation may never occur), and *reachability* (some particular situation can be reached) are examples of such properties. Model checking techniques offer the possibility of obtaining an early integration of verification in the design process and reducing the verification time. However, they are only applicable for finite state systems and they generally operate on system models and not on the actual system. In fact, the system is represented by a finite model M and the specification is represented by a formula ϕ using an appropriate logic. The verification method consists of computing whether the model M satisfies ϕ (i.e. $M \models \phi$) or not (i.e. $M \not\models \phi$).

Recently, model checking Multi-Agent Systems (MASs) has seen an increasing interest [33, 61, 62, 232, 266, 267, 337, 354, 356, 440]. However, although research in agent communication has received much attention during the past years, only few research works tried to address the verification of agent protocols [4, 24, 163, 195, 244, 430]. Several dialogue game protocols have been proposed for specifying agent communication interactions [37, 304, 307, 381]. These games aim at offering more flexibility by combining different small games to construct complete and more complex protocols. Dialogue games can be thought of as interaction games in which each agent plays a move in turn by performing utterances according to a pre-defined set of rules.

The verification problem of agent communication protocols is fundamental for the MASs community. Endriss et al. [163] have proposed abductive logic-based agents and some means of determining whether or not these agents behave in conformance with agent communication protocols. Baldoni et al. [24] have addressed the problem of verifying that a given protocol implementation using a logical language conforms to its AUML specification. Alberti et al. [4] have considered the problem of verifying on the fly the compliance of the agents' behavior to protocols specified using a logic-based framework. These approaches are different from the technique presented in this chapter in the sense that they are not based on model checking techniques and they do not address the problem of verifying if a protocol satisfies given properties. Giordano et al. [195] have addressed the problem of specifying and verifying agent interaction protocols using a Dynamic Linear Time Temporal Logic (DLTL). The authors have addressed three kinds of verification problems: 1) the compliance of a protocol execution to its specification; 2) the satisfaction of a property in the protocol; 3) the compliance of agents to the protocol. They have shown that these problems can be solved by model checking DLTL. This model checking technique uses a tableau-based algorithm for obtaining a Büchi automaton from a formula in DLTL and the construction of this automaton uses proof rules. However, the protocols are only specified in an abstract way in terms of the effects of communicative actions and some precondition laws.

In this chapter, we describe model checking-based verification of dialogue game protocols for agent communication using an action and temporal logic (ACTL*) based on the Extended Computation Tree Logic CTL*. Using a model checking technique for this verification is motivated by the fact that model-checking is a successful technique for automatically and computationally verifying protocol specifications using a suitable logic. This technique can be used to verify the protocol correctness in the sense that the protocol satisfies the expected properties. It allows us to verify agent communication properties specified using ACTL* logic. Therefore, we can specify the protocol in a logical way and verify its correctness in terms of the satisfaction of the expected properties. The definition of a new logic is motivated by the fact that dialogue game protocols should be specified using not only temporal properties, but also action properties. In addition, in these protocols, actions that agents perform by communicating are expressed in terms of "Social Commitments" (SCs) and arguments. These protocols are specified as transition systems (TSs) using ACTL* logic and Commitment and Argument Network (CAN) [38]. These TSs are labeled with actions that agents perform on SCs and SC contents [115,182,404].

The model checking technique we describe in this chapter is based on the translation of the formula expressing the property to be verified into a variant of alternating tree automata called Alternating Büchi Tableau Automata (ABTA). This technique is an extension of the ABTA-based algorithm for CTL* proposed in [44]. The choice of this technique is motivated by the fact that unlike other model checking techniques, this technique allows us to check temporal and action formulas. In addition, this technique is one of the most efficient techniques proposed in the literature. The translation procedure uses a set of inference rules called tableau rules. Like automata-based model checking of Linear Temporal Logic LTL, our technique is based on the product graph of the model and the automaton representing the formula to be verified (Fig. 3.1). This technique allows us to verify not only that the dialogue game protocol satisfies a given property, but also that this protocol respects the decomposition rules of the communicative acts. Consequently, if agents respect these protocols, then they also respect the decomposition semantics of the communicative acts. Thus, we have only one procedure to verify both:

1. the correctness of the protocols relative to the properties that the protocols should satisfy;
2. the conformance of agents to the decomposition semantics of the communicative acts.

The rest of this chapter is organized as follows. Section 3.2 presents an overview of model checking MASs. Section 3.3 introduces tableau-based algorithms for model checking, which we use in the verification procedure. Section 3.4 presents the ACTL* logic: syntax, semantics and associated tableau rules. In Section 3.5, we use this logic to define the TS that we use to specify dialogue game protocols. The problem of verifying these protocols is addressed in Section 3.6. The ABTA's definition that we use in our verification technique along with some running examples of the model checking steps are presented in this section. Section 3.7 presents two

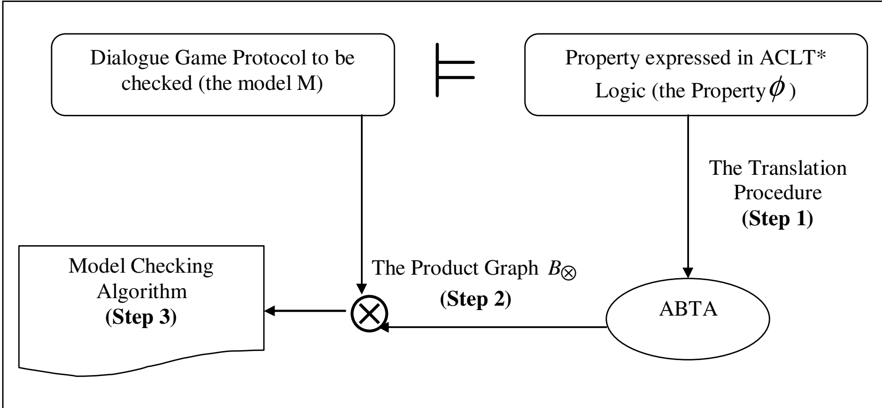

Fig. 3.1 The model checking approach

case studies and Section 3.8 concludes the chapter by discussing open challenges in the area of verifying MASs and identifying some directions for future work.

3.2 Brief Overview of Model Checking Multi-Agent Systems

3.2.1 Extending and Adapting Existing Model Checkers

Bordini and his colleagues [61, 62, 73] have addressed the problem of verifying MASs specified using the AgentSpeak(F) language (a simplified version of AgentSpeak) against BDI specifications. They have shown how programs written in AgentSpeak(F) can be automatically transformed into Promela and into Java and how the BDI specifications are transformed into LTL formulae. The Spin model checker[1] based on Promela [236] and Java PathFinder 2 (JPF2) model checker[2] based on translating Java to Promela [211] are then used to verify the MAS specifications. The idea behind using AgentSpeak(F) instead of the original AgentSpeak is to make the system to be checked finite in terms of state space, which is a fundamental condition of using model checking techniques. To this end, the maximum sizes of types, data structures and communication channels are specified. Examples of these maximum sizes are: M_{Term}: maximum number of terms in a predicate or an action; M_{Conj}: maximum number of conjuncts (literals) in a plan's context;

[1] The Spin model checker can be downloaded from:
http://spinroot.com/spin/Man/README.html

[2] The JPF2 model checker is open source and can be downloaded from:
http://javapathfinder.sourceforge.net/

M_{Var}: maximum number of different variables in a plan; M_{Bel}: maximum number of beliefs an agent can have at any moment in time in its belief base; and M_{Msg}: maximum number of messages (generated by inter-agent communication) that an agent can handle at a time.

The main constructs in a Promela program are Promela channels and in order to translate AgentSpeak(F) into Promela, the following channels are used to capture the data structures used in an AgentSpeak(F) program: (1) channel b for the agent's belief base with M_{Bel} messages as maximum size and each message has $M_{Terms}+1$ as maximum size; (2) channel p for the environment's percepts where the maximum size is the same as for channel b; (3) channel m for sending agent communication messages where the bound is M_{Msg} messages; (4) channel e for events, which are related to intentions; (5) channel i for scheduling intentions; and channel a for storing actions. Promela inline procedures are used to code the bodies of agents' plans. The environment is implemented as a Promela process type defined by the user.

Channel m is used to handle messages when the agent interpretation cycle starts, and channels p and b are used by the agent to run its belief revision. Events are handled according to FIFO policy: when new events are generated, they are inserted in the end of channel e, and the first message in that channel is selected as the event to be handled in the current cycle. Translating a formula that appears in a plan body is done as follows: basic actions are appended to channel a; addition and deletion of beliefs is translated as adding or removing messages to/from channel b; and test goals are simply an attempt to match the associated predicate with any message from channel b.

To check BDI properties, BDI modalities are interpreted in terms of Promela data structures associated to an agentSpeak(F) agent. For instance, an AgentSpeak(F) agent believes a formula ϕ iff it is included in the agent's belief base, and this agent intends ϕ iff it has ϕ as an achievement goal that currently appears in its set of intentions, or ϕ is an achievement goal that appears in the (suspended) intentions associated with the set of events.

In the same line of research, Rao and Georgeff [356] have proposed an adaptation of CTL and CTL* model checking to verify BDI (beliefs, desires and intentions) logics. Furthermore, van der Hoek and Wooldridge [232] have reduced the problem of model checking knowledge for multi-agent systems to linear temporal logic model checking using the logic of local propositions [165]. The Spin model checker is then used to check temporal epistemic properties. In [440], Wooldridge et al. have presented the translation of the MABLE language for the specification and verification of MASs into Promela. MABLE is an imperative and agent-oriented programming language where agents have mental states consisting of beliefs, desires and intentions and communicate using *request* and *inform* performatives. The inputs of the MABLE compiler are the MABLE system and associated claims expressed in $\mathcal{MORA}$, a BDI logic. As output, MABLE generates a description of the MABLE system in Promela and a translation of the claims into LTL. In another work, Huget and Wooldridge [244] have used a variation of the MABLE language to define a semantics of agent communication and have shown that the compliance to

this semantics can be reduced to a model checking problem. In [430], Walton has applied model checking techniques in order to verify the correctness of agent protocol communication using the SPIN model checker. Benerecetti and Cimatti [33] have proposed a general approach for model-checking MASs together with modalities for BDI attitudes by extending symbolic model checking and using NuSMV[3] [98], a model checker for computation tree logic CTL. In [355], Lomuscio et al. have introduced a methodology for model checking multi-dimensional temporal-epistemic logic CTLK by extending NuSMV. The methodology is based on reducing the model checking of CTLK to the problem of model checking ARCTL, an extension of CTL with action labels and operators to reason about actions [335].

3.2.2 Developing New Algorithms and Tools

To model MASs, the authors in [354, 355] use the formalism of interpreted systems [205]. This formalism is defined as follows. Assume a set of agents $Ag = \{1,...,n\}$, where each agent i is characterized by a finite set of local states L_i and possible actions Act_i together with a protocol $P_i : L_i \rightarrow 2^{Act_i}$. The set $S = L_1 \times \cdots \times L_n \times L_E$ represents global states for the system where L_E is the set of local states associated to the environment. Agents' local states evolve in time according to the evolution function $t_i : L_i \times L_E \times Act \rightarrow L_i$, where $Act = Act_1 \times \cdots \times Act_n$. Given a set of initial global states $I \subseteq S$, the set of reachable states $Rs \subseteq S$ is generated by the possible runs of the system using the evolution function and the protocol. An interpretation system is then a tuple: $IS = \langle (L_i, Act_i, P_i, t_i)_{i\in Ag}, I, V\rangle$, where $V : S \rightarrow 2^{AP}$ is the evaluation function over the set of atomic propositions AP. The MAS is analyzed using a logic combining epistemic logic $S5_n$ with CTL logic. The syntax is as follows:
$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid EX\varphi \mid EG\varphi \mid E[\varphi U \varphi] \mid K_i\varphi$.
$K_i\varphi$ means i knows φ. The meanings of the other operators are as in CTL, where E is the existential path quantifier, X is the next operator, G is the globally operator and U is the until operator.

To evaluate the formulae, a Kripke model $M_{IS} = (Rs, I, R_t, \sim_1, \ldots, \sim_n, V)$ is associated with a given interpreted system IS. The temporal relation $R_t \subseteq Rs \times Rs$ is obtained using the protocols P_i and the evolutions functions t_i, and the epistemic relations $\sim_1, \ldots, \sim_n$ are defined by checking the equality of the i-th local component of two global states (i.e., $(l_1, \ldots, l_n) \sim_i (l'_1, \ldots, l'_n)$ iff $l_i = l'_i$). The semantics is defined in M_{IS} in the standard way.

To check the desired properties, the authors use symbolic model checking based on ordered binary decision diagrams (OBDDS). The model and formula to be checked are not represented as automata, but symbolically using boolean functions. This makes the technique efficient to deal with large systems. NuSMV [98] is

[3] The NuSMV2 model checker is open source and can be downloaded from:
http://nusmv.fbk.eu/NuSMV/download/getting-v2.html

the most popular symbolic model checker based on OBDDS. The MCMAS model checker[4] proposed in [355] is an extension of NuSMV for the epistemic properties. The idea is to represent the elements of the interpreted system M_{IS} by means of boolean formulas and then develop a propositional satisfiability solver (SAT) based on this representation for the verification of the properties associated with the interpreted system.

Agents' local states and actions are encoded as boolean vectors, which are identified by boolean formulae. Protocols and evolutions functions associated with local states and actions are also represented via boolean formulae. The SAT algorithm is an extension of CTL SAT solver for the knowledge operator $K_i\varphi$ whose semantics is defined using the accessibility relation $\sim_i$. Let R_i be the boolean function representing $\sim_i$, the SAT component of this operator is defined as follows:

$SAT_K(\varphi, i)\{$
$X = SAT(\neg\varphi)$
$Y = \{s | R_i(s) \cap X = \emptyset\}$
return $Y \cap Rs$
$\}$

The idea of the algorithm is to compute the set of global states X in which the negation of φ holds. Then, the set Y of states of which the $\sim_i$ accessible states are not in X is computed. This means that these states satisfy the semantics of $K_i\varphi$. Among these states, the algorithm returns those are reachable (i.e. those in Rs).

MCMAS model checker takes as input an interpreted system, which is parsed using Lex and Yacc parser. OBDDs are then built for the input parameters. The formula to be checked is then parsed and the SAT algorithm is executed to compute the set of states in which the formula holds, which is then compared with the set of reachable states. The tool is developed in C++.

In the same research direction, Penczek and Lomuscio [337] have developed a bounded model checking algorithm for branching time logic for knowledge (CTLK). In a similar way, Kacprzak et al. [266] have investigated the problem of verifying epistemic properties using CTLK by means of an unbounded model checking algorithm. Kacprzak and Penczek [267] have addressed the problem of verifying game-like structures by means of unbounded model checking. Recently, Cohen et al. [108] have introduced a new abstraction-based model checking technique for MASs aiming at saving representation space and verification time. The MAS is defined in the interpreted systems framework and the abstraction is performed by simplifying and collapsing the local states, local protocol and local evolution function of each agent in the system. Thus, the set L_i of local states of agent i is partitioned into equivalence classes called abstract local states of agent i. Similarly, the set ACT_i of possible actions of agent i is partitioned into equivalence classes called abstract actions of agent i. Local protocols and local evolution functions are abstracted by uniformly replacing any local state with its equivalence class

[4] The MCMAS model checker can be downloaded from:
http://www-lai.doc.ic.ac.uk/mcmas/download.html

and replacing any action with its equivalence class. The authors have shown that the resulting abstract system simulates the concrete system so that if a temporal-epistemic specification holds on the abstract system, the specification also holds on the concrete one.

3.3 Tableau-based Model Checking Dialogue Games

Unlike traditional proof systems which are bottom-up approaches, tableau-based algorithms used for model checking work in a *top-down* or *goal-oriented* fashion [106]. In the tableau-based approach, *tableau rules* are used in order to prove a certain formula by inferring when a state in a Kripke structure satisfies such a formula. According to this approach, we start from a goal (a formula), and we apply a tableau rule and determine the sub-goals (sub-formulae) to be proven. The tableau rules are designed so that the goal is true if all the sub-goals are true. The advantage of this method is that the state space to be checked is explored in a need-driven fashion [44]. The model checking algorithm searches only the part of the state space that needs to be explored to prove or disprove a certain formula. The state space is constructed while the algorithm runs. This kind of model checking algorithms is referred to as *on-the-fly* or *local* algorithms [44, 45, 106, 408].

The tableau decision algorithm that we use in our verification technique provides a systematic search for a model which satisfies a particular formula expressed using ACTL* logic. It is a graph construction algorithm. Nodes of the graph are sets of ACTL* formulae and tableau rule names. The interpretation of vertex labeling is that for the vertex to be satisfied, it must be possible to satisfy all the formulae in the set together. Each edge in the graph represents a satisfaction step of the formula contained in the starting vertex. These steps correspond to the application of a set of tableau rules. These rules express how the satisfaction of a particular formula (the goal) can be obtained by the satisfaction of its constituent formulae (sub-goals).

3.4 ACTL* Logic

3.4.1 Syntax

In this section, we present ACTL* logic that we use to specify dialogue game protocols and express the properties to be verified (See Fig. 3.1). This specification will be addressed in Section 3.5. ACTL* is a simplification of our logic for agent communication [38]. ACTL* extends CTL* by allowing formulae to constrain actions as well as propositions. The difference between ACTL* and CTL* is that the former contains action formulae and two new operators: SC for social commitments and $\therefore$

for arguments. The set of atomic propositions is denoted Γp. The set of action labels is denoted Γa. In what follows we use $p, p_1, p_2, \ldots$ to range over the set of atomic propositions and $\theta, \theta_1, \theta_2, \ldots$ to range over action labels. The syntax of this logic is as follows:

$$\mathcal{S} ::= p \mid \neg\mathcal{S} \mid \mathcal{S} \wedge \mathcal{S} \mid \mathcal{S} \vee \mathcal{S} \mid A\mathcal{P} \mid E\mathcal{P} \mid SC(Ag_1, Ag_2, \mathcal{P})$$

$$\begin{aligned}\mathcal{P} ::= {} & \theta \mid \neg\mathcal{P} \mid \mathcal{S} \mid \mathcal{P} \wedge \mathcal{P} \mid \mathcal{P} \vee \mathcal{P} \mid X\mathcal{P} \mid \mathcal{P} U \mathcal{P} \mid \mathcal{P} \therefore \mathcal{P} \\ & \mid ACT_1(Ag_1, SC(Ag_1, Ag_2, \mathcal{P})) \mid ACT_2(Ag_2, SC(Ag_1, Ag_2, \mathcal{P})) \\ & \mid ACT_1^+(Ag_1, SC(Ag_1, Ag_2, \mathcal{P}), \mathcal{P}) \mid ACT_2^+(Ag_2, SC(Ag_1, Ag_2, \mathcal{P}), \mathcal{P})\end{aligned}$$

$$ACT_1 ::= Cr \mid Wit \mid Sat \mid Vio$$

$$ACT_2 ::= Ac \mid Ref \mid Ch$$

$$ACT_1^+ ::= Def \mid Jus$$

$$ACT_2^+ ::= At$$

The formulae generated by $\mathcal{S}$ are called state formulae, while those generated by $\mathcal{P}$ are called path formulae. We use $\psi, \psi_1, \psi_2, \ldots$ to range over state formulae and $\phi, \phi_1, \phi_2, \ldots$ to range over path formulae. The formula $A\phi$ (respectively $E\phi$) means in all paths (resp. some paths) starting from the current state ϕ is satisfied. The formula $SC(Ag_1, Ag_2, \phi)$ means that agent Ag_1 commits towards agent Ag_2 that the path formula ϕ is true. Committing to path formulae is more expressive than committing to state formulae since state formulae are path formulae. In fact, by committing to path formulae, agents can commit to state formulae and express commitments toward the future, for example committing that $X\phi$ (ϕ holds from the next state), $\phi_1 U \phi_2$ (ϕ_1 holds until ϕ_2 becomes true) and $EF\phi$ (there is a path such that in its future ϕ holds)[5]. Ag_1 and Ag_2 are respectively called the *debtor* and *creditor* of the commitment. The formula $\phi_1 \therefore \phi_2$ means that ϕ_1 is an argument for ϕ_2. We can read this formula: ϕ_1, so ϕ_2. This operator introduces argumentation as a logical relation between path formulae. $Action(Ag, SC(Ag_1, Ag_2, \phi))$ and $Action^+(Ag, SC(Ag_1, Ag_2, \phi), \phi_1)$, where $Action$ corresponds to ACT_1 and ACT_2 and $Action^+$ corresponds to ACT_1^+ and ACT_2^+, indicate the action an agent Ag ($Ag \in \{Ag_1, Ag_2\}$) performs on $SC(Ag_1, Ag_2, \phi)$. The actions we consider are: Cr (create), Wit (withdraw), Sat (satisfy), Vio (violate), Ac (accept), Ref (refuse), Ch (challenge), At (attack), Def (defend) and Jus (justify).

[5] Operator F (in the future) is an abbreviation defined from operator U: $F\phi \equiv true U\phi$

3.4.2 Semantics

Semantically, this logic is interpreted with respect to the model M defined as follows: $M = \langle S_m, Lab, Act_m, \xrightarrow{Act_m}, Agt, R_{sc}, s_{m_0}\rangle$ where: S_m is a set of states; $Lab : S_m \rightarrow 2^{\Gamma p}$ is the labeling state function; Act_m is a set of actions; $\xrightarrow{Act_m} \subseteq S_m \times Act_m \times S_m$ is the transition relation; Agt is a set of communicating agents; $R_{sc} : S_m \times Agt \times Agt \rightarrow 2^{\sigma}$ with σ is the set of all paths in M is an accessibility modal relation that associates to a state s_m the set of paths along which an agent can commit towards another agent; s_{m_0} is the start state. The paths that path formulae are interpreted over have the form $x = s_{m_0} \xrightarrow{\alpha_1} s_{m_1} \xrightarrow{\alpha_2} s_{m_2} \ldots$ where $x \in \sigma$, $s_{m_0}, s_{m_1}, \ldots$ are states and $\alpha_1, \alpha_2, \ldots$ are actions. $x^i = s_{m_i} \xrightarrow{\alpha_{i+1}} s_{m_{i+1}} \ldots$ is the suffix of the path x starting from the *ith* state. The set of paths starting from a state s_m is denoted σ_m. $x[i]$ is the *ith* state in the path x. In the rest, $\Rightarrow$ stands for implies.
$s_m \models_M p$ iff $p \in Lab(s_m)$
$s_m \models_M \neg\psi$ iff not$(s_m \models_M \psi)$
$s_m \models_M \psi_1 \wedge \psi_2$ iff $s_m \models_M \psi_1$ and $s_m \models_M \psi_2$
$s_m \models_M \psi_1 \vee \psi_2$ iff $s_m \models_M \psi_1$ or $s_m \models_M \psi_2$

A state s_m satisfies $A\phi$ ($E\phi$) if every path (some path) emanating from this state satisfies ϕ. Formally:
$s_m \models_M A\phi$ iff $\forall x \in \sigma_m\ x \models_M \phi$
$s_m \models_M E\phi$ iff $\exists x \in \sigma_m\ x \models_M \phi$

A state s_m satisfies $SC(Ag_1, Ag_2, \phi)$ if every accessible path to Ag_1 towards Ag_2 from this state using R_{sc} satisfies ϕ. Formally:
$s_m \models_M SC(Ag_1, Ag_2, \phi)$ iff $\forall x \in R_{sc}(s_m, Ag_1, Ag_2)\ x \models_M \phi$.

A path satisfies a state formula if the initial state in the path does. Formally:
$x \models_M \psi$ iff $s_{m_0} \models_M \psi$

To define the satisfiability of action labels over paths, we introduce the notation $\theta \trianglerighteq \alpha_i$ where $i \geq 1$ to indicate that the action label θ becomes true when performing the action α_i, that is α_i brings about θ (for example, by performing the action of opening the door the action label "door is open" becomes true. If not, we write $\theta \ntrianglerighteq \alpha_i$. A path x satisfies an action label θ if θ is in the label of the first transition on this path and this path is not a *deadlocked* path. A path is deadlocked if it has no transitions. A path satisfies $\neg\theta$ if either θ is not in the label of the first transition on this path or this path is a deadlocked path. Formally:
$x \models_M \theta$ iff $\theta \trianglerighteq \alpha_1$ and *x is not a deadlocked path*
$x \models_M \neg\theta$ iff $\theta \ntrianglerighteq \alpha_1$ or *x is a deadlocked path*
where the action α_1 is the label of the first transition on the path x.
$x \models_M \neg\phi$ iff not$(x \models_M \phi)$

$x \models_M \phi_1 \wedge \phi_2$ iff $x \models_M \phi_1$ and $x \models_M \phi_2$
$x \models_M \phi_1 \vee \phi_2$ iff $x \models_M \phi_1$ or $x \models_M \phi_2$

X represents the next time operator and has the usual semantics when the path is not deadlocked. On a deadlocked path, $X\phi$ holds if the current state satisfies ϕ. Formally:
$x \models_M X\phi$ iff (x *is not a deadlocked path* $\Rightarrow x^1 \models_M \phi$) and
(x *is a deadlocked path* $\Rightarrow x[0] \models_M \phi$)

In the rest, the path x is supposed non-deadlocked. Along this path, $\phi_1 U \phi_2$ holds if ϕ_1 remains true along this path until ϕ_2 becomes true (strong until). Formally:
$x \models_M \phi_1 U \phi_2$ iff $\exists i \geq 0 : x^i \models_M \phi_2$ and $\forall j < i, x^j \models_M \phi_1$

Along a path x, $\phi_1 \therefore \phi_2$ holds if ϕ_1 is true and at next time if ϕ_1 is true then ϕ_2 is true. Formally:
$x \models_M \phi_1 \therefore \phi_2$ iff $x \models_M \phi_1$ and $x^1 \models_M \phi_1 \Rightarrow \phi_2$
Because the semantics of $\therefore$ operator is defined using existing operators, it is introduced here as a useful abbreviation, which will be used to define the semantics of some actions performed on SCs.

To specify dialogue game protocols in this logic according to the CAN framework, we use a set of actions performed by the communicating agents on SCs and SC contents. The idea behind the CAN framework is that agents communicate by performing actions on SCs (for example creating, accepting and challenging SCs) and by supporting these actions by argumentation relations (attack, defense, and justification). Such an approach, called the social approach [318] is considered as an alternative to the private approach based on the agents' mental states like beliefs, desires, and intentions [109]. The semantics of the action formulae is defined as follows:
$x \models_M Cr(Ag_1, SC(Ag_1, Ag_2, \phi))$ iff $\alpha_1 = Cr$ and $s_{m_1} \models_M SC(Ag_1, Ag_2, \phi)$
$x \models_M Wit(Ag_1, SC(Ag_1, Ag_2, \phi))$ iff $\alpha_1 = Wit$ and $s_{m_1} \models_M \neg SC(Ag_1, Ag_2, \phi)$
$x \models_M Sat(Ag_1, SC(Ag_1, Ag_2, \phi))$ iff $\alpha_1 = Sat$ and $s_{m_1} \models_M \phi$
$x \models_M Vio(Ag_1, SC(Ag_1, Ag_2, \phi))$ iff $\alpha_1 = Vio$ and $s_{m_1} \models_M \neg\phi$
$x \models_M Ac(Ag_2, SC(Ag_1, Ag_2, \phi))$ iff $\alpha_1 = Ac$ and $s_{m_1} \models_M SC(Ag_2, Ag_1, \phi)$
$x \models_M Ref(Ag_2, SC(Ag_1, Ag_2, \phi))$ iff $\alpha_1 = Ref$ and $s_{m_1} \models_M SC(Ag_2, Ag_1, \neg\phi)$
$x \models_M Ch(Ag_2, SC(Ag_1, Ag_2, \phi))$ iff $\alpha_1 = Ch$ and $s_{m_1} \models_M SC(Ag_2, Ag_1, ?\phi)$
$x \models_M At(Ag_2, SC(Ag_1, Ag_2, \phi_1), \phi_2)$ iff $\alpha_1 = At$ and $s_{m_1} \models_M SC(Ag_2, Ag_1, \phi_2 \therefore \neg\phi_1)$
$x \models_M Def(Ag_1, SC(Ag_1, Ag_2, \phi_1), \phi_2)$ iff $\alpha_1 = Def$ and $s_{m_1} \models_M SC(Ag_1, Ag_2, \phi_2 \therefore \phi_1)$
$x \models_M Jus(Ag_1, SC(Ag_1, Ag_2, \phi_1), \phi_2)$ iff $\alpha_1 = Jus$ and $s_{m_1} \models_M SC(Ag_1, Ag_2, \phi_2 \therefore \phi_1)$

$Cr(Ag_1, SC(Ag_1, Ag_2, \phi))$ is satisfied along the path x iff the first transition is labeled by Cr and the underlying commitment holds in the next state on that path. The semantics of the other formulae is defined in the same way. The commitment is withdrawn iff after performing the action, the commitment does not hold in the next state. It is satisfied (resp. violated) iff after the action, the content becomes true

(resp. false) in the next state. When Ag_2 accepts (resp. refuses) the commitment content, it becomes committed to the same content (resp. the negation of the same content) in the next state. For simplification reasons, the semantics of challenge is defined by introducing a syntactical construct "?" to indicate that the debtor Ag_2 of the resulting commitment $SC(Ag_2, Ag_1, ?\phi)$ does not have an argument supporting ϕ or $\neg\phi$. For the purpose of model checking dialogue games, this syntactical construct is useful for the tableau-based verification technique we will present in Section 3.6. The content ϕ_1 of Ag_1's commitment is attacked by Ag_2 using ϕ_2 iff after performing the attack action, Ag_2's commitment about $\phi_2 \therefore \neg\phi_1$ holds in the next state. Ag_1 defends its commitment (against an attack) and justifies it (against a challenge) iff after performing the action, the Ag_1's commitment about $\phi_2 \therefore \phi_1$ holds in the next state.

ACTL* logic is the fusion of CTL* logic and a logic for commitments. The logic for commitments has the following properties, where $\rightarrow$ is the classical implication:

1. R_{sc} is serial (axiom D);
2. R_{sc} is reflexive (axiom M) because accessible paths start from the current state where the commitment has been made and a formula is satisfied along a path if it is satisfied in the initial state of this path, which means on an accessible path we have $SC(Ag_1, Ag_2, \phi) \rightarrow \phi$
3. R_{sc} is transitive (axiom 4): $SC(Ag_1, Ag_2, \phi) \rightarrow SC(Ag_1, Ag_2, SC(Ag_1, Ag_2, \phi))$.

This makes the logic an $S4$ system.

3.4.3 Tableau Rules

In this section, we present the tableau rules that we use to translate the ACTL* property to be verified to an ABTA (see Fig. 3.1). The definition of ABTA and the translation procedure will be presented in Sections 3.6.1 and 3.6.2. The tableau rules allow us to build the ABTA representing the formula to be verified. These rules [106] are specified in terms of the decomposition of formulae to sub-formulae. They enable us to define top-down proof systems. The idea is: given a formula (the top part of the rule), we apply a tableau rule and determine the sub-formulae (the down part of the rule) to be proven (see Section 3.3). Tableau rules are inference rules used in order to prove a formula by proving all the sub-formulae. The labels of these rules are the labels of states in the ABTA constructed from the given formula (Section 3.6.1). These rules are presented in Table 3.1. In these rules, Φ is any set of path formulae. The symbol "," indicates a conjunction. For example, $E(\Phi, \psi)$ means that, there is a path along which the set of path formulae Φ and the state formula ψ are true. Adding the set Φ to these rules allows us to deal with any form of formulae written under the form of any set of path formulae and a formula of our logic. We

Table 3.1 Tableau rules

$$R1 \wedge : \frac{\psi_1 \wedge \psi_2}{\psi_1 \psi_2} \quad R2 \vee : \frac{\psi_1 \vee \psi_2}{\psi_1}, \frac{\psi_1 \vee \psi_2}{\psi_2} \quad R3 \vee : \frac{E(\psi)}{\psi} \quad R4 \neg : \frac{\neg\psi}{\psi} \quad R5 \neg : \frac{A(\Phi)}{E(\neg\Phi)}$$

$$R6 <Cr> : \frac{E(\Phi, Cr(Ag_1, SC(Ag_1, Ag_2, \phi)))}{E(\Phi, SC(Ag_1, Ag_2, \phi))} \quad R11 <Ac> : \frac{E(\Phi, Ac(Ag_2, SC(Ag_1, Ag_2, \phi)))}{E(\Phi, SC(Ag_2, Ag_1, \phi))}$$

$$R7 <Wit> : \frac{E(\Phi, Wit(Ag_1, SC(Ag_1, Ag_2, \phi)))}{E(\Phi, \neg SC(Ag_1, Ag_2, \phi))} \quad R12 <Ref> : \frac{E(\Phi, Ref(Ag_2, SC(Ag_1, Ag_2, \phi)))}{E(\Phi, SC(Ag_2, Ag_1, \neg\phi))}$$

$$R8 <Sat> : \frac{E(\Phi, Sat(Ag_1, SC(Ag_1, Ag_2, \phi)))}{E(\Phi, \phi)} \quad R13 <Jus> : \frac{E(\Phi, Jus(Ag_1, SC(Ag_1, Ag_2, \phi_1), \phi_2))}{E(\Phi, SC(Ag_1, Ag_2, \phi_2 \therefore \phi_1))}$$

$$R9 <Vio> : \frac{E(\Phi, Vio(Ag_1, SC(Ag_1, Ag_2, \phi)))}{E(\Phi, \neg\phi)} \quad R14 <At> : \frac{E(\Phi, At(Ag_2, SC(Ag_1, Ag_2, \phi_1), \phi_2))}{E(\Phi, SC(Ag_2, Ag_1, \phi_2 \therefore \neg\phi_1))}$$

$$R10 <Ch> : \frac{E(\Phi, Ch(Ag_2, SC(Ag_1, Ag_2, \phi)))}{E(\Phi, SC(Ag_2, Ag_1, ?\phi))} \quad R15 <Def> : \frac{E(\Phi, Def(Ag_1, SC(Ag_1, Ag_2, \phi_1), \phi_2))}{E(\Phi, SC(Ag_1, Ag_2, \phi_2 \therefore \phi_1))}$$

$$R16\ [SC_{Ag_1}] : \frac{E(\Phi, SC(Ag_1, Ag_2, \phi))}{E(\Phi, \phi)} \quad R17 <\equiv> : \frac{E(\Phi, \Psi)}{E(\Phi)E(\Psi)} \quad R18 \wedge : \frac{E(\Phi, \phi_1 \wedge \phi_2)}{E(\Phi, \phi_1, \phi_2)}$$

$$R19 \vee : \frac{E(\Phi, \phi_1 \vee \phi_2)}{E(\Phi, \phi_1)E(\Phi, \phi_2)} \quad R20\ X : \frac{E(\Phi, X\phi_1, \ldots, X\phi_n)}{E(\Phi, \phi_1, \ldots, \phi_n)} \quad R21 \wedge : \frac{E(\Phi, \phi_1 \therefore \phi_2)}{E(\Phi, \phi_1, X(\neg\phi_1 \vee \phi_2))}$$

$$R22 \vee : \frac{E(\Phi, \phi_1 U \phi_2)}{E(\Phi, \phi_2)E(\Phi, \phi_1, X(\phi_1 U \phi_2))}$$

also recall that we use $\psi, \psi_1, \psi_2, \ldots$ to range over state formulae and $\phi, \phi_1, \phi_2, \ldots$ to range over path formulae.

Rule $R1$ labeled by "$\wedge$" indicates that ψ_1 and ψ_2 are the two sub-formulae of $\psi_1 \wedge \psi_2$. This means that, in order to prove that a state labeled by "$\wedge$" satisfies the formula $\psi_1 \wedge \psi_2$, we have to prove that the two children of this state satisfy ψ_1 and ψ_2 respectively. According to rule $R2$, in order to prove that a state labeled by "$\vee$" satisfies the formula $\psi_1 \vee \psi_2$, we have to prove that one of the two children of this state satisfies ψ_1 or ψ_2. $R3$ labeled by "$\vee$" indicates that ψ is the sub-formula to be proved in order to prove that a state satisfies $E(\psi)$. E is the existential path-quantifier. According to $R4$, the formula $\neg\psi$ is satisfied in a state labeled by "$\neg$" if this state has a successor representing the sub-formula ψ, which is not satisfied. $R5$ is defined in the usual way.

The label "$<Cr>$" ($R6$) is the label associated with the creation action of a social commitment. According to this rule, in order to prove that a state labeled by "$<Cr>$" satisfies $Cr(Ag_1, SC(Ag_1, Ag_2, \phi))$, we have to prove that the child state satisfies the sub-formula $SC(Ag_1, Ag_2, \phi)$. The idea is that by creating a social commitment, this commitment becomes true in the child state. In the model representing the dialogue game protocol, the idea behind the creation action is that by creating a social commitment, this commitment becomes true in the accessible state via the transition labeled by the creation action. The label "$<Wit>$" ($R7$) is the label associated with the withdrawal action of a social commitment. According to this rule, in order to prove that a state labeled by "$<Wit>$" satisfies $Wit(Ag_1, SC(Ag_1, Ag_2, \phi))$, we have to prove that the child state satisfies the sub-formula $\neg SC(Ag_1, Ag_2, \phi)$. Rules $R8$ to $R15$ are defined in the same way. For example, the idea of rule $R11$ is that by accept-

ing a social commitment whose content is ϕ by an agent Ag_2, this agent commits about this content in the child state. In this state, the commitment of Ag_2 becomes true. In rule $R10$, we use the syntactical construct "?" introduced in Section 3.4.2 meaning that the debtor Ag_2 does not have an argument supporting ϕ or $\neg\phi$. The idea of this rule is that by challenging a social commitment, Ag_2 commits in the child state that it does not have an argument for or against the content ϕ. Because "?" is only a syntactical construct, $?\phi$ does not have a sub-formula, so there is no rule for "?".

Rule $R16$ indicates that $E(\phi)$ is the sub-formula of $E(SC(Ag_1, Ag_2, \phi))$. Thus, in order to prove that a state labeled by "$[SC_{Ag_1}]$" satisfies formula $E(SC(Ag_1, Ag_2, \phi))$, we have to prove that the child state satisfies the sub-formula $E(\phi)$. According to the semantics of social commitments (Section 3.4), the idea of this rule is that if an agent commits about a content along a path, this content is true along this path (we recall that the commitment content is a path formula).

Rules $R17$, $R18$, and $R19$ are straightforward. According to rule $R20$ and in accordance with the semantics of "X", in order to prove that a state labeled with "X" satisfies $E(X\phi)$, we have to prove that the child state satisfies the sub-formula $E(\phi)$. According to $R21$ and in accordance with the semantics of "$\therefore$" (Section 3.4), in order to prove that a state labeled with "$\wedge$" satisfies $E(\phi_1 \therefore \phi_2)$, we have to prove that the child state satisfies the sub-formula $E(\phi_1 \wedge X(\neg\phi_1 \vee \phi_2))$. This mean that the support is true and next if the support is true then the conclusion is true. Finally, rule $R22$ is defined in accordance with the usual semantics of *until* operator "U".

3.5 Dialogue Game Protocols as Transition Systems

In Section 3.4, we presented ACTL* logic and CAN-based actions. In this section, we specify the dialogue game protocols to be checked as models for this logic (see Fig. 3.1). This specification uses CAN-based actions and the labels of the tableau rules that we will present in Section 3.4.3. Dialogue game protocols are specified as a set of rules describing the entry condition, the dynamics and the exit condition [37]. These rules can be specified as CAN-based actions.

Dialogue game protocols are defined as TSs. The purpose of these TSs is to describe not only the sequence of the allowed actions (classical TSs), but also the tableau rules-based decomposition of these actions (Section 3.4.3). The states of these systems are sub-TSs (that we call decomposition TSs) describing the tableau rules-based decomposition of the actions labeling the entry transitions. Defining TSs in such a way allows us to verify: (1) The correctness of the protocol (if the model of the protocol satisfies the properties that the protocol should specify); (2) The compliance to the decomposition semantics of the communicative actions (if the specification of the protocol respects the decomposition semantics). In Section 3.6, we present a model checking procedure in order to verify both (1) and (2) at

the same time. The definition of the TSs of dialogue game protocols is given by the following definitions:

Definition 3.1 (Decomposition TSs). A decomposition transition system (DT) describing the tableau-rules-based decomposition semantics of a CAN based-action formula is a 7-tuple $\langle S', Lab', F, L', R, \xrightarrow{R}, s'_0 \rangle$ where: S' is a set of states; $Lab' : S' \rightarrow 2^{\Gamma p}$ is the labeling state function; F is a set of ACTL* formulae; $L' : S' \rightarrow 2^F$ is a function associating a set of formulae to a state; $R \in \{\wedge, \vee, \neg, <\!\equiv\!>, X, SC_{Ag}\}$ is a tableau rule label (without the rules for CAN-based action formulae) (see Section 3.4.3); $\xrightarrow{R} \subseteq S' \times R \times S'$ is the transition relation; s'_0 is the start state.

Intuitively, states S' contain the sub-formulae of the CAN-based action formulae, and the transitions are labeled by operators associated with the formula of the starting state. Decomposition TSs enable us to describe the decomposition semantics of formulae by sub-formulae connected by logical operators. Thus, there is a transition between states S'_i and S'_j iff $L'(S'_j)$ is a sub-formula of $L'(S'_i)$.

Definition 3.2 (TSs for Dialogue Game Protocols). A transition system T for a dialogue game protocol is a 7-tuple $\langle S, Lab, \wp, L, Act, \xrightarrow{Act}, s_0 \rangle$ where: S is a set of states; $Lab : S \rightarrow 2^{\Gamma p}$ is the labeling state function; $\wp$ is a set of decomposition TSs with $\varepsilon \in \wp$ is the empty decomposition TS; $L : S \rightarrow \wp$ is the function associating to a state $s \in S$ a decomposition transition system $DT \in \wp$ describing the tableau-based decomposition of the CAN-based action labeling the entry transition; Act is the set of CAN-based actions; $\xrightarrow{Act} \subseteq S \times Act \times S$ is the transition relation; s_0 is the start state with $L(s_0) = \varepsilon$.

We write $s \xrightarrow{\bullet} s'$ instead of $< s, \bullet, s' > \in \xrightarrow{Act}$ where $\bullet \in Act$. Fig. 3.2 illustrates a part of a TS for a dialogue game protocol. According to this protocol, if Ag_1 plays a creation game ($a1$), Ag_2 can, for instance, play a challenge game ($a2$). Thereafter, Ag_1 must plays a justification game ($a3$), etc.

States $S1$, $S2$, and $S3$ are decomposition TS associated respectively with creation, challenge, and justification actions. For example, for the creation action ($S1$), the first state ($s1.0$) is associated with the SC formula according to the rule $R6$ (Table 3.1, Section 3.4.3). The next state is associated with the SC content according to the rule $R16$ (Table 3.1). The transition is labeled with the label of this rule. An example of the properties to be verified in this protocol is:

$$AG(Ch(Ag_2, SC(Ag_1, Ag_2, \phi_1)) \Rightarrow F(Jus(Ag_1, SC(Ag_1, Ag_2, \phi_1), \phi_2))) \qquad (3.1)$$

This property says that in all paths (A) globally (G)[6], if an agent Ag_2 challenges (Ch) the content of a SC made by an agent Ag_1, then in the future (F), Ag_1 justifies (Jus) the content of its SC. In the rest of this chapter, we refer to this formula as **Formula 1**.

[6] Operator G (globally in the future) is an abbreviation defined from operator F: $G\phi \equiv \neg F \neg \phi$

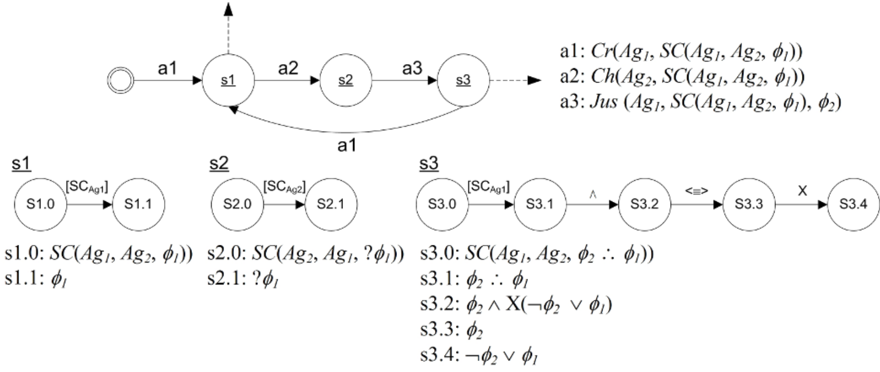

Fig. 3.2 A part of a transition system for a dialogue game protocol

3.6 Verification of Dialogue Game Protocols

In previous sections, we presented the elements needed for the verification of dialogue game protocols: the logic along with the associated tableau rules and the specification of dialogue game protocols. In this section, we present the verification technique, which is based upon (1) the ABTA for ACTL* logic (Section 3.6.1); and (2) the translation of the property to be verified to an ABTA (Section 3.6.2) (see Fig. 3.1). This translation is the step 1 of Fig. 3.1. The step 2, which is the construct of the product graph of the model and the ABTA is addressed in Section 3.6.3. Finally, the model checking algorithm acting on the product graph (step 3) is presented in Section 3.6.4. Examples illustrating each step are also presented.

*3.6.1 Alternating Büchi Tableau Automata (ABTA) for ACTL**

As a kind of Büchi automata, ABTAs [44] are used in order to prove properties of *infinite* behavior. These automata can be used as an intermediate representation for system properties. Let Γp be the set of atomic propositions and let $\mathfrak{R}$ be a set of tableau rule labels defined as follows: [7]
$\mathfrak{R} = \{\wedge, \vee, \neg\} \cup \mathfrak{R}_{Act} \cup \mathfrak{R}_{\neg Act} \cup \mathfrak{R}_{SC} \cup \mathfrak{R}_{Set}$ where: $\mathfrak{R}_{Act} = \{<Cr>, <Wit>, <Sat>, <Vio>, <Ch>, <Ac>, <Ref>, <Jus>, <At>, <Def>\}$, $\mathfrak{R}_{SC} = \{[SC_{Ag}]\}$, and $\mathfrak{R}_{Set} = \{<\equiv>, X\}$.
We define ABTAs for ACTL* logic as follows:

[7] The partition of the set of tableau rule labels is only used for readability and organizational reasons.

Definition 3.3 (ABTA). An ABTA for ACTL* is a 5-tuple $\langle Q, l, \rightarrow, q_0, F \rangle$, where: Q is a finite set of states; $l : Q \rightarrow \Gamma p \cup \Re$ is the state labeling; $\rightarrow \subseteq Q \times Q$ is the transition relation; q_0 is the start state; $F \subseteq 2^Q$ is the acceptance condition[8].

ABTAs allow us to encode "*top-down proofs*" for temporal formulae. Indeed, an ABTA encodes a proof schema in order to prove, in a goal-directed manner, that a TS satisfies a temporal formula. Let us consider the following example. We would like to prove that a state s in a TS satisfies a temporal formula of the form $F_1 \wedge F_2$, where F_1 and F_2 are two formulae. Regardless of the structure of the system, there would be two sub-goals. The first would be to prove that s satisfies F_1, and the second would be to prove that s satisfies F_2. Intuitively, an ABTA for $F_1 \wedge F_2$ would encode this "proof structure" using states for the formulae $F_1 \wedge F_2$, F_1, and F_2. A transition from $F_1 \wedge F_2$ to each of F_1 and F_2 should be added to the ABTA and the labeling of the state for $F_1 \wedge F_2$ being "$\wedge$" which is the label of a certain rule. Indeed, in an ABTA, we can consider that: 1) states correspond to "formulae", 2) the labeling of a state is the "logical operator" used to construct the formula, and 3) the transition relation represents a "sub-goal" relationship.

3.6.2 Translating ACTL* into ABTA (Step 1)

The procedure for translating an ACTL* formula $p = E(\phi)$ to an ABTA B uses goal-directed rules in order to build a tableau from this formula. Indeed, these proof rules are conducted in a top-down fashion in order to determine if states satisfy properties. The tableau is constructed by exhaustively applying the tableau rules presented in Table 3.1 to p. Then, B can be extracted from this tableau as follows. First, we generate the states and the transitions. Intuitively, states will correspond to state formulae, with the start state being p. To generate new states from an existing state for a formula p', we determine which rule is applicable to p', starting with $R1$, by comparing the form of p' to the formula appearing in the "*goal position*" of each rule. Let *rule*(q) denote the rule applied at node q. The labeling function l of states is defined as follows. If q does not have any successor, then $l(q) \in \Gamma p$. Otherwise, the successors of q are given by *rule*(q). The label of the rule becomes the label of the state q, and the sub-goals of the rule are then added as states related to q by transitions.

A tableau for a ACTL* formula p is a maximal proof tree having p as its root and constructed using our tableau rules (see Section 3.4.3). If p' results from the application of a rule to p, then we say that p' is a child of p in the tableau. The height of a tableau is defined as the length of the longest sequence $< p_0, p_1, \ldots >$, where p_{i+1} is the child of p_i [106].

---

[8] The notion of acceptance condition is related to the notion of accepting run that we define in Section 3.6.3.

Example 3.1. In order to illustrate the translation procedure and the construction of an ABTA from an ACTL* formula, let us consider our formula Formula 1 given in Section 3.5. Table 3.2 is the tableau to build for translating Formula 1 into an ABTA. The form of Formula 1 is: $AG(p \Rightarrow q)(\equiv AG(\neg p \vee q))$ (the root of Table 3.2). The first rule we can apply is $R5$ labeled by $\neg$ in order to transform *all paths* to *exists a path*. We also use the equivalence $(F(p) \equiv \neg G(\neg p))$. We then obtain the child number (2). The next rule we can apply is $R22$ labeled by $\vee$ because F is an abbreviation of U $(F(p) \equiv True\ U\ p)$. Consequently, we obtain two children (3) and (4). From the child (3) we obtain the child (5) by applying the rule $R10$, and from the child (4) we obtain the child (2) by applying the rule $R20$ etc. The ABTA obtained from this tableau is illustrated by Fig. 3.3. States are labeled by the child's number in the tableau and the label of the applied rule according to Table 3.2.

Table 3.2 The tableau of Formula 1

$\neg : AG(\neg Ch(Ag_2,SC(Ag_1,Ag_2,\phi_1)) \vee F(Jus(Ag_1,SC(Ag_1,Ag_2,\phi_1),\phi_2)))$ **(1)**	
$\vee : EF(Ch(Ag_2,SC(Ag_1,Ag_2,\phi_1)) \wedge G(\neg Jus(Ag_1,SC(Ag_1,Ag_2,\phi_1),\phi_2)))$ **(2)**	
$\langle Ch\rangle: E(Ch(Ag_2,SC(Ag_1,Ag_2,\phi_1)) \wedge G(\neg Jus(Ag_1,SC(Ag_1,Ag_2,\phi_1),\phi_2)))$ **(3)**	$\langle X\rangle: EX(F(Ch(Ag_2,SC(Ag_1,Ag_2,\phi_1)) \wedge G(\neg Jus(Ag_1,SC(Ag_1,Ag_2,\phi_1),\phi_2))))$ **(4)**
$[SC_{Ag_2}] : E(SC(Ag_2,Ag_1,?\phi_1) \wedge G(\neg Jus(Ag_1,SC(Ag_1,Ag_2,\phi_1),\phi_2)))$ **(5)**	$EF(Ch(Ag_2,SC(Ag_1,Ag_2,\phi_1)) \wedge G(\neg Jus(Ag_1,SC(Ag_1,Ag_2,\phi_1),\phi_2)))$ **(2)**
$\langle\equiv\rangle: E(?\phi_1 \wedge G(\neg Jus(Ag_1,SC(Ag_1,Ag_2,\phi_1),\phi_2)))$ **(6)**	
$?\phi_1$ **(7)**	$\vee : E(G(\neg Jus(Ag_1,SC(Ag_1,Ag_2,\phi_1),\phi_2)))$ **(8)**
	$\langle\neg Jus\rangle: E(\neg Jus(Ag_1,SC(Ag_1,Ag_2,\phi_1),\phi_2), XG(\neg Jus(Ag_1,SC(Ag_1,Ag_2,\phi_1),\phi_2)))$ **(9)**
	$[SC_{Ag_1}] : E(SC(Ag_1,Ag_2,\phi_1 \therefore \phi_2), XG(\neg Jus(Ag_1,SC(Ag_1,Ag_2,\phi_1),\phi_2)))$ **(10)**
	$\wedge : E(\phi_2 \therefore \phi_1, XG(\neg Jus(Ag_1,SC(Ag_1,Ag_2,\phi_1),\phi_2)))$ **(11)**
	$\langle\equiv\rangle: E(\phi_2, X(\neg\phi_2 \vee \phi_1), XG(\neg Jus(Ag_1,SC(Ag_1,Ag_2,\phi_1),\phi_2)))$ **(12)**
ϕ_2 **(13)**	$X : E(X(\neg\phi_2 \vee \phi_1), XG(\neg Jus(Ag_1,SC(Ag_1,Ag_2,\phi_1),\phi_2)))$ **(14)**
	$\langle\equiv\rangle: E((\neg\phi_2 \vee \phi_1), XG(\neg Jus(Ag_1,SC(Ag_1,Ag_2,\phi_1),\phi_2)))$ **(15)**
$\neg\phi_2 \vee \phi_1$ **(16)**	$X : E(XG(\neg Jus(Ag_1,SC(Ag_1,Ag_2,\phi_1),\phi_2)))$ **(17)**
	$\vee : E(G(\neg Jus(Ag_1,SC(Ag_1,Ag_2,\phi_1),\phi_2)))$ **(8)**

The termination proof of the translation procedure is based on the finiteness of the tableau. This proof is based on the length of formulae and an ordering relation between these formulae. The proof is detailed in [35].

3.6.3 Run of an ABTA on a Transition System (Step 2)

Like the automata-based model checking of LTL, in order to decide about the satisfaction of formulae, we use the notion of the *accepting runs*. In our technique, we need to define accepting runs of an ABTA on a TS. Firstly, we have to define the

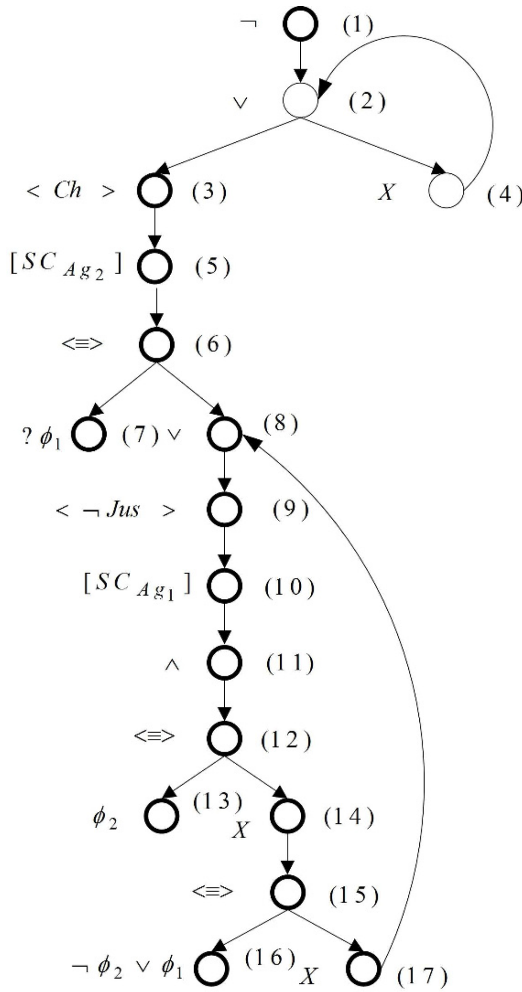

Fig. 3.3 The ABTA of Formula 1

notion of *ABTA's run*. For this reason, we need to introduce two types of nodes: *positive* and *negative*. Intuitively, nodes classified positive are nodes that correspond to a formula without negation, and negative nodes are nodes that correspond to a formula with negation. Definition 3.4 gives the definition of this notion of run. In this definition, elements of the set S of states are denoted s_i or t_i.

Definition 3.4 (Run of an ABTA). A run of an ABTA $B = \langle Q, l, \rightarrow, q_0, F \rangle$ on a transition system $T = \langle S, Lab, \wp, L, Act, \xrightarrow{Act}, s_0 \rangle$ is a graph in which the nodes are classified as positive or negative and are labeled by elements of $Q \times S$ as follows:

1. The root of the graph is a positive node and is labeled by $< q_0, s_0 >$.

2. For a positive node φ with label $<q, s_i>$:

   a. If $l(q) = \neg$ and $q \to q'$, then φ has one negative successor labeled $<q', s_i>$ and vice versa.

   b. If $l(q) \in \Gamma p$, then φ is a leaf.

   c. If $l(q) \in \{\wedge, <\!\!\equiv\!\!>\}$ and $\{q' | q \to q'\} = \{q_1, \ldots, q_m\}$, then φ has positive successors $\varphi_1, \ldots, \varphi_m$ with φ_j labeled by $<q_j, s_i>$ $(1 \leq j \leq m)$.

   d. If $l(q) = \vee$, then φ has one positive successor φ' labeled by $<q', s_i>$ for some $q' \in \{q' | q \to q'\}$.

   e. If $l(q) = X$ and $q \to q'$ and $\{s' | s_i \stackrel{\bullet}{\longrightarrow} s'\} = \{t_1, \ldots, t_m\}$ where $\bullet \in Act$, then φ has positive successors $\varphi_1, \ldots, \varphi_m$ with φ_j labeled by $<q', t_j>$ $(1 \leq j \leq m)$.

   f. If $l(q) = <\bullet>$ where $\bullet \in Act$ and $q \to q'$, and $s_i \stackrel{\bullet}{\longrightarrow} s_{i+1}$, then φ has one positive successor φ' labeled by $<q', s_{i+1,0}>$ where $s_{i+1,0}$ is the initial state of the decomposition TS of s_{i+1}.

   g. If $l(q) = <\bullet>$ where $\bullet \in \neg Act$ and $q \to q'$, and $s_i \stackrel{\bullet'}{\longrightarrow} s_{i+1}$ where $\bullet \neq \bullet'$ and $\bullet' \in Act$, then φ has one positive successor φ' labeled by $<q', s_{i+1}>$.

3. For a negative node φ labeled by $<q, s_i>$:

   a. If $l(q) \in \Gamma p$, then φ is a leaf.

   b. If $l(q) \in \{\vee, <\!\!\equiv\!\!>\}$ and $\{q' | q \to q'\} = \{q_1, \ldots, q_m\}$, then φ has negative successors $\varphi_1, \ldots, \varphi_m$ with φ_j labeled by $<q_j, s_i>$ $(1 \leq j \leq m)$.

   c. If $l(q) = \wedge$, then φ has one negative successor φ' labeled by $<q', s_i>$ for some $q' \in \{q' | q \to q'\}$.

   d. If $l(q) = X$ and $q \to q'$ and $\{s' | s_i \stackrel{\bullet}{\longrightarrow} s'\} = \{t_1, \ldots, t_m\}$ where $\bullet \in Act$, then φ has negative successors $\varphi_1, \ldots, \varphi_m$ with φ_j labeled by $<q', t_j>$ $(1 \leq j \leq m)$.

   e. If $l(q) = <\bullet>$ where $\bullet \in Act$ and $q \to q'$, and $s_i \stackrel{\bullet}{\longrightarrow} s_{i+1}$, then φ has one negative successor φ' labeled by $<q', s_{i+1,0}>$ where $s_{i+1,0}$ is the initial state of the decomposition TS of s_{i+1}.

   f. If $l(q) = <\bullet>$ where $\bullet \in \neg Act$ and $q \to q'$, and $s_i \stackrel{\bullet'}{\longrightarrow} s_{i+1}$ where $\bullet \neq \bullet'$ and $\bullet' \in Act$, then φ has one negative successor φ' labeled by $<q', s_{i+1}>$.

4. Otherwise, for a positive (negative) node φ labeled by $<q, s_{i,j}>$:

   a. If $l(q) = <\!\!\equiv\!\!>$ and $\{q' | q \to q'\} = \{q_1, q_2\}$ such that q_1 is a leaf, and $s_{i,j}$ has a successor $s_{i,j+1}$, then φ has one positive leaf successor φ' labeled by $<q_1, s_{i,j}>$ and one positive (negative) successor φ'' labeled by $<q_2, s_{i,j+1}>$.

   b. If $l(q) = <\!\!\equiv\!\!>$ and $\{q' | q \to q'\} = \{q_1, q_2\}$ such that q_1 is a leaf, and $s_{i,j}$ has no successor, then φ has one positive leaf successor φ' labeled by $<q_1, s_{i,j}>$ and one positive (negative) successor φ'' labeled by $<q_2, s_i>$.

c. If $l(q) \in \{\wedge, \vee, X, [SC_{Ag}]\}$ and $\{q' | q \rightarrow q'\} = \{q_1\}$, and $s_{i,j} \xrightarrow{r} s_{i,j+1}$ such that $r = l(q)$, then φ has one positive (negative) successor φ' labeled by $< q_1, s_{i,j+1} >$.

The notion of run of an ABTA on a TS is a non-synchronized product graph of the ABTA and TS (see Fig. 3.1). This run uses the label of nodes in the ABTA ($l(q)$), transitions in the ABTA ($q \rightarrow q'$), and transitions in the TS ($s_i \xrightarrow{\bullet} s_j$). The product is not synchronized in the sense that it is possible to use transitions in the ABTA while staying in the same state in the TS (this is the case for example of clauses $2.a$, $2.c$, and $2.d$).

The clause $2.a$ in the definition says that if we have a positive node φ in the product graph such that the corresponding state in the ABTA is labeled with $\neg$ and we have a transition $q \rightarrow q'$ in this ABTA, then φ has one negative successor labeled with $< q', s_i >$. In this case we use a transition from the ABTA and we stay in the same state of the TS. In the case of a positive node and if the current state of the ABTA is labeled with $\wedge$, all the transitions of this current state of the ABTA are used (clause $2.c$). However, if the current state of the ABTA is labeled with $\vee$, only one arbitrary transition from the ABTA is used (clause $2.d$). The intuitive idea is that in the case of $\wedge$, all the sub-formulae must be true in order to decide about the formula of the current node of the ABTA. However, in the case of $\vee$ only one sub-formula must be true.

The cases in which a transition of the TS is used are:

1. The current node of the ABTA is labeled with X (which means a next state in the TS). This is the case of clauses $2.e$ and $3.d$. In this case we use all the transitions from the current state s_i to next states of the TS.
2. The current state of the ABTA and a transition from the current state of the TS are labeled with the same action. This is the case of clauses $2.f$ and $3.e$. In this case, the current transition of the ABTA and the transition from the current state s_i of the TS to a state $s_{i+1,0}$ of the associated decomposition TS are used. The idea is to start the parsing of the formula coded in the decomposition TS.
3. The current state of the ABTA and a transition from the current state of the TS are labeled with different actions where the state of the ABTA is labeled with a negative formula. This is the case of clauses $2.g$ and $3.f$. In this case, the formula is satisfied. Consequently, the current transition of the ABTA and the transition from the current state s_i of the TS to a next state s_{i+1} are used. Finally, clauses $4.a$, $4.b$, and $4.c$ deal with the case of verifying the structure of the commitment formulae in the sub-TS. In these clauses, transitions $s_{i,j} \xrightarrow{r} s_{i,j+1}$ are used. We note here that when $s_{i,j}$ has no successor, the formula contained in this state is an atomic formula or a boolean formula whose all the sub-formulae are atomic (for example $p \wedge q$ where p and q are atomic).

Example 3.2. Fig. 3.4 illustrates an example of the run of an ABTA. This figure illustrates a part of the automaton $B_{\otimes}$ resulting from the product of the TS of Fig. 3.2 and the ABTA of Fig. 3.3. According to the clause 1 (Definition 3.4), the root is

a positive node and it is labeled by $<\neg, s_0>$ because the label of the ABTA's root is $\neg$ (Fig. 3.3). Consequently, according to the clause 2.a, the successor is a negative node and it is labeled by $<\vee, s_0>$. According to the clause 3.b, the second node has two negative successors labeled by $<<Ch>, s_0>$ and $< X, s_0 >$ etc.

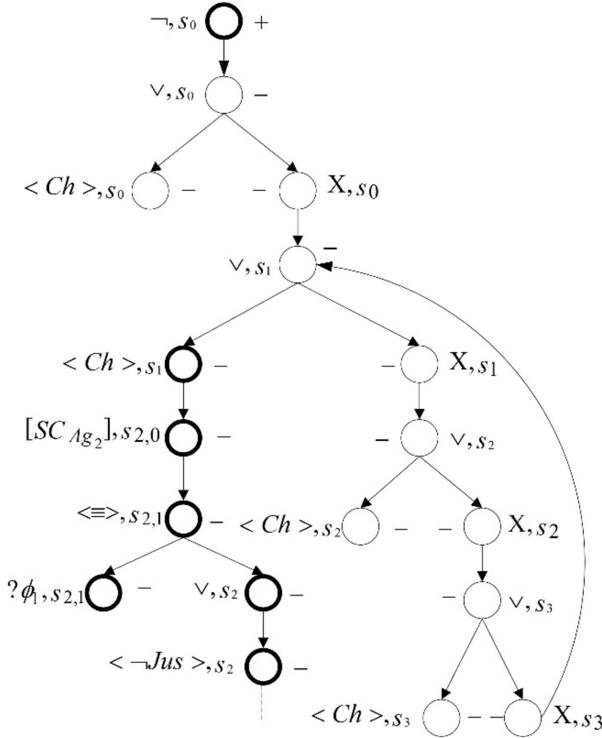

Fig. 3.4 An example of an ABTA's run

In an ABTA, every infinite path has a suffix that contains either positive or negative nodes, but not both. Such a path is referred to as positive in the former case and negative in the latter. Now we can define the notion of accepting runs (or successful runs). Let $p \in \Gamma p$ and let s_i be a state in a TS T. Then $s_i \models_T p$ iff $p \in Lab(s_i)$ and $s_i \models_T \neg p$ iff $p \notin Lab(s_i)$. Let $s_{i,j}$ be a state in a decomposition TS of a TS T. Then $s_{i,j} \models_T p$ iff $p \in Lab'(s_{i,j})$ and $s_{i,j} \models_T \neg p$ iff $p \notin Lab'(s_{i,j})$.

Definition 3.5 (Successful Run). Let r be a run of an ABTA $B = \langle Q, l, \rightarrow, q_0, F\rangle$ on a TS $T = \langle S, Lab, \wp, L, Act, \xrightarrow{Act}, s_0\rangle$. The run r is successful iff every leaf and every infinite path in r is successful. A successful leaf is defined as follows:

1. A positive leaf labeled by $<q, s_i>$ is successful iff $s_i \models_T l(q)$ or $l(q) = <\bullet>$ where $\bullet \in Act$ and there is no s_j such that $s_i \xrightarrow{\bullet} s_j$.

2. A positive leaf labeled by $<q, s_{i,j}>$ is successful iff $s_{i,j} \models_T l(q)$
3. A negative leaf labeled by $<q, s_i>$ is successful iff $s_i \models_T \neg l(q)$ or $l(q) = <\bullet>$ where $\bullet \in Act$ and there is no s_j such that $s_i \xrightarrow{\bullet} s_j$.
4. A negative leaf labeled by $<q, s_{i,j}>$ is successful iff $s_{i,j} \models_T \neg l(q)$

A successful infinite path is defined as follows:

1. A positive path is successful iff $\forall f \in F, \exists q \in f$ such that q occurs infinitely often in the path. This condition is called the Büchi condition.
2. A negative path is successful iff $\exists f \in F, \forall q \in f, q$ does not occur infinitely often in the path. This condition is called the co-Büchi condition.

We note here that a positive or negative leaf labeled by $<q, s>$ such that $l(q) = <\bullet>$ where $\bullet \in Act$ and there is no s' such that $s \xrightarrow{\bullet} s'$ is considered a successful leaf. The reason is that it is possible to find a transition labeled by $\bullet$ and starting from another state s'' in the TS. In fact, if we consider such a leaf unsuccessful, then even if we find a successful infinite path, the run will be considered unsuccessful, which is false.

An ABTA B accepts a TS T iff there exists a successful run of B on T. In order to compute the successful run of the generating ABTA, we should compute the acceptance states F. For this purpose we use the following definition.

Definition 3.6 (Acceptance States). Let q be a state in an ABTA B and Q the set of all states. Suppose $\phi = \phi_1 U \phi_2 \in q$ [9]. We define the set F_ϕ as follows: $F_\phi = \{q' \in Q | (\phi \notin q' \textit{ and } X\phi \notin q') \textit{ or } \phi_2 \in q'\}$. The acceptance set F is defined as follows: $F = \{F_\phi | \phi = \phi_1 U \phi_2 \textit{ and } \exists q \in B, \phi \in q\}$.

According to this definition, a state that contains the formula ϕ or the formula $X\phi$ is not an acceptance state. The reason is that according to Definition 3.4, there is a transition from a state containing ϕ to a state containing $X\phi$ and vice versa. Therefore, according to Definition 3.5, there is a successful run in the ABTA B. However, we can not decide about the satisfaction of a formula using this run. The reason is that in an infinite cycle including a state containing ϕ and a state containing $X\phi$, we can not be sure that a state containing ϕ_2 is reachable. However, according to the semantics of U, the satisfaction of ϕ needs that a state containing ϕ_2 is reachable while passing by states containing ϕ_1.

Example 3.3. In order to compute the acceptance states of the ABTA of Fig. 3.3, we use the formula associated with the child number (2) in Table 3.2:

$$F(Ch(Ag_2, SC(Ag_1, Ag_2, \phi_1)) \wedge G(\neg Jus(Ag_1, SC(Ag_1, Ag_2, \phi_1)\phi_2)))$$

[9] Here we consider until formula because it is the formula that allows paths to be infinite.

We consider this formula, denoted ϕ, instead of the root's formula because its form is $E(\phi)$ (see Section 3.6.2). Consequently, state (1) and states from (3) to (17) are the acceptance states according to Definition 3.6. For example, state (1) is an acceptance state because ϕ and $X\phi$ are not in this state, and state (3) is an acceptance state because ϕ_2 is in this state. States (2) and (4) are not acceptance states. Because only the first state is labeled by $\neg$, all finite and infinite paths are negative paths. Consequently, the only infinite path that is a valid proof of Formula 1 is (1, (2, 4)*). In this path there is no acceptance state that occurs infinitely often. Therefore, this path satisfies the Büchi condition. The path visiting the state (3) and infinitely often the state (9) does not satisfy Formula 1 because there is a challenge action (state (3)), and globally no justification action of the content of the challenged commitment (state (9)).

3.6.4 Model Checking Algorithm (Step 3)

Our model checking algorithm (see Fig. 3.5) for verifying that a dialogue game protocol satisfies a given property and that it respects the decomposition semantics of the underlying communicative acts is inspired by the procedure proposed by [44]. Like the algorithm proposed by [117], our algorithm explores the product graph of an ABTA representing an ACLT* formula and a TS for a dialogue game protocol. This algorithm is on-the-fly (or local) algorithm that consists of checking if a TS is accepted by an ABTA. This ABTA-based model checking is reduced to the emptiness of the Büchi automata [422]. The emptiness problem of automata is to decide, given an automaton A, whether its language $L(A)$ is empty. The language $L(A)$ is the set of words accepted by A.

Let $T = \langle S, Lab, \wp, L, Act, \xrightarrow{Act}, s_0\rangle$ be a TS for a dialogue game and let $B = \langle Q, l, \rightarrow, q_0, F\rangle$ be an ABTA for ACTL*. The procedure consists of building the ABTA product $B_\otimes$ of T and B while checking if there is a successful run in $B_\otimes$. The existence of such a run means that the language of $B_\otimes$ is non-empty. The automaton $B_\otimes$ is defined as follows: $B_\otimes = \langle Q \times S, \rightarrow_{B_\otimes}, q_{0B_\otimes}, F_{B_\otimes}\rangle$. There is a transition between two nodes $<q, s>$ and $<q', s'>$ iff there is a transition between these two nodes in some run of B on T. Intuitively, $B_\otimes$ simulates all the runs of the ABTA. The set of accepting states $F_{B_\otimes}$ is defined as follows: $q_{0B_\otimes} \in F_{B_\otimes}$ iff $q \in F$.

Unlike the algorithms proposed in [44, 117], our algorithm uses only one depth-first search (DFS) instead of two. This is due to the fact that our algorithm explores directly the product graph using the sign of the nodes (positive or negative). In addition, our algorithm does not distinguish between recursive and non-recursive nodes. Therefore, we do not take into account the strongly-connected components in the ABTA, but we use a marking algorithm that directly works on the product graph.

The idea of this algorithm is to construct the product graph while exploring it. The construction procedure is directly obtained from Definition 3.4. The algorithm

uses the label of nodes in the ABTA, and the transitions in the product graph obtained from the TS and the ABTA as explained in Definition 3.4. In order to decide if the ABTA contains an infinite successful run, all the explored nodes are marked "visited". Thus, when the algorithm explores a visited node, it returns false if the infinite path is not successful. If the node is not already visited, the algorithm tests if it is a leaf. In this case, it returns false if the node is a non-successful leaf. If the explored node is not a leaf, the algorithm explores recursively the successors of this node. If this node is labeled by "$\wedge$", and signed positively, then it returns false if one of the successors is false. However, if the node is signed negatively, it returns false if all the successors are false. A dual treatment is applied when the node is labeled by "$\vee$".

Example 3.4. In order to check if the language of the automaton illustrated by Fig. 3.4 is empty, we check if there is a successful run. The idea is to verify if $B_{\otimes}$ contains an infinite path visiting the state (3) and infinitely often the state (9) of the ABTA of Fig. 3.3. If such a path exists, then we conclude that Formula 1 is not satisfied by the TS of Fig. 3.2. Indeed, the only infinite path of $B_{\otimes}$ is successful because it does not touch any accepted state and all leaves are also successful. For instance, the leaf labeled by $(<Ch>, s_0)$ is successful since there is no state s_i such that $s_0 \xrightarrow{Ch} s_i$. Therefore, the TS of Fig. 3.2 is accepted by the ABTA of Formula 1. Consequently, this TS satisfies Formula 1 and respects its decomposition semantics.

Soundness and completeness of our model checking method are stated by the following theorem.

Theorem 3.1 (Soundness and Completeness). *Let ψ be a ACTL* formula and B_{ψ} the ABTA obtained by the translation procedure described above, and let $T = \langle S, Lab, \wp, L, Act, \xrightarrow{Act}, s_0 \rangle$ be a TS that represents a dialogue game protocol. Then, $s_0 \models_T \psi$ iff T is accepted by B_{ψ}.*

Proof. (**Direction** $\Rightarrow$). To prove that T is accepted by B_{ψ}, we have to prove that there exists a run r of B_{ψ} on T such that all leaves and all infinite paths in the run are successful. Let us assume that $s_0 \models_T \psi$. First, let us suppose that there exists a leaf $<q, s>$ in r such that $s \models_T \neg l(q)$. Since the application of tableau rules does not change the satisfaction of formulae, it follows from Definition 3.4 that $s_0 \models_T \neg\psi$ which contradicts our assumption.
Now, we will prove that all infinite paths are successful. The proof proceeds by contradiction. ψ is a state formula that we can write under the form $E\Phi$, where Φ is a set of path formulae. Let us assume that there exists an unsuccessful infinite path x_r in r and prove that $x_T \models_T \neg\Phi$ where x_T is the path in T that corresponds to x_r (x_r is the product of B_{ψ} and T). The fact that x_r is infinite implies that $R22$ occurs at infinitely many positions in x_r. Because x_r is unsuccessful, $\exists \phi_1, \phi_2, q_i$ such that $\phi_1 U \phi_2 \in q_i$ and $\forall j \geq i$ we have $\phi_2 \notin q_j$. When this formula appears in the ABTA at the position q_i, we have $l(q_i) = \vee$. Thus, according to Definition 3.4 and the form of $R22$, the current node φ_1 of r labeled by $<q_i, s>$ has one successor φ_1 labeled

```
DFS(v = (q, s)): boolean {
    if v marked visited {
      if (sign(v) = "+" and not accepting(v)) or (sign(v) = "-" and accepting(v))
        return false
    } // end of if v marked visited
    else {
      mark v visited
      switch(l(q)) {
          case (p ∈ Γp):
              switch(sign(v)) {
                  case("+"): if s is a sub-state and l(q) ∉ L'(s) return false
                  case("-"): if s is a sub-state and l(q)) ∉ L'(s) return false
                  case("neutral"): return false
              } // end of switch(sign(v))
          case(∧):
              if s is a leaf return false
              else
                  switch(sign(v)) {
                      case(neutral): for all v'' ∈ {v' / v →B⊗ v'}
                                        if not DFS(v'') return false
                      case("+"): for all v'' ∈ {v' / v →B⊗ v'}
                                    if not DFS(v'') return false
                      case("-"): for all v'' ∈ {v' / v →B⊗ v'}
                                   if DFS(v'') return true else return false
                  } // end of switch(sign (v))
          case(∨):
              if s is a leaf return false
              else
                  switch(sign(v)) {
                      case(neutral): for all v'' ∈ {v' / v →B⊗ v'}
                                        if DFS(v'') return true else return false
                      case("+"): for all v'' ∈ {v' / v →B⊗ v'}
                                    if DFS(v'') return true else return false
                      case("-"): for all v''  {v' / v →B⊗ v'}
                                   if not DFS(v'') return false
                  } // end of switch(sign (v))
          case(<•>):
              if s is a leaf return true
              else for the v'' ∈ {v' / v →B⊗ v'} if not DFS(v'') return false
          case(X, SC_Ag, <≡>, ?):
              if s is a leaf return false
              else for the v'' ∈ {v' / v →B⊗ v'} if not DFS(v'') return false
      } // end of switch(l(q))
    } // end of else
    return true }
```

Fig. 3.5 The model checking algorithm

by $<q_{i+1}, s>$ with $\phi_1 U \phi_2 \in q_i$ and $\{\phi_1, X(\phi_1 U \phi_2)\} \subseteq q_{i+1}$. Therefore, $l(q_{i+1}) = \wedge$, and φ_2 has a successor φ_3 labeled by $<q_{i+2}, s>$ with $X(\phi_1 U \phi_2) \in q_{i+2}$. Using $R20$ and the fact that $l(q_{i+2}) = X$, the successor φ_4 of φ_3 is labeled by $<q_{i+3}, s'>$ with $\phi_1 U \phi_2 \in q_{i+3}$ and s' is a successor of s. This process will be repeated infinitely since the path is unsuccessful. It follows that there is no s in T such that $s \models_T \phi_2$. Thus, according to the semantics of U, there is no s in T such that $s \models_T \phi_1 U \phi_2$. Therefore, $x_T \models_T \neg\Phi$.
(**Direction** $\Leftarrow$). The proof proceeds by an inductive construction of x_r and an analysis of the different tableau rules. A detailed proof of this theorem is presented in [35].

3.7 Case Studies

In this section, we will exemplify the model checking technique presented in this chapter by means of two case studies: 1) the persuasion/negotiation protocol for agent-based web services ($\mathcal{PNAWS}$) [36]; and 2) the NetBill protocol, a system of micropayments for goods on the Internet [405]. We will also discuss their implementations using an extension of the Concurrency Workbench of New Century (CWB-NC) model checker[10] [107, 446], which has been used to check many large-scale protocols in communication networking and process control systems. As benchmark, we will show the simulation results of these two case studies using the MCMAS model checker [355].

3.7.1 Verifying $\mathcal{PNAWS}$

$\mathcal{PNAWS}$ is a dialogue game-based protocol allowing web services to interact in a negotiation setting via argumentative agents. Agents can negotiate their participation in composite web services and persuade each other to perform some actions such as joining some existing business communities. In this case, two agents are used: the Master agent that manages the community and the Slave agent that is invited to join the community. $\mathcal{PNAWS}$ is specified using two special moves: refusal and acceptance as well as five dialogue games: entry game (to open the interaction), defense game, challenge game, justification game, and attack game. The $\mathcal{PNAWS}$ protocol can be defined as follows using a BNF-like grammar where "|" is the choice symbol and ";" the sequence symbol:

$\mathcal{PNAWS}$ = entry game; defense game; WSDG

---

[10] The CWB-NC model checker can be downloaded from:
http://www.cs.sunysb.edu/ cwb/

WSDG = acceptance move | CH | ATT
CH = challenge game; justification game; (WSDG | refusal move)
ATT = attack game; (WSDG | refusal move)

Each game is specified by a set of moves using a set of logical rules. Fig. 3.6 illustrates the different actions of this protocol using a finite state machine. Many properties can be checked in this protocol, such as deadlock freedom (a safety property), and liveness (something good will eventually happen). Deadlock freedom means that there is always a possibility for an action and can be expressed as follows, where $Ag \in \{Ag_1, Ag_2\}$:

$$AG(Cr(Ag_1, SC(Ag_1, Ag_2, \phi)) \Rightarrow AF\underline{(}Action(Ag, SC(Ag_1, Ag_2, \phi)) \\ \vee Action^+(Ag, SC(Ag_1, Ag_2, \phi), \phi_1)\underline{)}) \quad (3.2)$$

An example of liveness can be expressed by the following formula stating that if there is a challenge, a justification will eventually follow:

$$AG(Ch(Ag_2, SC(Ag_1, Ag_2, \phi_1)) \Rightarrow F(Jus(Ag_1, SC(Ag_1, Ag_2, \phi_1), \phi_2))) \quad (3.3)$$

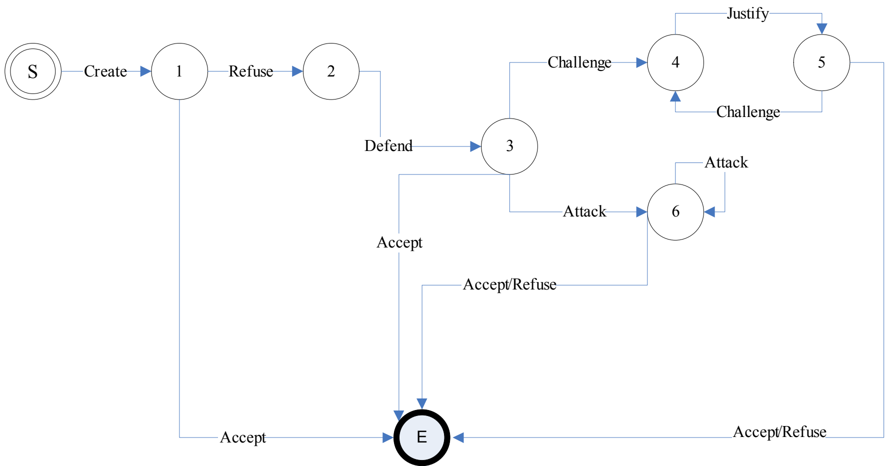

Fig. 3.6 Actions of the $\mathcal{PNAWS}$ protocol

We have extended the CWB-NC model checker by adding SC and argument operators and implemented this case study. CWB-NC supports GCTL*, which is close to our logic (without SC and argument operators) and allows modeling concurrent systems using Calculus of Communicating Systems (CCS) developed in [316]. CCS is a process algebra language, which is a prototype specification language for reactive systems. CCS can be used not only to describe implementations of processes, but also specifications of their expected behaviors. To implement this case study,

CCS is used to describe the model M to be checked by specifying the states and labeled transitions. ACTL* is used to specify the properties and the extended CWB-NC tool takes as input the CCS code and the ACTL* property and automatically builds the dialogue game protocol and checks the property by building the ABTA and executing the model checking algorithm presented in Fig. 3.5 (see the methodology in Fig. 3.1). To use CCS as the design language to describe the $\mathcal{PNAWS}$ protocol, we need first to introduce its syntax. Let A be the set of actions performed on SC we consider in ACTL* logic. For all $a \in A$, we associate a complementary action $'a$. An action a represents the receipt of an input action, while $'a$ represents the deposit of an output action. The syntax is given by the following BNF grammar:

$$P ::= nil|\alpha(\phi).P|(P+P)|(P|P)|proc\ C = P$$

"." represents the prefixing operator, "+" is the choice operator, "|" is the parallel operator and "*proc* =" is used for defining processes. The semantics can be defined using operational semantics in the usual way. $\alpha(\phi).P$ is the processes of performing the action α on the SC content ϕ and then evolves into process P. For representation reasons, we consider only the commitment content and we omit the other arguments. In addition, we abstract away from the internal states and we focus only on the global states. $P+Q$ is the process which non-deterministically makes the choice of evolving into either P or Q. $P|Q$ is the process which evolves in parallel into P and Q. To implement $\mathcal{PNAWS}$, we need to model the protocol and the agents using this protocol (the Master and Slave agents). For this reason, four particular processes should be defined: the states process describing the protocol dynamics; the two agents processes describing the agents legal decisions; and the communication synchronization process. The formulae to be checked are then encoded in CWB-NC input language. A simplified version of the states process is as follows:

```
proc Spec = create(ϕ).S1
proc Accept = accept(ϕ).Spec
proc Accept' = 'accept(ϕ).Spec
proc Refuse = refuse(ϕ).Spec
proc Refuse' = 'refuse(ϕ).Spec
proc S1 = 'refuse(ϕ).S2 + Accept'
proc S2 = defend(ϕ′).S3
proc S3 = 'challenge(ϕ′).S4 + 'attack(ϕ′).S6 + 'accept(ϕ′).Spec
proc S4 = justify(ϕ).S5
proc S5 = 'challenge(ϕ).S4 + 'Accept +'Refuse
proc S6 = attack(ϕ').S7 + Accept +Refuse
proc S7 = 'attack(ϕ).S6 + 'accept(ϕ′).Spec + 'refuse(ϕ′).Spec
set Internals = {create, challenge, justify, accept,
                 refuse, attack, defend}
```

The Master agent process has the form:

```
proc Master = create(φ).'accept(φ).master
             + create(φ).'refuse(φ).defend(φ).'accept(φ).master
             + create(φ).'refuse(φ).defend(φ).'refuse(φ).master
              ...
```

The Slave agent process has a similar form except the fact that it does not initiate the communication. The process describing the communication synchronization activity of an agent is as follows:

```
proc Ag = 'create(φ).Ag +
  create(φ).('refuse(φ ).Ag + 'accept(φ ).Ag) +
  refuse(φ).(Ag + 'defend( φ').Ag) +
  defend(φ').('challenge(φ′).Ag + 'attack(φ).Ag + 'accept(φ').Ag)
+
  challenge(φ).'justify(φ′).Ag +
  justify(φ').('challenge(φ′).Ag + 'accept(φ′).Ag +'refuse(φ′).Ag)
+
  attack(φ').('attack(φ).Ag + 'accept(φ′).Ag+'refuse(φ′).Ag) +
  accept(φ).Ag
```

The model size is $|M| = |S| + |R|$, where $|S|$ is the state space and $|R|$ is the relation space. $|S| = |S_{Ag_1}| \times |S_{Ag_2}| \times |S_{PNAWS}|$, where $|S_{Ag_i}|$ is the number of states for Ag_i and $|S_{PNAWS}|$ is the number of states of the protocol. An agent state is described in terms of the possible actions and each action is described by a set of states. For example, create action needs 2 states, challenge needs 3 states, and justify needs 5 states (see Fig. 3.2). Thus, for each agent we have 35 states. The protocol is described by the legal actions (Fig. 3.6), so it needs 29 states. In total, the number of states needed for this case study is $|S| = 35525 \approx 3.5 \cdot 10^4$. To calculate $|R|$, we have to consider the operators of ACTL* and the actions, where the total number is $6 + 11 = 17$. We can then approximate $|R|$ by $17 \cdot |S|^2$. So we have $|M| \approx 17 \cdot |S|^2 \approx 2 \cdot 10^{10}$. This is a theoretically estimated size if all possible transitions are considered. However, in the implementation, not all these transitions are used. On the other hand, the system considers additional states for the internal coding of variable states and actions. Some simulation results on a laptop Intel Core 2 Duo CPU T6400 2.20 GHz with 3.00 GB of RAM running Windows Vista Home Premium are given in Table 3.3. Fig. 3.7 shows the results screenshot. In fact, CWB-NC does not search the whole model, but it proceeds by simplifying the ABTA, minimizing the sets of accepting states, and computing bisimulation before starting the model checking.

As benchmark, we use MCMAS [355] that supports agent specifications. As discussed in Section 3.2.2, MCMAS is a symbolic model checker based on OBDDS, where the model and formula to be checked are not represented as automata, but using boolean functions. in MCMAS, models are described into a modular language called Interpreted Systems Programming Language (ISPL). An ISPL program includes: 1) a list of agents' descriptions; 2) an evaluation function indicating the

Table 3.3 Statistics of verifying $\mathcal{PNAWS}$ using CWB-NC

Model size (states/transitions)	35709/77244
Time for building the model (sec)	1.763
Verification time (sec)	5.912
Total execution time (sec)	7.675

```
cwb-nc> load PNAWS.ccs
Execution time (user,system,gc,real):(0.016,0.000,0.016,0.016)
cwb-nc> size Spec
Building automaton...
.........1000.........2000.........3000.........4000.........5000
.........6000.........7000.........8000.........9000.........10000
.........11000.........12000.........13000.........14000.........15000
.........16000.........17000.........18000.........19000.........20000
.........21000.........22000.........23000.........24000.........25000
.........26000.........27000.........28000.........29000.........30000
.........31000.........32000.........33000.........34000.........35000
......
States: 35709
Transitions: 77244
Done building automaton.
States: 35709
Transitions: 77244
Execution time (user,system,gc,real):(1.763,0.000,0.172,1.763)
cwb-nc> chk -L gctl Spec can_deadlock
Generating ABTA from GCTL* formula...done
Initial ABTA has 6 states.
Simplifying ABTA:
Minimizing sets of accepting states...done
Performing constant propagation...done
Joining operations...done
Shrinking automaton...done
Computing bisimulation...
Done computing bisimulation.
Simplification completed.
Simplified ABTA has 4 states.
Starting ABTA model checker..........*...*.*..*..*.*.*..*.*.*..*.*.*.*.*.*.*.*..
*.*.*.*.*.*.*.*.*.*.**.*.*..*.*.*.*.*.*.*.*.*.*.**.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*
.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.**.*.*.*.**.*.*.**.*.**.*.**.***.***.***.****
*****.*
Model checking completed.
Expanded state-space 107127 times.
Stored 112953 dependencies.
FALSE, the agent does not satisfy the formula.
Execution time (user,system,gc,real):(5.912,0.000,1.393,5.912)
cwb-nc>
```

Fig. 3.7 $\mathcal{PNAWS}$ simulation results with CWB-NC

states where atomic propositions are true; 3) a set of initial states; and 4) a list of formulae. Each agent is composed by: a set of local states, a set of actions, a rule (protocol) describing which action can be performed by the agent, and evolution functions that describe how the local states of the agent evolve based on the current local states and agent's actions.

To implement the $\mathcal{PNAWS}$ protocol with ISPL, commitments are encoded as variables. The Master and Slave agents are specified in two Agent sections along with the Protocol and its Evolution. The atomic propositions are evaluated in Evaluation section. Formulae are then encoded in the same file in the Formulae section. As example, we show here the form of the Master agent:

```
Agent Master
   Vars:
     state : {M0, M1, M2, ...};
     ...
   end Vars
   Actions = {create, defend, ...};
```

```
   Protocol:
     -- initiate the contract by creating
     state = M0 : {create};
     ...
   end Protocol
   Evolution:
     state = M1 if state = M0 and Slave.Action = reject
     ...
   end Evolution
end Agent
```

Some simulation results using the same machine as for CWB-NC are given in Table 3.4. Fig. 3.8 shows the results screenshot. This simulation reveals that MCMAS uses greater number of reachable states, which are needed to encode commitments and agent local states. The execution time is very close to the previous experiment.

Table 3.4 Statistics of verifying $\mathcal{PNAWS}$ using MCMAS

Number of reachable states	39475
Number of BDD and ADD nodes	152093
Total execution time (sec)	8

```
$ ./mcmas -bdd_stats PNAWS.ispl
*******************************************************************************

                          MCMAS v0.9.8.6

 This software comes with ABSOLUTELY NO WARRANTY, to the extent
 permited by applicable law.

 Please check
 http://www.cs.ucl.ac.uk/staff/f.raimondi/MCMAS/
 for the latest release.
 Report bugs to <hongyang.qu@imperial.ac.uk> or <f.raimondi@cs.ucl.ac.uk>
*******************************************************************************

PNAWS.ispl has been parsed successfully.
Global syntax checking...
Done
Encoding BDD parameters...
Building partial transition relation...
Building OBDD for initial states...
Building reachable state space...
Building OBDD for group modalities...
Done.
Checking formulae...
Verifying properties...
  Formula number 0: (EG complianceTermination), is TRUE in the model
done, 1 formulae successfully read and checked
execution time = 8
number of reachable states = 39575
BDD memory in use = 8276116
**** CUDD modifiable parameters ****
Hard limit for cache size: 5592405
Cache hit threshold for resizing: 30%
Garbage collection enabled: yes
Limit for fast unique table growth: 3355443
Maximum number of variables sifted per reordering: 1000
Maximum number of variable swaps per reordering: 2000000
Maximum growth while sifting a variable: 1.2
```

Fig. 3.8 $\mathcal{PNAWS}$ simulation results with MCMAS

3.7.2 Verifying NetBill

We consider a modified version of the NetBill protocol where two agents, Customer (Cus) and Merchant (Mer), are interacting about some goods. The protocol starts when the Customer requests a quote, which means creating a commitment about a content ϕ_1. The merchant can then either reject the request, which means refuse the commitment and the protocol will end, or accept the request (i.e. accept the commitment) and then make an offer (i.e. create another commitment about a content ϕ_2). The protocol is self-described in Fig.3.9. An example of liveness in this protocol can be expressed by the following formula stating that if a commitment is created, then there is a possibility of satisfying it.

$$AG(Cr(Ag_1, SC(Ag_1, Ag_2, \phi_1)) \Rightarrow EF(Sat(Ag_1, SC(Ag_1, Ag_2, \phi_1))) \quad (3.4)$$

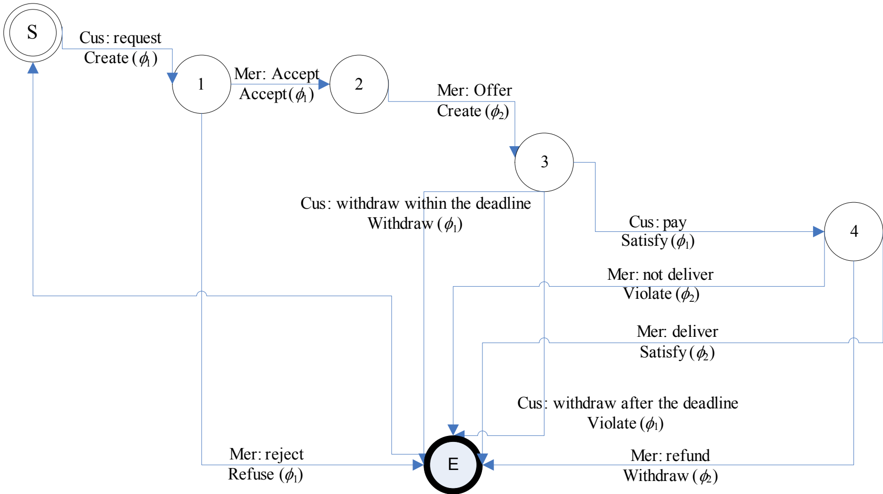

Fig. 3.9 Actions of the NetBill protocol

NetBill size is $|M| = (|S_{Ag_1}| \times |S_{Ag_2}| \times |S_{NetBill}|) + |R|$, where Ag_1 is the Customer and Ag_2 is the Merchant. According to the actions the Customer and Merchant are allowed to perform, we have $|S_{Ag_1}| = 9$ and $|S_{Ag_2}| = 13$. The NetBill protocol is described by the legal actions, and by considering the size of each action, we obtain $|S_{NetBill}| = 22$. In total, the number of states needed for this case study is $|S| = 2574 \approx 2.5 \cdot 10^3$. As we did in the previous case study, the theoretical estimation of $|R|$ if all possible transitions are considered is $|R| \approx 17 \cdot |S|^2$. So we have $|M| \approx 10^{10}$. As illustrated in Table 3.5, which shows the NetBill simulation results with CWB-NC using the same machine as in the previous case study, the number of transitions that are effectively considered is much more smaller. Table 3.6 shows the simulation

results with MCMAS. Fig. 3.10 shows the results screenshot with the two model checkers. Because NetBill is 14 times smaller than $PNAWS$, its execution time is shorter.

```
cwb-nc> load NetBillnew.ccs
Execution time (user,system,gc,real):(0.016,0.000,0.000,0.016)
cwb-nc> load formula.gctl
Execution time (user,system,gc,real):(0.000,0.000,0.000,0.000)
cwb-nc>
cwb-nc>
cwb-nc> size Spec
Building automaton...
.........1000.........2000.....
States: 2593
Transitions: 5911
Done building automaton.
States: 2593
Transitions: 5911
Execution time (user,system,gc,real):(0.125,0.000,0.000,0.125)
cwb-nc>
cwb-nc>
cwb-nc> chk -L gctl Spec can_deadlock
Generating ABTA from GCTL* formula...done
Initial ABTA has 6 states.
Simplifying ABTA:
Minimizing sets of accepting states...done
Performing constant propagation...done
Joining operations...done
Shrinking automaton...done
Computing bisimulation...
Done computing bisimulation.
Simplification completed.
Simplified ABTA has 4 states.
Starting ABTA model checker....*.*.**.*.***
Model checking completed.
Expanded state-space 7779 times.
Stored 8504 dependencies.
FALSE, the agent does not satisfy the formula.
Execution time (user,system,gc,real):(0.359,0.000,0.046,0.359)
cwb-nc>
```

```
$ ./mcmas -bdd_stats NetBill.ispl
*************************************************************************

                    MCMAS v0.9.8.6

 This software comes with ABSOLUTELY NO WARRANTY, to the extent
 permited by applicable law.

 Please check
 http://www.cs.ucl.ac.uk/staff/f.raimondi/MCMAS/
 for the latest release.
 Report bugs to <hongyang.qu@imperial.ac.uk> or <f.raimondi@cs.ucl.ac.uk>
*************************************************************************


NetBill.ispl has been parsed successfully.
Global syntax checking...
Done
Encoding BDD parameters...
Building partial transition relation...
Building OBDD for initial states...
Building reachable state space...
Done.
Checking formulae...
Verifying properties...
  Formula number 0: (AF purchase_violation), is FALSE in the model
  Formula number 1: (AF purchase_compliance), is FALSE in the model
done, 2 formulae successfully read and checked
execution time = 0
number of reachable states = 2851
BDD memory in use = 6537844
**** CUDD modifiable parameters ****
Hard limit for cache size: 5592405
Cache hit threshold for resizing: 30%
Garbage collection enabled: yes
Limit for fast unique table growth: 3355443
```

Fig. 3.10 Simulation results of NetBill

Table 3.5 Statistics of verifying NetBill using CWB-NC

Model size (states/transitions)	2593/5911
Time for building the model (sec)	0.125
Verification time (sec)	0.359
Total execution time (sec)	0.484

Table 3.6 Statistics of verifying NetBill using MCMAS

Number of reachable states	2851
Number of BDD and ADD nodes	9332
Total execution time (sec)	≈ 0.5

3.8 Discussion and Future Work

Model checking is an effective technique to verify finite state systems. Compared to classical software systems, model checking multi-agent systems raise new challenges related to the need of considering: 1) epistemic properties where the semantics is expressed in terms of accessibility relations; and 2) agent communication protocols that integrate agent properties and message meaning, which make them more complex than simple message exchanging mechanisms. These two fundamental issues need new and efficient verification techniques considering computational interpretations of accessibility relations and message meaning.

In this chapter we described a verification technique for dialogue game protocols. The proposed model checking algorithm allows us to verify both protocols' correctness and agents' compliance to the decomposition semantics of communicative acts. This technique uses a combination of automata and tableau-based algorithms to verify temporal and action specifications. The formal properties to be verified are expressed in ACTL* logic and translated to ABTA using tableau rules. Our model checking algorithm that works on a product graph is an efficient on-the-fly procedure.

The field of automatic verification of multi-agent systems has manifested significant advances in the past few years, as efficient algorithms and techniques have been proposed. However, many issues still need investigations. The most challenging among them are: 1) verifying the compliance of agents' joint actions to the norms and rules of the multi-agent system in which they operate; 2) integrating the verification of mental and social attitudes in the same framework; 3) allowing the use of expressive logical languages to specify agents and their communication and coordination, multi-agent environments, and requirements (i.e. desired properties); and 4) developing tools integrating the whole aforementioned issues.

We plan to extend this work to address some of these issues. In fact, we intend to use the proposed tableau-based technique to verify MAS specifications and the conformance of agents to these specifications. We also plan to extend the technique and logic in order to consider epistemic properties, so that we will have a same framework for private and social attitudes. We plan to use this technique to specify

and verify the compliance of agents' actions to the norms and policies of multi-agent systems. Although the technique discussed in this chapter is computationally efficient, it has the problem of state explosion. For this reason, we plan to consider symbolic and bounded model checking to verify agent commitments and their dialogue games. We are investigating the extension of the MCMAS model checker to integrate LTL logic with commitment modalities and action formulae, so it will be possible to symbolically model check dialogue games with ACTL* logic.

Acknowledgement

We would like to thank the reviewers for their valuable comments and suggestions. Jamal Bentahar would like to thank Natural Sciences and Engineering Research Council of Canada (NSERC: Project 341422-07), Fonds québécois de la recherche sur la nature et les technologies (FQRNT: Project 2008-NC-119348) and Fonds québécois de la recherche sur la société et la culture (FQRSC: Project 2007- 111881) for their financial support.