
eleventh
international
conference
on
autonomous
agents and
multiagent
systems
AAMAS
2012
4th- 8th June 2012
Valencia
W13
Workshop on
Declarative
Agent
Languages and
Technologies
(DALT)

Matteo Baldoni, Louise Dennis,
Viviana Mascardi, Wamberto Vasconcelos (eds.)

Declarative Agent Languages and Technologies

Tenth International Workshop, DALT 2012
Valencia, Spain, June 4th, 2012

Workshop Notes

DALT 2012 Home Page:
`http://www.di.unito.it/~baldoni/DALT-2012/`

Preface

The workshop on Declarative Agent Languages and Technologies (DALT), in its *tenth edition* this year, is about investigating, studying, and using the declarative paradigm for specifying, programming and verifying both individual agents and and multi-agent systems. As one of the well-established workshops in the multi-agent systems area, DALT aims to provide a forum for researchers interested in linking theory to practical applications by combining declarative and formal approaches with engineering and technology aspects of agents and multi-agent systems.

Declarative approaches provide smoother and more natural ways to connect theory with practical computing aspects. Algebras, logics and functions, to name a few, have been used as declarative formalisms, with which (together with their associated mechanisms) one can specify, verify, program and analyze computational systems. The well-understood mathematical underpinnings of declarative approaches provide clean, solid and natural techniques for bridging the gap between theory and practice, providing formalisms, tools and techniques to support the development of applications. They offer useful abstractions for studying computational phenomena, which are necessarily more compact than procedural accounts. Software agents and multi-agent systems have been pursued as a means to realize a new generation of very large-scale, distributed information systems. Declarative approaches to agent languages and technologies raise many fresh challenges with exciting prospects for agent programming, communication languages, reasoning and decision-making. These challenges include, for instance, which formal foundations to use, how pragmatic concerns are addressed formally, how expressive approaches are, and so on.

In the tradition of DALT, the 2012 meeting is being held as a satellite workshop of AAMAS 2012, the 11th International Joint Conference on Autonomous Agents and Multiagent Systems, in Valencia, Spain. Following the success of DALT 2003 in Melbourne (LNAI 2990), DALT 2004 in New York (LNAI 3476), DALT 2005 in Utrecht (LNAI 3904), DALT 2006 in Hakodate (LNAI 4327), DALT 2007 in Honolulu (LNAI 4897), DALT 2008 in Estoril (LNAI 5397), DALT 2009 in Budapest (LNAI 5948), DALT 2010 in Toronto (LNAI 6619), DALT 2011 in Taiwan (LNAI 7169), DALT will aim at providing a discussion forum to both (i) support the transfer of declarative paradigms and techniques to the broader community of agent researchers and practitioners, and (ii) to bring the issue of designing complex agent systems to the attention of researchers working on declarative languages and technologies.

This edition of DALT received *eight* long paper submissions, and *three* short paper submissions, describing work by researchers coming from seven different countries. *Six* long papers and *three* short papers have been selected by the Programme Committee and are included in this volume. Each long paper received at least three reviews in order to supply the authors with helpful feedback that

could stimulate the research as well as foster discussion. The short papers are a new innovation introduced to celebrate DALT's 10th edition and the Alan Turing year, aimed at encouraging the exchange of views among scientists sharing the same interests. Each short paper received two "light touch" reviews and was evaluated on the basis of its potential for stimulating discussion. As has happened for all the nine previous editions, we plan to publish the DALT 2012 post-proceedings as a volume in Lecture Notes in Artificial Intelligence by Springer.

We would like to thank all authors for their contributions, the members of the Steering Committee for the valuable suggestions and support, and the members of the Programme Committee for their excellent work during the reviewing phase.

April 18th, 2012

Matteo Baldoni
Louise Dennis
Viviana Mascardi
Wamberto Vasconcelos

Workshop Organisers

Matteo Baldoni	University of Torino, Italy
Louise Dennis	University of Liverpool, UK
Viviana Mascardi	University of Genova, Italy
Wamberto Vasconcelos	University of Aberdeen, UK

Programme Committee

Thomas Ågotnes	Bergen University College, Norvey
Marco Alberti	Universidade Nova de Lisboa, Portugal
Natasha Alechina	University of Nottingham, UK
Cristina Baroglio	University of Torino, Italy
Rafael Bordini	Pontificia Universidade Católica do Rio Grande do Sul, Brasil
Jan Broersen	University of Utrecht, The Netherlands
Federico Chesani	University of Bologna, Italy
Flavio Correa Da Silva	Universidade de São Paulo, Brasil
Marina De Vos	University of Bath, UK
Francesco Donini	Tuscia Universityas, Italy
Michael Fink	Vienna University of Technology, Austria
James Harland	RMIT University, Australia
Andreas Herzig	Paul Sabatier University, France
Koen Hindriks	Delt University of Technology, The Netherlands
João Leite	Universidade Nova de Lisboa, Portugal
Shinichi Honiden	National Institute of Informatics, Japan
Yves Lespérance	York University, Canada
Nicolas Maudet	University of Paris-Dauphine, France
John-Jules Ch. Meyer	Utrecht University, The Netherlands
Peter Novak	Czech Technical University in Prague, Czech Republic
Fabio Patrizi	Imperial College London, UK
Enrico Pontelli	New Mexico State University, USA
David Pym	University of Aberdeen, UK
Alessandro Ricci	University of Bologna, Italy
Michael Rovatsos	The University of Edinburgh, UK
Guillermo Simari	Universidad Nacional del Sur, Argentina
Tran Cao Son	New Mexico State University, USA

Steering Committee

Matteo Baldoni	University of Torino, Italy
Andrea Omicini	University of Bologna-Cesena, Italy
M. Birna van Riemsdijk	Delft University of Technology, The Netherlands
Tran Cao Son	New Mexico State University, USA
Paolo Torroni	University of Bologna, Italy
Pınar Yolum	Bogazici University, Turkey
Michael Winikoff	University of Otago, New Zealand

Additional Reviewers

Michal Cap	Czech Technical University in Prague, Czech Republic

Table of Contents

Automatic Generation of Self-Monitoring MASs from Multiparty Global Session Types in Jason

Davide Ancona[1], Sophia Drossopoulou[2], and Viviana Mascardi[1]

[1] DISI, University of Genova, Italy
{davide.ancona,viviana.mascardi}@unige.it
[2] Imperial College, London, UK
scd@doc.ic.ac.uk

Abstract. Global session types are behavioral types designed for specifying in a compact way multiparty interactions between distributed components, and verifying their correctness. We take advantage of the fact that global session types can be naturally represented as cyclic Prolog terms - which are directly supported by the Jason implementation of AgentSpeak - to allow simple automatic generation of self-monitoring MASs: given a global session type specifying an interaction protocol, and the implementation of a MAS where agents are expected to be compliant with it, we define a procedure for automatically deriving a self-monitoring MAS. Such a generated MAS ensures that agents conform to the protocol at run-time, by adding a *monitor* agent that checks that the ongoing conversation is correct w.r.t. the global session type.
The feasibility of the approach has been experimented in Jason for a non-trivial example involving recursive global session types with alternative choice and fork type constructors. Although the main aim of this work is the development of a unit testing framework for MASs, the proposed approach can be also extended to implement a framework supporting self-recovering MASs.

1 Introduction

A protocol represents an agreement on how participating systems interact with each other. Without a protocol, it is hard to do a meaningful interaction: participants simply cannot communicate effectively.
The development and validation of programs against protocol descriptions could proceed as follows:

- *A programmer specifies a set of protocols to be used in her application.*

...

- *At the execution time, a local monitor can validate messages with respect to given protocols, optionally blocking invalid messages from being delivered.*

This paper starts with a few sentences drawn from the manifesto of Scribble, a language to describe application-level protocols among communicating systems

initially designed by Kohei Honda and Gary Brown[1]. The team working on Scribble involves both scientists active in the agent community and scientists active in the session types one. Their work inspired the proposal presented in this paper where multiparty global session types are used on top of the Jason agent oriented programming language for runtime verification of the conformance of a MAS implementation to a given protocol. This allows us to experiment our approach on realistic scenarios where messages may have a complex structure, and their content may change from one interaction to another.

Following Scribble's manifesto, we ensure runtime conformance thanks to a Jason monitor agent that can be automatically generated from the global session type, represented as a Prolog cyclic term. Besides the global session type, the developer must specify the type of the actual messages that are expected to be exchanged during a conversation.

In order to verify that a MAS implementation is compliant with a given protocol, the Jason code of the agents that participate in the protocol is extended seamlessly and automatically. An even more transparent approach would be possible by overriding the underlying agent architecture methods of Jason responsible for sending and receiving messages, which could intercept all messages sent by the monitored agents, and send them to the monitor which could manage them in the most suitable way. In this approach message "sniffing" would have to occur at the Java (API) level, gaining in transparency but perhaps loosing in flexibility.

In this paper we show the feasibility of our approach by testing a MAS against a non-trivial protocol involving recursive global session types with alternative choice and fork type constructors.

The paper is organized in the following way: Section 2 provides a gentle introduction to the notion of global session type adopted in this work; Section 3 discusses our implementation of the protocol testing mechanism and presents the results of our experiments; Section 4 discusses the related literature and outlines future directions of our work.

2 A gentle introduction to global session types for agents

In this section we informally introduce global session types (global types for short) and show how they can be smoothly integrated in MASs to specify multiparty communication protocols between agents. To this aim, we present a typical protocol that can be found in literature as our main running example used throughout the paper.

Our example protocol involves three different agents playing the roles of a seller `s`, a broker `b`, and a client `c`, respectively. Such a protocol is described by the FIPA AUML interaction diagram [14] depicted in Figure 1: initially, `s` communicates to `b` the intention to sell a certain item to `c`; then the protocol enters a negotiation loop of an arbitrary number n (with $n \geq 0$) of iterations,

[1] http://www.jboss.org/scribble/

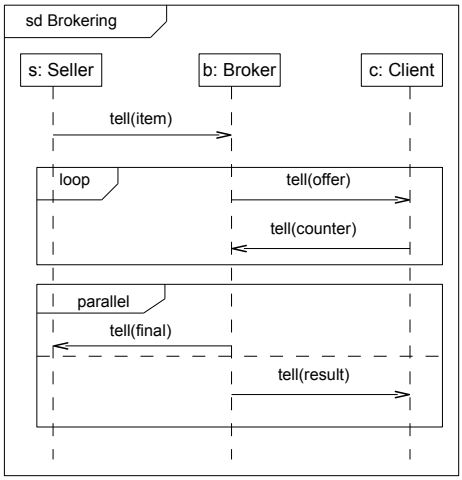


Fig. 1. The Brokering interaction protocol in FIPA AUML.

where `b` sends an offer to `c` and `c` replies with a corresponding counter-offer. After such a loop, `b` concludes the communication by sending in an arbitrary order the message of type `result` to `c`, and of type `final` to `s`.

Even though the AUML diagram of Figure 1 is very intuitive and easy to understand, a more compact and formal specification of the protocol is required to perform verification or testing of a MAS, in order to provide guarantees that the protocol is implemented correctly. Global session types [5, 11] have been introduced and studied exactly for this purposes, even though in the more theoretical context of calculi of communicating processes. A global type describes succinctly all sequences of sending actions that may occur during a correct implementation of a protocol.

Depending on the employed type constructors, a global type can be more or less expressive. Throughout this paper we will use a fixed notion of global type, but our proposed approach can be easily adapted for other kinds of global types. The notion of global type we adopt is a slightly less expressive version of that proposed by Deniélou and Yoshida [7] (which, however, allows us to specify the protocol depicted in Figure 1), defined on top of the following type constructors:

– *Sending Actions*: a sending action occurs between two agents, and specifies the sender and the receiver of the message (in our case, the names of the agents, or, more abstractly, the role they play in the communication), and the type of the performative and of the content of the sent message; for

instance, `msg(s, b, tell, item)` specifies that agent `s` (the seller) sends the **tell** performative to agent `b` (the broker) with content of type `item`.

- *Empty Type*: the constant `end` represents the empty interaction where no sending actions occur.
- *Sequencing*: sequencing is a binary constructor allowing a global type t to be prefixed by a sending action a; that is, all valid sequences of sending actions denoted by `seq(`a`,`t`)` are obtained by prefixing with a all those sequences denoted by t. For instance,

```
seq(msg(alice,bob,tell,ping),
    seq(msg(bob,alice,tell,pong),end))
```

  specifies the simple interaction where first `alice` sends **tell**`(ping)` to `bob`, then `bob` replies to `alice` with **tell**`(pong)`, and finally the interaction stops.
- *Choice*: the choice constructor has variable arity[2] n (with $n \geq 0$) and expresses an alternative between n possible choices. Because its arity is variable we use a list to represent its operands. For instance,

```
choice([
    seq(msg(c,b,tell,counter),end),
    seq(msg(b,s,tell,final),end),
    seq(msg(b,c,tell,result),end)
])
```

  specifies an interaction where either `c` sends **tell**`(counter)` to `b`, or `b` sends **tell**`(final)` to `s`, or `b` sends **tell**`(result)` to `c`.
- *Fork*: the fork binary[3] constructor specifies two interactions that can be interleaved. For instance,

```
fork(
    seq(msg(b,s,tell,final),end),
    seq(msg(b,c,tell,result),end)
)
```

  specifies the interaction where first `b` sends **tell**`(final)` to `s`, and then `b` sends **tell**`(result)` to `c`, or the other way round.

Recursive types: the example types shown so far do not specify any interaction loop, as occurs in the protocol of Figure 1. To specify loops we need to consider recursive global types; for instance, the protocol consisting of infinite sending actions where first `alice` sends **tell**`(ping)` to `bob`, and then `bob` replies **tell**`(pong)` to `alice`, can be represented by the recursive type `T` s.t.

```
T = seq(msg(alice,bob,tell,ping),
        seq(msg(bob,alice,tell,pong),T))
```

If we interpret the equation above syntactically (that is, as a unification problem), then the unique solution is an infinite term (or, more abstractly, an infinite

[2] Arity 0 and 1 are not necessary, but make the definition of predicate `next` simpler.

[3] For simplicity, the operator has a fixed arity, but it could be generalized to the case of n arguments (with $n \geq 2$) as happens for the `choice` constructor.

tree) which is *regular*, that is, whose set of subterms is finite. In practice, the unification problem above is solvable in most modern implementations of Prolog, where cyclic terms are supported; this happens also for the Jason implementation, where Prolog-like rules can be used to derive beliefs that hold in the current belief base[4]. As another example, let us consider the type `T2` s.t.

```
T2 = seq(msg(alice,bob,tell,ping),
         seq(msg(bob,alice,tell,pong),choice([T2,end])))
```

Such a type contains the infinite interaction denoted by `T` above, but also all finite sequences of length $2n$ (with $n \geq 1$) of alternating sending actions `msg(alice,bob,tell,ping)` and `msg(bob,alice,tell,pong)`.

We are now ready to specify the `Brokering` protocol with a global type `BP`, where for sake of clarity we use the auxiliary types `OffOrFork`, `Off`, and `Fork`:

```
BP         = seq(msg(s,b,tell,item),OffOrFork),
OffOrFork  = choice([Off,Fork])
Off        = seq(msg(b,c,tell,offer),
                 seq(msg(c,b,tell,counter),OffOrFork))
Fork       = fork(seq(msg(b,s,tell,final),end),
                  seq(msg(b,c,tell,result),end))
```

Note that for the definition of global types we consider in this paper, the `fork` constructor does not really extend the expressiveness of types: any type using `fork` can be transformed into an equivalent one without `fork`. However, such a transformation may lead to an exponential growth of the type [2].

Formal definitions

Figure 2 defines the abstract syntax of the global session types that will be used in the rest of the paper. As already explained in the previous section, global

$$
\begin{aligned}
\mathrm{GT} ::= {} & \mathrm{choice}([\mathrm{GT}_1, \ldots, \mathrm{GT}_n])\ (n \geq 0) \mid \\
& \mathrm{seq}(\mathrm{SA}, \mathrm{GT}) \mid \\
& \mathrm{fork}(\mathrm{GT1}, \mathrm{GT1}) \mid \\
& \mathrm{end} \\
\mathrm{SA} ::= {} & \mathrm{msg}(\mathrm{AId}_1,\mathrm{AId}_2,\mathrm{PE},\mathrm{CT})
\end{aligned}
$$

Fig. 2. Syntax of Global Types.

types are defined coinductively: GT is the greatest set of regular terms defined by the productions of Figure 2.

The meta-variables AId, PE and CT range over agent identifiers, performatives, and content types, respectively. Content types are constants specifying the types of the contents of messages.

[4] Persistency of cyclic terms is supported by the very last version of Jason; since testing of this feature is still ongoing, it has not been publicly released yet.

The syntactic definition given so far still contains global types that are not considered useful, and, therefore, are rejected for simplicity. Consider for instance the following type `NC`:

```
NC = choice([NC,NC])
```

Such a type is called *non contractive* (or *non guarded*), since it contains an infinite path with no `seq` type constructors. These kinds of types pose termination problems during dynamic global typechecking. Therefore, in the sequel we will consider only *contractive* global types (and we will drop the term "contractive" for brevity), that is, global types that do not have paths containing only the `choice` and `fork` type constructors. Such a restriction does not limit the expressive power of types, since it can be shown that for every non contractive global type, there exists a contractive one which is equivalent, in the sense that it represents the same set of sending action sequences. For instance, the type `NC` as defined above corresponds to the empty type `end`.

Interpretation of global types. We have already provided an intuition of the meaning of global types. We now define their interpretation, expressed in terms of a `next` predicate, specifying the possible transitions of a global type. Intuitively, a global type represents a state from which several transition steps to other states (that is, other global types) are possible, with a resulting sending action. Consider for instance the type `F` defined by

```
fork(seq(msg(b,s,tell,final),end),
     seq(msg(b,c,tell,result),end))
```

Then there are two possible transition steps: one yields the sending action `msg(b,s,tell,final)` and moves to the state corresponding to the type

```
fork(end,
     seq(msg(b,c,tell,result),end))
```

while the other yields the sending action `msg(b,c,tell,result)` and moves to the state corresponding to the type

```
fork(seq(msg(b,s,tell,final),end),
     end)
```

Predicate `next` is defined below, with the following meaning: if `next(GT1,SA,GT2)` succeeds, then there is a one step transition from the state represented by the global type GT1 to the state represented by the global type GT2, yielding the sending action SA. The predicate is intended to be used with the mode indicators `next(+,+,-)`, that is, the first two arguments are input, whereas the last is an output argument.

```
1 next(seq(msg(S, R, P, CT),GT),msg(S, R, P, C),GT) :-
      has_type(C, CT).
2 next(choice([GT1|_]),SA,GT2) :- next(GT1,SA,GT2).
3 next(choice([_|L]),SA,GT) :- next(choice(L),SA,GT).
4 next(fork(GT1,GT2),SA,fork(GT3,GT2)) :- next(GT1,SA,GT3).
5 next(fork(GT1,GT2),SA,fork(GT1,GT3)) :- next(GT2,SA,GT3).
```

We provide an explanation for each clause:

1. For a sequence `seq(msg(S, R, P, CT),GT)` the only allowed transition step leads to state `GT`, and yields a sending action `msg(S, R, P, C)` where `C` is required to have type `CT`; we assume that all used content types are defined by the predicate `has_type`, whose definition is part of the specification of the protocol, together with the initial global type.
2. The first clause for `choice` states that there exists a transition step from `choice([GT1|_])` to `GT2` yielding the sending action `SA`, whenever there exists a transition step from `GT1` to `GT2` yielding the sending action `SA`.
3. The second clause for `choice` states that there exists a transition step from `choice([_|L])` to `GT` yielding the sending action `SA`, whenever there exists a transition step from `choice(L)` (that is, the initial type where the first choice has been removed) to `GT` yielding the sending action `SA`.
   Note that both clauses for `choice` fail for the empty list, as expected (since no choice can be made).
4. The first clause for `fork` states that there exists a transition from `fork(GT1,GT2)` to `fork(GT3,GT2)` yielding the sending action `SA`, whenever there exists a transition step from `GT1` to `GT3` yielding the sending action `SA`.
5. The second clause for `fork` is symmetric to the first one.

We conclude this section by a claim stating that contractive types ensure termination of the resolution of `next`.

Proposition 1. *Let us assume that* `has_type(`c`,`ct`)` *always terminates for any ground atoms* c *and* ct*. Then,* `next(`gt`,`sa`,`X`)` *always terminates, for any ground terms* gt *and* sa*, and logical variable* X*, if* gt *is a contractive global type.*

Proof. By contradiction, it is straightforward to show that if `next(`gt`,`sa`,`X`)` does not terminate, then gt must contain a (necessarily infinite) path with only `choice` and `fork` constructors, hence, gt is not contractive.

3 A Jason Implementation of a Monitor for Checking Global Session Types

As already explained in the Introduction, the main motivation of our work is a better support for testing the conformance of a MAS to a given protocol, even though we envisage other interesting future application scenarios (see Section 4). From this point of view our approach can be considered as a first step towards the development of a unit testing framework for MASs where testing, types, and – more generally – formal verification can be reconciled in a synergistic way.

In more detail, given a Jason implementation of a MAS[5], our approach allows automatic generation[6] of an extended MAS from it, that can be run on a set of tests to detect possible deviations of the behavior of a system from a given

[5] We assume that the reader is familiar with the AgentSpeak language [17].
[6] Its implementation has not been completed yet.

protocol. To achieve this the developer is required to provide (besides the original MAS, of course) the following additional definitions:

- The Prolog clauses for predicate `next` defining the behavior of the used global types (as shown in Section 2); such clauses depend on the notion of global type needed for specifying the protocol; depending on the complexity of the protocol, one may need to adopt more or less expressive notions of global types, containing different kinds of type constructors, and for each of them the corresponding behavior has to be defined in terms of the `next` predicate. However, we expect the need for changing the definition of `next` to be a rare case; the notion of global type we present here captures a large class of frequently used protocols, and it is always possible to extend the testing unit framework with a collection of predefined notions of global types among which the developer can choose the most suitable one.
- The global type specifying the protocol to be tested; this can be easily defined in terms of a set of unification equations.
- The clauses for the `has_type` predicate (already mentioned in Section 2), defining the types used for checking the content of the messages; also in this case, a set of predefined primitive types could be directly supported by the framework, leaving to the developer the definition of the user-defined types.

The main idea of our approach relies on the definition of a centralized *monitor* agent that verifies that a conversation among any number of participants is compliant with a given global type, and warns the developer if the MAS does not progress. Furthermore, the code of the agents of the original MAS requires minimal changes that, however, can be performed in an automatic way.

In the sequel, we describe the code of the monitor agent, and the changes applied to all other agents (that is, the participants of the implemented protocol).

3.1 Monitor

We illustrate the code for the monitor by using our running brokering example. The monitor can be automatically generated from the global type specification in a trivial way. The global type provided by the developer is simply a conjunction *UnifEq* of unification equations of the form $X = GT$, where X is a logical variable, and GT is a term (possibly containing logical variables) denoting a global type. The use of more logical variables is allowed for defining auxiliary types that make the definition of the main type more readable. Then from *UnifEq* the following Prolog rule is generated:

`initial_state(`X`) :-` *UnifEq*`.`

where X is the logical variable contained in *UnifEq* corresponding to the main global type. The definition of the type of each message content must be provided as well. In fact, the protocol specification defines also the expected types (such as `item`, `offer`, `counter`, `final` and `result`) for the correct content of all possible messages. For example, the developer may decide that the type `offer` defines all

terms of shape `offer(Item, Offer)`, where `Item` is a string and `Offer` is an integer; similarly, the type `item` corresponds to all terms of shape `item(Client, Item)` where both `Client` and `Item` are strings.

Consequently, the developer has to provide the following Prolog rules that formalize the descriptions given above:

```
has_type(offer(Item, Offer), offer) :-
   string(Item) & int(Offer).
has_type(item(Client, Item), item) :-
   string(Client) & string(Item).
```

The monitor keeps track of the runtime evolution of the protocol by saving its current state (corresponding to a global type), and checking that each message that a participant would like to send, is allowed by the current state. If so, the monitor allows the participant to send the message by explicitly sending an acknowledgment to it. We explain how participants inform the monitor of their intention to send a message in Section 3.2.

The correctness of a sending action is directly checked by the `next` predicate, that also specifies the next state in case the transition is correct. In other words, verifying the correctness of the message sent by `S` to `R` with performative `P` and content `C` amounts to checking if it is possible to reach a `NewState` from the `CurrentState`, yielding a sending action `msg(S, R, P, C)` (`type_check` predicate).

```
/* Monitor's initial beliefs and rules */

// user-defined predicates
initial_state(Glob) :-
   Merge = choice([Off,Fork]) &
   Off= seq(msg(b, c, tell, offer),
            seq(msg(c, b, tell, counter), Merge)) &
   Fork= fork(seq(msg(b, s, tell, final),end),
              seq(msg(b, c, tell, result),end)) &
   Glob = seq(msg(s, b, tell, item),Merge).

has_type(offer(Item, Offer), offer) :-
   string(Item) & int(Offer).
has_type(counter(Item, Offer), counter) :-
   string(Item) & int(Offer).
has_type(final(Res, Client, Item, Offer), final) :-
   string(Res) & string(Client) & string(Item) & int(Offer).
has_type(result(Res, Item, Offer), result) :-
   string(Res) & string(Item) & int(Offer).
has_type(item(Client, Item), item) :-
   string(Client) & string(Item).
// end of user-defined predicates

timeout(4000).

type_check(msg(S, R, P, C), NewState) :-
   current_state(CurrentState) &
   next(CurrentState, msg(S, R, P, C), NewState).

// Rules defining the next predicate follow
........
```

The monitor prints every information relevant for testing on the console with the `.print` internal action. The `.send(R, P, C)` internal action implements the

asynchronous delivery of a message with performative `P` and content `C` to agent `R`.

A brief description of the main plans follow.

- Plan `test` is triggered by the initial goal `!test` that starts the testing, by setting the current state to the initial state.
- Plan `move2state` upgrades the belief about the current state.
- Plan `successfulMove` is triggered by the `!type_check_message(msg(S, R, P, C))` internal goal. If the `type_check(msg(S, R, P, C), NewState)` context is satisfied, then `S` is allowed to send the message with performative `P` and content `C` to `R`. The state of the protocol changes, and `monitor` notifies `S` that the message can be sent.
- Plan `failingMoveAndProtocol` is triggered, like `successfulMove`, by the `!type_check_message(msg(S, R, P, C))` internal goal. It is used when `successfulMove` cannot be applied because its context is not verified. This means that `S` is not allowed to send message `P` with content `C` to `R`, because a dynamic type error has been detected: the message does not comply with the protocol.
- Plan `messageReceptionOK` is triggered by the reception of a tell message with `msg(S, R, P, C)` content; the message is checked against the protocol, and the progress check is activated (`!check_progress` succeeds either if a message is received before a default timeout, or if the timeout elapses, in which case `!check_progress` is activated again: `.wait(+msg(S1, R1, P1, C1), MS, Delay)` suspends the intention until `msg(S1, R1, P1, C1)` is received or `MS` milliseconds have passed, whatever happens first; `Delay` is unified to the elapsed time from the start of `.wait` until the event or timeout).

All plans whose context involves checking the current state and/or whose body involves changing it are defined as atomic ones, to avoid problems due to interleaved check-modify actions.

```
/* Initial goals */

!test.

/* Monitor's plans */

@test[atomic]
+!test : initial_state(InitialState)
   <- +current_state(InitialState).

@move2state[atomic]
+!move_to_state(NewState) : current_state(LastState)
   <- -current_state(LastState);
      +current_state(NewState).

@successfulMove[atomic]
+!type_check_message(msg(S, R, P, C)) : type_check(msg(S, R, P, C), NewState)
   <- !move_to_state(NewState);
      .print("\nMessage ", msg(S, R, P, C), "\nleads to state ", NewState, "\n");
      .send(S, tell, ok_check(msg(S, R, P, C))).

@failingMoveAndProtocol
+!type_check_message(msg(S, R, P, C)) : current_state(Current)
```

```
   <- .print("\n*** DYNAMIC TYPE-CHECKING ERROR ***\nMessage ", msg(S, R, P, C),
      "\ncannot be accepted in the current state ", Current, "\n");
       !move_to_state(failure).

@messageReceptionOK
+msg(S, R, P, C)[source(S)]: true
   <- -msg(S, R, P, C)[source(S)];
      !type_check_message(msg(S, R, P, C));
      !check_progress.

+!check_progress : timeout(MS)
   <- .wait({+msg(S1, R1, P1, C1)}, MS, Delay);
      !aux_check_progress(Delay).

+!aux_check_progress(Delay) : timeout(MS) & Delay < MS.

+!aux_check_progress(Delay) : timeout(MS) & current_state(Current) & Delay >= MS
   <- .print("\n*** WARNING ***\nNo progress for ", Delay, " milliseconds
      in the current state ", Current, "\n");
      !check_progress.
```

3.2 Participants

We assume that participants interact via asynchronous exchange of messages with `tell` performatives.

To keep the implementation as general and flexible as possible, in the participants' code extended as explained below we use the `Perf` logical variable where the message performative is expected. Under the assumption that only `tell` performatives will be used, `Perf` will always be bound to the `tell` ground atom.

Only two changes are required to the code of participants:

1. `.send` is replaced by `!my_send` and
2. two plans are added for managing the interaction with the monitor.

The first plan is triggered by the `!my_send` internal goal; `my_send` has the same signature as the `.send` internal action, but, instead of sending a message with performative `Perf` and `Content` to `Receiver`, it sends a `tell` message to the monitor in the format `msg(Sender, Receiver, Perf, Content)`. When received, this message will be checked by the monitor against the global type, as explained in Section 3.1.

The second plan is triggered by the reception of the monitor's message that allows the agent to actually send `Content` to `Receiver`, by means of a message with performative `Perf`. In reaction to the reception of such a message, the agent sends the corresponding message to the expected agent.

```
/* Plans for runtime type checking */

+!my_send(Receiver, Perf, Content) : true
   <- .my_name(Sender);
      .send(monitor, tell, msg(Sender, Receiver, Perf, Content)).

+ok_check(msg(Sender, Receiver, Perf, Content))[source(monitor)] : true
   <- -ok_check(msg(Sender, Receiver, Perf, Content))[source(monitor)];
      .send(Receiver, Perf, Content).
```

3.3 Experiments

Table 1 summarizes the results of some of the experiments we carried out with the brokering protocol. The full implementation of the seller, client and broker agents, as well as the messages printed by the monitor on the console are described in [2].

- *Broker: i.o.* is the initial offer the broker makes to the client.
- *Broker: a.o.* is the lowest price the broker is willing to accept for selling oranges to the client.
- *Client: c.o.* is the client's initial counter offer.
- *Code* is the agents code used to run the experiment.
- *Expected res.* and *Obtained res.* are the expected and obtained results.
- *Bug1*: instead of sending a counter offer upon reception of the broker's offer, the client sends an `offer` followed by a message with unknown type.
- *Bug2*: the client autonomously starts to interact with the broker before the initial messages that the protocol enforces have been sent.
- *Bug3*: we deleted all the plans triggered by the reception of `+counter(Item, Offer)[source(Client)]` from the broker's code, making the broker agent unable to react to a counter offer.

Our "meta-testing" of the testing mechanism was successful. We run the MAS with many other values of the initial and acceptable offers, and with other communication errors, always obtaining the expected result. The simpler protocols involving `alice` and `bob` agents described in Section 2 have been successfully tested as well.

Broker: i.o.	**Broker: a.o.**	**Client: c.o.**	**Code**	**Expected res.**	**Obtained res.**
11	6	3	Correct	ok	ok
8	6	2	Correct	noDeal	noDeal
8	6	5	Bug1	protocol error	protocol error
8	6	5	Bug2	protocol error	protocol error
8	6	5	Bug3	no progress	no progress

Table 1. Some of our experiments with the brokering protocol

3.4 Discussion

Alternative implementations. We opted to implement the proof-of-concept of our approach by extending the code of the existing participants rather than modifying the code of the Jason interpreter, because this was the simplest and quickest solution we could devise for developing a prototype, and easily experimenting different design choices. However, the same results could be obtained by

directly modifying the `.send` internal action by overriding the underlying agent architecture methods of Jason responsible for sending and receiving messages.

This solution would not require any modification of the code of the participants, and would allow the monitor to forward the message, when correct, directly to the recipient agent, thus reducing the number of interactions required among agents.

Another interesting solution would consist in creating a monitor agent for each agent participating to the interaction, thus avoiding the communication problems of the centralized approach where the unique monitor is required to exchange a large amount of messages with the other agents; however, this solution requires to project the global session type to end-point types (a.k.a. local types), specifying the expected behavior of each single agent involved in the interaction. Depending on the considered notion of global type, it might be non trivial to find an efficient and complete projection algorithm.

Global type transition. We have already shown that the `next` predicate is ensured to terminate on contractive global types; however, a developer may erroneously define a non contractive type for testing its system. Fortunately, there exist algorithms for automatically translating a non contractive global type into an equivalent contractive one.

Another issue concerns non deterministic global types, that is, global types where transitions are not deterministic. Consider for instance the following global type:

```
fork(seq(msg(alice,bob,tell,ping),
         seq(msg(bob,alice,tell,pong),end)),
     seq(msg(alice,bob,tell,ping),
         seq(msg(alice,bob,tell,bye),end)))
```

In this case the `next` predicate has to guess which of the two operand types must progress upon reception of the message matching with `msg(alice,bob,tell,ping)`; this means that in case of non deterministic global types the monitor may detect false positives. To avoid this problem one could determinize the type, but depending on the considered notion of global type, it would not be easy, or even possible, to devise a determinization algorithm. Alternatively, the monitor could store the whole sequence of received sending actions to allow backtracking in case of failure, thus making the testing procedure much less efficient.

Finally, it is worth mentioning that the proposed approach makes an efficient use of memory space if the initial global type does not contain loops with the `fork` constructor. In this case the space required by a global type representing an intermediate state is bounded by the size of the initial global type; since only one type at a time is kept in the belief base of the monitor, this implies a significant space optimization when the total number of all possible states is exponential w.r.t. the size of the initial global type. As already pointed out, this consideration does not apply to types with loops involving the `fork` constructor, like in the following example:

```
T = fork(seq(msg(alice,bob,tell,ping),T),
```

```
seq(msg(bob,alice,tell,pong),T)).
```

In this case the term grows at each transition step (and there are cases where the type cannot be simplified to a smaller one); however, we were not able to come up with examples of realistic protocols that require types with `fork` in a loop to be specified.

4 Related and Future Work

Our work represents a first step in two directions: extending an existing agent programming language with session types, and supporting testing of protocol conformance within a MAS. In this section we consider the related works in both areas, discuss the (lack of) proposals of integrating session types in existing MASs frameworks, and outline possible extensions of our work.

Session types on top of existing programming languages. The integration of session types into existing languages is a recent activity, dating back to less than ten years ago for object oriented calculi, and less than five years for declarative ones. The research field is very lively and open, with the newest proposals published just a few months ago.

Session types have been integrated into object calculi starting from 2005 [8, 9]. The first full implementation of a language and run-time for session-based distributed programming on top of Java, featuring asynchronous message passing, delegation, session subtyping and interleaving, combined with class downloading and failure handling, dates back to 2008 [13]. More recently, a Java language extension has been proposed, that counters the problems of traditional event-based programming with abstractions and safety guarantees based on session types [12].

Closer to our work on declarative languages, the paper [18] discusses how session types have been incorporated into Haskell as a standard library that allows the developer to statically verify the use of the communication primitives provided without an additional type checker, preprocessor or modification to the compiler. A session typing system for a featherweight Erlang calculus that encompasses the main communication abilities of the language is presented in [16]. Structured types are used to govern the interaction of Erlang processes, ensuring that their behavior is safe with respect to a defined protocol.

Protocol representation and verification in MASs. Because of the very nature of MASs as complex systems consisting of autonomous communicating entities that must adhere to a given protocol in order to allow the MAS correct functioning, the problem of how representing interaction protocols has been addressed since the dawning of research on MASs (one of the most well known outcomes being FIPA AUML interaction diagrams [14]), and the literature on protocol conformance verification is extremely rich.

Although a bit dated, [3] still represents one of the most valuable contributions to *verification of a priori conformance*. In that paper the authors propose

an approach based on the theory of formal languages. The ability to formally prove the interoperability of two policies (the actual protocol implementations), each of which is compliant with a protocol specification, is one of the main features of the proposed approach whose aim is however deeply different from ours, being devoted to a static analysis carried out before the interaction takes place.

The problem of *verifying the compliance of protocols at run time* has been tackled – among others – within the SOCS project[7], where the SCIFF computational logic framework [1] is used to provide the semantics of social integrity constraints. Such a semantics is based on abduction: expectations on the possibly observable, yet unknown, events are modeled as abducibles and social integrity constraints are represented as integrity constraints. To model MAS interaction, expectation-based semantics specifies the links between the observed events and the expected ones. The recent paper "Modelling Interactions via Commitments and Expectations" [20] discusses that and related approaches. Although aimed at testing run-time conformance of an actual conversation with respect to a given protocol, our approach differs from the expectation-based one in many respects, including the lack of notion of expectation in the agent language, and the implementation of the testing mechanism in a seamless way on top of an existing and widespread agent-oriented programming language.

As far as formalisms for representing agent interaction protocols are concerned, the reader may find a concise but very good survey in Section 4 of [19]. In that paper, the authors propose a commitment-based semantics of protocols. Commitments involve a debtor, a creditor, an antecedent, and a consequent: the debtor stakes a claim or makes a promise to the creditor about the specified consequent provided that the antecedent holds. Protocols specify business interactions by stating how messages affect the participants' commitments. That setting allows the authors to determine if a protocol refines another protocol, how protocols may be aggregated into other protocols, and to verify interoperability properties of agents and roles (safety, liveness, or alignment), conformance of roles, and compliance of agents. Our approach is currently limited to the run-time verification of the MAS compliance to the interaction protocol, but the exploitation of session types as the formalism to represent protocols allows us to take advantage of all the results achieved in the session types research field, which include session subtyping and algorithms for static verification of protocol properties such as safety and liveness.

The ability to specify the type of messages (`has_type(`c`,`ct`)` predicate) in order to relate actual messages to messages specified in the protocol, usually given at a more abstract level, is a characterizing feature of our approach and seems to be supported by none of the proposals mentioned above.

Session Types and MASs. As demonstrated for example by the Scribble language mentioned in the Introduction and by [10], using session types to represent and verify protocol conformance inside MASs is not a new idea but, to the best of our knowledge, no attempts of taking advantage of global session types to verify

[7] http://lia.deis.unibo.it/research/projects/SOCS/

MASs programmed in some widespread agent oriented programming languages had been made so far, and our proposal is an original one.

Future extensions. Our work can be extended in many ways, as already discussed to some extent in Section 3. Besides the specific extensions mentioned there, and the fully automatic generation of the monitor and participants code, our short term goals include analyzing how our approach could be extended to other Prolog-based agent-programming languages, such as GOAL [4] or 2APL [6], and designing more complex protocols to stress-test our system and provide a quantitative assessment of its runtime behavior and scalability.

In the medium term, we plan to work for evolving our mechanism towards a framework supporting self-recovering MASs. This evolution would require to modify the way we extend the code of the participant agents, in order to automatically select other messages to send in the current state, if any, in case the monitor realizes that the chosen one does not respect the protocol. Default recovery actions for the situation where no other choices are available, should be defined as well. In such a context – more oriented towards verification of interoperability of deployed systems rather than testing of systems-to-be –, agents might advertise to the monitor the services they offer and the protocols to follow in order to obtain them. Besides ensuring the protocol's compliance, the monitor could then act as a repository of <*service specification, protocol specification*> couples, helping agents to locate services in an open MAS in a similar way the Universal Description, Discovery and Integration (UDDI) registry does for web services.

In the long term, the integration of ontology-based meaning into protocol specifications, leading to "ontology-aware session types", will be addressed. Our previous work on CooL-AgentSpeak [15] will represent the starting point for that extension.

Acknowledgments

We are grateful to J. F. Hübner and R. H. Bordini for their effort in making cyclic terms in Jason belief base persistent, thus making the implementation of our monitor agent possible. We also thank the anonymous reviewers for their careful reading of the paper and for the valuable suggestions provided to improve its quality.

References

1. M. Alberti, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. The *S*CIFF abductive proof-procedure. In *AI*IA*, pages 135–147, 2005.
2. D. Ancona, S. Drossopoulou, and V. Mascardi. Automatic generation of self-monitoring MASs from multiparty global session types - extended version. Technical report, University of Genova, Department of Computing, 2012. Online at http://www.disi.unige.it/person/MascardiV/Download/sessionTypes4MASs.pdf.

3. M. Baldoni, C. Baroglio, A. Martelli, and V. Patti. Verification of protocol conformance and agent interoperability. In *CLIMA VI, 2005, Revised Selected and Invited Papers*, volume 3900 of *LNCS*, pages 265–283. Springer, 2005.
4. L. Braubach, A. Pokahr, D. Moldt, and W. Lamersdorf. Goal representation for BDI agent systems. In *ProMAS 2004, Selected Revised and Invited Papers*, volume 3346 of *LNCS*, pages 44–65. Springer, 2004.
5. M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In *ESOP'07 (part of ETAPS 2007)*, volume 4421 of *LNCS*, pages 2–17. Springer, 2007.
6. M. Dastani. 2APL: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, 2008.
7. P.-M. Deniélou and N. Yoshida. Multiparty session types meet communicating automata. In *ESOP'12 (part of ETAPS 2012)*, LNCS. Springer, 2012.
8. M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session types for object-oriented languages. In *ECOOP 2006*, volume 4067 of *LNCS*, pages 328–352. Springer, 2006.
9. M. Dezani-Ciancaglini, N. Yoshida, A. Ahern, and S. Drossopoulou. A distributed object-oriented language with session types. In *TGC 2005, Revised Selected Papers*, volume 3705 of *LNCS*, pages 299–318. Springer, 2005.
10. C. Grigore and R. Collier. Supporting agent systems in the programming language. In *WI/IAT*, pages 9–12. IEEE Computer Society, 2011.
11. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL 2008*, pages 273–284. ACM, 2008.
12. R. Hu, D. Kouzapas, O. Pernet, N. Yoshida, and K. Honda. Type-safe eventful sessions in Java. In *ECOOP 2010*, volume 6183 of *LNCS*, pages 329–353. Springer, 2010.
13. R. Hu, N. Yoshida, and K. Honda. Session-based distributed programming in Java. In *ECOOP 2008*, volume 5142 of *LNCS*, pages 516–541. Springer, 2008.
14. M.-P. Huget, B. Bauer, J. Odell, R. Levy, P. Turci, R. Cervenka, and H. Zhu. FIPA modeling: Interaction diagrams. Working Draft Version 2003-07-02. Online at http://www.auml.org/auml/documents/ID-03-07-02.pdf.
15. V. Mascardi, D. Ancona, R. H. Bordini, and A. Ricci. CooL-AgentSpeak: Enhancing AgentSpeak-DL agents with plan exchange and ontology services. In *IAT 2011*, pages 109–116. IEEE Computer Society, 2011.
16. D. Mostrous and V. T. Vasconcelos. Session typing for a featherweight Erlang. In *COORDINATION 2011*, volume 6721 of *LNCS*, pages 95–109. Springer, 2011.
17. A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *MAAMAW'96*, volume 1038 of *LNCS*, pages 42–55. Springer, 1996.
18. M. Sackman and S. Eisenbach. Session types in Haskell: Updating message passing for the 21st century. Technical report, Imperial College, Department of Computing, 2008. Online at http://spiral.imperial.ac.uk:8080/handle/10044/1/5918.
19. M. P. Singh and A. K. Chopra. Correctness properties for multiagent systems. In *DALT 2009, Revised Selected and Invited Papers*, volume 5948 of *LNCS*, pages 192–207. Springer, 2009.
20. P. Torroni, P. Yolum, M. P. Singh, M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, and P. Mello. Modelling interactions via commitments and expectations. In *Handbook of Research on Multi-Agent Systems: Semantics and Dynamics of Organizational Models*. IGI Global, 2009.

A Generalized Commitment Machine for 2CL Protocols and its Implementation

Matteo Baldoni, Cristina Baroglio, Federico Capuzzimati,
Elisa Marengo, and Viviana Patti

Università degli Studi di Torino — Dipartimento di Informatica
c.so Svizzera 185, I-10149 Torino (Italy)

Abstract. In practical contexts where protocols model business interactions (e.g. trading, banking), designers need tools allowing them to analyse the impact on the possible interactions of regulations, preferences, conventions and the like. This work faces the issue of how to equip commitment protocols with formal and practical instruments aimed at supporting such an analysis by identifying the possible risks of violation and, thus, enabling the definition of operational strategies aimed at reducing risks of violation. Specifically, we present an operational semantics for the commitment protocol language 2CL as well as a tool for visualizing as a graph the possible interactions, labelling the states of the interaction so as to highlight legal situations and violations.

Keywords: Commitment-based Interaction Protocols, Constraints among Commitments, Commitment Machine, Risks of Violation

1 Introduction and Motivation

Agent interaction is generally specified by defining *interaction protocols* [16]. For communicating with one another, agents must follow the schema that the protocol shapes. Different protocol models can be found in the literature, this work concerns *commitment-based protocols* [14, 19]. This kind of protocols relies on the notion of commitment, which in turn encompasses the notions of debtor and creditor: when a commitment is not fulfilled, the debtor is liable for that violation but as long as agents reciprocally satisfy their commitments, any course of action is fine.

In many practical contexts where protocols model *business interactions* (e.g. trading, banking), designers must be able to regulate and constrain the possible interactions as specified by *conventions*, *regulations* [4], *preferences* or *habits*. Some proposals attack the issue of introducing similar regulations inside commitment protocols [3, 8, 5, 12], however, none of them brought yet to the realization of a *tool* that allows visualizing and analysing how regulations or constraints impact on the interactions allowed by a commitment-based protocol. The availability of intuitive and possibly graphical tools of this kind would support the identification of possible violations, thus enabling an *analysis of the risks* the

interaction could encounter. As a consequence, it would be possible to raise alerts concerning possible violations before the protocol is enacted, and to reduce risks by defining proper operational strategies, like regimentation (aimed at preventing the occurrence of violations) or enforcement (introduction of warning mechanisms) [11].

The work presented in this paper aims at filling this gap. To this purpose, we started from the commitment protocol language 2CL described in [3], whose key characteristic is the extension of the regulative nature of commitments by featuring the definition of patterns of interaction as sets of *constraints*. Such constraints declaratively specify either conditions to be achieved or the order in which some of them should be achieved. The first contribution is, therefore, a formal, operational semantics for the proposal in [3], which relies on the Generalized Commitment Machine in [15]. We named our extension 2CL-Generalized Commitment Machines (2CL-GCM for short). On top of this, it was possible to realize the second contribution of this work: a Prolog implementation for 2CL-GCM, which extends the implementation in [17], and is equipped with a graphical tool to explore all the possible executions, showing both commitment and constraint violations. The implementation is part of a plug-in Eclipse which supports 2CL-protocol design and analysis.

The chief characteristic of our solution is that it performs a *state evaluation* of protocol constraints, rather than performing path evaluation (as, instead, done by model checking techniques). State evaluation allows considering each state only once, labelling it as a state of violation if some constraint is violated in it or as a legal state when no constraint is violated. This is a great difference with respect to path evaluation, where a state belonging to different paths can be classified as a state of violation or not depending on the path that is considered. The advantage is practical: state evaluation allows to easily supply the user an overall view of the possible alternatives of action, highlighting those which will bring to a violation and those that will not. State evaluation, however, is possible only by making some restriction on the proposal in [3]. Specifically, we assume that the domain is expressed in terms of positive facts only.

The paper is organized as follows. Section 2 briefly summarizes 2CL interaction protocol specification. Section 3 describes the formalization of 2CL-GCM. Section 4 presents a Prolog implementation of 2CL-GCM. Section 5 describes the 2CL Tools that supply features for supporting the protocol design and analysis. Related Work and Conclusions end the paper. Along the paper we use as a running example the well-known NetBill interaction protocol.

2 Background: 2CL Interaction Protocols

Let us briefly recall the chief characteristics of commitment protocols, as defined in [3]. In this approach, commitment protocols feature an explicit distinction between a *constitutive* and a *regulative* specification. The former defines the protocol actions, while the latter encodes the constraints the interaction should respect. Both specifications are based on *commitments*. Commitments are di-

	Relation	*Operator*	*Repr.*	*LTL formula*
Relation Operators	Correlation	*A correlate B*	$A \bullet\!\!- B$	$\Diamond A \supset \Diamond B$
		A not correlate B	$A \bullet\!\!\not- B$	$\Diamond A \supset \neg\Diamond B$
	Co-existence	*A co-exist B*	$A \bullet\!\!-\!\!\bullet B$	$A \bullet\!\!- B \wedge B \bullet\!\!- A$
		A not co-exist B	$A \bullet\!\!\not-\!\!\bullet B$	$A \bullet\!\!\not- B \wedge B \bullet\!\!\not- A$
Temporal Operators	Response	*A response B*	$A \bullet\!\!\rightarrow B$	$\Box(A \supset \Diamond B)$
		A not response B	$A \bullet\!\!\not\rightarrow B$	$\Box(A \supset \neg\Diamond B)$
	Before	*A before B*	$A \multimap\!\!\bullet B$	$\neg B \cup A$
		A not before B	$A \not\multimap\!\!\bullet B$	$\Box(\Diamond B \supset \neg A)$
	Cause	*A cause B*	$A \bullet\!\!\rightarrow\!\!\bullet B$	$A \bullet\!\!\rightarrow B \wedge A \multimap\!\!\bullet B$
		A not cause B	$A \bullet\!\!\not\rightarrow\!\!\bullet B$	$A \bullet\!\!\not\rightarrow B \wedge A \not\multimap\!\!\bullet B$

Table 1. 2CL operators and their meaning.

rected from a debtor to a creditor. The notation $C(x, y, r, p)$ denotes that agent x commits to an agent y to bring about the consequent condition p when the antecedent condition r holds. When r equals *true*, we use the short notation $C(x, y, p)$. The interacting partners share a *social state* that contains commitments and other facts that are relevant to their interaction. Every partner can affect the social state by executing actions, whose definition is given in terms of operations onto the social state, see [19]. The partners' behaviour is affected by commitments, which have a *regulative* nature, in that debtors should act in accordance with the commitments they have taken.

Definition 1 (Interaction protocol). *An interaction protocol $\mathcal{P}$ is a tuple $\langle Ro, F, s_0, A, C\rangle$, where Ro is a set of roles, identifying the interacting parties, F is a set of facts and commitments that can occur in the social state, s_0 is the set of facts and commitments in the initial state of the interaction, A is a set of actions, and C is a set of constraints.*

The set of social actions A, defined on F and on Ro, forms the *constitutive specification* of the protocol. The social effects are introduced by the construct *means*, and their achievement can depend on a precondition (conditional effects). Both preconditions and effects are given in terms of the set F specified in the protocol, which contains commitments and facts (i.e. the conditions that are brought about). The means construct amounts to a *counts-as* relation [13, 11]. For instance, consider the action *sendGoods* reported in Table 2. Its social meaning is that it makes the facts *goods* true (the goods were delivered to the customer) and creates the commitment $\mathsf{C}(m, c, pay, receipt)$ that corresponds to a promise by the merchant to send a receipt after the customer has paid. Further examples can be found in the first part of Table 2, which reports all the actions of the NetBill protocol. The formalization is inspired by those in [19, 17].

The *regulative specification* of the protocol is made of the set of 2CL constraints C, defined on F and on Ro as well. 2CL is a declarative language, which allows expressing what is mandatory and what is forbidden without the

Action Definitions
(a1) *sendRequest* **means** *request* **if** $\neg quote \wedge \neg goods$
(a2) *sendQuote* **means** $quote \wedge create(\mathsf{C}(m, c, \mathsf{C}(c, m, goods, pay), goods))$
$\wedge\ create(\mathsf{C}(m, c, pay, receipt))$
(a3) *sendAccept* **means** $create(\mathsf{C}(c, m, goods, pay))$ **if** $\neg pay$
(a4) *sendGoods* **means** $goods \wedge create(\mathsf{C}(m, c, pay, receipt))$
(a5) *sendEPO* **means** pay
(a6) *sendReceipt* **means** $receipt$ **if** pay

Constraints
(c1) $quote \rightarrowtriangle \mathsf{C}(c, m, goods, pay) \vee \mathsf{C}(c, m, pay)$
(c2) $\mathsf{C}(m, c, pay, receipt) \wedge goods \rightarrowtriangle pay$
(c2) $pay \bullet\!\!\rightarrowtriangle receipt$

Table 2. Actions and constraints for the NetBill protocol: m stands for merchant while c stands for customer.

need of listing the possible executions extensionally. Constraints have the form "dnf_1 op dnf_2", where dnf_1 and dnf_2 are disjunctive normal forms of facts and commitments, and op is one of the 2CL operators, reported in Table 1 together with their Linear-time Temporal Logic [7] interpretation and with their graphical notation. Basically, there are two kinds of operators: *relational* and *temporal*. The former kind expresses constraints on the co-occurrence of conditions (if this condition is achieved then also that condition must be achieved, but the order of the two achievements does not matter). For instance, one may wish to express that both the payment for some item and its delivery must occur without constraining the order of the two conditions: no matter which occurs first, when one is met, also the other must be achieved. Temporal operators, instead, capture the relative order at which different conditions should be achieved. The second part of Table 2 reports the constraints imposed by the NetBill protocol: *(c1)* means that a quotation for a price must occur before a commitment to pay or a conditional commitment to pay given that some goods were delivered; *(c2)* that the conditional commitment to send a receipt after payment and the delivery of goods must occur before the payment is done; *(c3)* that after payment a receipt must be issued and if a receipt is issued a payment must have occurred before.

Among the *possible* interactions, derivable from the action specification, those that respect the constraints are said to be *legal*. Violations amounting to the fact that a constraint is not respected can be detected *during the execution*. The following section provides the operational semantics 2CL lacked of.

3 2CL Generalized Commitment Machine

In order to provide the semantics of commitment protocols as specified in [3] (see Definition 1 of this paper), we define the 2CL *generalized commitment machine*

(2CL-GCM). Briefly, 2CL-GCM relies on the notion of *generalized commitment machine* (GCM) (introduced in [15]) for what concerns the inference of the possible evolutions of the social state, that can be obtained by taking into account only the protocol actions and the commitment life cycle. Additionally, 2CL-GCM also accounts for 2CL constraints.

According to [15], a GCM features a set S of possible *states*, each of which is represented by a logical expression of facts and commitments. S represents the possible configurations of the social state.

Example 1. Considering NetBill, the expression $goods \wedge \mathsf{C}(c, m, pay)$ represents one possible configuration of the social state, i.e. it is a state in S. This expression means that the goods were shipped and that there is a commitment from c (customer) to m (merchant) to pay for them. Another example is $goods \wedge pay \wedge \mathsf{C}(m, c, receipt)$, meaning that not only the goods were shipped but that the payment also occurred, and that there is an active commitment from m to c to having a receipt sent.

Particularly relevant is the subset of S, whose elements are named *good states*: they are the desired final states of the interaction. The characterization of good states depends on the particular application. For instance, they may be only those that do not contain unsatisfied active commitments, or they could be the ones which satisfy a condition of interest (e.g. payment done and goods shipped).

In GCM, transitions between the states are logically inferred on the basis of an action theory Δ, that contains a set of axioms of the kind $p \overset{a}{\hookrightarrow} q$, meaning that q is a consequence of performing action a in a state where p holds. When q is false the meaning is that a is impossible if p holds. In general, Δ contains all the axioms deriving from the specification of the protocol actions. Additionally, Δ also contains an axiom for each action a and for each couple of states s and s' such that the execution of a in s causes a transition to s': for instance, if the precondition p of a is satisfied in s and its effect q is satisfied in s', then $s \overset{a}{\hookrightarrow} s'$ is in Δ. The way in which these axioms are obtained is explained in [15].

Example 2. According to the 2CL protocol syntax, the action *sendAccept*, performed by the customer to accept a quote of the merchant, is defined as *sendAccept* **means** CREATE(C(c,m,goods,pay)) **if** ¬ pay. The corresponding axiom is $\neg pay \overset{sendAccept}{\hookrightarrow} \mathsf{C}(c, m, goods, pay)$. Now, if one considers a state in which $\neg pay \wedge quote$ holds, it is also possible to infer the axiom $\neg pay \wedge quote \overset{sendAccept}{\hookrightarrow} \mathsf{C}(c, m, goods, pay)$.

In GCM paths must be infinite. All the finite paths are transformed into infinite ones by adding a transition from the last state of the finite path towards an artificial new state with a self loop [15]. In 2CL-GCM we adopt the same assumption and the same mechanism for transforming finite paths into infinite ones. We are now ready to define 2CL-GCM. The definition adopts the same notation in [15].

Definition 2 (2CL **Generalized Commitment Machine).** *Let* $\vdash$ *and* $\equiv$ *be, respectively, the logical consequence and the logical equivalence of propositional logic. A* 2CL-GCM *is a tuple* $\mathbf{P} = \langle S, A, s_0, \Delta, G, C \rangle$*, where:*

- S *is a finite set of states;*
- A *is a finite set of actions;*
- $s_0 \in S$ *is the initial state;*
- Δ *is an action theory;*
- $G \subseteq S$ *is a set of good states;*
- C *is a set of* 2CL *constraints.*

(i) *Members of* S *are logically distinct, that is:* $\forall s, s' \in S, s \not\equiv s'$*;* (ii) $\mathsf{false} \notin S$*; and* (iii) $\forall s \in G, s' \in S : (s' \vdash s) \Rightarrow (s' \in G)$*, i.e. any state that logically derives a good state is also good.*

A sequence of states is a *path* of a 2CL-GCM if it satisfies all of the constraints in C. Since 2CL constraints are defined in terms of LTL formulas, to perform the verification one can consider the *transition system* corresponding to the path. Given a sequence of states interleaved by actions, the corresponding transition system can be derived quite straightforwardly. Intuitively, the set of states and transitions of the system is the same set of states and transitions in the sequence. A requirement on transition systems is that each state has at least one outgoing transition (i.e. runs are infinite).

Definition 3 (Transition System). *Let* $\tau = \langle (\tau_0, a_0, \tau_1), (\tau_1, a_1, \tau_2), \dots \rangle$ *be an infinite and ordered sequence of triples, where* τ_i *is the state at position* i *in* τ *and* a_i *is the action that causes the transition from state* τ_i *to state* τ_{i+1}*. The transition system* $T(\tau)$ *corresponding to* τ *is a triple* $\langle S_\tau, \delta_\tau, L_\tau \rangle$ *where:*

- $S_\tau = \{\tau_i |\ \tau_i \in \tau\}$ *is a set of states;*
- $\delta_\tau : S_\tau \to S_\tau$ *is a transition function where:* $\delta(\tau_j) = \tau_k$ *iff* $(\tau_j, a, \tau_k) \in \tau$*;*
- $L : S_\tau \to 2^F$ *is a labelling function, where:* F *is a set of facts and commitments and given* $l \in F$*, then* $l \in L(\tau_i)$ *iff* $\tau_i \vdash l$*.*

To define a 2CL-GCM path, we adapt the definition of GCM path by adding the requirement that the sequence of states satisfies all the constraints of the 2CL-GCM. This condition is checked on the transition system corresponding to the path, by means of the LTL satisfaction relation [1]. We denote it with the symbol $\models_{LTL}$. In the following definition we adopt the same notation in [15].

Definition 4 (2CL-GCM **path).** *Let* $\mathbf{P} = \langle S, A, s_0, \Delta, G, C \rangle$ *be a* 2CL-GCM*. Let* $\tau = \langle (\tau_0, a_0, \tau_1), \dots \rangle$ *be an infinite sequence of triples and* $T(\tau)$ *be the corresponding transition system. Let* $inf(\tau)$ *be the set of states that occur infinitely often in* τ*.* τ *is a path generated from* $\mathbf{P}$ *when:*

(i) $\forall (\tau_i, a_i, \tau_{i+1})$ *in* τ *then* $\tau_i, \tau_{i+1} \in S$ *and* $a_i \in A$ *and* $\tau_i \overset{a_i}{\hookrightarrow} \tau_{i+1} \in \Delta$*; and*
(ii) $inf(\tau) \cap G \neq \emptyset$*; and*
(iii) $\forall c \in C :\ T(\tau), \tau_0 \models_{LTL} c$

In the above definition, *(i)* and *(ii)* are the conditions for a path to be generated from a GCM [15]. Condition *(i)* requires that each *state* in the sequence is a state of the 2CL-GCM, that the *action* that causes the transition from a state to the subsequent one in the sequence is an action of the 2CL-GCM, and that the transition is inferable according to the *axioms* in Δ. It also requires that the path is *infinite*. Condition *(ii)* requires that at least one *good state* occurs infinitely often in the sequence. Condition *(iii)* was added to account for the evaluation of the protocol constraints. According to the LTL semantics, the notation $\mathcal{M}, s \models_{LTL} \varphi$ means that every execution path π of $\mathcal{M}$, starting at s, is such that $\pi \models_{LTL} \varphi$. Since $T(\tau)$ is a transition system made only of *one* linear path (by construction), whose starting state is the starting state of τ, the condition $T(\tau), \tau_0 \models_{LTL} c$ amounts to checking if c is satisfied in the path of the transition system, corresponding to τ.

Given a protocol specification it is possible to build the corresponding 2CL-GCM:

Definition 5 (2CL-GCM of a protocol). *Let $\mathcal{P} = \langle Ro, F, s_0, A, C\rangle$ be a protocol, S be a set of states and $G \subseteq S$ be a set of good states. $\mathbf{P} = \langle S, L_A, s_0, \Delta, G, C\rangle$ is a 2CL-GCM of $\mathcal{P}$ when (i) L_A is the set of action labels in A; and (ii) Δ is the action theory of A, i.e.:*

- *for each (a **means** e **if** p) belonging to A, then $p \overset{a}{\hookrightarrow} e$ belongs to Δ;*
- *Δ is closed under inference rules in [15].*

Since the state s_0 and the constraints C of a 2CL-GCM are the same of the protocol, the definition uses the same symbols. By varying the sets S and G different 2CL-GCMs associated to the same protocol are obtained: when S contains all the states that can be reached from s_0, applying the protocol actions, the machine can infer all the possible interactions; when S is smaller, only a subset of the possible interactions is determined.

4 Implementation of the 2CL Commitment Machine

This section describes a Prolog implementation for the 2CL-GCM, formalized above. It allows exploring all the possible executions of an interaction protocol, showing the *regulative violations*— i.e. both those states in which some constraint is violated and those that contain unsatisfied commitments. We prove that if a path is legal according to the implementation, then it is a path of the corresponding 2CL-GCM.

The implementation is realized in *tuProlog*[1] and it builds upon the *enhanced commitment machine* realized by Winikoff et al. [17]. By relying on it, we inherit the mechanisms for the computation of the possible interactions. Specifically, enhanced commitment machines feature the generation of the reachable states, the transitions among them and the management of commitments (like the operations of discharge, creation and so on). Our extension equips them with the

[1] `http://www.alice.unibo.it/xwiki/bin/view/Tuprolog/`

Relation	*State Condition*
Correlation	$\psi(A \bullet\!\!- B) = A \wedge B$
	$\psi(A \bullet\!\!\not- B) = \neg(A \wedge B)$
Co-existence	$\psi(A \bullet\!\!-\!\!\bullet B) = \psi(A \bullet\!\!- B) \ \wedge \psi(B \bullet\!\!- A)$
	$\psi(A \bullet\!\!\not-\!\!\bullet B) = \psi(A \bullet\!\!\not- B) \ \wedge \psi(B \bullet\!\!\not- A)$
Response	$\psi(A \bullet\!\!-\!\!\triangleright B) = A \wedge B$
	$\psi(A \bullet\!\!\not-\!\!\triangleright B) = \neg(A \wedge B)$
Before	$\psi(A \rightarrowtriangle\!\!\bullet B) = \neg(B \wedge \neg A)$
	$\psi(A \not\rightarrowtriangle\!\!\bullet B) = \neg(A \wedge B)$
Cause	$\psi(A \bullet\!\!-\!\!\triangleright\!\!\bullet B) = \psi(A \bullet\!\!-\!\!\triangleright B) \ \wedge \psi(A \rightarrowtriangle\!\!\bullet B)$
	$\psi(A \bullet\!\!\not-\!\!\triangleright\!\!\bullet B) = \psi(A \bullet\!\!\not-\!\!\triangleright B) \ \wedge \psi(A \not\rightarrowtriangle\!\!\bullet B)$

Table 3. State conditions corresponding to 2CL operators.

possibility of evaluating 2CL constraints. The aim is to provide a qualitative view of the possible interactions, highlighting those that violate some constraints. The interacting parties are not prevented from entering in illegal paths (autonomy is preserved), but they are made aware of the risks they are encountering and that they may incur in penalties as a consequence of the violations they caused [4].

In order to provide a compact but global view of the possible interactions, the evaluation of constraints is performed on *one state at a time* rather than on paths (as, instead, usually done in LTL model checking). Specifically, the *state content* is given in terms of positive facts and commitments. A fact that is not true in a state has not been achieved yet, so we use negation as failure in the conditions of the action definitions to verify whether a fact is present or not in the social state. In this setting, the evaluation of 2CL constraints can be made on single states. For instance, if in a state b holds but a does not, we can infer that the constraint '*a before b*' is violated. This kind of verification, however, can be performed only on a subset of 2CL formulas, specifically, only on constraints corresponding to conditions that persist (i.e. that involve DNFs of facts without negation). Since commitments do not persist because they can be cancelled, discharged, etc., another requirement is to *associate a fact to each operation* that is performed on commitments. These facts are automatically asserted whenever an operation is performed on a commitment and they can be used in constraint formulas. For instance, when a commitment $C(x, y, r, p)$ is created, the fact CREATED$(C(x, y, r, p))$ is added to the state, and so forth for the other operations.

Given a constraint c, we denote by $\psi(c)$ the corresponding condition to be verified one state at a time (*state condition*). The above assumptions allow the simplification of the LTL formulas, corresponding to the 2CL operators, in the way that is reported in Table 3. Consider, for instance, the *before* operator ($\rightarrowtriangle\!\!\bullet$): it requires that A is met before or in the same state of B. So, given a run π, if in π there is a state j such that B holds while A does not, that is a state where a violation occurred. In formulas: $\pi_i \models_{LTL} A \rightarrowtriangle\!\!\bullet B \ \Leftrightarrow \ \neg\exists j \geq i \ s.t. \ \pi_j \models_{LTL}$

$(B \wedge \neg A)$ (when a formula does not contain temporal operators, the relation $\models_{LTL}$ checks the condition in the first state of a path).

The other 2CL operators can be divided in two groups. *Correlation* ($\bullet\!\!-$) and *response* ($\bullet\!\!\rightarrow$) are part of the same group. $A \bullet\!\!- B$ requires that if A is achieved in a run, then also B is achieved in the same run (before or after A is not relevant). If B is achieved before A it will remain true also after. Therefore, in those cases in which the constraint is satisfied, from a certain time onwards both conditions will hold. In formulas: $\pi_i \models_{LTL} A \bullet\!\!- B \Leftrightarrow \neg\exists j \geq i \; s.t. \; \pi_j \models_{LTL} A$ and $\forall j' \geq j, \; \pi_{j'} \models_{LTL} (A \wedge \neg B)$. The same equivalence holds for $\pi_i \models_{LTL} A \bullet\!\!\rightarrow B$. In 2CL $A \bullet\!\!\rightarrow B$ requires that when A is met, B is achieved at least once later (even if it already occurred in the past) but under our assumptions it can be checked in the same way of correlation. The state condition amounts to verifying whether a state satisfies A but does not satisfy B. Notice that states that satisfy the test cannot be marked as states of violation because the constraint does not require B to hold *whenever* A holds. A state of violation is signalled when the interaction does not continue after it: we say that there is a *pending* condition.

Negated correlation, *response* and *before* correspond to the same formula: $\pi_i \models_{LTL} A \; \mathsf{op} \; B \Leftrightarrow \neg\exists j \geq i \; s.t. \; \pi_j \models_{LTL} (A \wedge B)$ where $\mathsf{op} \in \{\bullet\!\!\not-, \bullet\!\!\not\rightarrow, \not\bullet\!\!\rightarrow\}$. Intuitively, a constraint of the kind $A \bullet\!\!\not- B$ (negative correlation) requires that if A holds, B is not achieved. Since facts persist, this amounts to check that the two conditions do not hold in the same state, otherwise a violation occurs. *Negative response* (*negative before*) adds a *temporal aspect* to not-correlation: if A holds, B cannot hold later (before, respectively). Since facts persist, the first achieved condition will remain true also after the other becomes true. Also in this case we only need to check that the two conditions do not hold together.

Derived operators are decomposed and the reasoning made for the operators, from which they derive, is applied. For instance, *cause* ($\bullet\!\!\rightarrow\!\!\bullet$) derives from *before* and *response*. If a state does not satisfy the response part of the cause, it is marked as "pending"; if it violates the before part, it is marked as a "violation". Both labels are applied when the state does not satisfy any of the two.

Summarizing, given a constraint formula and a state in which to verify it, we have three possible cases: (i) the state satisfies the formula; (ii) the state does not satisfy the formula and this leads to a violation; and (iii) the state does not satisfy the formula but the violation is potential, depending on future evolution. Considering all the constraints of a protocol, a state can both violate some constraint and have pending conditions. Moreover, states are also evaluated based on the presence of unsatisfied active commitments.

These considerations enable the generation and the labelling of all the states that can be reached by applying the protocol actions. The result is a labelled graph, as defined in Definition 6, where each state is associated a set of labels.

Definition 6 (Labelled Graph). *Let $\mathcal{P} = \langle Ro, F, s_0, A, C \rangle$ be a protocol, the corresponding labelled graph $G(\mathcal{P})$ is a triple (S, δ, L) where:*

- *S is a set of states reachable from s_0, such that $\forall s, s' \in S, s \not\equiv s'$;*
- *$\delta \subseteq S \times A \times S$ is a transition relation such that $\forall (s, a, s') \in \delta$ then $s, s' \in S$ and $\exists a \in A$ s.t. when a is executed in s it determines s';*

– $L \subseteq S \times \{$*pending,violation,final,non-final*$\}$ *is a labelling relation such that given* $s \in S$*:*
  - *violation* $\in L(s)$ *iff* $\exists c \in C$ *s.t.* $s \not\models_{LTL} \psi(c)$ *and* c *is not a response or a correlation;*
  - *pending* $\in L(s)$ *iff* $\exists c \in C$ *s.t.* $s \not\models_{LTL} \psi(c)$ *and* c *is a response or a correlation;*
  - *final* $\in L(s)$ *iff there are no unsatisfied active commitments in* s*;*
  - *not-final* $\in L(s)$ *iff* s *contains unsatisfied active commitments.*

Following Definition 6, our implementation starts from the initial state and determines all the reachable states, by applying a depth-first search, as in [17]. The difference is that our representation of the states contains also a list of labels, which identify the presence of active commitments and of pending or violated conditions. Listing 1.1 reports part of the Prolog program that generates the labelled graph. The mechanism is as follows: given a state, *explore* finds the set of the possible successors by applying the effects of the actions, whose preconditions are satisfied in the state. A state is added only if it is new (not explored yet). Before adding it, *find_labels* considers all the constraints and checks them on the state. Constraints are represented as $constraint(A, B, Id)$, where *constraint* is the 2CL operator used by the constraint, A and B are the antecedent and the consequent conditions of the constraint, and Id is the identifier of the constraint. Listing 1.1 reports, as an example of tests performed on states, the verification of a *response* and of a *before*. The clause *check_pending*, that is reported here, verifies response constraints: it is satisfied if there is a constraint of kind response, whose antecedent condition can be derived in the state, while the consequent condition cannot. In this case, the label *pending* is added to the list of labels of the state. A similar clause checks the correlation constraint. Instead, the clause *check_violation*, checks before constraints, which are violated if the consequent condition can be derived in the state while their antecedent cannot. Other similar clauses, checking different conditions, are defined for the other operators. Finally, the program checks the presence of unsatisfied commitments (*check_commitments*) and adds the label *final* or *not-final* consequently. The result of running this program on a protocol specification is a graph of the reachable annotated states. Annotations follow the graphical convention explained in Section 5.

Given a labelled graph we are now able to define when a path is legal.

Definition 7 (Legal path). *Let* $G(\mathcal{P}) = (S, \delta, L)$ *be a labelled graph,* $\pi = \langle(\pi_0, a_0, \pi_1), \ldots, (\pi_{n-1}, a_{n-1}, \pi_n)\rangle$ *be a path of at least one state.* π *is a legal path of* $G(\mathcal{P})$ *when:*

(i) $\forall i \geq 0, \pi_i$ *is a state of the graph and* $(\pi_i, a_i, \pi_{i+1}) \in \delta$*.*
(ii) $\nexists i \geq 0$ *such that* $\pi_i \in \pi$ *and* $violation \in L(\pi_i)$*;*
(iii) $pending \notin L(\pi_n)$ *and* $final \in L(\pi_n)$*.*

Condition (i) requires that the transitions in the path find correspondence in the graph; (ii) requires that none of the states of the path violates a constraint; (iii)

```
explore(StateNum,Free,NextFree) :-
  state(StateNum,State,_),
  findall(t(StateNum,A,S2),nextstate(State,A,S2),Ts),
  add_states(Ts,Free,NextFree), add_transitions(Ts).

add_states([],N,N).
add_states([t(_,_,S)|Ss],N,N1) :-
  state(_,St,_), seteq(St,S), !, add_states(Ss,N,N1).
add_states([t(_,_,S)|Ss],N,N3) :-
  labels(S,L), assert(state(N,S,L)),
  N1 is N+1, explore(N,N1,N2), add_states(Ss,N2,N3).

labels(State,Labels) :- find_labels(State,[],Labels).

find_labels(S,L1,R) :-
  check_violation(S,L1,L2),
  check_pending(S,L2,L3),
  check_commitments(S,L3,R).

check_pending(State,L,[pending(Constr)|L]) :-
  response(A,B,Constr),
  consequence(A,State), \+consequence(B,State).

check_violation(State,L,[violation(Constr)|L]) :-
  before(A,B,Constr),
  consequence(B,State), \+consequence(A,State).

check_commitments(State,L,[final|L]) :-
  \+member(c(_,_,_),State).
check_commitments(State,L,[non-final|L]) :-
  member(c(_,_,_),State).
```

Listing 1.1. Prolog clauses that generate the labelled graph: consequence(A,State) is a clause that determines if the fact (or the DNF formula) A can be derived in State; response and before are constraints. The complete program is downloadable at the URL http://di.unito.it/2cl.

requires that the last state of the path does not contain pending conditions or unsatisfied commitments.

Given a legal path π, of a labelled graph produced by Listing 1.1, we can prove its correctness w.r.t. the 2CL-GCM built on the same protocol specification. This actually corresponds to prove that π is a path of the 2CL-GCM.

Theorem 1 (Soundness). *Let $\mathcal{P} = \langle Ro, F, s_0, A, C\rangle$ be a protocol; let $G(\mathcal{P})$ be the corresponding labelled graph; let $\pi = \langle(\pi_0, a_0, \pi_1), \ldots, (\pi_{n-1}, a_{n-1}, \pi_n)\rangle$ be a path; and let $\mathbf{P} = \langle S_\pi, A, s_0, \Delta, G, C\rangle$ be the 2CL-GCM of $\mathcal{P}$, where S_π is the set of states in π and G is the set of states in π that do not contains unsatisfied commitments. If π is a legal path of G then π is a path of $\mathbf{P}$.*

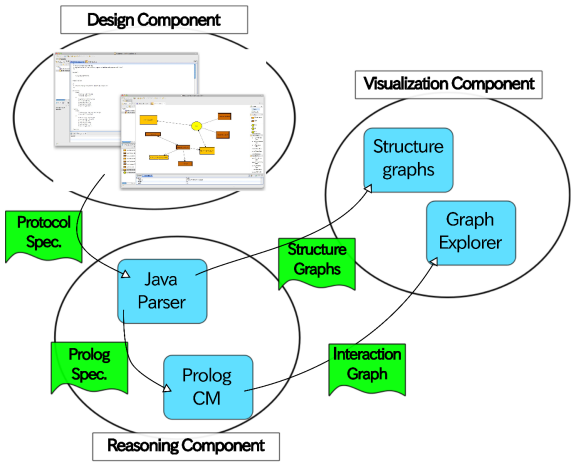


Fig. 1. Components and functionalities supplied by the system.

The proof is by contradiction. It is omitted for lack of space.

5 2CL Tool for Protocol Design and Analysis

Let us now present the tool that we developed based on the technical framework, described in the previous sections. The tool supports the user in two different ways: (i) it features two graphical editors for specifying the protocol actions and the constraints; (ii) it generates different kinds of graphs for supporting the user in the analysis of the possible interactions and in understanding which of them are legal. The system is realized as an Eclipse plug-in, available at the URL `http://di.unito.it/2cl`.

The functionalities that the system supports can be grouped into three components: *design*, *reasoning* and *visualization* (see Figure 1).

Design Component. The design component provides the tools that are necessary for defining the protocol. It supplies two editors: one for the definition of the actions and one for the definition of constraints (Figure 2). The *action definition* editor is basically a text editor, where actions can be specified following the grammar in [4]. The *regulative specification editor* allows the user to graphically define a set of constraints. Constraints are represented by drawing facts, connecting them with 2CL arrows (following the graphical representation of Table 1) or with logical connectives so as to design DNF formulas. The advantage of having a graphical editor is that it supplies a global view of constraints, thus giving the perception of the *flow* imposed by them, without actually specifying any rigid sequence (no-flow-in-flow principle [2]). Figure 2 shows a snapshot of the constraint editor with a representation of the NetBill constraints. On the

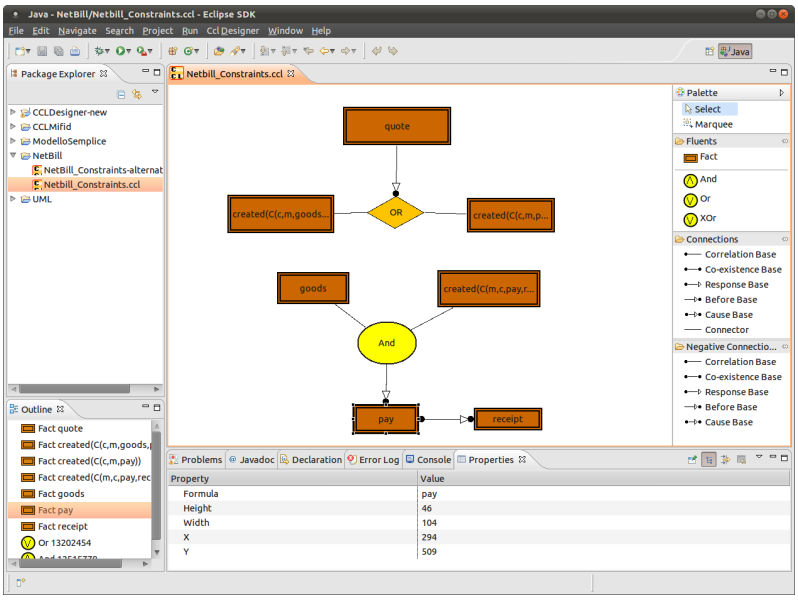


Fig. 2. Editor for constraint specification.

right the user can select the element to introduce in the graph. By editing the properties (bottom of the figure), instead, he/she can specify the name of facts and other graphical aspects.

Reasoning Component. The reasoning component consists of a Java Parser and of the Prolog implementation of the commitment machine described in Section 4. The former generates different kinds of graphs as well as the Prolog program corresponding to the protocol specification. The latter is the input of the Prolog implementation of the commitment machine for the generation of the labelled graph. As explained, the labelled graph represents all the possible interactions where each state is labelled according to the evaluation of the protocol constraints. The graphical conventions is: (i) a state of violation is represented as a red diamond, with an incoming red arrow (e.g. states 54, 57, 108 in Figure 3); (ii) a state in which there is a pending condition is yellow (e.g. states 45, 53, 108); (iii) a state with a single outline, independently from the shape (e.g. 49, 57, 60), is a state that contains unsatisfied commitments; (iv) a state with a double outline, independently from the shape, does not contain active commitments (e.g. 41, 108). Graphical notations can be combined, e.g. a yellow diamond with single outline is a state where there are unsatisfied active commitments, where a constraint is violated and where there is a pending condition (e.g. 53, 57, 114).

Visualization Component. All the graphs produced by the reasoning component can be visualized as images. *Labelled graph*, however, can be explored

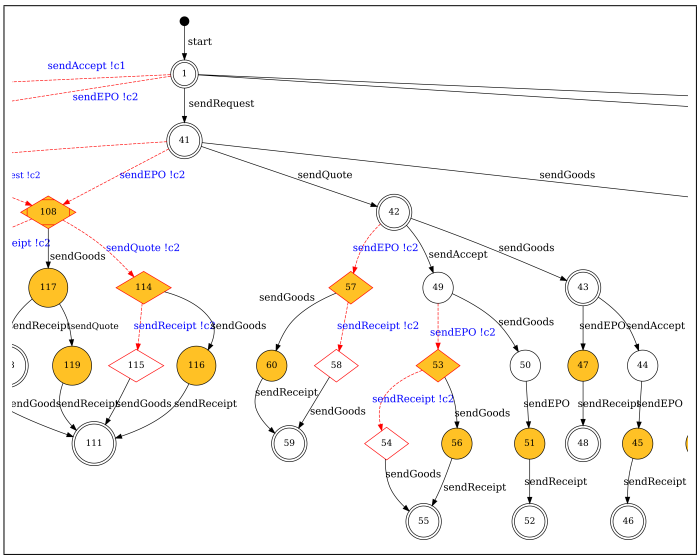


Fig. 3. Part of the labelled Graph for NetBill.

by means of the tool *Graph Explorer*, which is realized in Java and relies on iDot (Incremental Dot Viewer) – an open source project that uses the prefuse[2] visualization framework for Dot graph display. The Graph Explorer supplies different functionalities, like the visualization of the shortest path given a source and a target state, and the visualisation of legal (or illegal) paths only. The user can add or delete a node in a path; search a state starting from its label; and search all the states that contain a certain fact or commitment. Moreover, the tool allows the exploration of the graph one state at a time, by choosing which node to expand. Figure 3 reports part of the labelled graph for NetBill.

Protocol Analysis. The tool can be used as a support in protocol analysis [4]. Particularly interesting is the possibility of exploring the labelled graph by means of the Graph Explorer, which can be used to predict whether performing a certain sequence of actions results in a violation and, in this case, if there is a way to return on a legal path. For what concerns the designer, it is not always easy, when specifying a protocol, to individuate which constraints to introduce but, with the help of the tool, it becomes easy to identify misbehaviours and revise the constraints so as to avoid them. Moreover, a designer can decide, by analysing the graph, to modify the specification so as to regiment some of the patterns expressed as constraints, or to remove some of them. For instance, considering the running example, from Figure 3 it is possible to infer that the protocol does

[2] `http://prefuse.org/`

not allow the customer to pay (*sendEPO*) before the merchant sends the goods. This is due to the constraint CREATED($\mathsf{C}(m, c, pay, receipt)) \wedge goods \rightarrowtriangle pay$. If this behaviour was not in the intention of the designer, he/she can discover it and, e.g., relax the *before* constraint ($\rightarrowtriangle$) transforming it into a *co-existence* ($\bullet\!\!-\!\!\bullet$). If, instead, that is exactly the desired behaviour, one may decide to regiment *sendEPO* so as to enable the payment only after the goods have been sent.

The complete NetBill protocol encoding and the corresponding labelled graph together with further examples, like 2CL specifications of classical agent interaction protocols (CNet) and of real-life protocols (OECD guidelines and MiFID [4]) are available at `http://di.unito.it/2cl` (section Examples).

6 Related Work and Conclusions

This work provides an operational semantics of 2CL protocols [3], based on an extension of the Generalized Commitment Machine [15], and describes a Prolog implementation of this formalization, where the constraint evaluation is performed thanks to state conditions rather than by considering paths. Our aim was to enrich commitment machines with a mechanism for constraint evaluation, in a way that is suitable to creating tools which are useful in application domains. The provided formalization allows the creation of compact and annotated graphs, which provide a global overview of the possible interactions, showing which are legal and which cause constraint (or commitment) violations. The aim was to support an implementation, which enables the verification of exposure to risk on the graph of the possible executions, and taking decisions concerning how to behave or to modify the protocol in order to avoid such a risk. Due to this aim, we decided to base our implementation on [17], rather than on formalizations which support, for instance, model checking. The reason is that this work already is along the same line of ours, the intent being to give a global view on desirable and undesirable states. Winikoff et al. [17], however, propose to cope with undesired paths or undesired final states by adding ad-hoc preconditions to the actions, or by adding active commitments to states that are desired not to be final. This, however, complicates the reuse and the adaptation of the specification to different domains. On the contrary, the proposal in [3] results to be easily adaptable and customizable so as to address different needs of different domains, and it also allows for the specification of more expressive patterns of interaction, given as 2CL constraints.

Concerning model checking, in [6] it is possible to find a proposal of a branching-time logic that extends CTL*, used to give a logical semantics to the operations on commitments. This approach was designed to perform verifications on commitment-protocol ruled interactions by exploiting symbolic model checking techniques. The properties that can be verified are those that are commonly checked in distributed systems: fairness, safety, liveness, and reachability. It would be interesting to integrate in this logical framework the 2CL constraints in order to combine the benefits of both approaches: on the one hand, the possibility to embed in the protocols expressive regulative specification, and, on the

other hand, the possibility to exploit the logical framework to perform the listed verifications.

For what concerns the semantics of commitment protocols, the literature proposes different formalizations. Some approaches present an operational semantics that relies on commitment machines to specify and execute protocols [19, 18, 17]. Some others, like [9], use interaction diagrams, operationally specifying commitments as an abstract data type, and analysing the commitment's life cycle as a trajectory in a suitable space. Further approaches rely on temporal logics to give a formal semantics to commitments and to the protocols defined upon them. Among these, [10] uses DLTL. All these approaches allow the inference of the possible executions of the protocol, but, differently than [3], all of them consider as the only regulative aspect of the protocol the regulative value of the commitments.

References

1. C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.
2. M. Baldoni, C. Baroglio, and E. Marengo. Behavior-Oriented Commitment-based Protocols. In *Proc. of ECAI*, vol. 215 of *Frontiers in Artificial Intelligence and Applications*, pages 137–142. IOS Press, 2010.
3. M. Baldoni, C. Baroglio, E. Marengo, and V . Patti. Constitutive and Regulative Specifications of Commitment Protocols: a Decoupled Approach. *ACM Trans. on Int. Sys. and Tech., Spec. Iss. on Agent Communication*, 2011.
4. M. Baldoni, C. Baroglio, E. Marengo, and V. Patti. Grafting Regulations into Business Protocols: Supporting the Analysis of Risks of Violation. In A. Antón, D. Baumer, T. Breaux, and D. Karagiannis, editors, *Forth International Workshop on Requirements Engineering and Law (RELAW 2011), held in conjunction with the 19th IEEE International Requirements Engineering Conference*, pages 50–59, Trento, Italy, August 30th 2011. IEEE Xplore.
5. A. K. Chopra and M. P. Singh. Constitutive Interoperability. In L. Padgham, D. C. Parkes, J. Müller, and S. Parsons, editors, *Proc. of 7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, vol. 2, pages 797–804, Estoril, Portugal, May 2008. IFAAMAS.
6. M. El-Menshawy, J. Bentahar, and R. Dssouli. Verifiable Semantic Model for Agent Interactions Using Social Commitments. In M. Dastani, A. Seghrouchni El Fallah, J. Leite, and P. Torroni, editors, *LAnguages, methodologies and Development tools for multi-agent systemS (LADS 2009)*, LNCS 6039, pages 128–152, Torino, Italy, September 2010. Springer.
7. E. A. Emerson. *Temporal and Modal Logic*, volume B. Elsevier, Amsterdam, The Netherlands, 1990.
8. N. Fornara and M. Colombetti. Defining Interaction Protocols using a Commitment-based Agent Communication Language. In J. S. Rosenschein, T. Sandholm, M. Wooldridge, and M. Yokoo, editors, *Proc. of the Second International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS 2003)*, pages 520–527, Melbourne, Australia, July 2003. ACM.
9. N. Fornara and M. Colombetti. A Commitment-Based Approach To Agent Communication. *Applied Artificial Intelligence*, 18(9-10):853–866, 2004.

10. L. Giordano, A. Martelli, and C. Schwind. Specifying and Verifying Interaction Protocols in a Temporal Action Logic. *Journal of Applied Logic*, 5(2):214–234, 2007.
11. A. J. I. Jones and M. Sergot. *On the Characterization of Law and Computer Systems: the Normative Systems Perspective*, pages 275–307. John Wiley & Sons, Inc., New York, NY, USA, 1994.
12. E. Marengo, M. Baldoni, C. Baroglio, A. K. Chopra, V. Patti, and M. P. Singh. Commitments with Regulations: Reasoning about Safety and Control in REGULA. In L. Sonenberg, P. Stone, K. Tumer, and P. Yolum, editors, *AAMAS*, vol. 1–3, pages 467–474, Taipei, Taiwan, May 2011. IFAAMAS.
13. J.R. Searle. *The construction of social reality.* Free Press, New York, 1995.
14. M. P. Singh. An Ontology for Commitments in Multiagent Systems. *Artificial Intelligence and Law*, 7(1):97–113, 1999.
15. M. P. Singh. Formalizing Communication Protocols for Multiagent Systems. In M. M. Veloso, editor, *IJCAI*, pages 1519–1524, Hyderabad, India, January 2007. AAAI Press.
16. G. Weiss, editor. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. The MIT Press, 1999.
17. M. Winikoff, W. Liu, and J. Harland. Enhancing Commitment Machines. In J. A. Leite, A. Omicini, P. Torroni, and P. Yolum, editors, *Proc. of the Second International Workshop on Declarative Agent Languages and Technologies II (DALT 2004)*, LNCS 3476, pages 198–220, New York, NY, USA, July 2005. Springer.
18. P. Yolum and M. P. Singh. Designing and Executing Protocols Using the Event Calculus. In *Agents*, pages 27–28, New York, NY, USA, 2001. ACM.
19. P. Yolum and M. P. Singh. Commitment Machines. In J.-J. Ch. Meyer and M. Tambe, editors, *Proc. of the 8th International Workshop on Intelligent Agents VIII (ATAL 2001)*, LNCS 2333, pages 235–247, Seattle, WA, USA, August 2002. Springer.

Solving Fuzzy Distributed CSPs: An Approach with Naming Games*

Stefano Bistarelli[1,2], Giorgio Gosti[3] and Francesco Santini[1,4**]

[1] Dipartimento di Matematica e Informatica, Università di Perugia
[bista,francesco.santini]@dmi.unipg.it
[2] Istituto di Informatica e Telematica (CNR), Pisa, Italy
stefano.bistarelli@iit.cnr.it
[3] Institute for Mathematical Behavioral Sciences, University Of California, Irvine, USA
ggosti@uci.edu
[4] Centrum Wiskunde & Informatica, Amsterdam, Netherlands
F.Santini@cwi.nl

Abstract. Constraint Satisfaction Problems (CSPs) are the formalization of a large range of problems that emerge in computer science. The solving methodology described here is based on the *Naming Game* (*NG*). The NG was introduced to represent N agents that have to bootstrap an agreement on a name to give to an object (*i.e.*, a word). In this paper we focus on solving Fuzzy NGs and Fuzzy Distributed CSPs (Fuzzy DCSPs) with an algorithm for NGs. In this framework, each potential answer evaluated by the agents is associated with a preference represented as a fuzzy score. The solution is the answer associated with the highest preference value. The two main features that distinguish this methodology from Fuzzy DCSP methods are that the system can react to small instance changes and and it does not require pre-agreed agent/variable ordering.

1 Introduction

This paper presents a distributed method to solve Fuzzy Constraint Satisfaction Problems (CSPs) [12, 16, 9, 10, 15] that comes from a generalization of the Naming Game (NG) model [13, 1, 11, 8]. In Fuzzy Distributed CSP (DCSP) protocols, the aim is to design a distributed architecture of processors, or more generally a group of agents, who cooperate to solve a Fuzzy CSP instantiation. Fuzzy CSP

* Research partially supported by MIUR PRIN 20089M932N project: "Innovative and multi-disciplinary approaches for constraint and preference reasoning", by CCOS FLOSS project "Software open source per la gestione dell'epigrafia dei corpus di lingue antiche", and by INDAM GNCS project "Fairness, Equità e Linguaggi".

** This work was carried out during the tenure of an ERCIM "Alain Bensoussan" Fellowship Programme. This Programme is supported by the Marie Curie Co-funding of Regional, National and International Programmes (COFUND) of the European Commission.

overcame the limitations of crisp CSP, because they allow constraints to be partially satisfied. This in turn allows us to model real-word situations in which at the aggregate level we may have only partial agreement solutions. Moreover, in a multi-agent environment, Fuzzy CSPs allow us to represent individual preferences, that most often are not strictly binary, but are better represented by a continuous variable which characterizes a degree of satisfaction.

In our approach, we see Fuzzy DCSP protocol as a dynamic system and we select the stable states of the system as the solutions to our CSP. To do this we design each agent so that it will move towards a stable local state. This system may be called "self-stabilizing" whenever the global stable state is obtained through the reinforcement of the local stable states [7]. When the system finds the stable state, the DCSP instantiation is solved. A protocol designed in this way is resistant to damage and external threats because it can react to changes in the problem instance. Moreover, in our approach all agents have equal chance to reveal private information, for this reason this algorithm is unbiased ("fair') with respect to privacy. These characteristics make this protocol an ideal service to coordinate cooperation among a party of independent clients such as banks, private companies, or local institutions. Because it is unbiased with respect to privacy and it allows the clients to be peers who are not forced into a pre-agreed agent/variable ordering.

The NG describes a set of problems in which a number of agents bootstrap a commonly agreed name for one or more objects. In this paper we discuss a NG generalization in which agents have individual fuzzy preferences over words. This is a natural generalization of the NG, because it models the endogenous agents's preferences and attitudes towards certain object naming system. This preferences may be driven by pragmatic or rational reasons: some words may be associated to other objects, some words may be too long or too complex, or other words may be easy to confuse. To model the agents preferences we add individual fuzzy preference levels to each word in the agents' domain. A NG can be viewed as a particular crisp CSP instance [3, 4].

When we add preference levels, the new game may be interpreted as an optimization problem. To represent the Fuzzy NG instance as a particular instance of a Fuzzy DCSP, we impose that only the solutions that optimize the Fuzzy DCSP are the ones in which all the agents connected by a communication edge share the same word as a naming. More specifically, we use fuzzy unary constraints to represent the preferences over words and crisp binary constraints that prevent the two variables of the scope to be assigned to different values (in our case, different names). This forces all the optimal solutions to be the states in which all variables have the same assignment (name). The resulting Fuzzy NG algorithm solves DCSPs with fuzzy unary constraints and crisp binary constraints.

In the algorithm we have an asymmetric interaction, in which one agent is the speaker and the other agents involved are listeners. To let this interaction occur our algorithm uses a central scheduler that randomly draws a speaker at each turn. This may be interpreted as a "central blind orchestrator" scheme; this central scheduler has no information on the DCSP instance, and has no

pre-determined agent/variable ordering (the algorithm "fairness" is preserved). Nonetheless, the "central blind orchestrator" can be thought of as an idealization of a true asynchronous system which allows us to analyze some of the interesting complexities of a distributed system without the complications introduced by asynchronism. [4] shows how a similar algorithm can be extended to an asynchronous system.

As a further novelty, in Sec. 5 we extend the algorithm to solve Fuzzy NG in order to solve a generic instance of a Fuzzy DCSP, that is a DCSP problem where both unary and binary constraints are associated with a fuzzy preference. This kind of distributed Fuzzy DSCP can be applied to deal with resource allocation, collaborative scheduling and distributed negotiation [9].

The paper extends [5] by refining the distributed algorithm and by providing a detailed description of a run of the algorithm. The paper is organized as follows: in Sec. 2 we present the background on NGs and Fuzzy DSCPs, while Sec. 3 summarizes the related work. Sec. 4 presents an algorithm that solves Fuzzy NGs. Sec. 5 extends the algorithm in Sec. 4 in order to solve generic Fuzzy DCSPs and discusses a simple example of a algorithm run. Sec. 6 presents the tests and the results for the Fuzzy NG algorithm. Lastly, Sec. 7 reports the conclusions and ideas about future work.

2 Background

2.1 Distributed Constraint Satisfaction Problem (DCSP)

A classical constraint can be seen as the set of value combinations for the variables in its scope that satisfy the constraint. In the fuzzy framework, a constraint is no longer a set, but rather a fuzzy set [12]. Therefore, for each assignment of values to its variables, we allow a gradual assessment of how much it belongs to the set, not whether it belongs to the set. This allows us to represent the fact that a combination of values for the variables of the constraint is partially permitted.

A Fuzzy CSP is defined as a triple $P = \langle X, D, C \rangle$ where, as in classical CSPs, X and D are the set of variables and their domain, and C is a set of fuzzy constraints. We suppose a single domain for all the variables. A fuzzy constraint is defined as a function c_V on a sequence of variables V, which is called the *scope* (or *support*) of the constraint. The scope is the set of variables on which the constraint is defined.

$$c_V : \prod_{x_i \in V} D_i \rightarrow [0, 1]$$

The function c_V indicates to what extent an assignment of the variables in V satisfies the constraint [12]. In fuzzy constraints, 1 usually corresponds to the best preference, and 0 to the worst preference value. The combination $c_V \otimes c_W$ of two fuzzy constraints c_V and c_W is a new fuzzy constraint $c_{V \cup W}$ defined as

$$c_{V \cup W}(\eta) = min(c_V(\eta), c_W(\eta))$$

where η is a complete assignment of the variables in the problem, *i.e.*, an assignment of the variables in X:

$$\eta \in \prod_{x_i \in X} D_i$$

If $c_1\eta > c_2\eta$ (*e.g.*, $c_1\eta = 0.8$ and $c_2\eta = 0.4$), it means that the assignment η satisfies c_1 better than c_2. In the rest of the paper we will use the expression $c\eta[x_i := d]$ to denote a constraint assignment with variable $x_i \in X$ assigned to value $d \in D$.

We can now define the preference of the complete set C of constants in the problem, by performing a combination of all the fuzzy constraints. Given any complete assignment η we have

$$(\bigotimes_{c_V \in C} c_V)(\eta) = \min_{c_V \in C} c_V(\eta)$$

Thus, the optimal solutions of a fuzzy CSP are the complete assignments whose satisfaction degree is maximum over all complete assignments, that is,

$$OptSol(P) = \{\eta \mid \max_{\eta} \min_{c_V \in C} \ c_V(\eta)\}$$

In DCSPs [16, 12], the main difference to a classical CSP is that each variable is controlled by a corresponding agent, meaning that this agent sets the variables value. Formally, a DCSP is a tuple $\langle X, D, C, A\rangle$, *i.e.*, a CSP with a set A of n agents. We suppose the number of variables m to be greater/equal than the number of agents n, *i.e.*, $m \geq n$. When an agent controls more than one variable, this would be modeled by a single variable whose values are combinations of values of the original variable. It is further assumed that an agent knows the domain of its variable and all constraints involving the variable, and that it can reliably communicate with all other agents. The main challenge is to develop distributed algorithms that solve the CSP by exchanging messages among the agents.

2.2 Introduction to Naming Games

The NGs [13, 1, 11, 8] describe a set of problems in which a number of agents bootstrap a commonly agreed name for one or more objects. The game is played by a population of n agents which play pairwise interactions in order to negotiate conventions, *i.e.*, associations between forms and meanings, and it is able to describe the emergence of a global consensus among them. For the sake of simplicity the model does not take into account the possibility of homonyms, so that all meanings are independent and one can work with only one of them, without loss of generality. An example of such a game is that of a population that has to reach the consensus on the name (*i.e.*, the form) to assign to an object (*i.e.*, the meaning) exploiting only local interactions. As we clarify later,

this model is appropriate to address all those situations in which negotiation rules a decision process (*i.e.*, opinion dynamics, *etc.*) [1].

Each NG is defined by an interaction protocol. There are two important aspects of the NG. First, the agents randomly interact and use a simple set of rules to update their state. Second, the agents converge to a consistent state in which all the objects of the set have a uniquely assigned name, by using a distributed social strategy. Generally, two agents are randomly extracted at each turn to perform the role of the speaker and the listener (or hearer as used in [13, 1]). The interaction between the speaker and the listener determines the agents' update of their internal state. DCSPs and NGs share a variety of common features [3, 4].

2.3 Self-Stabilizing Algorithms

The definition of *Self-stabilizing algorithm* in distributed computing was first introduced by [7]. A system is *self-stabilizing* whenever, each system configuration associated with a *solution* is an absorbing state (global stable state), and any initial state of the system is in the basin of attraction of at least one *solution*.

In a self-stabilizing algorithm, we program the agents of our distributed system to interact with their neighbors. The agents update their state through these interactions by trying to find a stable state in their neighborhood. Since the algorithm is distributed, many legal configurations of the agents' states and their neighbors' states start arising sparsely. Not all of these configurations are mutually compatible, and so they form mutually inconsistent potential cliques. The self-stabilizing algorithm must find a way to make the global legal state emerge from the competition between these potential cliques. Dijkstra [7] and Collin [6] suggest that an algorithm designed in this way can not always converge, and a special agent is needed to break the system symmetry. In this paper, we show a different strategy based on the concept of random behavior and probabilistic transition function, which we discuss in Sec. 4.3.

3 Related Work

This paper extends the results of [3, 4], in which the authors discuss how to solve (non fuzzy) DCSP with NGs. Despite the fact that a number of approaches have been proposed to solve DCSPs [12, 16] or centralized FCSP [12] alone, there is less work related to the combination of DCSPs and Fuzzy CSPs.

It is important to notice the fundamental difference with the DCSP algorithms designed by Yokoo [16]. Yokoo addresses three fundamental kinds of DCSP algorithms: *Asynchronous Backtracking*, *Asynchronous Weak-commitment Search* and *Distributed Breakout Algorithm* [16]. Although these algorithms share the property of being asynchronous, they require a pre-agreed agent/variable ordering. The algorithm presented in this paper does not need this initial condition.

Fuzzy DCSPs has been of interest to the Multi-Agent System community, especially in the context of distributed resource allocation, collaborative scheduling, and negotiation (*e.g.*, [9]). In these contexts, the work focuses on bilateral

negotiations; where a central coordinating agent may be required when many agents take part.

For example, the work in [9] promotes a rotating coordinating agent which acts as a central point to evaluate different proposals sent by other agents. Thus the network model employed is not totally distributed. Another important note is that this work focuses on competitive negotiation where agents try to outsmart each other as opposed to collaborative negotiation. Therefore, it does not use techniques from DCSP algorithms.

In [14, 15] the authors define the fuzzy GENET model for solving binary FCSPs. Fuzzy GENET is a neural network model for solving binary FCSPs. Through transforming FCSPs into $[0, 1]$ integer programming problems, they display an equivalence between the underlying working mechanism of fuzzy GENET and the discrete Lagrangian method. Benchmarking results confirm its feasibility in tackling CSPs and flexibility in dealing with over-constrained problems.

In [10] the authors propose two approaches to solve these problems: An iterative method and an adaptation of the *Asynchronous Distributed constraint OPTimization* algorithm (*ADOPT*) for solving Fuzzy DCSPs. They also present experiments on the performance comparison of the two approaches, showing that ADOPT is more suitable for low density problems (density = num of links / number of agents).

4 An Algorithm for Fuzzy Naming Games

In this section we extend the NGs to take into account Fuzzy scores associated with words. Therefore, we propose an algorithm that solves Fuzzy NGs. Since we deal with fuzzy values associated only with words, we can consider the Fuzzy NG as a particular instance of a Fuzzy DCSP $P = \langle X, D, C, A\rangle$ (see Sec. 2.1). We assume that each agent a_i controls a variable $x_i \in X$, and searches its domain $d_i \in D$ for a name convention that optimizes P. Each agent a_i has a fuzzy unary constraints $c_i \in C$ which describes its preferences over the competing names, and controls all the binary constraint $c_{i,j} \in C$ that act on its variable x_i. In the Fuzzy NG, we restrict C to constraints that are satisfied only if the words chosen from a_i and a_j are the same (*i.e.*, $x_i = x_j$). Therefore, the solution of the Fuzzy NG is a naming convention that maximizes the individual preferences of the agents. In Sec. 5 we extend the algorithm in order to solve fuzzy binary constraints among agents, and consequently, to solve all Fuzzy DCSPs.

The algorithm is based on two entities. The first is the *speaker*, who communicates by broadcasting a word he considers a possible solution and the related fuzzy preference. The second, is a set of *listeners*, who are all the neighboring agents. We define the neighbors as those agents who share a binary constraint with the speaker. At each turn t, an agent is drawn with uniform probability to be the speaker. In the following paragraph we describe in detail each step of the interaction scheme that defines the behavior between the speaker and the listeners: we consider three phases, *i) broadcast*, *ii) feedback* and *iii) update*. Each agent marks the element that it expects to be the final shared name.

4.1 Interaction Protocol

Broadcast The speaker s executes the broadcast protocol. The speaker computes $top = \{x_s | x_s = \arg\max_{x_s} \bigotimes c_{\{s\}}\eta[s := x]\}$, and checks if the marked variable assignment b is in *top*. If the marked variable assignment is not in *top* it selects a new variable assignment b with uniform probability from *top*, and marks it. Then it sends the couple $(b, \max_{b \in D_s}(\bigotimes c_{\{s\}}\eta[s := b]))$ to all its neighboring listeners. $c_{\{s\}}$ is the set of all the constraints whose scope contains the variable s.

Feedback All the listeners receive the broadcast message (b, u) from the speaker. Each listener l computes $\forall d_k$, the aggregate preference $v_k = \bigotimes c_{\{s,l\}}\eta[s := b][l := d_k]$, that is it computes the combination of the fuzzy preferences (*i.e.*, v_k) for each d_k assignment, supposing that s chooses word b. $c_{\{s,l\}}$ is the set of all the constraints whose scope contains the variable s or l. Each listener sends back to s a feedback message according to the following two cases:

- *Failure*. If $u > \max_k(v_k)$ there is a *failure*, and the listener feedbacks a failure message containing the maximum value and the corresponding assignment for l, **Fail(**$\max_k(v_k)$**)**.
- *Success*. If $u \leq \max_k(v_k)$, there is a *success*, the listener feedbacks **Succ**, and marks the assigment b.

Update The overall listeners' feedback determines the update of the listeners and of the speaker. When a listeners feedback a **Succ**, then it lowers the preference level for all its v_k with a preference value higher then the speaker's preference level: $\forall v_k.v_k > u$ and it sets $v_k = u$. If the speaker receives only **Succ** feedback messages from all its listeners, then it does not need to update.

Otherwise, that is if the speaker receives a number of **Fail(**v_j**)** feedback messages from h listeners (with $h \geq 1$ and $1 \leq j \leq h$), then it selects the worst v_w fuzzy preference such that $\forall j, v_w \leq v_j$, and it sends to all listeners a **FailUpdate(**v_w**)**. Thus, the speaker sets its preference for the assignment b to v_w, *i.e.*, $c_{\{s\}}\eta[s := b] = v_w$. In addition, each listener l sets $v_l = v_w$, *i.e.*, $c_{\{s,l\}}[l := d_l] = v_w$.

4.2 A Simple Execution

In this section, we describe a run of the algorithm on a Fuzzy NG instance. We consider a problem with three agents (X_1, X_2, and X_3) that try to come to an agreement on the naming convention A or B, and the unary/binary constraints as depicted in Fig. 1.

At $t = 1$ (see Fig. 5b), X_2 is the first agent to speak. It computes the elements with the highest preference over the constraints $c_{\{s\}}$ and puts them in *top*. $\forall b \in D_s = \{A, B\}$ it computes $\bigotimes c_{\{s\}}\eta$, it finds that $\bigotimes c_{\{s\}}\eta[s := A] = 0.8$,

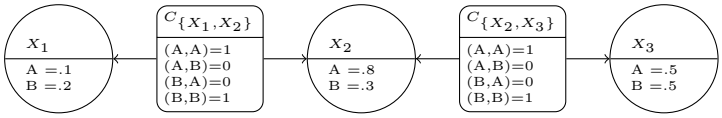


Fig. 1: Fuzzy NG example at time $t = 0$: the initial state of the problem.

and $\bigotimes c_{\{s\}}\eta[s := B] = 0.3$. Thus, $top = \{A\}$. Because this is the first interaction, the agent has no marked element, thus it draws the only element in *top*. Thus, it marks A, and chooses to broadcast $(A, 0.8)$ (Fig. 2).

Listener X_1 computes $v_k = \bigotimes c_{\{s\}}\eta[s := A][l := d_k]$. For $d_1 = A$ it finds $v_1 = \bigotimes c_{\{s\}}\eta[s := A][l := A] = 0.1$, and for $d_2 = B$ it finds $v_2 = \bigotimes c_{\{s\}}\eta[s := A][l := B] = 0$. Thus, it returns **Fail(0.1)**. Simultaneously, listener X_3 computes $v_k = \bigotimes c_{\{s\}}\eta[s := b][l := d_k]$. For $d_1 = A$ it finds $v_1 = \bigotimes c_{\{s\}}\eta[s := A][l := A] = 0.5$, and for $d_2 = B$ it finds $v_2 = \bigotimes c_{\{s\}}\eta[s := A][l := B] = 0$. Thus, it returns **Fail(0.5)**. In the update phase the listeners X_1 and X_3 change the preference levels of the $v_k > 0.2$ to $v_k = 0.2$ (the changed values are colored in blue in Fig. 2). Since X_2 receives a failure feedback, it calls **FailUpdate(0.1)**. Then, the speaker update its preference level, $A = 0.1$ (Fig. 2), and the listeners X_1 and X_3 change the preference levels $v_k = 0.1$ (Fig. 2).

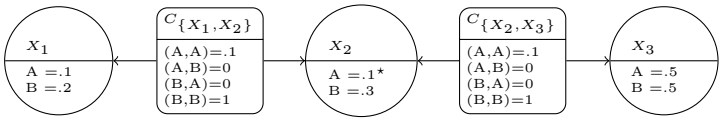


Fig. 2: Fuzzy NG example at time $t = 1$.

At $t = 2$ (see Fig. 5b), X_1 is the second agent to speak. It finds that $\bigotimes c_{\{s\}}\eta[s := A] = 0.1$, and $\bigotimes c_{\{s\}}\eta[s := B] = 0.3$. Thus, $top = \{B\}$. Thus, it marks B, and chooses to broadcast $(B, 0.3)$ (Fig. 3).

Listener X_2 computes $v_k = \bigotimes c_{\{s\}}\eta[s := B][l := d_k]$. For $d_1 = A$ it finds $v_1 = \bigotimes c_{\{s\}}\eta[s := B][l := A] = 0$, and for $d_2 = B$ it finds $v_2 = \bigotimes c_{\{s\}}\eta[s := B][l := B] = 0.2$. Thus, it returns **Succ**. In the update phase the listeners X_2 change the preference levels of the $v_k > 0.2$ to $v_k = 0.2$ (Fig. 3). Then, the listeners X_2 change the preference levels $v_k = 0.2$ (Fig. 3).

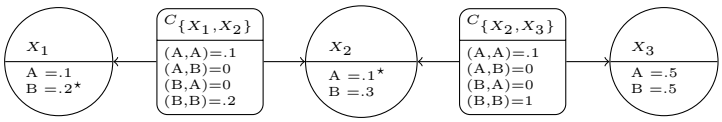


Fig. 3: Fuzzy NG example at time $t = 2$.

At time $t = 3$, X_3 is selected to be a speaker, and chooses to broadcast B. The listener X_2 feedbacks success and adjusts $v_2 = \bigotimes c_{\{s\}}\eta[s := B][l := B] = 0.2$. At time $t = 4$, X_2 is selected to be a speaker, and chooses to broadcast B. X_1 and X_2 feedback success. From now on all interactions are successful, and the agents agree on the convention B (Fig. 4).

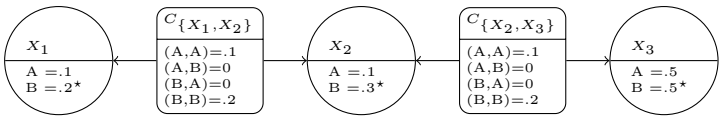


Fig. 4: Fuzzy NG example at time $t \geq 4$.

4.3 Theorems

In this Section we show the lemmas and theorems that lead to the convergence property of the algorithm in Sec. 4.1. We formally prove that the algorithm always terminates with the best solution, *i.e.*, the word with the highest fuzzy preference. With Lemma 1 we state that a subset of constraints $C' \subseteq C$ has a higher fuzzy preference w.r.t. C. We say that a fuzzy constraint problem is α-consistent if it can be solved with a level of satisfiability of at least α (see also [2]).

Lemma 1 ([2]). *Consider a set of constraints C and any subset C' of C. Then we have $\bigotimes C \leq \bigotimes C'$.*

The speaker selection rule defines a probability distribution function F that tells us the probability that a certain domain assignment is selected. $\bigotimes c_{\{s\}}\eta[s := b]$) and the marked word determine F. In Lemma 2 we relate F and convergence of the algorithm with probability 1, to the level of satisfiability of the problem.

Lemma 2. *If the F function selects only the domain elements with preference level larger then α, and the algorithm converges with probability 1, then it converges only to solutions such that $Sol(P) \geq \alpha$.*

From [3, 4], if the F function chooses a random element in the word domain, then the algorithm converges to the same word, but this word could not be the optimal one, *i.e.*, the word with the highest fuzzy preference. If we choose F in order to select only words with a preference greater than α, then the algorithm converges to a solution with a global preference greater than α.

With Prop. 1 and Prop. 2 we prepare the background for the main theorem of this section, *i.e.*, Th. 1. Proposition 1 shows the stabilization of the algorithm after some time, while Prop. 2 states that the algorithm converges with a probability of 1.

Proposition 1. *For time $t \to +\infty$, the weight associated to the optimal solution is equal for all the agents, and it is equal to the minimum preference level of that word.*

Proposition 2. *For any probability distribution F such that the system is an absorbing homogeneous Markov process the algorithm converges with a probability of* 1.

At last, we state that the presented algorithm always converge to the best solution of the Fuzzy DCSP.

Theorem 1. *The algorithm described in Sec. 4.1 is an absorbing homogeneous Markov process thus it always converges to the best solution of the represented Fuzzy NG,* i.e.*,, to the solution with the highest preference possible.*

The proof comes from the fact that, *i)* according to Prop. 2, the algorithm always converges, and *ii)* we choose a proper function F as described in Lem. 2.

5 Solving Fuzzy Distributed Constraint Satisfaction Problems with Naming Games

In this section, we propose an generalization of the Fuzzy NG algorithm given in Sec. 4 that intended to solve Fuzzy DCSPs in general. As proposed in [16], we assign to each variable $x_i \in X$ of the $P = \langle X, D, C, A \rangle$, an agent $a_i \in A$. We assume that each agent knows all the constraints that act over its variables [16]. Each agent $i = 1, 2, \ldots, n$ (where $|A| = n$) searches its own variable domain $d_i \in D$ for its variable assignment that optimizes P. The degree of satisfaction of a fuzzy constraint tells us to what extent it is satisfied (see Sec. 2.1). Otherwise stated, the goal of the game is to make the agents find an assignment of their variables that maximizes the overall fuzzy score result for the problem; fuzzy preferences of constraints are combined with *min* function (see Sec. 2.1).

In our algorithm we solve unary and binary constraints only, since we know from literature that any CSP can be translated to an equivalent one adopting only unary/binary constraints [12] . Each agent has a unary constraint c_i with support defined over its variable $x_i \in X$; these unary constraints represent the local preference of the agents a_i for each variable assignment $d_i \in D$, $c_{a_i}\eta[a_i := d_i]$. Each agent interacts only with its neighbors, we define neighbors as any two agents that share a constraint $c_{i,j} \in C$. Any binary constraint $c_{i,j}$ returns a preference value $p \in [0, 1]$ that states the combined preference over the assignment of x_i and x_j together.

At each turn t, an agent is drawn with uniform probability to be the speaker. As illustrated in Sec. 4 we have a speaker s and a set of listeners l_j, each of them sharing a binary constraint with s. The phases of the algorithm are the same three as in Sec. 4: *i) broadcast*, *ii) feedback* and *iii) update*. The two main features that distinguish this methodology from Fuzzy DCSPs methods are: the system can react to small instance changes, and it does not require pre-agreed agent/variable ordering [3, 4].

5.1 Interaction Protocol

Broadcast The speaker s executes the broadcast protocol. The speaker computes $top = \{x_s | x_s = \max_{x_s} \bigotimes c_{\{s\}}\eta[s := x]\}$, and checks if the marked variable assignment b is in top. If the marked variable assignment is not in top it selects a new variable assignment b with uniform probability from top, and marks it. Then it sends the couple $(b, \max_{b \in D_s}(\bigotimes c_{\{s\}}\eta[s := b]))$ to all its neighboring listeners. $c_{\{s\}}$ is the set of all the constraints whose scope contains the variable s.

Feedback All the listeners receive the broadcast message (b, u) from the speaker. Each listener l computes $\forall d_k$, $v_k = \bigotimes c_{\{s,l\}}\eta[s := b][l := d_k]$, that is it computes the combination of the fuzzy preferences (*i.e.*, v_k) for each d_k word, supposing that s chooses word b. $c_{\{s,l\}}$ is the set of all the constraints whose scope contains the variable s or l. Each listener sends back to s a feedback message according to the following two cases:

- *Failure*. If $u > \max_k(v_k)$ there is a *failure*, and the listener feedbacks the failure message **Fail(**$\max_k(v_k)$**)**.
- *Success*. If $u \leq \max_k(v_k)$, there is a *success*, the listener feedbacks **Succ**.

Update As in Sec. 4.1, the overall listeners' feedback determines the update of the listener and of the speaker. When a listener feedbacks a **Succ**, it also lowers the preference level for all its v_k with a preference value higher than the speaker's preference level u: $\forall v_k$, $v_k > u$ and it sets $v_k = u$. If the speaker receives only **Succ** feedback messages from all its listeners, then it does not need to update.

Otherwise, that is, if the speaker receives a number of **Fail(**$v_j, l_j = d_j$**)** feedback messages from h listeners (with $h \geq 1$ and $1 \leq j \leq h$), then it selects the worst v_w fuzzy preference such that $\forall j, v_w \leq v_j$. Then it sends to all listeners a **FailUpdate(**v_w**)**. Thus, the speaker sets its assignment to b with the worst fuzzy preference level among the failure feedback messages of the listeners, *i.e.*, $c_{\{s\}}\eta[s := b] = v_w$. In addition, each listener l sets $v_l = v_w$, *i.e.*, $c_{\{s,l\}}\eta[s := b][l := d_l] = v_w$.

5.2 A Simple Algorithm Execution

In this section we provide a simple run of the algorithm. We consider a problem with three agents (X_1, X_2 and X_3) and the unary/binary constraints as depicted in Fig. 5a.

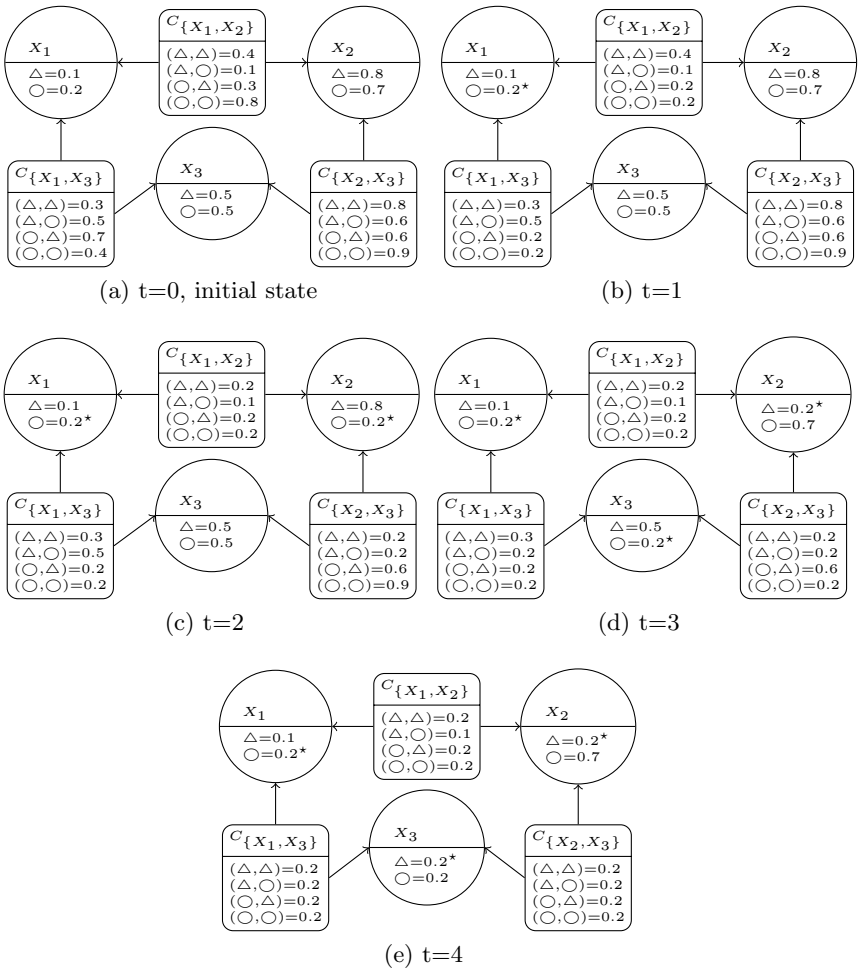


Fig. 5: Example Fuzzy DCSP problem.

At $t = 1$ (see Fig. 5b), X_1 is the first agent to speak. It computes the elements with the highest preference over the constraints $c_{\{s\}}$ and puts them in *top*. It finds that $\bigotimes c_{\{s\}}\eta[s := \triangle] = 0.1$, and $\bigotimes c_{\{s\}}\eta[s := \bigcirc] = 0.2$. Thus, $top = \{\bigcirc\}$, it marks $\bigcirc$, and chooses to broadcast $(\bigcirc, 0.2)$ (Fig. 5b), and marks it.

Listener X_2 finds $v_1 = \bigotimes c_{\{s\}}\eta[s := \bigcirc][l := \triangle] = 0.2$, and $v_2 = \bigotimes c_{\{s\}}\eta[s := \bigcirc][l := \bigcirc] = 0.2$. Thus, it returns **Succ**. Simultaneously, listener X_3 finds $v_1 = \bigotimes c_{\{s\}}\eta[s := \bigcirc][l := \triangle] = 0.2$, and $v_2 = \bigotimes c_{\{s\}}\eta[s := \bigcirc][l := \bigcirc] = 0.2$. Thus, it returns **Succ**. In the update phase the listeners X_2 and X_3 change the preference levels of the $v_k > 0.2$ to $v_k = 0.2$ (Fig. 5b).

At $t = 2$ (see Fig. 5c), X_2 is the second agent to speak. It finds that $\bigotimes c_{\{s\}}\eta[s := \triangle] = 0.4$, and $\bigotimes c_{\{s\}}\eta[s := \bigcirc] = 0.2$. Thus, it sends $(\triangle, 0.4)$

(Fig. 5c), and marks it. Listener X_1 finds $d_1 = \bigotimes c_{\{s\}}\eta[s := \triangle][l := \triangle] = 0.1$, and $d_2 = \bigotimes c_{\{s\}}\eta[s := \triangle][l := \bigcirc] = 0.2$. Thus, it returns **Fail(0.2)**. Simultaneously, listener X_3 finds $d_1 = \bigotimes c_{\{s\}}\eta[s := b][l := d_k] = 0.4$, and $d_2 = \bigotimes c_{\{s\}}\eta[s := b][l := d_k] = 0.4$. Thus, it returns **Succ**. Since X_2 receives a failure feedback, it calls **FailUpdate(0.2)**. Then, the speaker update its preference level, $\triangle = 0.2$. The listeners X_1 and X_2 change their preference levels $v_k = 0.2$ (5c).

At $t = 3$ (see Fig. 5d), X_3 is the third agent to speak. It finds that $\bigotimes c_{\{s\}}\eta[s := \triangle] = 0.3$, and $\bigotimes c_{\{s\}}\eta[s := \bigcirc] = 0.5$. Thus, it sends $(\bigcirc, 0.5)$, and marks it. Listener X_1 finds $v_1 = \bigotimes c_{\{s\}}\eta[s := \bigcirc][l := \triangle] = 0.1$, and $v_2 = \bigotimes c_{\{s\}}\eta[s := \bigcirc][l := \bigcirc] = 0.2$. Thus, it returns **Fail(0.2)**. Listener X_3 finds $v_1 = \bigotimes c_{\{s\}}\eta[s := \bigcirc][l := \triangle] = 0.2$, and $v_2 = \bigotimes c_{\{s\}}\eta[s := \bigcirc][l := \bigcirc] = 0.7$. Thus, it returns **Succ**. Since X_2 receives a failure feedback, it calls **FailUpdate(0.2)**. Then, the speaker update its preference level, $\triangle = 0.2$ (Fig. 5d), and the listeners X_1 and X_2 change the preference levels $v_k = 0.2$ (Fig. 5d).

At $t = 4$ (see Fig. 5e), X_3 is the fourth agent to speak. It finds that $\bigotimes c_{\{s\}}\eta[s := \triangle] = 0.3$, and $\bigotimes c_{\{s\}}\eta[s := \bigcirc] = 0.2$. Thus, it marks $(\triangle, 0.3)$ and chooses to broadcast it. Listener X_1 finds $v_1 = \bigotimes c_{\{s\}}\eta[s := \triangle][l := \triangle] = 0.1$, and $v_2 = \bigotimes c_{\{s\}}\eta[s := \triangle][l := \bigcirc] = 0.2$. Thus, it returns **Fail(0.2)**. Listener X_2 finds $v_1 = \bigotimes c_{\{s\}}\eta[s := \triangle][l := \triangle] = 0.2$, and $v_2 = \bigotimes c_{\{s\}}\eta[s := \triangle][l := \bigcirc] = 0.6$. Thus, it returns **Succ**. Since X_3 receives a failure feedback, it calls **FailUpdate(0.2)**. Then, the speaker update its preference level, $\triangle = 0.2$ (green in Fig. 5e), and the listeners X_1 and X_2 change the preference levels $v_k = 0.2$ (blue in Fig. 5e).

At $t = 5$, X_2 is the fifth agent to speak. It finds that $\bigotimes c_{\{s\}}\eta[s := \triangle] = 0.2$, and $\bigotimes c_{\{s\}}\eta[s := \bigcirc] = 0.2$. Since $\triangle$ is marked, the agent chooses to broadcast it. Listener X_1 finds $v_1 = \bigotimes c_{\{s\}}\eta[s := b][l := d_k] = 0.1$, and $v_2 = \bigotimes c_{\{s\}}\eta[s := b][l := d_k] = 0.2$. Thus, it returns **Succ**. Listener X_3 finds $v_1 = \bigotimes c_{\{s\}}\eta[s := b][l := d_k] = 0.2$, and $v_2 = \bigotimes c_{\{s\}}\eta[s := b][l := d_k] = 0.2$. Thus, it returns **Succ**. Since all interactions are successful the speaker calls a success update, the listeners X_2 and X_3 do not change the preference levels, because all $v_k \leq 0.2$. From $t \geq 5$ the system converges to a absorbing state in which all interactions are successes, and the preference levels do not change. This state is also a solution of the fuzzy DCSP.

6 Experimental Results

In this Section we show some performance results related to the algorithm presented in Sec. 4.

To evaluate the runs we define the probability of a successful interaction at time t, $P_t(succ)$. $P_t(succ)$ is determined by the probability that an agent is a speaker at time t, and the probability the interaction is a success, $P_t(succ) = \sum P_t(succ|s = a_i)P(s = a_i)$. At each time t, by querying the state of the speaker and the listeners we can compute $P_t(succ|s = a_i)$ for each agent in the system. Since $P(s = a_i) = 1/N$, we can compute $P_t(succ) = \sum P_t(succ|s = a_i)/N$.

For our benchmark, let us define a *Random Fuzzy NG instance* (RFNG). To generate such an instance, we assign the same domain of names D to each agent, and for each agent and each agent's name we draw a preference level between $[0, 1]$ from a uniform distribution. Moreover, RFNG can only have crisp binary equality constraints. We also define the *Path RFNG Instance* [4] which is a RFNG instance, in which the constraint network is a *path graph*. A path graph (or linear graph) is a particularly simple example of a tree, which has two terminal vertices (vertices that have degree 1), while all others (if any) have degree 2.

We generated 5 such random instances, with 10 agents and 10 words each. we computed using a For each one of these instances, brutal force algorithm the best preference level and the word associated to this solution. Then, we ran this algorithm 10 times on each instance. In Fig. 6 we measure the evolution in time of $P_t(succ)$ for the path RFNG instance. When $P_t(succ) = 1$, all interactions are going to be successful, thus we are in an absorbing state. We also checked that the system is at a optimal naming convention when the probability of success becomes one.

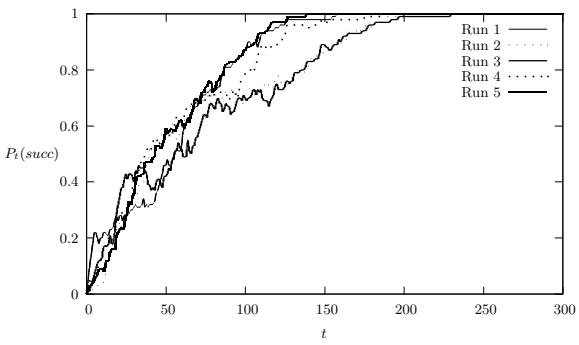


Fig. 6: Evolution of the mean $P_t(succ)$ over 5 different path RFNG instances. For each instance, we computed the mean $P_t(succ)$ over 10 different runs. We set $N = 10$, and the number of words to 10.

In Fig. 7, we show the scaling of the mean number of messages MNM needed to the system to find a solution for different numbers of N variables in the path RFNG instances. For each N, the MNM was measured over 5 different path RFNG instances. We notice that the points approximately overlap the polinomial $t = cN^{1.8}$.

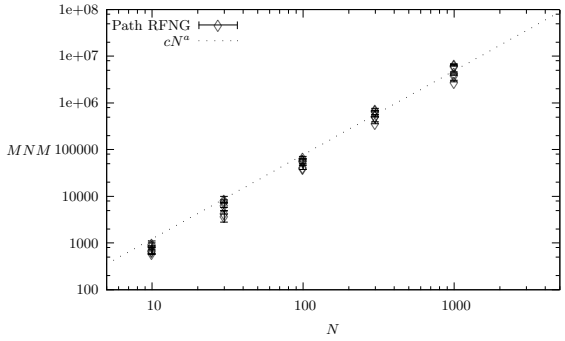


Fig. 7: Scaling of the mean number of messages MNM needed to the system to find a solution for different numbers of variables N in path RFNG instances. For each N, the MNM was measured over 5 different path RFNG instances. We notice that the points approximately overlap the function $cN^{1.8}$.

7 Conclusions and Future Work

In this paper we have shown an algorithm to solve an extension of NG problems [13, 1, 11, 8] with fuzzy preferences over words and we have also extended this algorithm in order to solve a generic instance of a Fuzzy DCSP [12, 16, 9, 10, 15]. In the study of this kind of algorithm we try to fully exploit the power of distributed calculation. Our algorithm is based on the random exploration of the system state space: it travels through the possible states until it finds the absorbing state, where it stabilizes. These goals are achieved through the union of new topics addressed in statistical physics (the NG), and the abstract framework posed by constraint solving.

Furthermore, in the real world applications, it may be quite restrictive to impose predetermined agent/variable ordering. For example, if we consider our agents to be corporations, institutions, or in general any collective of peers, we may find that a predetermined order may not be acceptable. Thus, it is very important to explore and understand how such distributed systems may cooperate and what problems may hinder them.

In future work, we intend to evaluate an asynchronous version of this algorithm in depth, and to test it using comparison metrics, such as communication costs (number of message sent), NCCCs (number of non-concurrent constraint checks). We then intend to compare our algorithm, against other distributed and asynchronous algorithms, such as the distributed stochastic search algorithm (DSA), and the distributed breakout algorithm (DBA). We also intend to investigate the "fairness" in the loss of privacy between algorithms with no pre-agreed agent/variable ordering, and algorithms with pre-agreed agent/variable ordering. We also plan to develop other functions used to select the speaker agent in the broadcast phase of the algorithm, and to study the their conver-

gence comparing the performance with the function F used in this paper (see Sec. 4.1). Furthermore, we will try to generalize it to generic semiring-based CSP instances [2], and not only Fuzzy CSPs.

References

1. A. Baronchelli, M. Felici, E. Caglioti, V. Loreto, and L. Steels. Sharp transition towards shared vocabularies in multi-agent systems. *CoRR*, abs/physics/0509075, 2005.
2. S. Bistarelli. *Semirings for Soft Constraint Solving and Programming*, volume 2962 of *LNCS*. Springer, 2004.
3. S. Bistarelli and G. Gosti. Solving CSPs with naming games. In A. Oddi, F. Fages, and F. Rossi, editors, *CSCLP*, volume 5655 of *LNCS*, pages 16–32. Springer, 2008.
4. S. Bistarelli and G. Gosti. Solving distributed CSPs probabilistically. *Fundam. Inform.*, 105(1-2):57–78, 2010.
5. S. Bistarelli, G. Gosti, and F. Santini. Solving fuzzy DCSPs with naming games. In *IEEE 23rd International Conference on Tools with Artificial Intelligence, ICTAI 2011*, pages 930–931, 2011.
6. Z. Collin, R. Dechter, and S. Katz. On the feasibility of distributed constraint satisfaction. In *IJCAI*, pages 318–324, 1991.
7. E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17:643–644, November 1974.
8. N. L. Komarova, K. A. Jameson, and L. Narens. Evolutionary models of color categorization based on discrimination. *Journal of Mathematical Psychology*, 51(6):359 – 382, 2007.
9. X. Luo, N. R. Jennings, N. Shadbolt, H. Leung, , and J. H. Lee. A fuzzy constraint based model for bilateral, multi-issue negotiations in semi-competitive environments. *Artif. Intell.*, 148:53–102, August 2003.
10. X. T. Nguyen and R. Kowalczyk. On solving distributed fuzzy constraint satisfaction problems with agents. In *Proceedings of the 2007 IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, IAT '07, pages 387–390, Washington, DC, USA, 2007. IEEE Computer Society.
11. M. A. Nowak, J. B. Plotkin, and D. C. Krakauer. The evolutionary language game. *Journal of Theoretical Biology*, 200(2):147–162, September 1999.
12. F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.
13. L. Steels. A self-organizing spatial vocabulary. *Artificial Life*, 2(3):319–332, 1995.
14. J. Wong, K. Ng, and H. Leung. A stochastic approach to solving fuzzy constraint satisfaction problems. In Eugene Freuder, editor, *Principles and Practice of Constraint Programming CP96*, volume 1118 of *LNCS*, pages 568–569. Springer Berlin Heidelberg, 1996. 10.1007/3-540-61551-2-119.
15. J. H. Y. Wong and H. Leung. Extending GENET to solve fuzzy constraint satisfaction problems. In *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, AAAI '98 IAAI '98, pages 380–385, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.
16. M. Yokoo and K. Hirayama. Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems*, 3:185–207, June 2000.

Commitment Protocol Generation

Akın Günay[1]* and Michael Winikoff[2] and Pınar Yolum[1]

[1] Computer Engineering Department, Bogazici University, Istanbul, Turkey
{akin.gunay,pinar.yolum}@boun.edu.tr
[2] Department of Information Science, University of Otago, Dunedin, New Zealand
michael.winikoff@otago.ac.nz

Abstract. Multiagent systems contain agents that interact with each other to carry out their activities. The agents' interactions are usually regulated with protocols that are assumed to be defined by designers at design time. However, in many settings, such protocols may not exist or the available protocols may not fit the needs of the agents. In such cases, agents need to generate a protocol on the fly. Accordingly, this paper proposes a method that can be used by an agent to generate commitment protocols to interact with other agents. The generation algorithm considers the agent's own goals and capabilities as well as its beliefs about other agents' goals and capabilities. This enables generation of commitments that are more likely to be accepted by other agents. We demonstrate the workings of the algorithm on a case study.

1 Introduction

Interaction is a key element of many multiagent systems. Agents need to interact for various reasons such as coordinating their activities, collaborating on tasks, and so on. These interactions are generally regulated by interaction protocols that define the messages that can be exchanged among agents. Traditionally, agents are supplied with interaction protocols at design time. Hence, they do not need to worry about which protocol to use at run time and can just use the given protocol as they see fit.

However, in open agent systems, where agents enter and leave, an agent may need to interact with another agent for which no previous interaction protocol has been designed. For example, a buyer may know of interaction protocols to talk to a seller, but may not be aware of an interaction protocol to talk to a deliverer. If these two agents meet, they need to figure out a protocol to complete their dealing. Additionally, even if there is an existing interaction protocol, the interaction protocols that are designed generically may make false assumptions about agents' capabilities, which would make the interaction protocol unusable in a real setting. For example, assume that an e-commerce protocol specifies that a buyer can pay by credit card upon receiving goods from a seller. If the buyer does not have the capability to pay by credit card, this protocol will not achieve its purpose. Even when the capabilities of the agents are aligned with those expected by the interaction protocol, the current context of the agents may not be appropriate to engage in the protocol. Following the previous example, an agent who

* Akın Günay is partially supported by TÜBİTAK Scholarships 2211 and 2214 and Pınar Yolum is partially supported by a TÜBİTAK Scholarship 2219.

can pay by credit card might have a current goal of minimizing bank transactions for that month and thus may find it more preferable to pay cash. That is, based on its current goals and existing commitments, the interactions that it is willing to engage in may differ. Therefore an interaction protocol that is blind to agents' current needs would not be applicable in many settings.

Accordingly, we argue that an agent needs to generate appropriate interaction protocols itself at run time. Since the agent would know its own capabilities, goals, and commitments precisely, it can generate an interaction protocol that respects these. However, for the interaction protocol to be successful, it should also take into account the participating agents' context.

Many times, even though the goals, commitments, or the capabilities of the other agents may not be known in full, partial information will exist. For example, agents may advertise their capabilities especially if they are offering them as services (e.g., selling goods). Existing commitments of the other agents may be known if the agent itself is part of those commitments (e.g., the agent has committed to deliver, after payment). The partial goal set of the participating agents may be known from previous interactions (e.g., the agent is interested in maximizing cash payments), or from domain knowledge (e.g. merchants in general have the goal of selling goods and/or services). Hence, the other agents' context can be approximated and using this approximate model a set of possible interaction protocols can be generated.

To realize this, we propose a framework in which agents are represented with their capabilities, goals, and commitments. The interactions of the agents are represented using commitments [2, 13] and the interaction protocols are modeled as commitment protocols. Commitments offer agents flexibility in carrying out their interactions and enable them to reason about them [8, 18, 20]. An agent that wants to engage in an interaction considers its own goals, makes assumptions about the other agents' goals, and proposes a set of commitments such that, if accepted by the other agent, will lead the initial agent to realize its goal. While doing this generation, the agent also considers its own capabilities, so that it generates commitments that it can realize. Note that even with a good approximation of the other agent, the proposed protocol may not be acceptable. For this reason, the agent generates a set of alternative protocols rather than a single one. The exact protocol that will be used is chosen after deliberations with other agents. Having alternative protocols is also useful for recoverability. That is, if a protocol is chosen by the agents, but if one of the agents then violates a commitment, the goals will not be realized as expected. In this case, agents can switch to an alternative protocol. This work is novel in that it situates commitment-based protocols in the larger context of agents by relating commitments to the agents goals, capabilities, and their knowledge of other agents' goals and capabilities.

The rest of this paper is organized as follows. Section 2 describes our technical framework in depth. Section 3 introduces our algorithm for generating commitment protocols based on agents' goals and capabilities. Section 4 applies the algorithm to a case study. Section 5 explains how our approach can be used in a multiagent system. Finally, Section 6 discusses our work in relation to recent work.

2 Technical Framework

In this section we define formally the necessary concepts: agents which have goals that they want to fulfill, and certain capabilities (formalized as propositions that they are able to bring about). We also define the notion of a social commitment between agents (in line with existing approaches, e.g. [20]). The concepts are captured using the following syntax, where *prop* is a proposition, and *agent* is an agent identifier.

commitment $\rightarrow$ *C(agent, agent, prop, prop)*^{cstate}
goal $\rightarrow$ G_{agent}*(prop, prop, prop)*^{gstate}
service $\rightarrow$ S_{agent}*(prop, prop)*
belief $\rightarrow$ BG_{agent}*(agent, prop, prop)* | BS_{agent}*(agent, prop, prop)*
cstate $\rightarrow$ *Null* | *Requested* | *Active* | *Conditional* | *Violated* | *Fulfilled* | *Terminated*
gstate $\rightarrow$ *Inactive* | *Active* | *Satisfied* | *Failed*

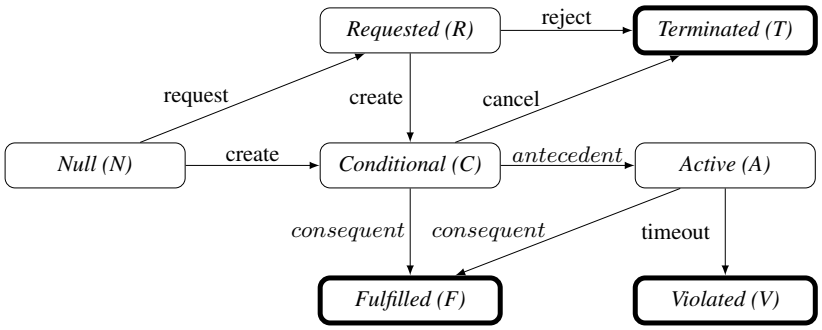


Fig. 1. Life cycle of a commitment.

Commitments. A *commitment* $C(debtor, creditor, antecedent, consequent)^{state}$ expresses the social contract between the agents $debtor$ and $creditor$, such that if the $antecedent$ holds, then the $debtor$ is committed to the $creditor$ to bring about the $consequent$. Each commitment has a $state$ that represents the current state of the commitment in its life cycle. The state of a commitment evolves depending on the state of the antecedent and the consequent and also according to the operations performed by the debtor and the creditor of the commitment. We show the life cycle of a commitment in Fig. 1. In this figure, the rectangles represent the states of the commitment and the directed edges represent the transitions between the states. Each transition is labeled with the name of the triggering event. A commitment is in *Null* state before it is created. The create operation is performed by the $debtor$ to create the commitment and the state of the commitment is set to *Conditional*. If the $antecedent$ already holds while creating the commitment, the state of the commitment becomes *Active* immediately. It is also possible for the $creditor$ of a commitment in *Null* state to make a request to the $debtor$ to create the commitment. In this case, the state of the commitment is *Requested*. The $debtor$ is free to create the requested commitment or reject it, which makes the commitment *Terminated*. A *Conditional* commitment becomes *Active* if the $antecedent$ starts

to hold, *Fulfilled* if the $consequent$ starts to hold or *Terminated* if the $debtor$ cancels the commitment. An *Active* commitment becomes *Fulfilled* if the $consequent$ starts to hold, *Violated* if the $debtor$ cancels the commitment or *Terminated* if the $creditor$ releases the $debtor$ from its commitment. *Fulfilled*, *Violated* and *Terminated* states are terminal states (depicted with thicker borders in Fig. 1)

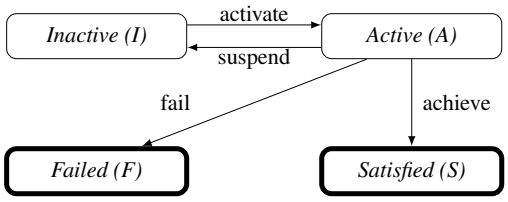


Fig. 2. Life cycle of a goal.

Goals. A *goal* $G_{agent}(precondition, satisfaction, failure)^{state}$ represents an aim of an agent such that the $agent$ has a goal to achieve $satisfaction$ if $precondition$ holds and the goal fails if $failure$ occurs (adapted from [19]). The state of the goal is represented by $state$. We show the life cycle of a goal in Fig. 2. A goal is in *Inactive* state if its $precondition$ does not hold. An inactive goal is not pursued by the agent. A goal is in *Active* state if its $precondition$ holds and neither $satisfaction$ nor $failure$ holds. An active goal is pursued by the agent. A goal is *Satisfied*, if $satisfaction$ starts to hold while in the *Active* state. A goal is *Failed*, if $failure$ occurs while in the *Active* state. An active goal may also be suspended, if the precondition ceases to hold. The *Satisfied* and *Failed* states are terminal states.

Capabilities. A *capability* $S_{agent}(precondition, proposition)$ states that an agent has the capability of performing an action (or actions) that will make $proposition$ true. However, this is only possible if the $precondition$ holds. Note that we use the terms "**capability**" and "**service**" interchangeably: in a context where an agent does something for itself "capability" makes more sense, but when an agent acts for another agent, then "service" is more appropriate.

Beliefs. Agents have their own *beliefs* about other agents' goals and capabilities. $BG_{agent_i}(agent_j, condition, satisfaction)$ represents that $agent_i$ believes $agent_j$ has the goal $satisfaction$ if $condition$ holds. Note that beliefs about other agents' goals do not include information about the failure conditions. Similarly $BS_{agent_i}(agent_j, condition, proposition)$ represents that $agent_i$ believes $agent_j$ is able to bring about the $proposition$, if the $condition$ holds. Beliefs about other agents' capabilities essentially correspond to services provided by other agents and interpreted as $agent_i$ believes that $agent_j$ provides a service to bring about $proposition$, if $condition$ is brought about (most probably by an effort of $agent_i$). As discussed in Section 1, although in general other agents' goals and capabilities are private, some information will be available. Although it is possible that advertised services may differ from the actual capabilities of the agent. For example, certain capabilities may not be

advertised, or some advertised services may in fact be realized by a third party (e.g. a merchant delegating delivery to a courier).

Agents and **Multiagent system**. An *agent* is a four tuple $A = \langle \mathcal{G}, \mathcal{S}, \mathcal{C}, \mathcal{B} \rangle$, where $\mathcal{G}$ is a set of goals that agent A has, $\mathcal{S}$ is a set of services (aka capabilities) that agent A can provide, $\mathcal{C}$ is a set of commitments that agent A is involved in and $\mathcal{B}$ is a set of beliefs that agent A has about other agents. A *multiagent system* $\mathcal{A}$ is a set of agents $\{A_1, \ldots, A_n\}$. We write $a.X$ to denote the X component of the agent, e.g. writing $a.\mathcal{G}$ to denote the agent's goals, $a.\mathcal{C}$ to denote its commitments etc.

Protocol. We adopt the definition of commitment protocols [6, 20] in which a *protocol* P is a set of (conditional) commitments. Hence, we do not have explicit message orderings. Each agent can manipulate the commitments as it sees fit. The manipulations of the commitments lead to state changes in the lifecycles of the commitments as depicted in Fig. 1. Unlike traditional approaches to capturing protocols, such as AUML, this approach, using social commitments, aims to provide minimal constraints on the process by which the interaction achieves its aims [14]. We emphasise that a set of commitments is a protocol in the sense that it allows for a range of possible concrete interactions, unlike the notion of contract used by Alberti *et al.* [1] which represents a single specific concrete interaction.

Definition 1 (Proposition Support). *Given a set Γ of propositions that hold, and a proposition p, the agent $a = \langle \mathcal{G}, \mathcal{S}, \mathcal{C}, \mathcal{B} \rangle$ supports p, denoted as $a \Vdash p$, iff at least one of the following cases holds:*

- *base case: $\Gamma \models p$, i.e. p already holds*
- *capability: $\exists S_a(pre, prop) \in \mathcal{S} : \{prop \rightarrow p \wedge a \Vdash pre\}$, i.e. the agent is able to bring about p (more precisely, a condition $prop$ which implies p) itself, and the required condition is also supported*
- *commitment: $\exists C(a', a, \top, cond)^A \in \mathcal{C} : \{cond \rightarrow p\}$, i.e. there is an active commitment from another agent to bring about p*
- *conditional: $\exists C(a', a, ant, cond)^C \in \mathcal{C} : \{cond \rightarrow p \wedge a \Vdash ant\}$, i.e. there is a conditional commitment from another agent to bring about p, and the antecedent of the commitment is supported by agent a*

The *capability* case states that p can be made true by agent a if p is one of the agent's capabilities. This is the strongest support for p, since p can be achieved by the agent's own capabilities. The *commitment* case states that the agent has a commitment in which it expects p to become true (because it is the creditor of an active commitment). Note that this is weaker than the *capability* condition since the commitment may be violated by its debtor. In the *conditional* case, the agent first needs to realize the antecedent for p to be achieved.

Definition 2 (Goal Support). *A goal $g = G_a(pre, sat, fail)^A$ is supported by the agent $a = \langle \mathcal{G}, \mathcal{S}, \mathcal{C}, \mathcal{B} \rangle$, denoted as $a \Vdash g$, if $a \Vdash sat$.*

Theorem 1. *If a proposition p (respectively goal g) is supported by agent a, then the agent is able to act in such a way that p (resp. g) eventually becomes true (assuming all active commitments are eventually fulfilled).*
Proof: *Induction over the cases in Definition 2 (details omitted).*

3 Commitment Protocol Generation Algorithm

We present an algorithm that uses the agent's capabilities, commitments and also beliefs about other agents, to generate a set of alternative commitment protocols[3] such that each generated protocol supports the given agent's set of goals. That is, for each given goal of the agent, either the agent is able to achieve the goal by using its own capabilities, or the agent is able to ensure that the goal is achieved by relying appropriately on a commitment from another agent which has the goal's satisfaction condition as its consequent. More precisely, if an agent a cannot achieve a desired proposition p using its own capabilities, then the algorithm generates a proposed commitment such as $C(a', a, q, p)^R$ (ensuring q is supported by a) to obtain (conditional) proposition support for p, which implies goal support for goal $g \equiv G_a(pre, p, fail)$.

Note that in general, we can only expect to be able to obtain *conditional* support (in terms of Definition 1). Obtaining *capability* support amounts to extending the agent's capabilities, and obtaining *commitment* support amounts to getting an active commitment $C(a', a, \top, q)^A$ which, in general, another agent a' would not have any reason to accept. Thus, the algorithm proposes commitments that are likely to be attractive to a' by considering its beliefs about the goals of a' and creating a candidate commitment $C(a', a, q, p)^R$ where q is a proposition that is believed to be desired by a' (i.e. satisfies one of its goals). Clearly, there are situations where a given goal cannot be supported (e.g. if no other agents have the ability to bring it about, or if no suitable q can be found to make the proposed commitments attractive), and hence the algorithm may not always generate a protocol.

We divide our algorithm into four separate functions (described below) for clarity:

- *generateProtocols* takes an agent and the set of proposition that hold in the world as arguments, and returns a set of possible protocols $\mathcal{P} = \{P_1, \ldots, P_n\}$, where each protocol is a set of proposed commitments (i.e. it returns a set of sets of commitments).
- *findSupport* takes as arguments an agent, a queue of goals, a set of propositions that are known to hold, and a set of commitments that are known to exist (initially empty); and does the actual work of computing the possible protocols, returning a set of possible protocols $\mathcal{P}$.
- *isSupported* takes as arguments an agent, a proposition, a set of propositions known to hold, and a set of commitments known to exist; and determines whether the proposition is supported, returning a Boolean value.
- *updateGoals* is an auxiliary function used by the main algorithm, and is explained below.

The *generateProtocols* function (see Algorithm 1) is the entry point of the algorithm. It has as parameters an agent a and a set of propositions Γ that hold in the world. Γ is meant to capture a's current world state. The algorithm finds possible, alternative protocols such that when executed separately, each protocol ensures that all of the goals of that agent are achievable.

[3] In practice, we may want to generate the set incrementally, stopping when a suitable protocol is found.

Algorithm 1 $\mathcal{P}$ generateProtocols(a, Γ)

Require: a, the agent that the algorithm runs for
Require: Γ, set of propositions known to be true
1: **queue** $\mathcal{G}' \leftarrow \{g | g \in a.\mathcal{G} \wedge g.state = Active\}$
2: **return** findSupport($a, \mathcal{G}', \Gamma, \emptyset$)

The *generateProtocols* function copies the agent's active goals into a queue structure $\mathcal{G}'$ for further processing and then calls the recursive function *findSupport* providing a (the agent), $\mathcal{G}'$ (its currently active goals), Γ (the propositions that currently hold), and $\emptyset$ (initial value for Δ) as arguments. The *generateProtocols* function returns the result of *findSupport*, which is a set of commitment protocols ($\mathcal{P}$), i.e. a set of sets of commitments. Recall that we use $a.\mathcal{G}$ to denote the goals $\mathcal{G}$ of agent a, and that for goal g we use $g.state$ to denote its state.

The main function is *findSupport* (see Algorithm 2). The function recursively calls itself to generate alternative commitment protocols which support every given goal of the agent a. The function takes as arguments an agent a, the queue of the agent's goals $\mathcal{G}'$ that need to be supported, a set Γ of propositions that are known to be true, and a set Δ of commitments that are known to exist. The function first defines sets $\mathcal{BG}$ and $\mathcal{BS}$ of (respectively) the beliefs of agent a about the goals and the services of other agents. It then pops the next goal g from the goal queue $\mathcal{G}'$ (Line 3). If all goals are considered (i.e. $g =$ Null), then there is no need to generate extra commitments. Hence, the algorithm simply returns one protocol: the set of the commitments already proposed. This corresponds to the base case of the recursion (Lines 4–5). If the agent already supports g (determined by *isSupported* function, see Algorithm 3), then the algorithm ignores g and calls itself for the next goal in $\mathcal{G}'$ (Line 8).

Otherwise, the function searches for one or more possible sets of commitments that will support the goal g. It first initializes the set of alternative protocols $\mathcal{P}$ to the empty set (Line 10). Then the algorithm searches for candidate commitments that will support g. As a first step it checks whether it has any capabilities that would support this goal if the precondition of the capability could be achieved through help from other agents (Line 11). Note that if the preconditions could be achieved by the agent itself then the algorithm would have detected this earlier in Line 3. Hence, here the specific case being handled is that the precondition of a capability cannot be achieved by the agent itself, but if it were achieved through other agents, then the capability would enable the agent to reach its goal g. For each such capability, we make the precondition pre a new goal for the agent, add it to the list of goals $\mathcal{G}'$ that it wants to achieve, and recursively call *findSupport* to find protocols.

After checking its own capabilities for achieving g, the agent then also starts looking for another agent with a known service $s' \in \mathcal{BS}$ such that s' achieves the satisfaction condition of the goal g (Line 14). For any such service s', we generate a proposed commitment of the form $C(a', a, sat', prop)^R$ (Line 16), where a' is the agent that is believed to be provide the service s', a is the agent being considered by the call to the function (its first argument), $prop$ implies the satisfaction condition of the desired goal g (i.e. $prop \rightarrow sat$), and sat' is an "attractive condition" to the proposed debtor agent

Algorithm 2 $\mathcal{P}$ findSupport$(a, \mathcal{G}', \Gamma, \Delta)$

Require: a, the agent that the algorithm runs for
Require: $\mathcal{G}'$, queue of agent's (active) goals
Require: Γ, set of propositions known to be true
Require: Δ, set of commitments already generated (initially called with $\emptyset$)
1: **define** $\mathcal{BG} \equiv \{b | b \in a.\mathcal{B} \wedge b = BG_a(a', gc, s)\}$
2: **define** $\mathcal{BS} \equiv \{b | b \in a.\mathcal{B} \wedge b = BS_a(a', c, p)\}$
3: $g \leftarrow$ pop$(\mathcal{G}')$
4: **if** $g =$ Null **then**
5: **return** $\{\Delta\}$
6: // else $g = G_a(gpre, sat, fail)^A$
7: **else if** isSupported(a, sat, Γ, Δ) **then**
8: **return** findSupport$(a, \mathcal{G}', \Gamma, \Delta)$
9: **else**
10: $\mathcal{P} = \emptyset$
11: **for all** $\{s \mid S_a(pre, prop) \in a.\mathcal{S} \wedge prop \rightarrow sat\}$ **do**
12: $\mathcal{P} \leftarrow \mathcal{P} \cup$ findSupport$(a, \{G_a(\top, pre, \bot)^A\} \cup \mathcal{G}', \Gamma, \Delta)$
13: **end for**
14: **for all** $\{s' \mid BS_a(a', cond, prop) \in \mathcal{BS} \wedge prop \rightarrow sat\}$ **do**
15: **for all** $\{g' \mid BG_a(a', pre', sat') \in \mathcal{BG} \wedge$ isSupported$(a, pre', \Gamma, \Delta)\}$ **do**
16: $c \leftarrow C(a', a, sat', prop)^R$
17: $\mathcal{G}'' \leftarrow$ updateGoals$(sat', prop, a.\mathcal{G}, \mathcal{G}')$
18: **if** $\neg$isSupported$(a, sat', \Gamma, \Delta)$ **then**
19: $\mathcal{G}'' \leftarrow \{G_a(\top, sat', \bot)^A\} \cup \mathcal{G}''$
20: **end if**
21: **if** $\neg$ isSupported$(a, cond, \Gamma, \Delta)$ **then**
22: $\mathcal{G}'' \leftarrow \{G_a(\top, cond, \bot)^A\} \cup \mathcal{G}''$
23: **end if**
24: $\mathcal{P} \leftarrow \mathcal{P} \cup$ findSupport$(a, \mathcal{G}'', \Gamma, \Delta \cup \{c\})$
25: **end for**
26: **end for**
27: **return** $\mathcal{P}$
28: **end if**

(a'). The notion of "attractive to agent a'" is defined in line 15: we look for a condition sat' that is believed to be a goal of agent a'. Specifically, we consider the known goals $\mathcal{BG}$ of other agents, and look for a $g' \in \mathcal{BG}$ such that $g' = BG_a(a', pre', sat')$ where pre' is already supported by agent a.

Next, having generated a potential commitment $C(a', a, sat', prop)^R$ where the debtor, a', has a service that can achieve the desired condition $prop$ and has a goal to bring about sat' (which makes the proposed commitment attractive), we update the goals of the agent (discussed below) and check whether (1) the promised condition sat' is supported by agent a, and (2) the precondition $cond$ for realizing $prop$ is supported by agent a. If they are supported, then a does not need to do anything else. Otherwise, it adds the respective proposition to the list of goals $\mathcal{G}''$ (Lines 19 and 22), so that appropriate support for these propositions can be obtained.

Finally, the agent calls the function recursively to deal with the remainder of the goals in the updated goal queue $\mathcal{G}''$. When doing this, it adds the currently created commitment c to the list of already generated commitments Δ. The result of the function call is added to the existing set of possible protocols $\mathcal{P}$ (line 24). Once the agent has completed searching for ways of supporting g, it returns the collected set of protocols $\mathcal{P}$. Note that if the agent is unable to find a way of supporting its goals, then $\mathcal{P}$ will be empty, and the algorithm returns the empty set, indicating that no candidate protocols could be found.

Algorithm 3 {**true** | **false**} isSupported(a, p, Γ, Δ)

Require: a, agent to check for support of p
Require: p, property to check for support
Require: Γ, set of propositions known to be true
Require: Δ, set of commitments already generated
1: **if** $\Gamma \models p$ **then**
2: **return true**
3: **end if**
4: **for all** $s = S_a(pre, prop) \in a.\mathcal{S}$ **do**
5: **if** $prop \rightarrow p \wedge$ isSupported(a, pre, Γ, Δ) **then**
6: **return true**
7: **end if**
8: **end for**
9: **for all** $\{c \mid C(a', a, cond, prop) \in (a.\mathcal{C} \cup \Delta)\}$ **do**
10: **if** $c.state = Active \wedge prop \rightarrow p$ **then**
11: **return true**
12: **else if** $(c.state = Conditional \vee c.state = Requested) \wedge prop \rightarrow p \wedge$ isSupported$(a, cond, \Gamma, \Delta)$ **then**
13: **return true**
14: **end if**
15: **end for**
16: **return false**

Algorithm 3 defines the *isSupported* function. This algorithm corresponds to Definition 1 and returns true if the given proposition p is supported by the given agent a, and false otherwise. The first case (line 1) checks whether the proposition is known to be true. The second case checks capability support. That is, whether p is supported by a capability s of the agent. More precisely, if the proposition $prop$ of s implies p and the precondition pre of s is supported by the agent (Lines 4-8). The third case checks commitment support by checking whether a has (or will have) an active commitment c, in which a is the creditor and the consequent $prop$ implies p (Lines 10-11). In the last case, the algorithm checks conditional support by checking whether a has (or will have) a conditional commitment c, in which a is the creditor, the consequent $prop$ implies p and a supports the antecedent $cond$ (Lines 12-14). If none of the above cases hold, then the algorithm returns false, indicating that p is not supported by a.

Algorithm 4 $\mathcal{G}''$ updateGoals(ant, $cons$, $\mathcal{G}$, $\mathcal{G}'$)

Require: ant, the antecedent of the new commitment
Require: $cons$, the consequent of the new commitment
Require: $\mathcal{G}$, set of agent's goals
Require: $\mathcal{G}'$, the current queue of (potentially) unsupported goals
1: **create new queue** $\mathcal{G}''$
2: $\mathcal{G}'' \leftarrow$ copy of $\mathcal{G}'$
3: **for all** $\{g \mid G_a(pre, sat, fail) \in \mathcal{G}\}$ **do**
4: **if** $g.state = Inactive \wedge (ant \rightarrow pre \vee cons \rightarrow pre)$ **then**
5: $g.state \leftarrow Active$
6: push($\mathcal{G}''$, g)
7: **end if**
8: **end for**
9: **return** $\mathcal{G}''$

Algorithm 4 defines the *updateGoals* function. This function is called when a new commitment is generated to support goal g of agent a. It takes propositions ant and $cons$ corresponding respectively to the antecedent and consequent of the new commitment. The function also takes as arguments the goals $\mathcal{G}$ of agent a, and the queue of currently unsupported goals $\mathcal{G}'$. The algorithm assumes that both ant and $cond$ will be achieved at some future point due to the generated commitment. Accordingly, the algorithm assumes that currently inactive goals which have ant or $cond$ as their precondition will be activated at some future point. Hence, these goals also need to be able to be achieved, i.e. to be supported by agent a. The algorithm thus generates these additional goals, and adds them to a (new queue) $\mathcal{G}''$. The algorithm first creates a new queue $\mathcal{G}''$ and copies into it the current contents of $\mathcal{G}'$ (Line 2). Then the goals in $\mathcal{G}$ that are inactive but will be activated are pushed into $\mathcal{G}''$ as active goals (Lines 3-8). Finally, $\mathcal{G}''$ is returned. Instead of pushing the goals that are assumed to be activated directly into $\mathcal{G}'$, the algorithm creates a new queue. This is done because every recursive call in line 24 of Algorithm 2 is related to a different commitment, which activates different goals depending on its antecedent and consequent. Hence each recursive call requires a different goal queue.

The algorithms presented are sound in the sense of Theorem 1: for any generated protocol, the agent is able to act in such a way as to ensure that the desired goal becomes achieved, without making any assumptions about the behaviour of other agents, other than that they fulfill their active commitments. The algorithms in this section have been implemented (available from `http://mas.cmpe.boun.edu.tr/akin/cpgen.html`), and have been used to generate protocols for a number of case studies, including the one we present next, which took 0.6 seconds to generate protocols (on a 2.GHz Intel Core i7 machine with 4 GB RAM running Ubuntu Linux).

4 Case Study

We illustrate our commitment generation algorithm's progress through an e-commerce scenario. In this scenario there is a customer (Cus), a merchant (Mer) and a bank

($Bank$). The goal of the customer is to buy some product from the merchant. The customer also has a goal of being refunded by the merchant, if the purchased product is defective. The customer is capable of making payment orders to the bank to pay to the merchant. The customer can also use a gift card, instead of payment. The merchant's goal is to be paid or to receive a gift card and the bank's goal is to get payment orders to earn commissions. We discuss the scenario from the customer's point of view, who runs our algorithm to generate a protocol in order to satisfy her goals. We first describe the propositions that we use and their meanings:

- *Delivered*: The purchased product is delivered to the customer.
- *Paid*: The merchant is paid.
- *HasGiftCard*: The customer has a gift card.
- *GiftCardUsed*: The customer uses the gift card.
- *Defective*: The delivered product is defective.
- *Returned*: The delivered product is returned to the merchant.
- *Refunded*: The customer is refunded.
- *PaymentOrdered*: The bank receives a payment order.

The customer has the following goals and capabilities: g_1 states that the goal of the customer is to have the product be delivered (without any condition) and g_2 represents the goal of the customer to be refunded, if the delivered product is defective, s_1 states that the customer is able to make payment orders (without any condition), and s_2 states that the customer is able to use a gift card (instead of payment), if she has one. Finally, s_3 states that the customer is capable of returning a product, if it is defective.

- $g_1 = G_{Cus}(\top, Delivered, \neg Delivered)$
- $g_2 = G_{Cus}(Defective, Refunded, \neg Refunded)$
- $s_1 = S_{Cus}(\top, PaymentOrdered)$
- $s_2 = S_{Cus}(HaveGiftCard, GiftCardUsed)$
- $s_3 = S_{Cus}(Defective, Returned)$

The customer has the following beliefs about the other agents: b_1 and b_2 state that the customer believes that the merchant provides a service to deliver a product, if the merchant is paid or a gift card is used, respectively. b_3 represents the belief that the merchant will give a refund, if a product is returned, and b_4 is the belief about the service of the bank to perform a money transaction for payment, if the bank receives such a request. The customer also believes that the goal of the merchant is to be paid (b_5) or to receive a gift card (b_6) and refund the customer if a sold product is defective (b_7), in order to ensure customer satisfaction. The goal of the bank is to receive payment orders (b_8), so that it can earn a commission from payment orders.

- $b_1 = BS_{Cus}(Mer, Paid, Delivered)$
- $b_2 = BS_{Cus}(Mer, GiftCardUsed, Delivered)$
- $b_3 = BS_{Cus}(Mer, Returned, Refunded)$
- $b_4 = BS_{Cus}(Bank, PaymentOrdered, Paid)$
- $b_5 = BG_{Cus}(Mer, \top, Paid)$
- $b_6 = BG_{Cus}(Mer, \top, GiftCardUsed)$

- $b_7 = BG_{Cus}(Mer, Defective, Returned)$
- $b_8 = BG_{Cus}(Bank, \top, PaymentOrdered)$

Let us first discuss the states of the merchant's goals g_1 and g_2. The algorithm considers both goals as active. g_1 is active, since its condition is $\top$. On the other hand, $Defective$ actually does not hold initially, which means g_2 should not be active. However, the algorithm assumes that $Defective$ holds, since its truth value is not controlled by any agent and therefore may or may not be true while executing the protocol. Using this assumption, the algorithm aims to create necessary commitments to capture all potential future situations during the execution of the protocol.

Let us walk through the protocol generation process. The algorithm starts with g_1. To support $Delivered$, which is the satisfaction condition of g_1, the algorithm generates the commitment $c_1 = C(Mer, Cus, Paid, Delivered)^R$ using the belief b_1, which is about the service to provide $Delivered$ and b_5, which is the goal of the merchant. However, the antecedent $Paid$ of c_1 is not supported by the customer. Hence, the algorithm considers $Paid$ as a new goal of the customer and starts to search for support for it. It finds the belief b_4, which indicates that the bank can bring about $Paid$ with a condition $PaymentOrdered$, which is also a goal of the bank due to b_8. $PaymentOrdered$ is already supported, since it is a capability of the customer (s_1). Hence, the algorithm generates the commitment $c_2 = C(Bank, Cus, PaymentOrdered, Paid)^R$. At this point, everything is supported to achieve g_1. The algorithm continues for g_2, which is achieved, if $Refunded$ holds. $Refunded$ can be achieved by generating the commitment $c_3 = C(Mer, Cus, Returned, Refunded)^R$ using the service b_3 and the goal b_7 of the merchant. The antecedent $Returned$ is a capability of the customer with a supported condition $Defective$. Hence, everything is supported to achieve g_2 and the algorithm returns the protocol that contains commitments c_1, c_2, and c_3.

Let us examine the protocol. c_1 states that the merchant is committed to deliver the product if the customer pays for it. However, the customer is not capable of payment (cannot bring about $Paid$ by itself). c_2 handles this situation, since the bank is committed to make the payment if the customer orders a payment. Finally, c_3 guarantees a refund, if the customer returns the product to the merchant. Note that the customer returns the product only if it is defective (s_2), hence there is no conflict with the goal (b_5) of the merchant.

Although the above protocol supports all the goals of the customer, the algorithm continues to search for other alternative protocols, since our aim is to generate all possible protocols to achieve the goals. Hence, it starts to search for alternative protocols that support the goals of the customer. It finds that it is possible to support g_1 also by using the service b_2. Accordingly, the algorithm initiates a new alternative protocol and generates the commitment $c_{2-1} = C(Mer, Cus, GiftCardUsed, Delivered)^R$ using the beliefs b_2 and b_6. However, the antecedent $GiftCardUsed$ of c_{2-1} is not supported by the customer, since $HasGiftCard$, which is the condition of service s_2, does not hold. The algorithm searches for support for $HasGiftCard$, but it fails, since neither the customer nor any other agent is able to bring it about.

Note that our algorithm also generates other protocols, which, due to information about other agents not being complete or correct, may be inappropriate. For instance, such a protocol may include a commitment such as $C(Mer, Cus, Paid, Refunded)^R$.

This happens because the algorithm considers all believed goals of the other agents while creating commitments. Specifically, to satisfy her goal $Refunded$, the customer considers the known goals of the merchant, and finds three options to offer to the merchant in return: $Paid$, $GiftCardUsed$ and $Returned$. Hence the algorithm creates three alternative commitments using each of these three goals of the merchant and each commitment is considered as an alternative protocol. Another example of this is a situation where the merchant actually replaces a defective product instead of refunding money (i.e. b_2 is incorrect). We deal with inappropriate protocols by requiring all involved agents to agree to a proposed protocol (see below). Specifically in this case when the customer requests the commitment from the merchant, the merchant would not accept the request.

5 Using Generated Protocols

The algorithm presented in the previous section generates candidate protocols, i.e. possible sets of proposed commitments that, if accepted, support the achievement of the desired propositions. In this section we consider the bigger picture and answer the question: *how are the generated candidate protocols used?*

The process is described in Algorithm 5, which uses two variables: the set of candidate protocols ($\mathcal{P}$), and the set of commitments (in the current candidate protocol, P) that agents have already accepted ($\mathcal{C}$). We begin by generating the set of protocols $\mathcal{P}$ (line 1). Next, we need to select one of the protocols[4] (line 2). The selected protocol is removed from $\mathcal{P}$. We then propose each commitment in the protocol to its debtor. This is needed because, as noted earlier, domain knowledge about other agents' goals may not be entirely correct or up-to-date. If any agent declines the proposed commitment then we cannot use the protocol, and so we clean up by releasing agents from their commitments in the protocol, and then try an alternative protocol. If all agents accept the commitments, then the protocol is executed.

Note that, since agents may not always fulfill their active commitments, we need to monitor the execution (e.g. [9]), and in case a commitment becomes violated, initiate action to recover. There are a range of possible approaches for recovery including simply abandoning the protocol and generating new protocols in the new state of the world; and using compensation [17].

6 Discussion

We developed an approach that enables agents to create commitment protocols that fit their goals. To achieve this, we proposed to represent agents' capabilities and commitments in addition to their goals. Agents reason about their goals as well as their beliefs about other agents' capabilities and goals to generate commitments. Our experiments on an existing case study showed that an agent can indeed generate a set of commitment protocols that can be used among agents. Hence, even agents who do not have any prior

[4] For the present we assume that the selection is done based on the simple heuristic that fewer commitments are preferred.

Algorithm 5 generateAndUseProtocols(a, Γ)

Require: a, the agent that the algorithm runs for
Require: Γ, set of propositions known to be true
1: $\mathcal{P} \leftarrow generateProtocols(a, \Gamma)$
2: select $P \in \mathcal{P}$
3: $\mathcal{P} \leftarrow \mathcal{P} \setminus \{P\}$
4: $\mathcal{C} \leftarrow \emptyset$
5: **for all** $C(x, y, p, q)^R \in P$ such that $x \neq a$ **do**
6: Propose $C(x, y, p, q)^R$ to agent x
7: **if** Agent x declines **then**
8: **for all** $C(x, y, p, q)^R \in \mathcal{C}$ **do**
9: Release agent x from the commitment $C(x, y, p, q)^R$
10: **end for**
11: Go to line 2
12: **else**
13: $\mathcal{C} \leftarrow \mathcal{C} \cup \{C(x, y, p, q)^R\}$
14: **end if**
15: **end for**
16: Execute Protocol P

protocols among them can communicate to carry out their interactions.

While we primarily discuss how our approach can be used at runtime, many of the underlying ideas can be used at design time as well. That is, a system designer who is aware of some of the goals and capabilities of the agents that will interact at runtime, can use the algorithm to generate protocols for them. This will enable a principled approach for designing commitment-based protocols.

Goals and commitments have been both widely studied in the literature. On the goals side, Thangarajah *et al.* [16] study relations and conflicts between goals. van Riemsdijk *et al.* [12] study different types of goals and propose to represent them in a unifying framework. On the commitments side, El-Menshawy *et al.* [7] study new semantics for commitments. Chopra and Singh [4, 5] study the interoperability and alignment of commitments. However, the interaction between goals and commitments has started to receive attention only recently.

Chopra *et al.* [3] propose a formalization of the semantic relationship between agents' goals and commitment protocols. Their aim is to check whether a given commitment protocol can be used to realize a certain goal. To do this, they define a capability set for each agent and first check if an agent can indeed carry out the commitments it participates in. This is important and can be used by agents to choose among possible commitment protocols. Chopra *et al.* assume that the commitment protocols are already available for agents. By contrast, in our work, we are proposing a method for the agents to generate a commitment protocol that they can use to realize their goals from scratch.

Işıksal [10] studies how an agent can create a single commitment to realize its goal with the help of other agents' in the system. She proposes reasoning rules that can be applied in various situations and she applies these rules on an ambient intelligence setting. She does not generate a set of alternative protocols and does not consider beliefs about other agents' goals as we have done here.

Desai *et al.* [6] propose Amoeba, a methodology to design commitment based protocols for cross-organizational business processes. This methodology enables a system designer to specify business processes through the participating agents' commitments. The methodology accommodates useful properties such as composition. Desai *et al.* model contextual changes as exceptions and deal with them through metacommitments. Their commitment-based specification is developed at design time by a human, based on the roles the agents will play. In this work, on the other hand, we are interested in agents generating their commitments themselves at run time. This will enable agents to interact with others even when an appropriate protocol has not been designed at design time.

Telang *et al.* [15] develop an operational semantics for goals and commitments. They specify rules for the evolution of commitments in light of agents' goals. These practical rules define when an agent should abandon a commitment, when it should negotiate, and so on. These rules are especially useful after a commitment protocol has been created and is in use. In this respect, our work in this paper is a predecessor to the execution of the approach that is described by Telang *et al.*, that is, after the protocol has been generated, the agents can execute it as they see fit, based on their current goals.

The work of Marengo *et al.* [11] is related to this work. Specifically, our notion of support (Definition 1) is analogous to their notion of control: intuitively, in order for an agent to consider a proposition to be supported, it needs to be able to ensure that it is achieved, i.e. be able to control its achievement. However, whereas the aim of their work is to develop a framework for reasoning about control and safety of given protocols, our aim is to derive protocols.

Future work includes studying how well our algorithms generate protocols in different scenarios, especially in cases where an agent's beliefs about other agents' goals and capabilities are incomplete or inconsistent. Also, we have not considered prioritization among generated alternative protocols. However, in real life, we would expect a ranking of different protocols based on how well a protocol satisfies an agents' goals or how much work a protocol requires an agent to carry out.

References

1. Alberti, M., Cattafi, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Montali, M., Torroni, P.: A Computational Logic Application Framework for Service Discovery and Contracting. International Journal of Web Services Research (IJWSR) 8(3), 1–25 (2011)
2. Castelfranchi, C.: Commitments: From Individual Intentions to Groups and Organizations. In: Lesser, V.R., Gasser, L. (eds.) ICMAS. pp. 41–48. The MIT Press (1995)
3. Chopra, A.K., Dalpiaz, F., Giorgini, P., Mylopoulos, J.: Reasoning about Agents and Protocols via Goals and Commitments. In: International Conference on Autonomous Agents and Multiagent Systems (AAMAS). pp. 457–464 (2010)
4. Chopra, A.K., Singh, M.P.: Constitutive Interoperability. In: International Conference on Autonomous Agents and Multiagent Systems (AAMAS). pp. 797–804 (2008)
5. Chopra, A.K., Singh, M.P.: Multiagent Commitment Alignment. In: International Conference on Autonomous Agents and Multiagent Systems (AAMAS). pp. 937–944 (2009)
6. Desai, N., Chopra, A.K., Singh, M.P.: Amoeba: A Methodology for Modeling and Evolving Cross-organizational Business Processes. ACM Transactions on Software Engineering and Methodology 19, 6:1–6:45 (October 2009)

7. El-Menshawy, M., Bentahar, J., Dssouli, R.: A New Semantics of Social Commitments Using Branching Space-Time Logic. In: WI-IAT '09: Proceedings of the 2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology. pp. 492–496 (2009)
8. Fornara, N., Colombetti, M.: Operational Specification of a Commitment-Based Agent Communication Language. In: International Conference on Autonomous Agents and Multiagent Systems (AAMAS). pp. 536–542 (2002)
9. Günay, A., Yolum, P.: Detecting Conflicts in Commitments. In: Sakama, C., Sardina, S., Vasconcelos, W., Winikoff, M. (eds.) Declarative Agent Languages and Technologies IX. LNAI, vol. 7169, pp. 51–66. Springer (2011)
10. Işıksal, A.: Use of Goals for Creating and Enacting Dynamic Contracts in Ambient Intelligence. Master's thesis, Bogazici University (2012)
11. Marengo, E., Baldoni, M., Baroglio, C., Chopra, A.K., Patti, V., Singh, M.P.: Commitments with Regulations: Reasoning about Safety and Control in REGULA. In: International Conference on Autonomous Agents and Multiagent Systems (AAMAS). pp. 467–474 (2011)
12. van Riemsdijk, M.B., Dastani, M., Winikoff, M.: Goals in Agent Systems: A Unifying Framework. In: International Conference on Autonomous Agents and Multiagent Systems (AAMAS). pp. 713–720 (2008)
13. Singh, M.P.: An Ontology for Commitments in Multiagent Systems. Artificial Intelligence and Law 7(1), 97–113 (1999)
14. Singh, M.P.: Information-Driven Interaction-Oriented Programming: BSPL, the Blindingly Simple Protocol Language. In: International Conference on Autonomous Agents and Multiagent Systems (AAMAS). pp. 491–498 (2011)
15. Telang, P.R., Yorke-Smith, N., Singh, M.P.: A Coupled Operational Semantics for Goals and Commitments. In: 9th International Workshop on Programming Multi-Agent Systems (ProMAS) (2011)
16. Thangarajah, J., Padgham, L., Winikoff, M.: Detecting & Avoiding Interference Between Goals in Intelligent Agents. In: Proceedings of the 18th International joint Conference on Artificial Intelligence. pp. 721–726 (2003)
17. Torroni, P., Chesani, F., Mello, P., Montali, M.: Social commitments in time: Satisfied or compensated. In: Declarative Agent Languages and Technologies (DALT). LNCS, vol. 5498, pp. 228–243 (2009)
18. Winikoff, M., Liu, W., Harland, J.: Enhancing Commitment Machines. In: Leite, J., Omicini, A., Torroni, P., Yolum, P. (eds.) Declarative Agent Languages and Technologies II, Lecture Notes in Computer Science, vol. 3476, pp. 198–220. Springer (2005)
19. Winikoff, M., Padgham, L., Harland, J., Thangarajah, J.: Declarative & procedural goals in intelligent agent systems. In: KR. pp. 470–481 (2002)
20. Yolum, P., Singh, M.P.: Flexible Protocol Specification and Execution: Applying Event Calculus Planning using Commitments. In: International Conference on Autonomous Agents and Multiagent Systems (AAMAS). pp. 527–534 (2002)

Goal-based Qualitative Preference Systems

Wietske Visser, Koen V. Hindriks, and Catholijn M. Jonker

Interactive Intelligence Group, Delft University of Technology, The Netherlands
{wietske.visser,k.v.hindriks,c.m.jonker}@tudelft.nl

Abstract. Goals are not only used to identify desired states or outcomes, but may also be used to derive qualitative preferences between outcomes. We show that Qualitative Preference Systems (QPSs) provide a general, flexible and succinct way to represent preferences based on goals. If the domain is not Boolean, preferences are often based on orderings on the possible values of variables. We show that QPSs that are based on such multi-valued criteria can be translated into equivalent goal-based QPSs that are just as succinct. Finally, we show that goal-based QPSs allow for more fine-grained updates than their multi-valued counterparts. These results show that goals are very expressive as a representation of qualitative preferences and moreover, that there are certain advantages of using goals instead of multi-valued criteria.

Key words: Qualitative multi-criteria preferences, goals

1 Introduction

In planning and decision making, goals are used to identify the desired states or outcomes. Essentially, goals provide a binary distinction between those states or outcomes that satisfy the goal and those that do not [1]. Outcomes that satisfy all goals are acceptable. However, it may happen that such outcomes are not available, but a decision still has to be made. Or there may be multiple outcomes that satisfy all goals and only one can be chosen. In these situations, goals provide no guidance to choose between the available alternatives [1, 2].

Instead of using goals in an absolute sense, it would be more convenient to use them to derive preferences between outcomes. There are multiple approaches to doing this in the literature, for example comparing the number of goals that are satisfied, or taking the relative importance of the (un)satisfied goals into account. We show in Section 2 that Qualitative Preference Systems [3] provide a general, flexible and succinct way to represent preferences based on goals. In this approach goals are modelled as criteria that can be combined to derive a preference between outcomes. We show that the best-known qualitative approaches to interpret goals as a representation of preferences are all expressible in a QPS.

Most goal-based approaches in the literature define outcomes as propositional models, i.e. all variables are Boolean, either true or false. In real-world applications, not all variables are Boolean. For example, variables may be numeric (e.g. cost, length, number, rating, duration, percentage) or nominal (e.g. destination, colour, location). Qualitative Preference Systems typically express preferences, in a compact way, based on

preference orderings on the possible values of variables. In Section 3 we show that such QPSs can be translated into equivalent goal-based QPSs, i.e. QPSs that express preferences based solely on goals. Such a translation requires at most polynomially more space, and hence is just as succinct as the original QPS. This result shows that goals are very expressive as a representation of qualitative preferences among outcomes. In [3], we discussed in detail the relation between Qualitative Preference Systems and two well-known frameworks that are representative for a large number of purely qualitative approaches to modelling preferences, namely Logical Preference Description language [4] and CP-nets [5]. We showed that for both of these approaches, a corresponding QPS can be defined straightforwardly. Since a QPS can be translated to a goal-based QPS, this result also holds for the goal-based QPSs that are the topic of the current paper.

In Section 4 we show that goal-based criterion trees also have some added value compared to trees with multi-valued criteria. We introduce basic updates on a QPS and show that goal-based QPSs allow for more fine-grained updates than their multi-valued counterparts. This is due to the different structure of goal-based criteria. We suggest a top-down approach to preference elicitation that starts with coarse updates and only adapts the criterion structure if more fine-grained updates are needed. Finally, Section 5 concludes the paper.

2 Modelling Goals as Criteria in a QPS

Several approaches to derive preferences over outcomes from goals can be found in the literature. Goals are commonly defined as some desired property that is either satisfied or not. As such, it is naturally represented as a propositional formula that can be true or false. Hence outcomes are often defined as propositional models, i.e. valuations over a set of Boolean variables $p, q, r, \ldots$. Sometimes all theoretically possible models are considered, sometimes the set of outcomes is restricted by a set of constraints. In the latter case, it is possible to specify which outcomes are actually available, or to use auxiliary variables whose values are derived from the values of other variables.

In [3] we introduced a framework for representing qualitative multi-criteria preferences called Qualitative Preference Systems (QPS). With this framework we aim to provide a generic way to represent qualitative preferences that are based on multiple criteria. A criterion can be seen as a preference from one particular perspective. We first summarize the general definition of a QPS from [3] in Section 2.1. We then propose in Section 2.2 that a goal can be straightforwardly modelled as a criterion in a QPS, thus providing the means to derive preferences over outcomes from multiple goals. In Section 2.3 we show that QPSs based on goal criteria can express different interpretations of what it means to have a goal p, such as absolute, ceteris paribus, leximin and discrimin preferences, and provide the possibility to state goals in terms of more fundamental interests.

2.1 Qualitative Preference Systems

The main aim of a QPS is to determine preferences between *outcomes* (or *alternatives*). An outcome is represented as an assignment of values to a set of relevant variables.

Every variable has its own domain of possible values. Constraints on the assignments of values to variables are expressed in a knowledge base. Outcomes are defined as variable assignments that respect the constraints in the knowledge base.

The preferences between outcomes are based on multiple *criteria*. Every criterion can be seen as a *reason* for preference, or as a preference from one particular *perspective*. A distinction is made between simple and compound criteria. Simple criteria are based on a single variable. Multiple (simple) criteria can be combined in a compound criterion to determine an overall preference. There are two kinds of compound criteria: cardinality criteria and lexicographic criteria. The subcriteria of a cardinality criterion all have equal importance, and preference is determined by counting the number of subcriteria that support it. In a lexicographic criterion, the subcriteria are ordered by priority and preference is determined by the most important subcriteria.

Definition 1. (Qualitative preference system [3]) *A* qualitative preference system (QPS) *is a tuple* $\langle Var, Dom, K, \mathcal{C} \rangle$. *Var is a finite set of* variables. *Every variable* $X \in Var$ *has a domain* $Dom(X)$ *of possible values. K (a* knowledge base*) is a set of constraints on the assignments of values to the variables in Var. A* constraint *is an equation of the form* $X = Expr$ *where* $X \in Var$ *is a variable and Expr is an algebraic expression that maps to* $Dom(X)$. *An* outcome α *is an assignment of a value* $x \in Dom(X)$ *to every variable* $X \in Var$, *such that no constraints in K are violated.* Ω *denotes the set of all outcomes:* $\Omega \subseteq \prod_{X \in Var} Dom(X)$. α_X *denotes the value of variable X in outcome* α. $\mathcal{C}$ *is a finite, rooted tree of criteria, where leaf nodes are simple criteria and other nodes are compound criteria. Child nodes of a compound criterion are called its subcriteria. The root of the tree is called the top criterion. Weak preference between outcomes by a criterion c is denoted by the relation* $\succeq_c$. $\succ_c$ *denotes the strict subrelation,* $\approx_c$ *the indifference subrelation.*

Definition 2. (Simple criterion [3]) *A* simple criterion *c is a tuple* $\langle X_c, \geq_c \rangle$, *where* $X_c \in Var$ *is a variable, and* $\geq_c$, *a preference relation on the possible values of* X_c, *is a preorder on* $Dom(X_c)$. $\succ_c$ *is the strict subrelation,* $\doteq_c$ *is the indifference subrelation. We call c a* Boolean simple criterion *if* X_c *is Boolean and* $\top \succ_c \bot$. *A simple criterion* $c = \langle X_c, \geq_c \rangle$ weakly prefers *an outcome* α *over an outcome* β, *denoted* $\alpha \succeq_c \beta$, *iff* $\alpha_{X_c} \geq_c \beta_{X_c}$.

Definition 3. (Cardinality criterion [3]) *A* cardinality criterion *c is a tuple* $\langle C_c \rangle$ *where* C_c *is a nonempty set of Boolean simple criteria (the* subcriteria *of c). A cardinality criterion* $c = \langle C_c \rangle$ weakly prefers *an outcome* α *over an outcome* β, *denoted* $\alpha \succeq_c \beta$, *iff* $|\{s \in C_c \mid \alpha \succ_s \beta\}| \geq |\{s \in C_c \mid \alpha \not\succeq_s \beta\}|$.

Note that a cardinality criterion can only have Boolean simple subcriteria. This is to guarantee transitivity of the preference relation induced by a cardinality criterion [3].

Definition 4. (Lexicographic criterion [3]) *A* lexicographic criterion *c is a tuple* $\langle C_c, \rhd_c \rangle$, *where* C_c *is a nonempty set of criteria (the* subcriteria *of c) and* $\rhd_c$, *a* priority relation *among subcriteria, is a strict partial order (a transitive and asymmetric relation) on* C_c. *A lexicographic criterion* $c = \langle C_c, \rhd_c \rangle$ weakly prefers *an outcome* α *over an outcome* β, *denoted* $\alpha \succeq_c \beta$, *iff* $\forall s \in C_c (\alpha \succeq_s \beta \vee \exists s' \in C_c (\alpha \succ_{s'} \beta \wedge s' \rhd_c s))$.

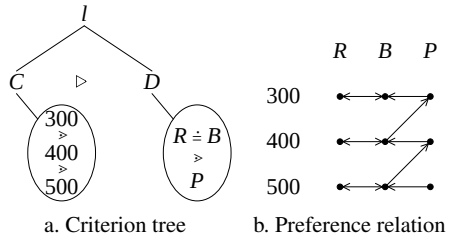


Fig. 1. Qualitative Preference System

This definition of preference by a lexicographic criterion is equivalent to the priority operator as defined by [6]. It generalizes the familiar rule used for alphabetic ordering of words, such that the priority can be any partial order and the combined preference relations can be any preorder.

Example 1. To illustrate, we consider a QPS to compare holidays. Holidays (outcomes) are defined by two variables: C (cost) and D (destination). $Dom(C) = \{300, 400, 500\}$ and $Dom(D) = \{R, B, P\}$ (Rome, Barcelona and Paris). For the moment, we do not use any constraints. We use the notation '300B', '500R' etc. to refer to outcomes. Preferences are determined by a lexicographic criterion l with two simple subcriteria: $\langle C, \geq_C \rangle$ such that $300 >_C 400 >_C 500$ and $\langle D, \geq_D \rangle$ such that $R \doteq_D B >_D P$. We slightly abuse notation and refer to these criteria by their variable, i.e. C and D. C has higher priority than D: $C \rhd_l D$. The criterion tree is shown in Figure 1a, the induced preference relation in Figure 1b. The black dots represent the outcomes, and the arrows represent preferences (arrows point towards more preferred outcomes). Superfluous arrows (that follow from reflexivity and transitivity of the preference relation) are left out for readability.

Priority between subcriteria of a lexicographic criterion ($\rhd$) is a strict partial order (a transitive and asymmetric relation). This means that no two subcriteria can have the same priority. If two criteria have the same priority, they have to be combined in a cardinality criterion, which can then be a subcriterion of the lexicographic criterion. To simplify the representation of such a lexicographic criterion with cardinality subcriteria, we define the following alternative specification.

Definition 5. (Alternative specification of a lexicographic criterion) *A tuple $\langle C'_c, \trianglerighteq'_c \rangle$, where C'_c is a set of criteria and $\trianglerighteq'_c$ is a preorder, specifies a lexicographic criterion $c = \langle C_c, \rhd_c \rangle$ as follows.*

- *Partition C'_c into priority classes based on $\trianglerighteq'_c$.*
- *For every priority class P, define a criterion c_P. If P contains only a single criterion s, then $c_P = s$. Otherwise c_P is a cardinality criterion such that for all $s \in P$: $s \in C_{c_P}$.*
- *Define $c = \langle C_c, \rhd_c \rangle$ such that $C_c = \{c_P \mid P \text{ is a priority class}\}$ and $c_P \rhd_c c_{P'}$ iff for all $s \in P, s' \in P'$: $s \rhd'_s s'$.*

For example, the specification $l = \langle \{g_1, g_2, g_3\}, \trianglerighteq \rangle$ such that $g_1 \trianglerighteq g_2 \triangleq g_3$ is short for $l = \langle \{g_1, c\}, \rhd \rangle$ such that $g_1 \rhd c$ and $c = \langle \{g_2, g_3\} \rangle$.

2.2 Goals in a QPS

In general, the variables of a QPS can have any arbitrary domain and simple criteria can be defined over such variables. Example 1 contains two such multi-valued simple criteria. In the goal-based case however, we define outcomes as propositonal models, and hence all variables are Booleans. Goals are defined as Boolean simple criteria, i.e. simple criteria that prefer the truth of a variable over falsehood.

Definition 6. (Goal) *A QPS* goal *is a Boolean simple criterion* $\langle X, \{(\top, \bot)\}\rangle$ *for some* $X \in Var$*. For convenience, we denote such a goal by its variable* X*.*

This is straightforward when goals are atomic, e.g. p. If goals are complex propositional formulas, e.g. $(p \vee q) \wedge \neg r$, an auxiliary variable s can be defined by the constraint $s = (p \vee q) \wedge \neg r$ (see [3] for details on auxiliary variables). As this is a purely technical issue, we will sometimes use the formula instead of the auxiliary variable in order not to complicate the notation unnecessarily.

Multiple goals can be combined in order to derive an overall preference. If multiple goals are equally important and it is the number of satisfied goals that determines preference, a cardinality criterion can be used. Actually, every cardinality criterion is already goal-based, since the subcriteria are restricted to Boolean simple criteria which are the same as goals. If there is priority between goals (or if goals have incomparable priority), they can be combined in a goal-based lexicographic criterion. Such a criterion can also be used to specify priority between sets of equally important goals (goal-based cardinality criteria).

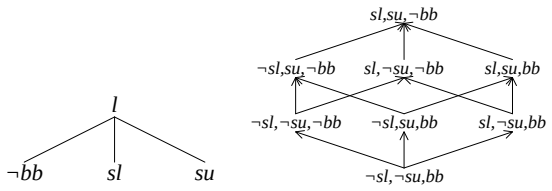
Definition 7. (Goal-based lexicographic criterion) *A* goal-based lexicographic criterion *is a lexicographic criterion all of whose subcriteria are either goals, goal-based cardinality criteria, or goal-based lexicographic criteria.*

Note that in the goal-based case, multi-valued simple criteria do not occur anywhere in the criterion tree; that is, all simple criteria are goals. The criterion tree in Figure 1a is not goal-based. However, we will see later that it can be translated to an equivalent goal-based criterion tree.

Example 2. Anne is planning to go on holiday with a friend. Her overall preference is based on three goals: that someone (she or her friend) speaks the language (sl), that it is sunny (su) and that she has not been there before ($\neg bb$). The set of variables is $Var = \{sl, su, bb\}$. Since every variable is propositional, the domain for each variable is $\{\top, \bot\}$ and there are eight possible outcomes. For the moment we do not constrain the outcome space and do not use auxiliary variables ($K = \varnothing$). Two goals (sl and su) are based on atomic propositions, the third ($\neg bb$) on a propositional formula that contains a negation. The overall preference between outcomes depends on the way that the goals are combined by compound criteria. In the next section we discuss several alternatives.

2.3 Expressivity of QPS as a Model of Goal-Based Preferences

What does it mean, in terms of preferences between outcomes, to have a goal p? Different interpretations can be found in the literature. We give a short overview of the best-known ones and show that QPSs can express the same preferences by means of some small examples.



a. Criterion tree b. Preference relation

Fig. 2. Ceteris paribus preference

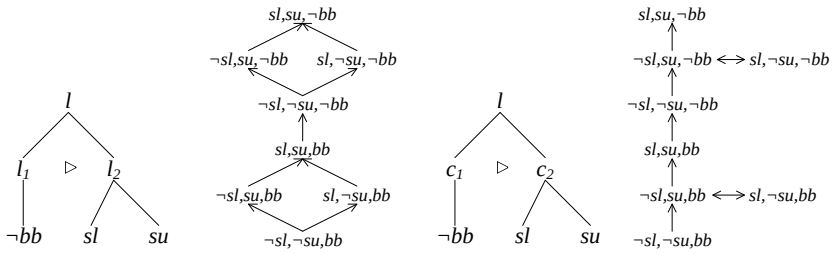
Ceteris Paribus Preference One interpretation of having a goal p is that p is preferred to $\neg p$ ceteris paribus (all other things being equal) [7, 1, 5]. The main question in this case is what the 'other things' are. Sometimes [5, 7], they are the other variables (atomic propositions) that define the outcomes. Wellman and Doyle [1] define ceteris paribus preferences relative to framings (a factorisation of the outcome space into a cartesian product of attributes). The preference relation over all outcomes is taken to be the transitive closure of the preferences induced by each ceteris paribus preference. So if we have p and q as ceteris paribus goals, then $p \wedge q$ is preferred to $\neg p \wedge \neg q$ since $p \wedge q$ is preferred to $\neg p \wedge q$ (by goal p) and $\neg p \wedge q$ is preferred to $\neg p \wedge \neg q$ (by goal q).

Example 3. Consider a lexicographic criterion l that has the three goals as subcriteria, and there is no priority between them, i.e. $l = \langle \{sl, su, \neg bb\}, \varnothing \rangle$ (Figure 2a). The resulting preference relation (Figure 2b) is a ceteris paribus preference.

This is a general property of qualitative preference systems: a lexicographic criterion with only goals as subcriteria and an empty priority relation induces a ceteris paribus preference, where the other things are defined by the other goals (see also [8]). The main advantage of the ceteris paribus approach is that it deals with multiple goals in a natural, intuitive way. However, the resulting preference relation over outcomes is always partial since there is no way to compare $p \wedge \neg q$ and $\neg p \wedge q$. This is why [1] claim that goals are inadequate as the sole basis for rational action. One way to solve this is to introduce relative importance between goals, which is done in the prioritized goals approach.

Prioritized Goals In e.g. [4], preferences are derived from a set of goals with an associated priority ordering (a total preorder). That is, there are multiple goals, each with an associated rank. There may be multiple goals with the same rank. Various strategies are possible to derive preferences from such prioritized goals. For example, the $\subseteq$ or discrimin strategy prefers one outcome over another if there is a rank where the first satisfies a strict superset of the goals that the second satisfies, and for every more important rank, they satisfy the same goals. The # or leximin strategy prefers one outcome over another if there is a rank where the first satisfies more goals than the second, and for every more important rank, they satisfy the same number of goals.

The prioritized goals strategies discrimin and leximin can also be expressed in a QPS. An exact translation is given in [3]. Here we just illustrate the principle. In the



a. Criterion tree b. Preference relation c. Criterion tree d. Preference relation

Fig. 3. (a, b) Discrimin preference (c, d) Leximin preference

prioritized goals approach, priority between goals is a total preorder, which can be expressed by assigning a rank to every goal. A QPS can model a discrimin or leximin preference with a lexicographic criterion that has one subcriterion for every rank. These subcriteria are compound criteria that contain the goals of the corresponding rank, and they are ordered by the same priority as the original ranking. For the discrimin strategy, the subcriteria are lexicographic criteria with no priority ordering between the goals. The leximin strategy uses the number of satisfied goals on each rank to determine overall preference. Therefore, each rank is represented by a cardinality criterion.

Example 4. Suppose that $\neg bb$ has the highest rank, followed by *sl* and *su* that have the same rank. The discrimin criterion tree for the example is shown in Figure 3a, where l is the top criterion and l_1 and l_2 the lexicographic criteria corresponding to the two ranks. The resulting preference relation is shown in Figure 3b. The leximin criterion tree for the example is shown in Figure 3c, where l is the top criterion and c_1 and c_2 the cardinality criteria corresponding to the two ranks. The resulting preference relation is shown in Figure 3d.

Preferential Dependence The above approaches all assume that goals are preferentially independent, that is, goalhood of a proposition does not depend on the truth value of other propositions. There are several options if goals are not preferentially independent. One is to specify conditional goals or preferences, as is done in e.g. [5, 2]. Another is to achieve preferential independence by restructuring the outcome space or expressing the goal in terms of more fundamental attributes [1, 9] or underlying interests [8].

Example 5. Actually, the variables *sl* and *bb* that we chose for the example already relate to some of Anne's underlying interests. It may have been more obvious to characterize the outcome holidays by the destination (where Anne may or may not have been before) and the accompanying friend (who may or may not speak the language of the destination country). In that case we would have had to specify that Anne would prefer Juan if the destination was Barcelona, but Mario if the destination was Rome. Instead of specifying several conditional preferences, we can just say that she prefers to go with someone who speaks the language. In this case, knowledge is used to create an abstraction level that allows one to specify more fundamental goals that are only indirectly related to the most obvious variables with which to specify outcomes [8].

3 Modelling Multi-valued Criteria as Goals

Preferences in a QPS are ultimately based on simple criteria, i.e. preferences over the values of a single variable. In general, the domain of such a variable may consist of many possible values. In the goal-based case, simple criteria are based on binary goals. In this section we show that the goal-based case is very expressive, by showing that every QPS can be translated into an equivalent goal-based QPS (provided that the domains of the variables used in the original QPS are finite). Moreover, we show that this translation is just as succinct as the original representation. In order to do this, we must first formalize the concept of equivalence between QPSs.

3.1 Equivalence

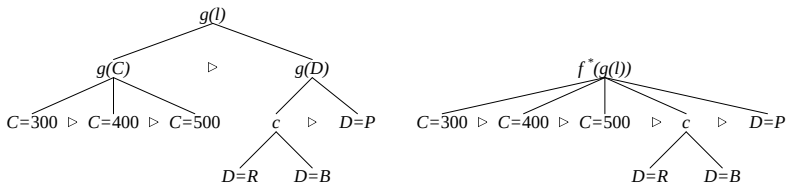
An obvious interpretation of equivalence between criteria is the equivalence of the preference relations they induce. I.e. two criteria c_1 and c_2 are equivalent if for all outcomes α, β, we have $\alpha \succeq_{c_1} \beta$ iff $\alpha \succeq_{c_2} \beta$. However, this definition only works if the criteria are defined with respect to the same outcome space, i.e. the same set of variables *Var*, the same domains *Dom* and the same constraints K. Since we will make use of auxiliary variables, we cannot use this definition directly. Fortunately, this is a technical issue that can be solved in a straightforward way.

Definition 8. (Equivalence of outcomes) *Let $S_1 = \langle Var_1, Dom_1, K_1, \mathcal{C}_1 \rangle$ and $S_2 = \langle Var_2, Dom_2, K_2, \mathcal{C}_2 \rangle$ be two QPSs such that $Var_1 \subseteq Var_2$, $\forall X \in Var_1 (Dom_1(X) \subseteq Dom_2(X))$ and $K_1 \subseteq K_2$. Let Ω_1 and Ω_2 denote the outcome spaces of S_1 and S_2, respectively. Two outcomes $\alpha \in \Omega_1$ and $\beta \in \Omega_2$ are* equivalent*, denoted $\alpha \equiv \beta$, iff $\forall X \in Var_1 : \alpha_X = \beta_X$.*

In the following, the only variables that are added are auxiliary variables. Such variables do not increase the outcome space because their value is uniquely determined by the values of (some of) the existing variables. We use special variable names of the form '$X = v$' to denote a Boolean variable that is true if and only if the value of variable X is v. For example, the variable $C = 300$ is true in outcomes $300R$, $300B$ and $300P$, and false in the other outcomes. When only auxiliary variables are added, every outcome in Ω_1 has exactly one equivalent outcome in Ω_2. We will represent such equivalent outcomes with the same identifier.

Definition 9. (Equivalence of criteria) *Let $S_1 = \langle Var_1, Dom_1, K_1, \mathcal{C}_1 \rangle$ and $S_2 = \langle Var_2, Dom_2, K_2, \mathcal{C}_2 \rangle$ be two QPSs such that $Var_1 \subseteq Var_2$, $\forall X \in Var_1 (Dom_1(X) \subseteq Dom_2(X))$ and $K_1 \subseteq K_2$. Let Ω_1 and Ω_2 denote the outcome spaces of S_1 and S_2, respectively. Two criteria c in $\mathcal{C}_1$ and c' in $\mathcal{C}_2$ are called* equivalent *iff $\forall \alpha, \beta \in \Omega_1, \forall \alpha', \beta' \in \Omega_2$, if $\alpha \equiv \alpha'$ and $\beta \equiv \beta'$, then $\alpha \succeq_c \beta$ iff $\alpha' \succeq_{c'} \beta'$.*

Definition 10. (Equivalence of QPSs) *Let $S_1 = \langle Var_1, Dom_1, K_1, \mathcal{C}_1 \rangle$ and $S_2 = \langle Var_2, Dom_2, K_2, \mathcal{C}_2 \rangle$ be two QPSs. S_1 and S_2 are equivalent if the top criterion of $\mathcal{C}_1$ is equivalent to the top criterion of S_2.*



a. Goal-based translation of the criterion tree in Figure 1a

b. The result of flattening the criterion in Figure 4a

Fig. 4. Goal-based translation and flattening

3.2 From Simple Criteria to Goals

A simple criterion on a variable with a finite domain can be translated to an equivalent goal-based criterion in the following way.

Definition 11. (Goal-based translation) *Let $c = \langle X, \geq \rangle$ be a simple criterion such that $Dom(X)$ is finite. The translation of c to a goal-based criterion, denoted $g(c)$, is defined as follows. If c is already a Boolean simple criterion, then $g(c) = c$. Otherwise:*

- *For every $x \in Dom(X)$, define a goal (Boolean simple criterion) c_x on variable $X = x$ with $\top \geq_{c_x} \bot$.*
- *Define a lexicographic criterion $g(c) = \langle C_{g(c)}, \trianglerighteq_{g(c)} \rangle$ such that $C_{g(c)} = \{c_x \mid x \in Dom(x)\}$ and $c_x \trianglerighteq_{g(c)} c_{x'}$ iff $x \geq_c x'$.*

Example 6. To illustrate, Figure 4a displays the translation of the criterion tree in Figure 1a. The simple criteria C and D have been replaced by their translations $g(C)$ and $g(D)$. These lexicographic criteria have a subgoal for every value of C resp. D. The priority between these goals corresponds to the value preferences of the original simple criteria.

Theorem 1. *Let $c = \langle X, \geq \rangle$ be a simple criterion such that $Dom(X_c)$ is finite. The goal-based translation $g(c)$ of c as defined in Definition 11 is equivalent to c.*

Proof. We distinguish five possible cases and show that in every case, c's preference between α and β is the same as $g(c)$'s preference between α and β.

1. If $\alpha_X = \beta_X$ then (a) $\alpha \approx_c \beta$ and (b) $\alpha \approx_{g(c)} \beta$.
2. If $\alpha_X \doteq_c \beta_X$ but $\alpha_X \neq \beta_X$ then (a) $\alpha \approx_c \beta$ and (b) $\alpha \approx_{g(c)} \beta$.
3. If $\alpha_X \succ_c \beta_X$ then (a) $\alpha \succ_c \beta$ and (b) $\alpha \succ_{g(c)} \beta$.
4. If $\beta_X \succ_c \alpha_X$ then (a) $\beta \succ_c \alpha$ and (b) $\beta \succ_{g(c)} \alpha$.
5. If $\alpha_X \not\succeq_c \beta_X$ and $\beta_X \not\succeq_c \alpha_X$ then (a) $\alpha \not\succeq_c \beta$ and $\beta \not\succeq_c \alpha$ and (b) $\alpha \not\succeq_{g(c)} \beta$ and $\beta \not\succeq_{g(c)} \alpha$.

1-5(a). This follows directly from the definition of simple criteria. **1(b).** If $\alpha_X = \beta_X$ then $\forall x \in Dom(X) : \alpha_{X=x} = \beta_{X=x}$, so also $\forall x \in Dom(X) : \alpha \approx_{c_x} \beta$. Hence, by the definition of a lexicographic criterion: $\alpha \approx_{g(c)} \beta$. **2-5(b).** If $\alpha_X \neq \beta_X$ then $\forall x \in Dom(X) \backslash \{\alpha_X, \beta_X\} : \alpha_{X=x} = \beta_{X=x}$ and $\alpha \approx_{g(c)} \beta$. Since a subcriterion s of a compound criterion such that $\alpha \approx_s \beta$ does not influence that compound criterion's preference between α and β, the

only criteria that can influence $g(c)$'s preference between α and β are c_{α_X} and c_{β_X}. Since $\alpha \succ_{c_{\alpha_X}} \beta$ and $\beta \succ_{c_{\beta_X}} \alpha$, preference between α and β by $g(c)$ is determined by the priority between c_{α_X} and c_{β_X}. **2(b).** If $\alpha_X \geq_c \beta_X$ then $c_{\alpha_X} \doteq_{g(c)} c_{\beta_X}$, so they are together in a cardinality criterion and we have $\alpha \approx_{g(c)} \beta$. **3(b).** If $\alpha_X \succ \beta_X$ then $c_{\alpha_X} \rhd_{g(c)} c_{\beta_X}$ so by the definition of a lexicographic criterion $\alpha \succ_{g(c)} \beta$. **4(b).** Analogous to 3(b). **5(b).** If $\alpha_X \not\geq_c \beta_X$ and $\beta_X \not\geq_c \alpha_X$ then $c_{\alpha_X} \not\rhd_{g(c)} c_{\beta_X}$ and $c_{\beta_X} \not\rhd_{g(c)} c_{\alpha_X}$ and $c_{\alpha_X} \not= _{g(c)} c_{\beta_X}$, so by the definition of a lexicographic criterion $\alpha \not\succ_{g(c)} \beta$ and $\beta \not\succ_{g(c)} \alpha$. □

By replacing every simple criterion c in a criterion tree with its goal-based translation $g(c)$, an equivalent goal-based criterion tree is obtained.

Definition 12. (Relative succinctness) *$g(c)$ is at least as succinct as c iff there exists a polynomial function p such that $size(g(c)) \leq p(size(c))$. (Adapted from [10].)*

Theorem 2. *Let $c = \langle X, \geq \rangle$ be a simple criterion such that $Dom(X_c)$ is finite. The translation $g(c)$ of c as defined in Definition 11 is just as succinct as c.*

Proof. The goal-based translation just replaces variable values with goals, and the preference relation between them with an identical priority relation between goals, so the translation is linear. □

The above two theorems are very important as they show that goals are very expressive as a way to represent qualitative preferences, and moreover, that this representation is just as succinct as a representation based on multi-valued criteria.

4 Updates in a QPS

In this section we show that goal-based criterion trees also have some added value compared to trees with multi-valued criteria. We introduce updates on a criterion tree as changes in the value preference of simple criteria or in the priority of lexicographic criteria. The number of updates of this kind that are possible depends on the structure of the tree. In general, the flatter a criterion tree, the more updates are possible. It is possible to make criterion tree structures flatter, i.e. to reduce the depth of the tree, by removing intermediate lexicographic criteria. The advantage of goal-based criterion trees is that they can be flattened to a greater extent than their equivalent non-goal-based counterparts. We first formalize the concept of flattening a criterion tree. Then we define what we mean by basic updates in a criterion tree and show the advantages of flat goal-based QPSs compared to other flat QPSs.

4.1 Flattening

Simple criteria are terminal nodes (leaves) and cannot be flattened. Cardinality criteria have only Boolean simple subcriteria and cannot be flattened either. Lexicographic criteria can have three kinds of subcriteria: simple, cardinality and lexicographic. They can be flattened by replacing each lexicographic subcriterion by that criterion's subcriteria and adapting the priority accordingly (as defined below).

Definition 13. (Removing a lexicographic subcriterion) *Let $c = \langle C_c, \rhd_c \rangle$ be a lexicographic criterion and $d = \langle C_d, \rhd_d \rangle \in C_c$ a lexicographic criterion that is a subcriterion of c. We now define a lexicographic criterion $f(c,d) = \langle C_{f(c,d)}, \rhd_{f(c,d)} \rangle$ that is equivalent to c but does not have d as a subcriterion. To this end, we define $C_{f(c,d)} = C_c \backslash \{d\} \cup C_d$ and $\forall i, j \in C_{f(c,d)} : i \rhd_{f(c,d)} j$ iff $i, j \in C_c$ and $i \rhd_c j$, or $i, j \in C_d$ and $i \rhd_d j$, or $i \in C_c$, $j \in C_d$ and $i \rhd_c d$, or $i \in C_d$, $j \in C_c$ anf $d \rhd_c j$.*

Theorem 3. *$f(c,d)$ is equivalent to c, i.e. $\alpha \succeq_c \beta$ iff $\alpha \succeq_{f(c,d)} \beta$.*

Proof. $\Rightarrow$. Suppose $\alpha \succeq_c \beta$. Then $\forall s \in C_c(\alpha \succeq_s \beta \vee \exists s' \in C_c(\alpha \succ_{s'} \beta \wedge s' \rhd_c s))$. We need to show that also $\forall s \in C_{f(c,d)}(\alpha \succeq_s \beta \vee \exists s' \in C_{f(c,d)}(\alpha \succ_{s'} \beta \wedge s' \rhd_{f(c,d)} s))$. We do this by showing that $\alpha \succeq_s \beta \vee \exists s' \in C_{f(c,d)}(\alpha \succ_{s'} \beta \wedge s' \rhd_{f(c,d)} s)$ holds for every possible origin of $s \in C_{f(c,d)}$. We have $\forall s \in C_{f(c,d)}$, either $s \in C_c \backslash \{d\}$ or $s \in C_d$.

- If $s \in C_c \backslash \{d\}$, we know that $\alpha \succeq_s \beta \vee \exists s' \in C_c(\alpha \succ_{s'} \beta \wedge s' \rhd_c s)$. If $\alpha \succeq_s \beta$, trivially also $\alpha \succeq_s \beta \vee \exists s' \in C_{f(c,d)}(\alpha \succ_{s'} \beta \wedge s' \rhd_{f(c,d)} s)$ and we are done. If $\exists s' \in C_c(\alpha \succ_{s'} \beta \wedge s' \rhd_c s)$, then either $s' \in C_c \backslash \{d\}$ or $s' = d$. If $s' \in C_c \backslash \{d\}$, then $s' \in C_{f(c,d)}$ and $s' \rhd_{f(c,d)} s$, so also $\alpha \succeq_s \beta \vee \exists s' \in C_{f(c,d)}(\alpha \succ_{s'} \beta \wedge s' \rhd_{f(c,d)} s)$ and we are done. If $s' = d$, then (since $\alpha \succ_{s'} \beta$) $\exists i \in C_{s'}$ (and hence $\in C_{f(c,d)}$): $\alpha \succ_i \beta$. Since $s' \rhd_c s$, we have $i \rhd_{f(c,d)} s$ and so also $\alpha \succeq_s \beta \vee \exists i \in C_{f(c,d)}(\alpha \succ_i \beta \wedge i \rhd_{f(c,d)} s)$ and we are done.
- Now consider the case that $s \in C_d$. Since $d \in C_c$, we know that either $\alpha \succeq_d \beta$ or $\exists s' \in C_c(\alpha \succ_{s'} \beta \wedge s' \rhd_c d)$. If $\alpha \succeq_d \beta$, we know $\alpha \succeq_s \beta \vee \exists s' \in C_d(\alpha \succ_{s'} \beta \wedge s' \rhd_d s)$ and hence $\alpha \succeq_s \beta \vee \exists s' \in C_{f(c,d)}(\alpha \succ_{s'} \beta \wedge s' \rhd_{f(c,d)} s)$ and we are done. If $\exists s' \in C_c(\alpha \succ_{s'} \beta \wedge s' \rhd_c d)$ then $\exists s' \in C_{f(c,d)}(\alpha \succ_{s'} \beta \wedge s' \rhd_{f(c,d)} s)$ so trivially also $\alpha \succeq_s \beta \vee \exists s' \in C_{f(c,d)}(\alpha \succ_{s'} \beta \wedge s' \rhd_{f(c,d)} s)$ and we are done.

$\Leftarrow$. Suppose $\alpha \not\succeq_c \beta$. Then $\exists s \in C_c(\alpha \not\succeq_s \beta \wedge \forall s' \in C_c(s' \rhd_c s \rightarrow \alpha \not\succ_{s'} \beta))$. We need to show that also $\exists t \in C_{f(c,d)}(\alpha \not\succeq_t \beta \wedge \forall t' \in C_{f(c,d)}(t' \rhd_{f(c,d)} t \rightarrow \alpha \not\succ_{t'} \beta))$. Either $s \neq d$ or $s = d$.

- If $s \neq d$, then $s \in C_{f(c,d)}$ and we know that $\alpha \not\succeq_s \beta$ and $\forall s' \in C_{f(c,d)} \backslash C_d(s' \rhd_{f(c,d)} s \rightarrow \alpha \not\succ_{s'} \beta)$. If $d \not\rhd_c s$, then $\forall s' \in C_{c*}(s' \rhd_{f(c,d)} s \rightarrow s' \in C_{f(c,d)} \backslash C_d)$. So we have $\exists s \in C_{f(c,d)}(\alpha \not\succeq_s \beta \wedge \forall s' \in C_{f(c,d)}(s' \rhd_{f(c,d)} s \rightarrow \alpha \not\succ_{s'} \beta))$. Take $t = s$ and we are done. If $d \rhd_c s$, then $\alpha \not\succ_d \beta$, i.e. $\alpha \not\succeq_d \beta$ or $\beta \succeq_d \alpha$. If $\alpha \not\succeq_d \beta$, then $\exists u \in C_d(\alpha \not\succeq_u \beta \wedge \forall u' \in C_d(u' \rhd_d u \rightarrow \alpha \not\succeq_{u'} \beta))$. Since $\forall s' \in C_c(s' \rhd_c s \rightarrow \alpha \not\succ_{s'} \beta)$ and $d \rhd_c s$, we also have $\exists u \in C_{f(c,d)}(\alpha \not\succeq_u \beta \wedge \forall u' \in C_{f(c,d)}(u' \rhd_{f(c,d)} u \rightarrow \alpha \not\succ_{u'} \beta))$. Take $t = u$ and we are done. If $\beta \succeq_d \alpha$, then $\forall v \in C_d(\beta \succeq_v \alpha \vee \exists v' \in C_d(\beta \succ_{v'} \alpha \wedge v' \rhd_d v))$. This means that either $\forall u \in C_d(\beta \succeq_u \alpha)$ or $\exists u \in C_d(\beta \succ_u \alpha \wedge \neg \exists u' \in C_d(u' \rhd_d u))$. If $\forall u \in C_d(\beta \succeq_u \alpha)$, then $\forall u \in C_d(\alpha \not\succ_u \beta)$. Take $t = s$ and we are done. If $\exists u \in C_d(\beta \succ_u \alpha \wedge \neg \exists u' \in C_d(u' \rhd_d u))$, then $\exists u \in C_d(\alpha \not\succeq_u \beta \wedge \forall u' \in C_d(u' \rhd_d u \rightarrow \alpha \not\succ_{u'} \beta))$. Take $t = u$ and we are done.
- If $s = d$, then $\alpha \not\succeq_d \beta$, so $\exists u \in C_d(\alpha \not\succeq_u \beta \wedge \forall u' \in C_d(u' \rhd_d u \rightarrow \alpha \not\succ_{u'} \beta))$. Since $\forall s' \in C_c(s' \rhd_c d \rightarrow \alpha \succ_{s'} \beta)$, we have $\forall s' \in C_c(s' \rhd_c u \rightarrow \alpha \succ_{s'} \beta)$. Take $t = u$ and we are done. □

Theorem 4. *$f(c,d)$ is just as succinct as c.*

Proof. When a lexicographic subcriterion is removed according to Definition 13, the total number of criteria decreases with 1: the subcriteria of d become direct subcriteria of c and d itself is removed. The priority between the original subcriteria of c (i.e. $C_c \backslash \{d\}$) and the priority between the original subcriteria of d (i.e. C_d) remains unaltered. Just the priority between the subcriteria in $C_c \backslash \{d\}$ and d is replaced by priority between the subcriteria in $C_c \backslash \{d\}$ and the subcriteria in C_d. Since $|C_d|$ is finite, the increase in size is linear. □

Definition 14. (Flat criterion) *All simple and cardinality criteria are flat. A lexicographic criterion is flat if all its subcriteria are either simple or cardinality criteria.*

Definition 15. (Flattening) *The flat version of a non-flat lexicographic criterion c, denoted $f^*(c)$, is obtained as follows. For an arbitrary lexicographic subcriterion $d \in C_c$, get $f(c,d)$. If $f(c,d)$ is flat, $f^*(c) = f(c,d)$. Otherwise, $f^*(c) = f^*(f(c,d))$.*

Example 7. (Flattening) The original criterion tree in Figure 1 is already flat. Its goal-based translation in Figure 4a can be flattened further, as shown in Figure 4b. Here the lexicographic subcriteria $g(C)$ and $g(D)$ have been removed.

4.2 Updates

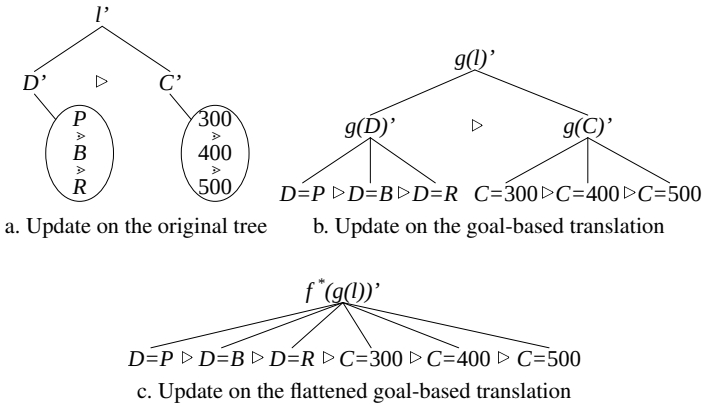
Criterion trees can be updated by leaving the basic structure of the tree intact but changing the priority between subcriteria of a lexicographic criterion ($\trianglerighteq$) or the value preferences of a multi-valued simple criterion ($\geq$). By performing these basic operations, the induced preference relation also changes. Therefore, such updates can be used to 'fine-tune' a person's preference representation.

Definition 16. (Update) *An update of a criterion tree is a change in (i) the preference between values ($\geq$) of a multi-valued simple criterion; and/or (ii) the priority ($\trianglerighteq$) between (in)direct subcriteria of a lexicographic criterion (in the alternative specification). The changed relations still have to be preorders.*

Theorem 5. *For every update on a criterion tree c, there exists an equivalent update on the goal-based translation $g(c)$ and vice versa.*

Proof. Every change in a value preference $\geq$ between two values x and y corresponds one-to-one to a change in priority between c_x and c_y. Every change in priority between two subcriteria s and s' corresponds one-to-one to a change in priority between $g(s)$ and $g(s')$. □

Example 8. Consider for example the criterion tree in Figure 1a. On the highest level, there are three possibilities for the priority: $C \triangleright D$, $D \triangleright C$ or incomparable priority. On the next level, each simple criterion has preferences over three possible values, which can be ordered in 29 different ways (this is the number of different preorders with three elements, oeis.org/A000798). So in total there are $3 \times 29 \times 29 = 2523$ possible updates of this tree. For the goal-based translation of this tree (in Figure 4a) this number is the same. Figure 5 shows one alternative update of the original criterion tree in Figure 1 as well as its goal-based translation in Figure 4a.



a. Update on the original tree b. Update on the goal-based translation

c. Update on the flattened goal-based translation

Fig. 5. Updates on criterion trees

Theorem 6. *For every update on a criterion tree c, there exists an equivalent update on the flattened criterion tree* $f^*(c)$.

Example 9. Figure 5c shows an update on the flat goal-based criterion tree in Figure 4b that is equivalent to the updates in Figure 5a and 5b.

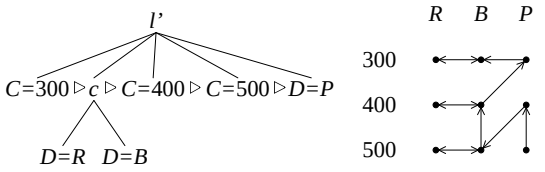
Theorem 7. *If a criterion tree c is not flat, there exist updates on* $f^*(c)$ *that do not have equivalent updates on c.*

We show this by means of an example.

Example 10. The goal-based tree in Figure 4a can be flattened to the equivalent flat tree in Figure 4b. This flattened tree can be updated in 209527 different ways (the number of different preorders with 6 elements, oeis.org/A000798), thereby allowing more preference relations to be represented by the same tree structure. Figure 6 shows an alternative flat goal-based tree that can be obtained from the previous one by updating it. It is not possible to obtain an equivalent criterion tree by finetuning the original criterion tree or its goal-based translation. This is because goals relating to different variables are 'mixed': the most important goal is that the cost is 300, the next most important goal is that the destination is Rome or Barcelona, and only after that is the cost considered again. This is not possible in a criterion tree that is based on simple criteria that are defined directly on the variables *C* and *D*.

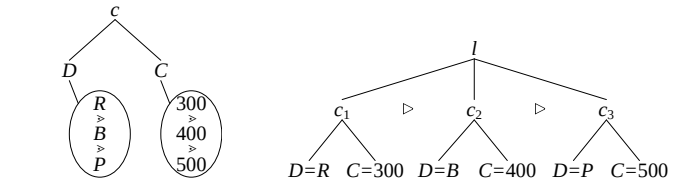
Theorem 8. *Let c be a non-flat, non-goal-based criterion. Then there exist updates on* $f^*(g(c))$ *that do not have equivalent updates on* $f^*(c)$.

In general, the flatter a criterion tree, the more different updates are possible. Since a goal-based tree can be made flatter than an equivalent criterion tree that is based on multi-valued simple criteria, the goal-based case allows more updates.



a. Criterion tree b. Preference relation

Fig. 6. Alternative flat goal-based tree obtained by updating the tree in Figure 4b



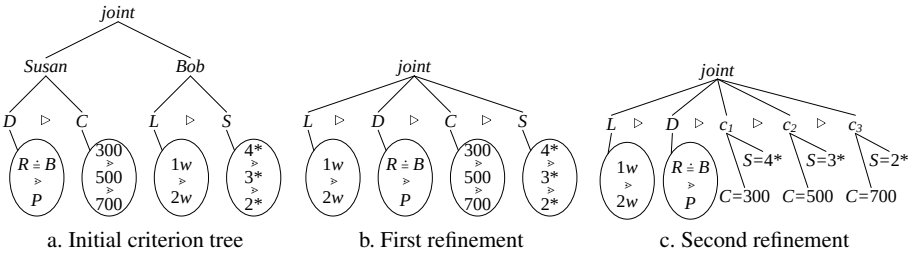
a. Invalid cardinality criterion b. Goal-based compensation

Fig. 7. Preferences where C and D are equally important

Example 11. This example shows how goals can be used for compensation between variables. The subcriteria of a cardinality criterion must be Boolean, to avoid intransitive preferences. So, for example, the criterion in Figure 7a is not allowed. It would result in $400B \approx 500R$ and $500R \approx 300B$, but $300B \succ 400B$. However, the underlying idea that the variables C and D are equally important is intuitive. Using goals we can capture it in a different way, as displayed in Figure 7b. This criterion tree results in a total preorder of preference between outcomes, where for instance $300B \succ 500R \succ 400B$.

The results above show that every update that can be applied on a criterion tree can also be applied on its flattened goal-based translation, and that this last criterion tree even allows more updates. However, if we look at the size of the updates, we can see that for equivalent updates, more value preference or priority relations have to be changed when the structure is flatter. For example, a simple inversion of the priority between $g(C)$ and $g(D)$ in Figure 4a corresponds to the inversion of priority between all of $C = 300$, $C = 400$ and $C = 500$ and all of $D = R$, $D = B$ and $D = P$ in Figure 4b. This suggests the following approach to finetuning a given preference representation during the preference elicitation process. First, one can fine-tune the current criterion tree as well as possible using (coarse) updates. If the result does not match the intended preferences well enough, one can start flattening, which will create more, fine-grained possibilities to update the tree. If this still does not allow to express the correct preferences, one can make a goal-based translation and flatten it. This allows for even more possible updates on an even lower level.

Example 12. Susan and Bob are planning a city trip together. Susan would like to go to a city that she has not been to before, and hence prefers Rome or Barcelona to Paris. She also does not want to spend too much money. Bob is a busy businessman who



a. Initial criterion tree b. First refinement c. Second refinement

Fig. 8. Successive criterion trees for Susan and Bob

only has a single week of holiday and would like some luxury, expressed in the number of stars of the hotel. There is no priority between Susan's and Bob's preferences. The initial criterion tree for Susan and Bob's joint preferences is displayed in Figure 8a. Susan and Bob decide that Bob's criterion on the length of the trip should be the most important, because he really does not have time to go for two weeks. They also decide that luxury is less important than the other criteria. In order to update the tree, it is first flattened by removing the subcriteria of Susan and Bob. The new tree, after flattening and updating, is shown in Figure 8b. However, Bob feels that luxury can compensate for cost. To represent this, the criteria for cost and number of stars are translated to goals and combined into three cardinality criteria, as shown in Figure 8c. At this point, the travel agent's website is able to make a good selection of offers to show and recommend to Susan and Bob.

5 Conclusion

We have shown that the QPS framework can be used to model preferences between outcomes based on goals. It has several advantages over other approaches. First, the QPS framework is general and flexible and can model several interpretations of using goals to derive preferences between outcomes. This is done by simply adapting the structure of the criterion tree. It is possible to specify an incomplete preference relation such as the ceteris paribus relation by using an incomplete priority ordering. But if a complete preference relation is needed, it is also easy to obtain one by completing the priority relation between subcriteria of a lexicographic criterion, or using cardinality criteria. Second, goals do not have to be independent. Multiple goals can be specified using the same variable. For example, there is no problem in specifying both p and $p \wedge q$ as a goal. Third, goals do not have to be consistent. It is not contradictory to have both p preferred to $\neg p$ (from one perspective) and $\neg p$ preferred to p (from another). This possibility is also convenient when combining preferences of multiple agents, who may have different preferences. Preferences of multiple agents can be combined by just collecting them as subcriteria of a new lexicographic criterion. Fourth, background knowledge can be used to express constraints and define abstract concepts. This in turn can be used to specify goals on a more fundamental level.

When the variables that define the outcomes are not Boolean, preferences are usually based on orderings of the possible values of each variable. We have shown that

such multi-valued criteria can be translated to equivalent goal-based criteria. Such a translation requires at most polynomially more space, and hence is just as succinct as the original QPS. This result shows that goals are very expressive as a representation of qualitative preferences among outcomes.

Goal-based criterion trees also have some added value compared to trees with multi-valued criteria. We introduced basic updates on a QPS and showed that goal-based QPSs allow for more fine-grained updates than their multi-valued counterparts. This is due to the different structure of goal-based criteria. In general, the flatter a criterion tree, the more updates are possible. It is possible to make criterion tree structures flatter, i.e. to reduce the depth of the tree, by removing intermediate lexicographic criteria. The advantage of goal-based criterion trees is that they can be flattened to a greater extent than their equivalent non-goal-based counterparts, and hence provide more possible updates.

We proposed a procedure to fine-tune a criterion tree during the preference elicitation process. Essentially, this is a top-down approach where a criterion tree is first updated as well as possible in its current state, and is only flattened and/or translated to a goal-based tree if more updates are necessary. This procedure gives rise to a more fundamental question. If it is really necessary to take all these steps, then maybe the original criteria were not chosen well in the first place. It may have been better to choose more fundamental interests as criteria. This is still an open question that we would like to address in the future.

Acknowledgements. This research is supported by the Dutch Technology Foundation STW, applied science division of NWO and the Technology Program of the Ministry of Economic Affairs. It is part of the Pocket Negotiator project with grant number VICI-project 08075.

References

1. Wellman, M.P., Doyle, J.: Preferential semantics for goals. In: Proc. AAAI. (1991) 698–703
2. Boutilier, C.: Toward a logic for qualitative decision theory. In: Proc. KR. (1994) 75–86
3. Visser, W., Aydoğan, R., Hindriks, K.V., Jonker, C.M.: A framework for qualitative multi-criteria preferences. In: Proc. ICAART. (2012)
4. Brewka, G.: A rank based description language for qualitative preferences. In: Proc. ECAI. (2004) 303–307
5. Boutilier, C., Brafman, R.I., Domshlak, C., Hoos, H.H., Poole, D.: CP-nets: A tool for representing and reasoning with conditional ceteris paribus preference statements. Journal of Artificial Intelligence Research **21** (2004) 135–191
6. Andréka, H., Ryan, M., Schobbens, P.Y.: Operators and laws for combining preference relations. Journal of Logic and Computation **12**(1) (2002) 13–53
7. Von Wright, G.H.: The Logic of Preference: An Essay. Edinburgh University Press (1963)
8. Visser, W., Hindriks, K.V., Jonker, C.M.: Interest-based preference reasoning. In: Proc. ICAART. (2011) 79–88
9. Keeney, R.L.: Analysis of preference dependencies among objectives. Operations Research **29**(6) (1981) 1105–1120
10. Chevaleyre, Y., Endriss, U., Lang, J.: Expressive power of weighted propositional formulas for cardinal preference modelling. In: Proc. KR. (2006)

SAT-based BMC for Deontic Metric Temporal Logic and Deontic Interleaved Interpreted Systems ⋆

Bożena Woźna-Szcześniak and Andrzej Zbrzezny

IMCS, Jan Długosz University. Al. Armii Krajowej 13/15, 42-200 Częstochowa, Poland.
{b.wozna,a.zbrzezny}@ajd.czest.pl

Abstract. We consider multi-agent systems' (MASs) modelled by deontic interleaved interpreted systems and we provide a new SAT-based bounded model checking (BMC) method for these systems. The properties of MASs are expressed by means of the metric temporal logic with discrete semantics and extended to include epistemic and deontic operators. The proposed BMC approach is based on the state of the art solutions to BMC. We test our results on a typical MASs scenario: train controller problem with faults.

1 Introduction

By *agents* we usually mean rational, independent, intelligent and high-tech entities that act autonomously on behalf of their users, across open and distributed environments, to solve a growing number of complex problems. *A multi-agent system* (MAS) [24] is a system composed of multiple interacting (communicating, coordinating, cooperating, etc.) agents which can be used to solve problems that are beyond the individual capacities or knowledge of a single agent.

Deontic interpreted systems (DISs) [17] are models of MASs that make possible reasoning about epistemic and correct functioning behaviour of MASs. They provide a computationally grounded semantics on which it is possible to interpret the $\mathcal{O}_i\alpha$ modality, representing the fact "in all correct functioning executions of agent i, α holds", as well as a traditional epistemic modalities and temporal operators. By *deontic interleaved interpreted systems* (DIISs) we mean a restriction of DISs that enforce the executions of agents to be interleaved. Thus we assume that agents act as network of synchronised automata; note that one can see DIISs as a deontic extension of the formalism of interleaved interpreted systems [16]. We consider DIISs since they allow for the distinction between correct (or ideal, normative, etc.) and incorrect states, and they enable more efficient verification of MASs, the behaviour of which is as the behaviour of synchronised automata. Note that although our method is described for DIISs, it can be applied to DISs [7] as well; as it will be clear below the main difference between DIISs and DISs is in the definition of the global evolution function. Thus, to apply our method to DISs it is enough to change the definition a propositional formula that encodes the transition relation. However, only DIISs can be combined with partial order reductions allowing for more efficient verification of MASs that are not so loosely coupled.

⋆ Partly supported by National Science Center under the grant No. 2011/01/B/ST6/05317.

Model checking [4, 21] has been developed as a method for automatic verification of finite state concurrent systems, and impressive strides have been made on this problem over the past thirty years.The main aim of model checking is to provide an algorithm determining whether an abstract model - representing, for example, a software project - satisfies a formal specification expressed as a modal formula. Moreover, if the property does not hold, the method discovers a counterexample execution that shows the source of the problem. The practical applicability of model checking in MASs settings requires the development of algorithms hacking the state explosion problem. In particular, to avoid this problem the following approaches have been developed: BDD-based bounded [12, 18] and unbounded [23, 22] model checking, SAT-based bounded [19, 20, 25] and unbounded [13] model checking.

To express the requirements of MASs, various extensions of temporal [5] or real time [2] temporal logics with epistemic (to represent knowledge) [7], doxastic (to represent beliefs) [15], and deontic (to represent norms and prescriptions) [17, 3] components have been proposed. In this paper we consider a deontic and epistemic extension of Metric Temporal Logic (MTL) [14], which we call MTLKD and interpret over discrete-time models; note that over the adopted discrete-time model, MTL is simply LTL, but with an exponentially succinct encoding [8]. MTLKD allows for the representation of the quantitative temporal evolution of epistemic states of the agents, as well as their correct and incorrect functioning behaviour. It can express multiple timing constraints on runs, which is really interesting for writing specifications. For example, MTLKD allows to express property asserting that whenever the system finds itself in a p-state, then agent c knows that the system will be in a q-state precisely one time unit later; note that this can be specified by the formula $\mathrm{G}(p \Rightarrow \mathrm{K}_c\mathrm{F}_{[1,1]}q)$.

In our past research we have provided a theoretical underpinnings of a bounded model checking (BMC) algorithm for DIS and an epistemic and deontic extension of CTL [25]; the method have not been implemented and experimentally evaluated. The main contributions of the paper are as follows. First, we introduce the MTLKD language. Second, we propose a SAT-based bounded model checking (BMC) technique for DIISs and the existential part of MTLKD. This is the first time when the BMC method for linear time epistemic (and deontic) logics uses a reduced number of paths to evaluate epistemic and deontic components what results in significantly smaller and less complicated propositional formulae that encode the MTLKD properties. Third, we implement the proposed BMC method and evaluate it experimentally. To the best of our knowledge, this is the first work which provides a practical (bounded) model checking algorithm for the MTLKD language, and the MTL itself.

The structure of the paper is the following. In Section 2 we shortly introduce DIISs and the MTLKD language. In Section 3 we define a bounded semantics for EMTLKD (existential part of MTLKD) and prove that there is a bound such that both bounded and unbounded semantics for EMTLKD are equivalent. In Section 4 we define a BMC method for MTLKD. In Section 5 we present performance evaluation of our newly developed SAT-based BMC algorithm and we conclude the paper.

2 Preliminaries

DIIS. We assume that a MAS consists of n agents, and by $Ag = \{1, \ldots, n\}$ we denote the non-empty set of agents; note that we do not consider the environment component. This may be added with no technical difficulty at the price of heavier notation. We assume that each agent $c \in Ag$ is in some particular local state at a given point in time, and that a set L_c of local states for agent $c \in Ag$ is non-empty and finite (this is required by the model checking algorithms). We assume that for each agent $c \in Ag$, its set L_c can be partitioned into *faultless (green)* and *faulty (red)* states. For n agents and n mutually disjoint and non-empty sets $\mathcal{G}_1, \ldots, \mathcal{G}_n$ we define the set G of all possible *global states* as the Cartesian product $L_1 \times \ldots \times L_n$, such that $L_1 \supseteq \mathcal{G}_1, \ldots, L_n \supseteq \mathcal{G}_n$. The set $\mathcal{G}_c$ is called the set of green states for agent c. The complement of $\mathcal{G}_c$ with respect to L_c (denoted by $\mathcal{R}_c$) is called the set of red states for the agent c. Note that $\mathcal{G}_c \cup \mathcal{R}_c = L_c$ for any agent c. Further, by $l_c(g)$ we denote the local component of agent $c \in Ag$ in a global state $g = (l_1, \ldots, l_n)$.

With each agent $c \in Ag$ we associate a finite set of *possible actions* Act_c such that a special "null" action (ϵ_c) belongs to Act_c; as it will be clear below the local state of agent c remains the same, if the null action is performed. We do not assume that the sets Act_c (for all $c \in Ag$) are disjoint. Next, with each agent $c \in Ag$ we associate a protocol that defines rules, according to which actions may be performed in each local state. The protocol for agent $c \in Ag$ is a function $P_c : L_c \to 2^{Act_c}$ such that $\epsilon_c \in P_c(l)$ for any $l \in L_c$, i.e., we insist on the null action to be enabled at every local state. For each agent c, there is a (partial) evolution function $t_c : L_c \times Act_c \to L_c$ such that for each $l \in L_c$ and for each $a \in P_c(l)$ there exists $l' \in L_c$ such that $t_c(l, a) = l'$; moreover, $t_c(l, \epsilon_c) = l$ for each $l \in L_c$. Note that the local evolution function considered here differs from the standard one (see [7]) by having the local action instead of the join action as the parameter. Further, we define the following sets $Act = \bigcup_{c \in Ag} Act_c$ and $Agent(a) = \{c \in Ag \mid a \in Act_c\}$.

The *global interleaved evolution function* $t : G \times Act_1 \times \cdots \times Act_n \to G$ is defined as follows: $t(g, a_1, \ldots, a_n) = g'$ iff there exists an action $a \in Act \setminus \{\epsilon_1, \ldots, \epsilon_n\}$ such that for all $c \in Agent(a)$, $a_c = a$ and $t_c(l_c(g), a) = l_c(g')$, and for all $c \in Ag \setminus Agent(a)$, $a_c = \epsilon_c$ and $t_c(l_c(g), a_c) = l_c(g)$. In brief we write the above as $g \xrightarrow{a} g'$.

Now, for a given set of agents Ag and a set of propositional variables $\mathcal{PV}$, which can be either true or false, a *deontic interleaved interpreted system* is a tuple: $DIIS = (\iota, < L_c, \mathcal{G}_c, Act_c, P_c, t_c >_{c \in Ag}, \mathcal{V})$, where $\iota \in G$ is an initial global state, and $\mathcal{V} : G \to 2^{\mathcal{PV}}$ is a valuation function. With such a DIIS it is possible to associate a *model* $M = (\iota, S, T, \{\sim_c\}_{c \in Ag}, \{\bowtie_c\}_{c \in Ag}, \mathcal{V})$, where ι is the initial global state; $S \subseteq G$ is a set of reachable global states that is generated from ι by using the global interleaved evolution functions t; $T \subseteq S \times S$ is a global transition (temporal) relation on S defined by: sTs' iff there exists an action $a \in Act \setminus \{\epsilon_1, \ldots, \epsilon_n\}$ such that $s \xrightarrow{a} s'$. We assume that the relation is total, i.e., for any $s \in S$ there exists an $a \in Act \setminus \{\epsilon_1, \ldots, \epsilon_n\}$ such that $s \xrightarrow{a} s'$ for some $s' \in S$; $\sim_c \subseteq S \times S$ is an indistinguishability relation for agent c defined by: $s \sim_c s'$ iff $l_c(s') = l_c(s)$; $\bowtie_c \subseteq S \times S$ is a deontic relation for agent c defined by: $s \bowtie_c s'$ iff $l_c(s') \in \mathcal{G}_c$; $\mathcal{V} : S \to 2^{\mathcal{PV}}$ is the valuation function of $DIIS$

restricted to the set S. $\mathcal{V}$ assigns to each state a set of propositional variables that are assumed to be true at that state.

Syntax of MTLKD. Let $p \in \mathcal{PV}$, $c \in Ag$, $\Gamma \subseteq Ag$, and I be an interval in $\mathbb{N} = \{0, 1, 2, \ldots\}$ of the form: $[a, b)$ and $[a, \infty)$, for $a, b \in \mathbb{N}$; note that the remaining forms of intervals can be defined by means of $[a, b)$ and $[a, \infty)$. Hereafter, let $left(I)$ denote the left end of the interval I, i.e., $left(I) = a$, and $right(I)$ the right end of the interval I, i.e., $right([a, b)) = b - 1$ and $right([a, \infty)) = \infty$. The language MTLKD is defined by the following grammar:

$$\alpha ::= \mathbf{true} \mid \mathbf{false} \mid p \mid \neg\alpha \mid \alpha \wedge \alpha \mid \alpha \vee \alpha \mid \mathrm{X}\alpha \mid \alpha \mathrm{U}_I \alpha \mid \mathrm{G}_I \alpha \mid \overline{\mathrm{K}}_c \alpha \mid \overline{\mathrm{D}}_\Gamma \alpha \mid \overline{\mathrm{E}}_\Gamma \alpha \mid \overline{\mathrm{C}}_\Gamma \alpha \mid \overline{\mathcal{O}}_c \alpha \mid \underline{\widehat{\mathrm{K}}}_c^d \alpha$$

The derived basic modalities are defined as follows: $\alpha \mathrm{R}_I \beta \stackrel{def}{=} \beta \mathrm{U}_I (\alpha \wedge \beta) \vee \mathrm{G}_I \beta$, $\mathrm{F}_I \alpha \stackrel{def}{=} \mathbf{true} \mathrm{U}_I \alpha$, $\mathcal{O}_c \alpha \stackrel{def}{=} \neg\overline{\mathcal{O}}_c \neg\alpha$, $\mathrm{K}_c \alpha \stackrel{def}{=} \neg\overline{\mathrm{K}}_c \neg\alpha$, $\widehat{\mathrm{K}}_c^d \alpha \stackrel{def}{=} \neg\underline{\widehat{\mathrm{K}}}_c^d \neg\alpha$, $\mathrm{D}_\Gamma \alpha \stackrel{def}{=} \neg\overline{\mathrm{D}}_\Gamma \neg\alpha$, $\mathrm{E}_\Gamma \alpha \stackrel{def}{=} \neg\overline{\mathrm{E}}_\Gamma \neg\alpha$, $\mathrm{C}_\Gamma \alpha \stackrel{def}{=} \neg\overline{\mathrm{C}}_\Gamma \neg\alpha$, where $c, d \in \mathcal{AG}$, and $\Gamma \subseteq \mathcal{AG}$. Intuitively, U_I and G_I are the operators, resp., for "bounded until" and "bounded always". $\mathrm{X}\alpha$ is true in a computation if α is true at the second state of the computation, $\alpha \mathrm{U}_I \beta$ is true in a computation if β is true in the interval I at least in one state and always earlier α holds, and $\mathrm{G}_I \alpha$ is true in a computation if α is true at all the states of the computation that are in the interval I. $\overline{\mathrm{K}}_c$ is the operator dual for the standard epistemic modality K_c ("agent c knows"), so $\overline{\mathrm{K}}_c \alpha$ is read as "agent c does not know whether or not α holds". Similarly, the modalities $\overline{\mathrm{D}}_\Gamma, \overline{\mathrm{E}}_\Gamma, \overline{\mathrm{C}}_\Gamma$ are the dual operators for $\mathrm{D}_\Gamma, \mathrm{E}_\Gamma, \mathrm{C}_\Gamma$ representing distributed knowledge in the group Γ, everyone in Γ knows, and common knowledge among agents in Γ. Further, we use the (double) indexed modal operators $\mathcal{O}_c$, $\overline{\mathcal{O}}_c$, $\widehat{\mathrm{K}}_c^d$ and $\underline{\widehat{\mathrm{K}}}_c^d$ to represent the *correctly functioning circumstances of agent* c. The formula $\mathcal{O}_c \alpha$ stands for "for all the states where agent c is functioning correctly, α holds". The formula $\overline{\mathcal{O}}_c \alpha$ can be read as "there is a state where agent c is functioning correctly, and in which α holds". The formula $\widehat{\mathrm{K}}_c^d \alpha$ is read as "agent c knows that α under the assumption that agent d is functioning correctly". $\underline{\widehat{\mathrm{K}}}_c^d$ is the operator dual for the modality $\widehat{\mathrm{K}}_c^d$. We refer to [17] for a discussion of this notion; note that the operator $\overline{\mathcal{O}}_c$ is there referred to as $\mathcal{P}_c$.

Note that MTL is the sublogic of MTLKD which consists only of the formulae built without epistemic operators. EMTLKD is the existential fragment of MTLKD, defined by the following grammar: $\alpha ::= \mathbf{true} \mid \mathbf{false} \mid p \mid \neg p \mid \alpha \wedge \alpha \mid \alpha \vee \alpha \mid \mathrm{X}\alpha \mid \alpha \mathrm{U}_I \alpha \mid \mathrm{G}_I \alpha \mid \overline{\mathrm{K}}_c \alpha \mid \overline{\mathrm{D}}_\Gamma \alpha \mid \overline{\mathrm{E}}_\Gamma \alpha \mid \overline{\mathrm{C}}_\Gamma \alpha \mid \overline{\mathcal{O}}_c \alpha \mid \underline{\widehat{\mathrm{K}}}_c^d \alpha$.

Semantics of MTLKD. Let M be a model for $DIIS$. A *path* in M is an infinite sequence $\pi = (s_0, s_1, \ldots)$ of states such that $(s_m, s_{m+1}) \in T$ for each $m \in \mathbb{N}$. For a path π and $m \in \mathbb{N}$, we take $\pi(m) = s_m$, the m-th suffix of the path π is defined as standard: $\pi^m = (s_m, s_{m+1}, \ldots)$, and the m-th prefix of π also is defined in the standard way: $\pi[..m] = (s_0, s_1, \ldots, s_m)$. By $\Pi(s)$ we denote the set of all the paths starting at $s \in S$. For the group epistemic modalities we define the following. If $\Gamma \subseteq Ag$, then $\sim_\Gamma^E \stackrel{def}{=} \bigcup_{c \in \Gamma} \sim_c$, $\sim_\Gamma^C \stackrel{def}{=} (\sim_\Gamma^E)^+$ (the transitive closure of $\sim_\Gamma^E$), and $\sim_\Gamma^D \stackrel{def}{=} \bigcap_{c \in \Gamma} \sim_c$. Given the above, the semantics of MTLKD is the following.

Let I be an interval in $\mathbb{N}$ of the form: $[a,b)$ or $[a,\infty)$, and $m \in \mathbb{N}$. Then, $I + m \stackrel{df}{=} [a+m, b+m)$ if $I = [a,b)$, and $I+m \stackrel{df}{=} [a+m,\infty)$ if $I = [a,\infty)$. A MTLKD formula φ is true (valid) along the path π (in symbols $M, \pi \models \varphi$) iff $M, \pi^0 \models \varphi$, where

$M, \pi^m \models \mathbf{true}$, $\quad M, \pi^m \not\models \mathbf{false}$,
$M, \pi^m \models p$ iff $p \in \mathcal{V}(\pi(m))$, $\quad M, \pi^m \models \neg\alpha$ iff $M, \pi^m \not\models \alpha$,
$M, \pi^m \models \alpha \wedge \beta$ iff $M, \pi^m \models \alpha$ and $M, \pi^m \models \beta$,
$M, \pi^m \models \alpha \vee \beta$ iff $M, \pi^m \models \alpha$ or $M, \pi^m \models \beta$,
$M, \pi^m \models \mathrm{X}\alpha$ iff $M, \pi^{m+1} \models \alpha$,
$M, \pi^m \models \alpha \mathrm{U}_I \beta$ iff $(\exists i \geq m)[i \in I + m$ and $M, \pi^i \models \beta$ and $(\forall m \leq j < i) M, \pi^j \models \alpha]$,
$M, \pi^m \models \mathrm{G}_I \alpha$ iff $(\forall i \in I + m)[M, \pi^i \models \alpha]$,
$M, \pi^m \models \overline{\mathrm{K}}_c \alpha$ iff $(\exists \pi' \in \Pi(\iota))(\exists i \geqslant 0)[\pi(m) \sim_c \pi'(i)$ and $M, \pi'^i \models \alpha]$,
$M, \pi^m \models \overline{\mathcal{O}}_c \alpha$ iff $(\exists \pi' \in \Pi(\iota))(\exists i \geqslant 0)[\pi(m) \bowtie_c \pi'(i)$ and $M, \pi'^i \models \alpha]$,
$M, \pi^m \models \underline{\widehat{\mathrm{K}}}_c^d \alpha$ iff $(\exists \pi' \in \Pi(\iota))(\exists i \geqslant 0)[\pi(m) \sim_c \pi'(i)$ and $\pi(m) \bowtie_d \pi'(i)$ and $M, \pi'^i \models \alpha]$,
$M, \pi^m \models \overline{Y}_\Gamma \alpha$ iff $(\exists \pi' \in \Pi(\iota))(\exists i \geqslant 0)[\pi'(i) \sim_\Gamma^Y \pi(m)$ and $M, \pi'^i \models \alpha]$, where $Y \in \{\mathrm{D}, \mathrm{E}, \mathrm{C}\}$.

A MTLKD formula φ holds in the model M, denoted $M \models \varphi$, iff $M, \pi \models \varphi$ for all the paths $\pi \in \Pi(\iota)$. An EMTLKD formula φ holds in the model M, denoted $M \models^\exists \varphi$, iff $M, \pi \models \varphi$ for some path $\pi \in \Pi(\iota)$. The *existential model checking problem* asks whether $M \models^\exists \varphi$.

3 Bounded semantics for EMTLKD

The proposed bounded semantics is the backbone of the SAT-based BMC method for EMTLKD, which is presented in the next section. The temporal part of this semantics is based on the bounded semantics presented in [26]. As usual, we start by defining *k-paths* and *loops*. Let M be a model for $DIIS$, $k \in \mathbb{N}$, and $0 \leqslant l \leqslant k$. A *$k$-path* is a pair (π, l), also denoted by π_l, where π is a finite sequence $\pi = (s_0, \ldots, s_k)$ of states such that $(s_j, s_{j+1}) \in T$ for each $0 \leqslant j < k$. A k-path π_l is a *loop* if $l < k$ and $\pi(k) = \pi(l)$. Note that if a k-path π_l is a loop it represents the infinite path of the form uv^ω, where $u = (\pi(0), \ldots, \pi(l))$ and $v = (\pi(l+1), \ldots, \pi(k))$. We denote this unique path by $\varrho(\pi_l)$. Note that for each $j \in \mathbb{N}$, $\varrho(\pi_l)^{l+j} = \varrho(\pi_l)^{k+j}$. By $\Pi_k(s)$ we denote the set of all the k-paths starting at s in M.

As in the definition of semantics one needs to define the satisfiability relation on suffixes of k-paths, we denote by π_l^m the pair (π_l, m), i.e., the k-path π_l together with the designated starting point m, where $0 \leqslant m \leqslant k$.

Let $M, \pi_l^m \models_k \varphi$, where $0 \leqslant m \leqslant k$, denotes that the EMTLKD formula φ is k-true along the suffix $(\pi(m), \ldots, \pi(k))$ of π. For convenience, in the following definition we write $\pi_l^m \models_k \varphi$ instead of $M, \pi_l^m \models_k \varphi$. The relation $\models_k$ is defined inductively as follows:

$\pi_l^m \models_k \mathbf{true}$, $\quad \pi_l^m \not\models_k \mathbf{false}$,
$\pi_l^m \models_k p$ iff $p \in \mathcal{V}(\pi_l(m))$, $\quad \pi_l^m \models_k \neg p$ iff $p \notin \mathcal{V}(\pi_l(m))$,
$\pi_l^m \models_k \alpha \wedge \beta$ iff $\pi_l^m \models_k \alpha$ and $\pi_l^m \models_k \beta$, $\pi_l^m \models_k \alpha \vee \beta$ iff $\pi_l^m \models_k \alpha$ or $\pi_l^m \models_k \beta$,
$\pi_l^m \models_k \mathrm{X}\alpha$ iff $(m < k$ and $\pi_l^{m+1} \models_k \alpha)$ or $(m = k$ and $\pi(k) = \pi(l)$ and $\pi_l^{l+1} \models_k \alpha)$,

$\pi_l^m \models_k \alpha \mathrm{U}_I \beta$ iff $(\exists m \leqslant j \leqslant k)(j \in I + m$ and $M, \pi_l^j \models_k \beta$ and $(\forall m \leqslant i < j)$
$M, \pi_l^i \models_k \alpha)$ or $(l < m$ and $\pi(k) = \pi(l)$ and $(\exists l < j < m)$
$(j + k - l \in I + m$ and $\pi_l^j \models_k \beta$ and $(\forall l < i < j)\pi_l^i \models \alpha$
and $(\forall m \leqslant i \leqslant k)\pi_l^i \models_k \alpha))$,

$\pi_l^m \models_k \mathrm{G}_I \alpha$ iff $(k \geq right(I + m)$ and $(\forall j \in I + m)(\pi_l^j \models_k \alpha))$
or $(k < right(I + m)$ and $\pi(k) = \pi(l)$ and $(\forall max \leqslant j < k)$
$\pi_l^j \models_k \alpha$ and $(\forall l \leqslant j < max)(j + k - l \in I + m$ implies
$\pi_l^j \models_k \alpha))$, where $max = max(left(I + m), m)$

$\pi_l^m \models_k \overline{\mathrm{K}}_c \alpha$ iff $(\exists \pi'_{ll} \in \Pi_k(\iota))(\exists 0 \leqslant j \leqslant k)({\pi'_{ll}}^j \models_k \alpha$ and $\pi(m) \sim_c \pi'(j))$,

$\pi_l^m \models_k \overline{Y}_\Gamma \alpha$ iff $(\exists \pi'_{ll} \in \Pi_k(\iota))(\exists 0 \leqslant j \leqslant k)({\pi'_{ll}}^j \models_k \alpha$ and $\pi(m) \sim_\Gamma^Y \pi'(j))$,

$\pi_l^m \models_k \overline{\mathcal{O}}_c \alpha$ iff $(\exists \pi'_{ll} \in \Pi_k(\iota))(\exists 0 \leqslant j \leqslant k)({\pi'_{ll}}^j \models_k \alpha$ and $\pi(m) \bowtie_c \pi'(j))$,

$\pi_l^m \models_k \widehat{\underline{\mathrm{K}}}_c^d \alpha$ iff $(\exists \pi'_{ll} \in \Pi_k(\iota))(\exists 0 \leqslant j \leqslant k)({\pi'_{ll}}^j \models_k \alpha$ and $\pi(m) \sim_c \pi'(j)$
and $\pi(m) \bowtie_d \pi'(j))$.

Let M be a model, and φ an EMTLKD formula. We use the following notations: $M \models_k^\exists \varphi$ iff $M, \pi_l \models_k \varphi$ for some $\pi_l \in \Pi_k(\iota)$. The *bounded model checking problem* asks whether there exists $k \in \mathbb{N}$ such that $M \models_k^\exists \varphi$.

Equivalence of the bounded and unbounded semantics. Now, we show that for some particular bound the bounded and unbounded semantics are equivalent.

Lemma 1. *Let M be a model, φ an EMTLKD formula, $k \geqslant 0$ a bound, π_l a k-path in M, and $0 \leqslant m \leqslant k$. Then, $M, \pi_l^m \models_k \varphi$ implies*

1. *if π_l is not a loop, then $M, \rho^m \models \varphi$ for each path $\rho \in M$ such that $\rho[..k] = \pi$.*
2. *if π_l is a loop, then $M, \varrho(\pi_l)^m \models \varphi$,*

Proof. (Induction on the length of φ) The lemma follows directly for the propositional variables and their negations. Assume that $M, \pi_l^m \models_k \varphi$ and consider the following cases:

1. $\varphi = \alpha \wedge \beta$. From $M, \pi_l^m \models_k \varphi$ it follows that $M, \pi_l^m \models_k \alpha$ and $M, \pi_l^m \models_k \beta$. Suppose first that π_l is not a loop. By induction we have that for each path ρ in M such that $\rho[..k] = \pi$, $M, \rho^m \models \alpha$ and $M, \rho^m \models \beta$. Hence, for every path ρ in M such that $\rho[..k] = \pi$, $M, \rho^m \models \alpha \wedge \beta$.
   Now suppose that π_l is a loop. By induction we have that $M, \varrho(\pi_l)^m \models \alpha$ and $M, \varrho(\pi_l)^m \models \beta$. Hence, $M, \varrho(\pi_l)^m \models \alpha \wedge \beta$.
2. $\varphi = \alpha \vee \beta$. From $M, \pi_l^m \models_k \varphi$ it follows that $M, \pi_l^m \models_k \alpha$ or $M, \pi_l^m \models_k \beta$. Suppose first that π_l is not a loop. By inductive hypothesis, for each path ρ in M such that $\rho[..k] = \pi$, $M, \rho^m \models \alpha$ or $M, \rho^m \models \beta$. Hence, for every path ρ in M such that $\rho[..k] = \pi$, $M, \rho^m \models \alpha \vee \beta$.
   Now suppose that π_l is a loop. By inductive hypothesis, $M, \varrho(\pi_l)^m \models \alpha$ or $M, \varrho(\pi_l)^m \models \beta$. Hence, $M, \varrho(\pi_l)^m \models \alpha \vee \beta$.
3. $\varphi = \mathrm{X}\alpha$. Suppose first that π_l is not a loop. Then $m < k$ and $M, \pi_l^{m+1} \models_k \alpha$. By inductive hypothesis, for every path ρ in M such that $\rho[..k] = \pi$, $M, \rho^{m+1} \models \alpha$. Hence, for every path ρ in M such that $\rho[..k] = \pi$, $M, \rho^m \models \varphi$.
   Now suppose that π_l is a loop. Then either $m < k$ and $M, \pi_l^{m+1} \models_k \alpha$ or $m = k$ and $\pi(k) = \pi(l)$ and $M, \pi_l^{l+1} \models_k \alpha$. By the inductive hypothesis it follows

that either $m < k$ and $M, \varrho(\pi_l)^{m+1} \models \alpha$ or $m = k$ and $M, \varrho(\pi_l)^{l+1} \models \alpha$. Since $\varrho(\pi_l)^{k+1} = \varrho(\pi_l)^{l+1}$, from $M, \varrho(\pi_l)^{l+1} \models \alpha$ we get $M, \varrho(\pi_l)^{k+1} \models \alpha$. Eventually, either $m < k$ and $M, \varrho(\pi_l)^{m+1} \models \alpha$ or $m = k$ and $M, \varrho(\pi_l)^{k+1} \models \alpha$. Hence, $M, \varrho(\pi_l)^{m+1} \models \alpha$. Thus, $M, \varrho(\pi_l)^m \models \varphi$.

4. $\varphi = \alpha \mathrm{U}_I \beta$. Assume that π_l is not a loop. Then $(\exists m \leqslant j \leqslant k)(j \in I + m$ and $M, \pi_l^j \models_k \beta$ and $(\forall m \leqslant i < j)(M, \pi_l^i \models_k \alpha)$. By inductive hypothesis, for every path ρ in M such that $\rho[..k] = \pi$, $(\exists m \leqslant j \leqslant k)(j \in I + m$ and $M, \rho^j \models \beta$ and $(\forall m \leqslant i < j) M, \rho^i \models \alpha)$. Thus, for every path ρ in M such that $\rho[..k] = \pi$, $M, \rho^m \models \varphi$.
   Now assume that π_l is a loop. Then $l < m$ and $\pi(k) = \pi(l)$ and $(\exists l < j < m)(j + k - l \in I + m$ and $M, \pi_l^j \models_k \beta$ and $(\forall l < i < j) M, \pi_l^i \models \alpha$ and $(\forall m \leqslant i \leqslant k)$ $M, \pi_l^i \models_k \alpha)$. By inductive hypothesis, $(\exists l < j < m)(j + k - l \in I + m$ and $M, \varrho(\pi_l)^j \models \beta$ and $(\forall l < i < j) M, \varrho(\pi_l)^i \models \alpha$ and $(\forall m \leqslant i \leqslant k) M, \varrho(\pi_l)^i \models \alpha)$. Since for each $n \in \mathbb{N}$, $\varrho(\pi_l)^{l+n} = \varrho(\pi_l)^{k+n}$, it follows that $M, \varrho(\pi_l)^{j+k-l} \models \beta$ and $(\forall k < i < j + k - l)(M, \varrho(\pi_l)^i \models \alpha)$ and $(\forall m \leqslant i \leqslant k)(M, \varrho(\pi_l)^i \models \alpha)$. Hence, $\varrho(\pi_l)^{j+k-l} \models \beta$ and $(\forall m \leqslant i < j + k - l)(M, \varrho(\pi_l)^i \models \alpha)$. Thus, $M, \varrho(\pi_l)^m \models \varphi$.
5. $\varphi = \mathrm{G}_I \alpha$. Assume that π_l is not a loop. Then $k \geq \mathit{right}(I + m)$ and $(\forall j \in I + m)$ $(M, \pi_l^j \models_k \alpha)$. By inductive hypothesis, for every path ρ in M such that $\rho[..k] = \pi$, $(\forall j \in I + m)(M, \rho^j \models \alpha)$. Thus, for every path ρ in M such that $\rho[..k] = \pi$, $M, \rho^m \models \varphi$.
   Now assume that π_l is a loop, and $max = max(\mathit{left}(I + m), m)$. Then, $k < \mathit{right}(I + m)$ and $\pi(k) = \pi(l)$ and $(\forall max \leqslant j < k) M, \pi_l^j \models_k \alpha$ and $(\forall l \leqslant j < max)(j + k - l \in I + m$ implies $M, \pi_l^j \models_k \alpha)$. By inductive hypothesis, $(\forall max \leqslant j < k)$ $M, \varrho(\pi_l)^j \models \alpha$ and $(\forall l \leqslant j < max)(j + k - l \in I + m$ implies $M, \varrho(\pi_l)^j \models \alpha)$. Since for each $n \in \mathbb{N}$, $\varrho(\pi_l)^{l+n} = \varrho(\pi_l)^{k+n}$, it follows that $(\forall n \in \mathbb{N})(\forall j \geqslant l + n)(j + k - l \in I + m$ implies $M, \varrho(\pi_l)^j \models \alpha)$. Thus, $M, \varrho(\pi_l)^m \models \varphi$.
6. $\varphi = \overline{\mathrm{K}}_c \alpha$. From $M, \pi_l^m \models_k \varphi$ it follows that $(\exists \pi'_{ll} \in \mathit{\Pi}_k(\iota))(\exists 0 \leqslant j \leqslant k)$ $(M, {\pi'_{ll}}^j \models_k \alpha$ and $\pi(m) \sim_c \pi'(j))$. Assume that both π_l and π'_{ll} are not loops. By inductive hypothesis, for every path ρ' in M such that $\rho'[..k] = \pi'$, $(\exists 0 \leqslant j \leqslant k)$ $(M, {\rho'}^j \models \alpha$ and $\pi(m) \sim_c \rho'(j))$. Further, for every path ρ in M such that $\rho[..k] = \pi$, we have that $\rho(m) \sim_c \rho'(j))$. Thus, for every path ρ in M such that $\rho[..k] = \pi$, $M, \rho^m \models \varphi$.
   Now assume that π'_{ll} is not a loop, and π_l is a loop. By inductive hypothesis, for every path ρ' in M such that $\rho'[..k] = \pi'$, $(\exists 0 \leqslant j \leqslant k)(M, {\rho'}^j \models \alpha$ and $\pi(m) \sim_c \rho'(j))$. Further, observe that $\varrho(\pi_l)(m) = \pi(m)$, thus $M, \varrho(\pi_l)^m \models \varphi$.
   Now assume that both π_l and π'_{ll} are loops. By inductive hypothesis, $(\exists 0 \leqslant j \leqslant k)$ $(M, \varrho(\pi'_{ll})^j \models \alpha$ and $\pi(m) \sim_c \varrho(\pi'_{ll})(j))$. Further, observe that $\varrho(\pi_l)(m) = \pi(m)$, thus $M, \varrho(\pi_l)^m \models \varphi$.
   Now assume that π'_{ll} is a loop, and π_l is not a loop. By inductive hypothesis, $(\exists 0 \leqslant j \leqslant k)$ $(M, \varrho(\pi'_{ll})^j \models \alpha$ and $\pi(m) \sim_c \varrho(\pi'_{ll})(j))$. Further, for every path ρ in M such that $\rho[..k] = \pi$, we have that $\rho(m) \sim_c \varrho(\pi'_{ll})(j))$. Thus, for every path ρ in M such that $\rho[..k] = \pi$, $M, \rho^m \models \varphi$.

7. Let $\varphi = \overline{Y}_\Gamma \alpha$, where $Y \in \{\mathrm{D}, \mathrm{E}, \mathrm{C}\}$, or $\varphi = \overline{\mathcal{O}}_c \alpha$, or $\varphi = \underline{\widehat{\mathrm{K}}}_c^d \alpha$. These cases can be proven analogously to the case 6.

Lemma 2. *Given an LTL formula α and a model M. Then, the following implication holds: if $M \models^\exists \alpha$, then there exists $k \leqslant |M| \cdot |\alpha| \cdot 2^{|\alpha|}$ with $M \models_k^\exists \alpha$.*

Proof. In [9] it is shown that the existential model checking problem for an LTL formula α can be reduced to checking for the emptiness of the product P of the original model and the Büchi automaton with at most $|\alpha| \cdot 2^{|\alpha|}$ states. So, if the LTL formula α is existentially valid in M, then there exists a path in the product P that starts with an initial state and ends with a cycle in the strongly connected component of accepting states. This path can be chosen to be a loop with k bounded by $|M| \cdot |\alpha| \cdot 2^{|\alpha|}$, which is the size of the product P. If we project this path onto its first component, the original model, then we get a path of the length k that is a loop and in addition fulfils α. By definition of the bounded semantics this also implies $M \models_k^\exists \alpha$.

Lemma 3. *Given are a model M, an EMTLKD formula φ, and a path π. Then the following implication holds: $M, \pi \models \varphi$ implies that there exists $k \geqslant 0$ such that $M, \pi \models_k \varphi$.*

Proof. In [8] it was shown that MTL is simply LTL, but with an exponentially succinct encoding. In [10] it was shown that it is possible to reduce the CKL_n model checking problem to the LTL model checking problem. The reduction is based on Proposition 1 of [6], which states that each epistemic modality (K_c, E_Γ, C_Γ, and D_Γ) is expressible in the Logic of Local Propositions. The CKL_n is the LTL logic augmented by an indexed set of modal operators K_c with their diamonds $\overline{\mathrm{K}}_c$, one for each agent $c \in Ag$, and common knowledge operators C_Γ with their diamonds $\overline{\mathrm{C}}_\Gamma$, where $\Gamma \subseteq Ag$.

Now, note that EMTLK language is an "epistemically existential" fragment of CKL_n augmented by the diamonds for D_Γ and E_Γ, representing distributed knowledge in the group Γ, and "everyone in Γ knows". Thus, to prove that the EMTLK model checking problem can be reduced to the LTL model checking problem, it is enough to observe that $\overline{\mathrm{D}}_\Gamma \alpha = \bigwedge_{c \in \Gamma} \overline{\mathrm{K}}_c \alpha$ and $\overline{\mathrm{E}}_\Gamma \alpha = \bigvee_{c \in \Gamma} \overline{\mathrm{K}}_c \alpha$. Further, the EMTLKD language is a deontic extension of EMTLK, which augments EMTLK by the diamonds for the deontic modalities $\mathcal{O}_c$ and $\widehat{\mathrm{K}}_c^d$. Thus, to prove that the EMTLKD model checking problem can be reduced to the LTL model checking problem, it is enough to express deontic modalities in the Logic of Local Propositions. This can be done in a similar way as presented in [10]. Consequently, by using Lemma 2, we can conclude that $M, \pi \models \varphi$ implies $M, \pi \models_k \varphi$ for some $k \geqslant 0$.

The following theorem states that for a given model and formula there exists a bound k such that the model checking problem ($M \models^\exists \varphi$) can be reduced to the bounded model checking problem ($M \models_k^\exists \varphi$).

Theorem 1. *Let M be a model and φ an EMTLKD formula. Then, the following equivalence holds: $M \models^\exists \varphi$ iff there exists $k \geqslant 0$ such that $M \models_k^\exists \varphi$.*

Proof. ("$\Rightarrow$"). This implication follows directly from Lemma 3. ("$\Leftarrow$"). This implication follows directly from Lemma 1.

Further, by straightforward induction on the length of an EMTLKD formula φ, we can show that φ is k-true in M if and only if φ is k-true in M with a number of k-paths reduced to $f_k(\varphi)$, where the function f_k : EMTLKD $\rightarrow \mathbb{N}$ gives a bound on the number of k-paths, which are sufficient to validate a given EMTLKD formula.

In the definition of f_k we assume that each EMTLKD formula is preceded by the "path" quantifier E with the meaning "there exists a path in $\Pi_k(\iota)$"; this assumption is only technical and it makes the definition of f_k easy to implement. Note that in the BMC method we deal with the existential validity ($\models^{\exists}$) only, so the above assumption is just another way to express this fact. More precisely, let φ be an EMTLKD formula. To calculate the value of $f_k(\varphi)$, we first extend the formula φ to the formula $\varphi' = \mathrm{E}\varphi$. Next, we calculate the value of f_k for φ' in the following way: $f_k(\mathrm{E}\varphi) = f_k(\varphi) + 1$; if $p \in \mathcal{PV}$, then $f_k(\mathbf{true}) = f_k(\mathbf{false}) = f_k(p) = f_k(\neg p) = 0$; $f_k(\alpha \wedge \beta) = f_k(\alpha) + f_k(\beta)$; $f_k(\alpha \vee \beta) = max\{f_k(\alpha), f_k(\beta)\}$; $f_k(\mathrm{X}\alpha) = f_k(\alpha)$; $f_k(\alpha \mathrm{U}_I \beta) = k \cdot f_k(\alpha) + f_k(\beta)$; $f_k(\mathrm{G}_I\alpha) = (k+1) \cdot f_k(\alpha)$; $f_k(\overline{\mathrm{C}}_I\alpha) = f_k(\alpha) + k$; $f_k(Y\alpha) = f_k(\alpha) + 1$ for $Y \in \{\overline{\mathrm{K}}_c, \overline{\mathcal{O}}_c, \underline{\widehat{\mathrm{K}}}^d_c, \overline{\mathrm{D}}_\Gamma, \overline{\mathrm{E}}_\Gamma\}$.

4 SAT-based BMC for EMTLKD

Let $M = (\iota, S, T, \{\sim_c\}_{c \in Ag}, \{\bowtie_c\}_{c \in Ag}, \mathcal{V})$ be a model, φ an EMTLKD formula, and $k \geqslant 0$ a bound. The proposed BMC method is based on the BMC encoding presented in [26], and it consists in translating the problem of checking whether $M \models_k \varphi$ holds, to the problem of checking the satisfiability of the propositional formula $[M, \varphi]_k := [M^{\varphi,\iota}]_k \ \wedge \ [\varphi]_{M,k}$. The formula $[M^{\varphi,\iota}]_k$ encodes sets of k-paths of M, whose size equals to $f_k(\varphi)$, and in which at least one path starts at the initial state of the model M. The formula $[\varphi]_{M,k}$ encodes a number of constraints that must be satisfied on these sets of k-paths for φ to be satisfied. Note that our translation, like the translation from [26], does not require that either all the k-paths used in the translation are loops or none is a loop. Once this translation is defined, checking satisfiability of an EMTLKD formula can be done by means of a SAT-solver.

In order to define the formula $[M, \varphi]_k$ we proceed as follows. We begin with an encoding of states of the given model M. Since the set of states of M is finite, each state s of M can be encoded by a bit-vector, whose length r depends on the number of agents' local states. Thus, each state s of M can be represented by a vector $w = (\mathtt{w}_1, \ldots, \mathtt{w}_r)$ (called a *symbolic state*) of propositional variables (called *state variables*). The set of all the propositional state variables we will denote by SV.

Since any k-path (π, l) is a pair consisting of a finite sequence of states of length k and a number $l < k$, to encode it by propositional formula, it suffices to take a finite sequence of symbolic states of length k and a formula that encodes the position $l < k$. The designated position l can be encoded as a bit vector of the length $t = max(1, \lceil log_2(k+1) \rceil)$. Thus, the position l can be represented by a valuation of a vector $u = (\mathtt{u}_1, \ldots, \mathtt{u}_t)$ (called a *symbolic number*) of propositional variables (called *propositional natural variables*), which not appear among propositional state variables. The set of all the propositional natural variables we will denote by NV, and we assume that $SV \cap NV = \emptyset$. Given the above we can define a *symbolic k-path* as a pair $((w_0, \ldots, w_k), u)$ consisting of a finite sequence of symbolic states

of length k and a symbolic number. Since in general we may need to consider more than one symbolic k-path, therefore we introduce a notion of the j-th symbolic k-path $\boldsymbol{\pi}_j = ((w_{0,j}, \ldots, w_{k,j}), u_j)$, where $w_{i,j}$ are symbolic states for $0 \leqslant j < f_k(\varphi)$ and $0 \leqslant i \leqslant k$, and u_j is a symbolic number for $0 \leqslant j < f_k(\varphi)$. Note that the exact number of symbolic k-paths depends on the checked formula φ, and it can be calculated by means of the function f_k.

Let $PV = SV \cup NV$, and $V : PV \to \{0,1\}$ be a *valuation of propositional variables* (a *valuation* for short). Each valuation induces the functions $\mathbf{S} : SV^r \to \{0,1\}^r$ and $\mathbf{J} : NV^t \to \mathbb{N}$ defined in the following way:

$$\mathbf{S}((\mathtt{w}_1, \ldots, \mathtt{w}_r)) = (V(\mathtt{w}_1), \ldots, V(\mathtt{w}_r)),\ \mathbf{J}((\mathtt{u}_1, \ldots, \mathtt{u}_t)) = \textstyle\sum_{i=1}^{t} V(\mathtt{u}_i) \cdot 2^{i-1}.$$

Moreover, for a symbolic state w and a symbolic number u, by $SV(w)$ and $NV(u)$ we denote, respectively, the set of all the state variables occurring in w, and the set of all the natural variables occurring in u. Next, let w and w' be two symbolic states such that $SV(w) \cap SV(w') = \emptyset$, and u be a symbolic number. We define the following auxiliary propositional formulae:

- $I_s(w)$ is a formula over $SV(w)$ that is true for a valuation V iff $\mathbf{S}(w) = s$.
- $p(w)$ is a formula over $SV(w)$ that is true for a valuation V iff $p \in \mathcal{V}(\mathbf{S}(w))$ (encodes a set of states of M in which $p \in \mathcal{PV}$ holds).
- $H(w, w')$ is a formula over $SV(w) \cup SV(w')$ that is true for a valuation V iff $\mathbf{S}(w) = \mathbf{S}(w')$ (encodes equality of two global states).
- $H_c(w, w')$ is a formula over $SV(w) \cup SV(w')$ that is true for a valuation V iff $l_c(\mathbf{S}(w)) = l_c(\mathbf{S}(w))$ (encodes equality of local states of agent c).
- $HO_c(w, w')$ is a formula over $SV(w) \cup SV(w')$ that is true for a valuation V iff $l_c(\mathbf{S}(w')) \in \mathcal{G}_c$ (encodes an accessibility of a global state in which agent c is functioning correctly).
- $\widehat{H}_c^d(w, w') := H_c(w, w') \wedge HO_d(w, w')$.
- $\mathcal{T}(w, w')$ is a formula over $SV(w) \cup SV(w')$ that is true for a valuation V iff $(\mathbf{S}(w), \mathbf{S}(w')) \in T$ (encodes the transition relation of M).
- $\mathcal{B}_j^{\sim}(u)$ is a formula over $NV(u)$ that is true for a valuation V iff $j \sim \mathbf{J}(u)$, where $\sim \in \{<, \leqslant, =, \geqslant, >\}$,
- $\mathcal{L}_k^l(\boldsymbol{\pi}_j) := \mathcal{B}_k^{>}(u_j) \wedge H(w_{k,j}, w_{l,j})$.
- Let $j \in \mathbb{N}$, and I be an interval. Then $In(j, I) := \mathbf{true}$ if $j \in I$, and $In(j, I) := \mathbf{false}$ if $j \notin I$.

Let $W = \{SV(w_{i,j}) \mid 0 \leq i \leq k \text{ and } 0 \leqslant j < f_k(\varphi)\} \cup \{NV(u_j) \mid 0 \leqslant j < f_k(\varphi)\}$ be a set of propositional variables. The propositional formula $[M^{\varphi,\iota}]_k$ is defined over the set W in the following way:

$$[M^{\varphi,\iota}]_k := I_\iota(w_{0,0}) \wedge \bigwedge_{j=0}^{f_k(\varphi)-1} \bigwedge_{i=0}^{k-1} \mathcal{T}(w_{i,j}, w_{i+1,j}) \wedge \bigwedge_{j=0}^{f_k(\varphi)-1} \bigvee_{l=0}^{k} B_l^{=}(u_j)$$

The next step of the reduction to SAT is the transformation of an EMTLKD formula φ into a propositional formula $[\varphi]_{M,k} := [\varphi]_k^{[0,0,F_k(\varphi)]}$, where $F_k(\varphi) = \{j \in \mathbb{N} \mid 0 \leqslant j < f_k(\varphi)\}$, and $[\varphi]_k^{[m,n,A]}$ denotes the translation of φ along the symbolic path $\boldsymbol{\pi}_{m,n}$ with starting point m by using the set A.

For every EMTLKD formula φ the function f_k determines how many symbolic k-paths are needed for translating the formula φ. Given a formula φ and a set A of k-paths such that $|A| = f_k(\varphi)$, we divide the set A into subsets needed for translating the subformulae of φ. To accomplish this goal we need some auxiliary functions that were defined in [26]. We recall the definitions of these functions. First, the relation $\prec$ is defined on the power set of $\mathbb{N}$ as follows: $A \prec B$ iff for all natural numbers x and y, if $x \in A$ and $y \in B$, then $x < y$.

Now, let $A \subset \mathbb{N}$ be a finite nonempty set, and $n, d \in \mathbb{N}$, where $d \leqslant |A|$. Then,

- $g_l(A, d)$ denotes the subset B of A such that $|B| = d$ and $B \prec A \setminus B$.
- $g_r(A, d)$ denotes the subset C of A such that $|C| = d$ and $A \setminus C \prec C$.
- $g_s(A)$ denotes the set $A \setminus \{min(A)\}$.
- if n divides $|A| - d$, then $hp(A, d, n)$ denotes the sequence $(B_0, \ldots, B_n)$ of subsets of A such that $\bigcup_{j=0}^{n} B_j = A$, $|B_0| = \ldots = |B_{n-1}|$, $|B_n| = d$, and $B_i \prec B_j$ for every $0 \leqslant i < j \leqslant n$.

Now let $h_k^{\mathrm{U}}(A, d) \stackrel{df}{=} hp(A, d, k)$ and $h_k^{\mathrm{G}}(A) \stackrel{df}{=} hp(A, 0, k+1)$. Note that if $h_k^{\mathrm{U}}(A, d) = (B_0, \ldots, B_k)$, then $h_k^{\mathrm{U}}(A, d)(j)$ denotes the set B_j, for every $0 \leqslant j \leqslant k$. Similarly, if $h_k^{\mathrm{G}}(A) = (B_0, \ldots, B_{k+1})$, then $h_k^{\mathrm{G}}(A)(j)$ denotes the set B_j, for every $0 \leqslant j \leqslant k+1$.

The functions g_l and g_r are used in the translation of the formulae with the main connective being either conjunction or disjunction: for a given EMTLKD formula $\alpha \wedge \beta$, if a set A is to be used to translate this formula, then the set $g_l(A, f_k(\alpha))$ is used to translate the subformula α and the set $g_r(A, f_k(\beta))$ is used to translate the subformula β; for a given EMTLKD formula $\alpha \vee \beta$, if a set A is to be used to translate this formula, then the set $g_l(A, f_k(\alpha))$ is used to translate the subformula α and the set $g_l(A, f_k(\beta))$ is used to translate the subformula β.

The function g_s is used in the translation of the formulae with the main connective $\mathrm{Q} \in \{\overline{\mathrm{K}}_c, \widehat{\underline{\mathrm{K}}}_c^j, \mathcal{O}_c, \overline{\mathrm{D}}_\Gamma, \overline{\mathrm{E}}_\Gamma\}$: for a given EMTLKD formula $\mathrm{Q}\alpha$, if a set A is to be used to translate this formula, then the path of the number $min(A)$ is used to translate the operator Q and the set $g_s(A)$ is used to translate the subformula α.

The function h_k^{U} is used in the translation of subformulae of the form $\alpha \mathrm{U}_I \beta$: if a set A is to be used to translate the subformula $\alpha \mathrm{U}_I \beta$ at the symbolic k-path $\boldsymbol{\pi}_n$ (with starting point m), then for every j such that $m \leqslant j \leqslant k$, the set $h_k^{\mathrm{U}}(A, f_k(\beta))(k)$ is used to translate the formula β along the symbolic path $\boldsymbol{\pi}_n$ with starting point j; moreover, for every i such that $m \leqslant i < j$, the set $h_k^{\mathrm{U}}(A, f_k(\beta))(i)$ is used to translate the formula α along the symbolic path $\boldsymbol{\pi}_n$ with starting point i. Notice that if k does not divide $|A| - d$, then $h_k^{\mathrm{U}}(A, d)$ is undefined. However, for every set A such that $|A| = f_k(\alpha \mathrm{U}_I \beta)$, it is clear from the definition of f_k that k divides $|A| - f_k(\beta)$.

The function h_k^{G} is used in the translation of subformulae of the form $\mathrm{G}_I \alpha$: if a set A is to be used to translate the subformula $\mathrm{G}_I \alpha$ along a symbolic k-path $\boldsymbol{\pi}_n$ (with starting point m), then for every j such that $m \leqslant j \leqslant k$ and $j \in I$, the set $h_k^{\mathrm{G}}(A)(j)$, is used to translate the formula α along the symbolic paths $\boldsymbol{\pi}_n$ with starting point j; Notice that if $k+1$ does not divide $|A|$, then $h_k^{\mathrm{G}}(A)$ is undefined. However, for every set A such that $|A| = f_k(\mathrm{G}_I \alpha)$, it is clear from the definition of f_k that $k+1$ divides $|A|$.

Let φ be an EMTLKD formula, and $k \geqslant 0$ a bound. We can define inductively the translation of φ over path number $n \in F_k(\varphi)$ starting at symbolic state $w_{m,n}$ as shown below. Let $min(A) = A'$, then:

$$
\begin{array}{ll}
[\mathbf{true}]_k^{[m,n,A]} & := \mathbf{true}, \qquad [\mathbf{false}]_k^{[m,n,A]} := \mathbf{false},\\
[p]_k^{[m,n,A]} & := p(w_{m,n}), \qquad [\neg p]_k^{[m,n,A]} := \neg p(w_{m,n}),\\
[\alpha \wedge \beta]_k^{[m,n,A]} & := [\alpha]_k^{[m,n,g_l(A,f_k(\alpha))]} \wedge [\beta]_k^{[m,n,g_r(A,f_k(\beta))]},\\
[\alpha \vee \beta]_k^{[m,n,A]} & := [\alpha]_k^{[m,n,g_l(A,f_k(\alpha))]} \vee [\beta]_k^{[m,n,g_l(A,f_k(\beta))]},\\
[\mathrm{X}\alpha]_k^{[m,n,A]} & := [\alpha]_k^{[m+1,n,A]}, \text{ if } m < k\\
 & \quad \textstyle\bigvee_{l=0}^{k-1}(\mathcal{L}_k^l(\boldsymbol{\pi}_n) \wedge [\alpha]_k^{[l+1,n,A]}), \text{ if } m = k\\
[\alpha \mathrm{U}_I \beta]_k^{[m,n,A]} & := \textstyle\bigvee_{j=m}^{k}(In(j,I+m) \wedge [\beta]_k^{[j,n,h_k^{\mathrm{U}}(A,f_k(\beta))(k)]} \wedge \bigwedge_{i=m}^{j-1}[\alpha]_k^{[i,n,h_k^{\mathrm{U}}(A,f_k(\beta))(i)]})\\
 & \quad \vee(\textstyle\bigvee_{l=0}^{m-1}(\mathcal{L}_k^l(\boldsymbol{\pi}_n)) \wedge \bigvee_{j=0}^{m-1}(\mathcal{B}_j^{>}(u_n) \wedge [\beta]_k^{[j,n,h_k^{\mathrm{U}}(A,f_k(\beta))(k)]} \wedge\\
 & \quad (\textstyle\bigvee_{l=0}^{m-1}(\mathcal{B}_l^{=}(u_n) \wedge In(j+k-l,I+m))) \wedge\\
 & \quad \textstyle\bigwedge_{i=0}^{j-1}(\mathcal{B}_i^{>}(u_n) \rightarrow [\alpha]_k^{[i,n,h_k^{\mathrm{U}}(A,f_k(\beta))(i)]}) \wedge \bigwedge_{i=m}^{k}[\alpha]_k^{[i,n,h_k^{\mathrm{U}}(A,f_k(\beta))(i)]})),\\
[\mathrm{G}_I\alpha]_k^{[m,n,A]} & := \textstyle\bigwedge_{j=max(left(I+m),m)}^{right(I+m)}[\alpha]_k^{[j,n,h_k^{\mathrm{G}}(A)(j)]}, \text{ if } right(I+m) \leqslant k\\
 & \quad \textstyle\bigvee_{l=0}^{k-1}(\mathcal{L}_k^l(\boldsymbol{\pi}_n)) \wedge \bigwedge_{j=max(left(I+m),m)}^{k-1}[\alpha]_k^{[j,n,h_k^{\mathrm{G}}(A)(j)]} \wedge\\
 & \quad \textstyle\bigwedge_{j=0}^{max(left(I+m),m)-1}((\mathcal{B}_j^{\geqslant}(u_n) \wedge (\bigvee_{l=0}^{max(left(I+m),m)-1}(\mathcal{B}_l^{=}(u_n) \wedge\\
 & \quad In(j+k-l,I+m)))) \rightarrow [\alpha]_k^{[j,n,h_k^{\mathrm{G}}(A)(j)]}), \text{ otherwise}\\
[\overline{\mathrm{K}}_c\alpha]_k^{[m,n,A]} & := I_\iota(w_{0,A'}) \wedge \textstyle\bigvee_{j=0}^{k}([\alpha]_k^{[j,A',g_s(A)]} \wedge H_c(w_{m,n},w_{j,A'})),\\
[\overline{\mathcal{O}}_c\alpha]_k^{[m,n,A]} & := I_\iota(w_{0,A'}) \wedge \textstyle\bigvee_{j=0}^{k}([\alpha]_k^{[j,A',g_s(A)]} \wedge HO_c(w_{m,n},w_{j,A'})),\\
[\underline{\widehat{\mathrm{K}}}_c^d\alpha]_k^{[m,n,A]} & := I_\iota(w_{0,A'}) \wedge \textstyle\bigvee_{j=0}^{k}([\alpha]_k^{[j,A',g_s(A)]} \wedge \widehat{H}_c^d(w_{m,n},w_{j,A'})),\\
[\overline{\mathrm{D}}_\Gamma\alpha]_k^{[m,n,A]} & := I_\iota(w_{0,A'}) \wedge \textstyle\bigvee_{j=0}^{k}([\alpha]_k^{[j,A',g_s(A)]} \wedge \bigwedge_{c\in\Gamma} H_c(w_{m,n},w_{j,A'})),\\
[\overline{\mathrm{E}}_\Gamma\alpha]_k^{[m,n,A]} & := I_\iota(w_{0,A'}) \wedge \textstyle\bigvee_{j=0}^{k}([\alpha]_k^{[j,A',g_s(A)]} \wedge \bigvee_{c\in\Gamma} H_c(w_{m,n},w_{j,A'})),\\
[\overline{\mathrm{C}}_\Gamma\alpha]_k^{[m,n,A]} & := [\textstyle\bigvee_{j=1}^{k}(\overline{\mathrm{E}}_\Gamma)^j\alpha]_k^{[m,n,A]}.
\end{array}
$$

Now, let α be an EMTLKD formula. For every subformula φ of α, we denote by $[\varphi]_k^{[\alpha,m,n,A]}$ the propositional formula $[M]_k^{F_k(\alpha)} \wedge [\varphi]_k^{[m,n,A]}$, where $[M]_k^{F_k(\alpha)} = \bigwedge_{j=0}^{f_k(\alpha)-1}\bigwedge_{i=0}^{k-1}\mathcal{T}(w_{i,j},w_{i+1,j}) \wedge \bigwedge_{j=0}^{f_k(\alpha)-1}\bigvee_{l=0}^{k}B_l^{=}(u_j)$. We write $V \Vdash \xi$ to mean that the valuation V satisfies the propositional formula ξ. Moreover, we write $s_{i,j}$ instead of $\mathbf{S}(w_{i,j})$, and l_j instead of $\mathbf{J}(u_j)$.

The lemmas below state the correctness and the completeness of the presented translation respectively.

Lemma 4 (Correctness of the translation). *Let M be a model, α an EMTLKD formula, and $k \in \mathbb{N}$. For every subformula φ of the formula α, every $(m,n) \in \{0,\ldots,k\}\times F_k(\alpha)$, every $A \subseteq F_k(\alpha) \setminus \{n\}$ such that $|A| = f_k(\varphi)$, and every valuation V, the following condition holds: $V \Vdash [\varphi]_k^{[\alpha,m,n,A]}$ implies $M,((s_{0,n},\ldots,s_{k,n}),l_n)^m \models_k \varphi$.*

Lemma 5 (Completeness of the translation). *Let M be a model, $k \in \mathbb{N}$, and α an EMTLKD formula such that $f_k(\alpha) > 0$. For every subformula φ of the formula α, every $(m,n) \in \{(0,0)\}\cup\{0,\ldots,k\}\times F_k(\alpha)$, every $A \subseteq F_k(\alpha)\setminus\{n\}$ such that $|A| = f_k(\varphi)$, and every k-path π_l, the following condition holds: $M,\pi_l^m \models_k \varphi$ implies that there exists a valuation V such that $\pi_l = ((s_{0,n},\ldots,s_{k,n}),l_n)$ and $V \Vdash [\varphi]_k^{[\alpha,m,n,A]}$.*

Theorem 2. *Let M be a model, and φ an EMTLKD formula. Then for every $k \in \mathbb{N}$, $M \models_k^{\exists} \varphi$ if, and only if, the propositional formula $[M, \varphi]_k$ is satisfiable.*

Proof. ($\Longrightarrow$) Let $k \in \mathbb{N}$ and $M, \pi_l \models_k \varphi$ for some $\pi_l \in \Pi_k(\iota)$. By Lemma 5 it follows that there exists a valuation V such that $\pi_l = ((s_{0,0}, \ldots, s_{k,0}), l_0)$ with $\mathbf{S}(w_{0,0}) = s_{0,0} = \iota$ and $V \Vdash [\varphi]_k^{[\varphi,0,0,F_k(\varphi)]}$. Hence, $V \Vdash I(w_{0,0}) \wedge [M]_k^{F_k(\varphi)} \wedge [\varphi]_k^{[0,0,F_k(\varphi)]}$. Thus $V \Vdash [M^{\varphi,\iota}]_k$.
($\Longleftarrow$) Let $k \in \mathbb{N}$ and $[M^{\varphi,\iota}]_k$ is satisfiable. It means that there exists a valuation V such that $V \Vdash [M^{\varphi,\iota}]_k$. So, $V \Vdash I(w_{0,0})$ and $V \Vdash [M]_k^{F_k(\varphi)} \wedge [\varphi]_k^{[0,0,F_k(\varphi)]}$. Hence, by Lemma 4 it follows that $M, ((s_{0,0}, \ldots, s_{k,0}), l_0) \models_k \varphi$ and $\mathbf{S}(w_{0,0}) = s_{0,0} = \iota$. Thus $M \models_k^{\exists} \varphi$.

Now, from Theorems 1 and 2 we get the following.

Corollary 1. *Let M be a model, and φ an EMTLKD formula. Then, $M \models^{\exists} \varphi$ if, and only if, there exists $k \in \mathbb{N}$ such that the propositional formula $[M, \varphi]_k$ is satisfiable.*

5 Experimental Results

Our SAT-base BMC method for EMTLKD is, to our best knowledge, the first one formally presented in the literature, and moreover there is no any other model checking technique for the considered EMTLKD language. Further, our implementation of the presented BMC method uses Reduced Boolean Circuits (RBC) [1] to represent the propositional formula $[M, \varphi]_k$. An RBC represents subformulae of $[M, \varphi]_k$ by fresh propositions such that each two identical subformulae correspond to the same proposition[1]. For the tests we have used a computer with Intel Core i3-2125 processor, 8 GB of RAM, and running Linux 2.6. We set the timeout to 5400 seconds, and memory limit to 8GB, and we used the state of the art SAT-solver MiniSat 2. The specifications for the described benchmark are given in the universal form, for which we verify the corresponding counterexample formula, i.e., the formula which is negated and interpreted existentially.

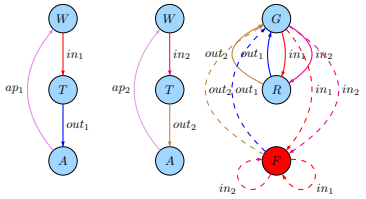


Fig. 1. An DIIS of FTC for two trains

To evaluate our technique, we have analysed a scalable multi-agent system, which is a faulty train controller system (FTC). Figure 1 presents a DIIS composed of three agents: a controller and two trains, but in general the system consists of a controller, and n trains (for $n \geqslant 2$) that use their own circular tracks for travelling in one direction

[1] Following van der Meyden at al. [11], instead of using RBCs, we could directly encode $[M, \varphi]_k$ in such a way that each subformula ψ of $[M, \varphi]_k$ occurring within a scope of a k-element disjunction or conjunction is replaced with a propositional variable p_ψ and the reduced formula $[M, \varphi]_k$ is conjuncted with the implication $p_\psi \Rightarrow \psi$. However, in this case our method, as the one proposed in [11], would not be complete.

(states Away (A)). At one point, all trains have to pass through a tunnel (states Tunnel 'T'), but because there is only one track in the tunnel, trains arriving from each direction cannot use it simultaneously. There are colour light signals on both sides of the tunnel, which can be either red (state 'R') or green (state 'G'). All trains notify the controller when they request entry to the tunnel or when they leave the tunnel. The controller controls the colour of the colour light signals, however it can be faulty (state 'F'), i.e., a faulty traffic light remains green when a train enters the tunnel, and thereby it does not serve its purpose. In the figure, the initial states of the controller and the trains are 'G' and 'W' (Waiting in front of the tunnel) respectively, and the transitions with the same label are synchronised. Null actions are omitted in the figure.

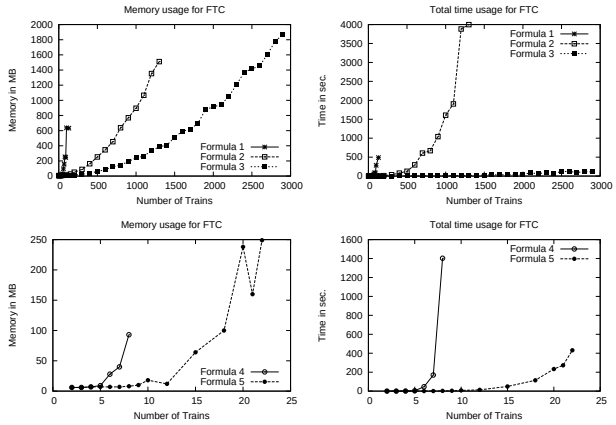
Let $\mathcal{PV} = \{inT_1, \ldots inT_n, Red\}$ be a set of propositional variables, which we find useful in analysis of the scenario of the FTC system. A valuation function $\mathcal{V} : S \rightarrow 2^{\mathcal{PV}}$ is defined as follows. Let $Ag = \{Train1\ (T1), \ldots, TrainN\ (TN), Controller\ (C)\}$. Then, $inT_c \in \mathcal{V}(s)$ if $l_c(s) = T$ and $c \in Ag \setminus \{C\}$; $Red \in \mathcal{V}(s)$ if $l_C(s) = R$. The specifications are the following:

- φ_1 = $\mathrm{G}_{[0,\infty]}\ \mathcal{O}_C(\bigwedge_{i=1}^{n-1} \bigwedge_{j=i+1}^{n} \neg(InT_i \wedge InT_j))$. "Always when *Controller* is functioning correctly, trains have exclusive access to the tunnel".
- φ_2 = $\mathrm{G}_{[0,\infty]}(inT_1 \Rightarrow \widehat{\mathrm{K}}^C_{T1}(\bigwedge_{i=2}^{n}(\neg inT_i)))$. "Always when *Train1* is in the tunnel, it knows under assumption that *Controller* is functioning correctly that none of the other trains is in the tunnel".
- φ_3 = $\mathrm{G}_{[0,\infty]}(inT_1 \Rightarrow \widehat{\mathrm{K}}^C_{T1}(Red))$. "Always when *Train1* is in the tunnel, it knows under assumption that *Controller* is functioning correctly that the colour of the light signal for other trains is red".
- φ_4 = $\mathrm{G}_{[0,\infty]}(InT_1 \Rightarrow \mathrm{K}_{T1}(\mathrm{F}_{[1,n+1]}(\bigvee_{i=1}^{n} InT_i)))$. "Always when *Train1* is in the tunnel, it knows that either he or other train will be in the tunnel during the next $n + 1$ time units".
- φ_5 = $\mathrm{G}_{[0,\infty]}(InT_1 \Rightarrow \mathrm{K}_{T1}(\mathrm{G}_{[3m-2,3m-2]} InT_1 \vee \mathrm{F}_{[1,n+1]}(\bigvee_{i=2}^{n} InT_i)))$, where $m \geqslant 2$. "Always when *Train1* is in the tunnel, it knows that either he is in the tunnel every $3m - 2$ time units or other train will be in the tunnel during the next $n + 1$ time units".

All the above properties are false in our DIIS model of the FTC system.

Since there is no model checker that supports the EMTLKD properties, we were not able to compare our results with others for the above formulae; McMAS [22] is the only model checker that supports deontic modalities, however it is designated for branching time logics only. Thus, we present results of our method only. An evaluation is given by means of the running time and the memory used, and it is presented on the included line-charts. It can be observed that for φ_1, φ_2, φ_3, φ_4 and φ_5 we managed to compute the results for 130, 1300, 2900, 8, and 22 trains, respectively, in the time of 5400 seconds. The exact data for the mentioned maximal number of trains are the following:

- φ_1: $k = 4$, $f_k(\varphi_4) = 2$, bmcT is 5.44, bmcM is 14.00, satT is 483.61, satM is 632.00, bmcT+satT is 489.05, max(bmcM,satM) is 632.00;
- φ_2: $k = 4$, $f_k(\varphi_4) = 2$, bmcT is 148.02, bmcM is 909.00, satT is 3850.09, satM 1511.00, bmcT+satT is 3998.11, max(bmcM,satM) is 1511.00;



φ_3: $k = 1$, $f_k(\varphi_4) = 2$, bmcT is 98.89, bmcM is 1114.00, satT is 9.69, satM 1869.00, bmcT+satT is 108.58, max(bmcM,satM) is 1869.00;

φ_4: $k = 24$, $f_k(\varphi_4) = 2$, bmcT is 2.00, bmcM is 3.57, satT is 1401.24, satM 93.00, bmcT+satT is 1403.24, max(bmcM,satM) is 93.00;

φ_5: $k = 65$, $f_k(\varphi_4) = 2$, bmcT is 281.50, bmcM is 18.13, satT is 149.59, satM 249.00, bmcT+satT is 431.10, max(bmcM,satM) is 249.00,

where k is the bound, $f_k(\varphi)$ is the number of symbolic paths, bmcT is the encoding time, bmcM is memory use for encoding, satT is satisfiability checking time, satM is memory use for satisfiability checking.

The formulae φ_1, φ_2 and φ_3 corroborates the efficiency of the SAT-based BMC methods when the length of the counterexamples does not grow with the number of agents (trains). On the other hand the formulae φ_4 and φ_5 demonstrate that SAT-based BMC becomes inefficient when the the length of the counterexamples grows with the number of agents (trains).

Our future work will involve an implementation of the method also for other models of multi-agent systems, for example for standard interpreted systems. Moreover, we are going to define a BDD-based BMC algorithm for EMTLKD, and compare it with the method presented in this paper.

References

1. P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on SAT-solvers. In *Proc. of TACAS'00*, volume 1785 of *LNCS*, pp. 411–425. Springer-Verlag, 2000.
2. R. Alur and T. Henzinger. Logics and models of real time: A survey. In *Proceedings of REX Workshop 'Real Time: Theory and Practice'*, volume 600 of *LNCS*, pp. 74–106. Springer-Verlag, 1992.
3. L. Aqvist. Deontic logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic: Volume II: Extensions of Classical Logic*, pp. 605–714. Reidel, Dordrecht, 1984.

4. E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons for branching-time temporal logic. In *Proceedings of Workshop on Logic of Programs*, volume 131 of *LNCS*, pp. 52–71. Springer-Verlag, 1981.
5. E. A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, volume B, chapter 16, pp. 996–1071. Elsevier Science Publishers, 1990.
6. K. Engelhardt, R. van der Meyden, and Y. Moses. Knowledge and the logic of local propositions. In *Proc. of TARK'98*, pp. 29–41, 1998.
7. R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about Knowledge*. MIT Press, Cambridge, 1995.
8. C. A. Furia and P. Spoletini. Tomorrow and all our yesterdays: MTL satisfiability over the integers. In *Proc. of ICTAC*, volume 5160 of *LNCS*, pp. 253–264. Springer-Verlag, 2008.
9. P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *Proc. of CAV*, volume 2102 of *LNCS*, pp. 53–65. Springer-Verlag, 2001.
10. W. van der Hoek and M. Wooldridge. Model checking knowledge and time. In *Proc. of SPIN*, 2002.
11. X. Huang, C. Luo, and R. van der Meyden. Improved bounded model checking for a fair branching-time temporal epistemic logic. In *Proc. of MoChArt'2010*, volume 6572 of *LNAI*, pp. 95–111. Springer, 2011.
12. A. Jones and A. Lomuscio. A BDD-based BMC approach for the verification of multi-agent systems. In *Proc. of CS&P*, volume 1, pp. 253264. Warsaw University, 2009.
13. M. Kacprzak, A. Lomuscio T. Lasica, W. Penczek, and M. Szreter. Verifying multiagent systems via unbounded model checking. In *Proc. of FAABS III*, volume 3228 of *LNCS*, pp. 189–212. Springer-Verlag, 2004.
14. R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.
15. H. Levesque. A logic of implicit and explicit belief. In *Proc. of the 6th National Conference of the AAAI*, pp. 198–202. Morgan Kaufman, 1984.
16. A. Lomuscio, W. Penczek, and H. Qu. Partial order reduction for model checking interleaved multi-agent systems. In *Proc. of AAMAS, IFAAMAS Press.*, pp. 659–666, 2010.
17. A. Lomuscio and M. Sergot. Deontic interpreted systems. *Studia Logica*, 75(1):63–92, 2003.
18. A. Męski, W. Penczek, and M. Szreter. Bounded model checking linear time and knowledge using decision diagrams. In *Proc. of CS&P*, pp. 363–375. Białystok University of Technology, 2011.
19. W. Penczek and A. Lomuscio. Verifying epistemic properties of multi-agent systems via bounded model checking. In *Proc. of AAMAS*, pp. 209–216. ACM, 2003.
20. W. Penczek, B. Woźna-Szcześniak, and A. Zbrzezny. Towards SAT-based BMC for LTLK over interleaved interpreted systems. In *Proc. of CS&P*, pp. 565–576. Białystok University of Technology, 2011.
21. J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. of the 5th International Symposium on Programming*, volume 131 of *LNCS*, pp. 337–351. Springer-Verlag, 1981.
22. F. Raimondi and A. Lomuscio. Symbolic model checking of deontic interpreted systems via OBDDs. In *Proc. of DEON04*, volume 3065 of *LNCS*, pp. 228–242. Springer Verlag, 2004.
23. F. Raimondi and A. Lomuscio. Automatic verification of multi-agent systems by model checking via OBDDs. *Journal of Applied Logic*, 5(2):235–251, 2005.
24. M. Wooldridge. *An introduction to multi-agent systems*. John Wiley, England, 2002.
25. B. Woźna, A. Lomuscio, and W. Penczek. Bounded model checking for deontic interpreted systems. In *Proc. of LCMAS'2004*, volume 126 of *ENTCS*, pp. 93–114. Elsevier, 2005.
26. A. Zbrzezny. A new translation from ECTL* to SAT. In *Proc. of CS&P*, pp. 589–600, http://csp2011.mimuw.edu.pl/proceedings/PDF/CSP2011589.pdf. Białystok University of Technology, 2011.

Some Thoughts about Commitment Protocols (Position Paper)

Matteo Baldoni and Cristina Baroglio

Università degli Studi di Torino
Dipartimento di Informatica
c.so Svizzera 185, I-10149 Torino (Italy)
{matteo.baldoni,cristina.baroglio}@unito.it

1 Introduction

Practical commitments lie at the level of *regulative* (or *preservative*) norms that, in turn, impact on the agents' behavior, creating social expectations, that should not be frustrated. By a practical commitment, in fact, an actor (*debtor*) is committed towards another actor (*creditor*) to bring about something [6, 17], i.e. to act either directly or by persuading others so as to make a condition of interest become true. Due to their *social nature*, practical commitments are a powerful tool that helps to overcome the controversial assumptions of the mentalistic approach that mental states are verifiable and that agents are sincere. Moreover, they support an *observational semantics* for communication that allows verifying an agent's compliance with its commitments based on observable behavior.

From the seminal paper by Singh [18], commitments protocols have been raising a lot of attention, see for instance [23, 14, 21, 9, 20, 11, 3]. The key feature of commitment protocols is their *declarative nature*, which allows specifying them in a way which abstracts away from any reference to the actual behaviour of the agents, thus avoiding to impose useless execution constraints [24]. By doing so, commitment-based protocols respect the autonomy of agents because whatever action they decide to perform is fine as long as they accomplished their commitments, satisfying each others' expectations. Now, after more than ten years from the introduction of commitments, it is time to ask (*i*) if a "commitment to do something" is *the only kind of regulative norm*, that we need in order to give a social semantics to a physical action, and (*ii*) if they realize what they promised. To this aim, we think that there are four intertwined aspects to be considered:

1. *Agent Coordination*: how to account for coordination patterns?
2. *Infrastructure for Execution*: which is the reference execution infrastructure?
3. *Observability of Events*: are events really observable by all agents?
4. *Composition of Coordination Patterns*: is composition influenced by the previous aspects?

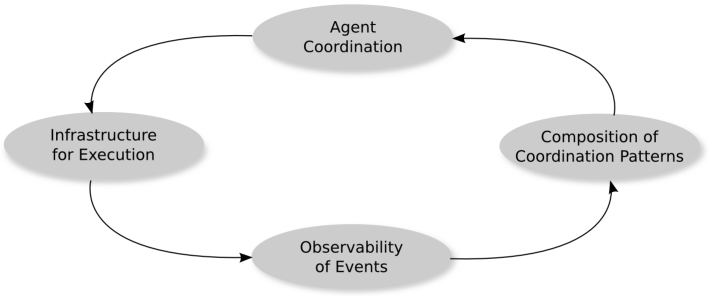


Fig. 1. The four considered intertwined aspects.

2 Agent Coordination

Commitment protocols leave the choice of which action to execute and when, totally up to the single agents. From a more general perspective, they do not impose constraints on the possible evolutions of the social state. However, in many practical cases there is the need to capture regulative aspects of *agent coordination*. For instance, a customer and a merchant may *agree* that payment should be done before shipping but how to represent this socially agreed constraint in commitment protocols? When a similar coordination is desired by the parties, one feels the lack of the means for capturing them *as regulations* inside the protocol. Notice that the desired coordination patterns, though restricting the choices up to the agents, would not prevent flexibility because, for instance, it is not mandatory that payment and shipping are one next to the other. What matters is their relative order. More importantly, an agreed coordination pattern establishes the boundaries within which each party can exercise his/her own autonomy without compromising the aims for which the agreement was taken. Citing Dwight Eisenhower[1] "*To be true to one's own freedom is, in essence, to honor and respect the freedom of all others.*" As long as agents respect such constraints, they are free to customize the execution at their will, e.g. by interleaving the two actions with others (like sending a receipt or asking a quote for another item). This need is felt by the research community, see [3] for an overview.

When regulations are expressed, agents can individually check whether their behavior conforms to the specification [2]. But in order to guarantee to the others that one will act in a way that conforms to the regulation, an agent should *formally bind* its behavior to the regulation itself. The proposal in [3], for instance, allows the representation of temporal regulations imposed on the evolution of the social state, however, it does not supply a deontic semantics to the constraints. Therefore the agents' behavior is not formally bound to them. On the other hand, the REGULA framework [16] uses precedence logic to express temporal patterns that can be used as antecedent (or consequent) conditions inside

[1] State of the Union Address, Feb. 2, 1953.

commitments. Since patterns may involve various parties, the framework also introduces a notion of condition control and of commitment safety, in order to allow agents to reason about the advisability of taking a commitment. However, patterns are not generally expressed on the evolution of the social state but are limited to events.

3 Infrastructure for Execution and Observability of Events

Commitments were introduced to support *run-time verification* in contrast to the mentalistic approach but despite this, they still lack of a *reference infrastructure* that practically enables such a verification. Verification is supported by proposals like [1, 7], although the authors do not draft an infrastructure, while commitment machines [24, 21, 19] have mainly been used to provide an operational semantics. Normative approaches, e.g. institutions [13, 14], provide an answer but with some limitation. Indeed, they tend to implicitly assume a centralized vision, often realized by introducing a new actor, whose task is to monitor the interaction: the institution itself. This assumption is coherent with the fact that commitment protocols tend to assume that events are *uniformly observed* by all the agents although in the real world this seldom happens; for instance, communications tend to be point-to-point. We need the infrastructure to support *this* kind of interaction and to monitor, in *this* context, the on-going enactment, checking whether it respects all the regulative aspects – that the designer identified as relevant or that the agents agreed. Chopra and Singh [8] addressed the issue of realizing an architecture that relaxes the centralization constraint by incorporating the notion of *commitment alignment*. In this way it becomes possible to answer to questions like "how to decide whether agents are acting in a way that complies to the regulations or not?", "How to know that an agent satisfied one of its commitments?" in contexts where events are not uniformly observable. Nevertheless, they relegated commitment alignment to the middleware, shielding the issue of *observability of events* from the agents and from the designer. Our claim is that this is a limitation and that in many real-world situations it is more desirable to have the means of making clear who can access to what information and who is accountable for reporting what event. This is especially true in the case when the protocol allows the representation of coordination patterns: there is the need of mechanisms for expressing who can observe what, tracking which part of a pattern was already followed, which is left to be performed, who is in charge of the next moves, and so on. As a consequence, we think that the specification of the coordination patterns and the design of the infrastructure cannot leave out the *observability of events*, which plays a fundamental role *at the level of the protocol specification* and, for this reason, it should be captured by *first-class abstractions* and appropriate regulations. Such abstractions/regulations should be represented in a way that makes them *directly manipulable* by the agents [4].

4 Composition of Coordination Patterns

Most of the works concerning software engineering aspects of commitment protocol specification focus on the formal verification to help the protocol designer to get rid of or to enforce given behaviors, [22, 15, 5, 12, 11]. An aspect that is not to be underestimated is the realization of a development methodology for commitment protocols. The most relevant representative is the Amoeba methodology [10], which allows the design of commitment protocols and their composition into complex business processes. With respect to the aspects that we are discussing, this methodology, however, has two main limits. On the one hand, when two or more protocols are composed, the designer is requested to define a set of temporal constraints among events and of data flow constraints to combine the various parts. However, such constraints do not have any regulatory flavour nor they have a deontic characterization. On the other hand, since a wider number of roles are involved, which of the actors of one protocol is entitled to (and physically can) observe events generated inside another protocol? The methodology does not explicitly account for this problem in the description of the various steps that compose it. For instance, suppose of composing a protocol that allows a merchant and a supplier to interact with one that allows the same merchant to interact with a customer. It is unrealistic to suppose that the client can observe events involving the supplier, even though after the composition both actors will play in the same protocol. Actually, it would be useful to incorporate in the protocol the means for letting the merchant tell the client that it received items from the supplier in a way that makes it accountable for its declarations.

References

1. M. Alberti, M. Gavanelli, E. Lamma, F. Chesani, P. Mello, and P. Torroni. Compliance verification of agent interaction: a logic-based software tool. *Applied Artificial Intelligence*, 20(2-4):133–157, 2006.
2. M. Baldoni, C. Baroglio, A. K. Chopra, N. Desai, V. Patti, and M. P. Singh. Choice, Interoperability, and Conformance in Interaction Protocols and Service Choreographies. In K. Decker, J. Sichman, C. Sierra, and C. Castelfranchi, editors, *Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2009*, pages 843–850, Budapest, Hungary, May 2009. IFAAMAS.
3. M. Baldoni, C. Baroglio, E. Marengo, and V . Patti. Constitutive and Regulative Specifications of Commitment Protocols: a Decoupled Approach. *ACM TIST, Spec. Iss. on Agent Communication*, 2011.
4. M. Baldoni, C. Baroglio, E. Marengo, V. Patti, and A. Ricci. Back to the future: An interaction-oriented framework for social computing. In *First Int. Workshop on Req. Eng. for Social Computing, RESC*, pages 2–5. IEEE, 2011.
5. J. Bentahar, J.-J. Ch. Meyer, and W. Wan. Model checking communicative agent-based systems. *Knowl.-Based Syst.*, 22(3):142–159, 2009.
6. C. Castelfranchi. Commitments: From Individual Intentions to Groups and Organizations. In *Proc. of ICMAS*, pages 41–48, 1995.

7. F. Chesani, P. Mello, M. Montali, and P. Torroni. Commitment Tracking via the Reactive Event Calculus. In C. Boutilier, editor, *Proc. of the 21st International Joint Conference on Artificial Intelligence, IJCAI*, pages 91–96, Pasadena, California, USA, July 2009.
8. A. K. Chopra and M. P. Singh. An Architecture for Multiagent Systems: An Approach Based on Commitments. In *Proc. of ProMAS*, volume 5919 of *LNAI*, pages 148–162, Heidelberg, 2009. Springer.
9. A.K. Chopra. *Commitment Alignment: Semantics, Patterns, and Decision Procedures for Distributed Computing*. PhD thesis, North Carolina State University, Raleigh, NC, 2009.
10. N. Desai, A.K. Chopra, and M.P. Singh. Amoeba: A methodology for modeling and evolving cross-organizational business processes. *ACM Trans. Softw. Eng. Methodol.*, 19(2), 2009.
11. M. El-Menshawy, J. Bentahar, and R. Dssouli. Verifiable Semantic Model for Agent Interactions Using Social Commitments. In *Proc. of LADS*, volume 6039, pages 128–152, 2010.
12. M. El-Menshawy, J. Bentahar, and R. Dssouli. Symbolic Model Checking Commitment Protocols Using Reduction. In *Declarative Agent Languages and Technologies VIII - 8th International Workshop, DALT 2010*, volume 6619 of *Lecture Notes in Computer Science*, pages 185–203. Springer, 2011.
13. N. Fornara. *Interaction and Communication among Autonomous Agents in Multiagent Systems*. PhD thesis, Università della Svizzera italiana, Facoltà di Scienze della Comunicazione, June 2003.
14. N. Fornara and M. Colombetti. Defining Interaction Protocols using a Commitment-based Agent Communication Language. In *Proc. of AAMAS*, pages 520–527. ACM, 2003.
15. A. Mallya and M. Singh. An algebra for commitment protocols. *Autonomous Agents and Multi-Agent Systems*, 14(2):143–163, 2007.
16. E. Marengo, M. Baldoni, C. Baroglio, A. K. Chopra, V. Patti, and M. P. Singh. Commitments with Regulations: Reasoning about Safety and Control in REGULA. In *Proc. of AAMAS*, pages 467–474, 2011.
17. M. P. Singh. An Ontology for Commitments in Multiagent Systems. *Artificial Intelligence and Law*, 7(1):97–113, 1999.
18. Munindar P. Singh. Agent communication languages: Rethinking the principles. *IEEE Computer*, 31(12):40–47, 1998.
19. Munindar P. Singh. Formalizing Communication Protocols for Multiagent Systems. In *Proc. of IJCAI*, pages 1519–1524, 2007.
20. P. Torroni, F. Chesani, P. Mello, and M. Montali. Social Commitments in Time: Satisfied or Compensated. In *Proc. of DALT*, volume 5948, pages 228–243, 2010.
21. M. Winikoff, W. Liu, and J. Harland. Enhancing Commitment Machines. In J. A. Leite, A. Omicini, P. Torroni, and P. Yolum, editors, *Proc. of DALT*, volume 3476 of *LNCS*, pages 198–220, New York, NY, USA, July 2005. Springer.
22. P. Yolum. Design time analysis of multiagent protocols. *Data Knowl. Eng.*, 63(1):137–154, 2007.
23. P. Yolum and M. P. Singh. Designing and Executing Protocols Using the Event Calculus. In *Agents*, pages 27–28, New York, NY, USA, 2001. ACM.
24. P. Yolum and M. P. Singh. Commitment Machines. In *Proc. of ATAL*, pages 235–247, 2002.

Semantic Web and Declarative Agent Languages and Technologies: Current and Future Trends (Position Paper)

Viviana Mascardi[1], James Hendler[2], and Laura Papaleo[3]

[1] DISI, University of Genova, Italy
viviana.mascardi@unige.it
[2] Rensselaer Polytechnic Institute, Troy, NY, USA
hendler@cs.rpi.edu
[3] ICT Department, Provincia di Genova, Genova, Italy
papaleo@disi.unige.it

1 Introduction

One of the first discussions about a Web enriched with semantics and its relationships with artificial intelligence (and hence, with intelligent agents) dates back to 1998 [4], but it was only ten years ago that the idea of a Semantic Web on top of which agent-based computing would have allowed computer programs to interact with non-local web-based resources, became familiar to a wide audience of scientists [5, 10].

The integration of Semantic Web concepts as first class entities inside agent languages, technologies, and engineering methodologies has different levels of maturity: many AOSE methodologies, organizational models and MAS architectures seamlessly integrate them (for example, [20], [19], and the FIPA "Ontology Service Specification", www.fipa.org/specs/fipa00086/, respectively), but few languages do.

In this position paper we review the state of the art in the integration of semantic web concepts in declarative agent languages and technologies and outline what we expect to be the future trends of this research topic.

2 State of the Art

Agent Communication Languages. In agent communication, the assumption that ontologies should be used to ensure interoperability had been made since the very beginning of the work on ontologies, even before they made the basis for the Semantic Web effort. Both KQML [15] and FIPA-ACL [9] allow agents to specify the ontology they are using, although none of them forces that. Agent communication languages were born with the Semantic Web in mind. The same does not hold for agent programming languages, that only recently started to address ontologies as first class objects.

Agent Programming Languages. AgentSpeak [17] underwent many extensions over time. However, what was considered only with the work [16] discussing AgentSpeak-DL, is that ontological reasoning could facilitate the development of AgentSpeak agents. The implementation of AgentSpeak-DL concepts is given in JASDL [12]. CooL-AgentSpeak [14], the "Cooperative Description-Logic AgentSpeak" language integrating Coo-BDI [1] and AgentSpeak-DL and enhancing them with *ontology matching capabilities* [8] is a further effort on this subject. The authors of [6] and [7] explore the use of a formal ontology as a constraining framework for the belief store of a rational agent and show the implementation of their proposal in the `Go!` multi-threaded logic programming language [6]. We are not aware of similar attempts made with non-declarative ones, apart from the support that JADE [3] offers to ontologies, which is limited to boosting agent communication by allowing the agents to refer to concepts belonging to ontologies in the messages they exchange, and is hence due to the respect of FIPA-ACL specifications.

Proof and Trust in MASs. Even if the Semantic Web is often incorrectly reduced to reasoning on semantic markups, it actually goes far beyond that, coping with proof and trust as well. Both these topics are extremely hot within the agent community, and on the DALT's one in particular. In the literature we can find dozens of works on trust and reputation in agent societies, and research on formally proving that an agent can enter an organization without damaging it has already produced many valuable results. Model checking declarative agent languages has a long tradition too (see for example the "MCAPL: Model Checking Agent Programming Languages" project, `http://cgi.csc.liv.ac.uk/MCAPL/index.php/Main_Page`, and [11]).

3 Future Trends

There are many promising directions that the research on integration of Semantic Web technologies and DALTs could take.

Semantic-Web based Proof and Trust. Although the maturity level of the aspects concerned with proof and trust in DALTs is satisfactory, mechanisms that give the developer the real power or putting all together are still missing. For example, to design and build MASs where agents can trust each other, the consistency of the agents' beliefs represented as ontologies should be always preserved, and formally demonstrated if required by the application.

Semantic-Web based Mediation. In [2], a semantic mediation going beyond the integration of ontologies within traditional message-based communication was envisaged. Mediation should occur at the level that characterizes the social approach where it is required to bind the semantics of the agent actions with their meaning in social terms (ontology-driven count-as rules).

Semantic Representation of the Environment. Although not yet formalized in published papers, the A&A model [18] is moving towards integrating semantic web concepts as first class objects for semantically representing the environment and the artifacts available to the agents[1]. This line of research should be pursued by other declarative approaches as well, where the environments is explicitly represented. Formally proving the consistency of the "Environment Ontology" should be possible, as well as evolving it, and learning it from sources of semi-structured information.

Adoption of Semantic-Web enriched DALTs for Real Applications. Many real applications involve scenarios where procedural rules for achieving a goal are expressed in an informal and fully declarative way, may require to achieve sub-goals, and the domain knowledge is hard-wired within the rules themselves, making them barely re-usable in other domains, even if they could. Think of the rules for getting a new identity card issued by Genova Municipality, which are declaratively defined by conditions to be met, other documents to be obtained before, and exactly the same as those for obtaining the document in another municipality, but nevertheless would be hard to compare. Expressing procedural rules of this kind using declarative agent languages fully integrated with semantic web concepts might help comparing and composing them in an automatic way, moving a step forward the automation of many services that are still completely performed by human agents.

Discussion. The first problem that the Semantic Web and Declarative Agent Languages and Technologies communities should struggle to solve together, is bringing usability to the world. Forthcoming technologies should be not only secure, efficient, self-*, etc. It is mandatory that *they will be usable* by average computer scientists, average professionals and even average users. "*Making intelligent software agents both powerful and easy to construct, manage, and maintain will require a very rich semantic infrastructure*" [13], and the rich semantic infrastructure seething with agents, must be there for anyone. In a few years, it must become a commodity, clearing the boundaries of academic research once and for all.

References

1. D. Ancona and V. Mascardi. Coo-BDI: Extending the BDI model with cooperativity. In *Proc. of the 1st Int. Workshop on Declarative Agent Languages and Technologies*, volume 2990 of *LNCS*, pages 109–134. Springer Verlag, 2003.
2. M. Baldoni, C. Baroglio, F. Bergenti, A. Boccalatte, E. Marengo, M. Martelli, V. Mascardi, L. Padovani, V. Patti, A. Ricci, G. Rossi, and A. Santi. MERCURIO: An interaction-oriented framework for designing, verifying and programming multi-agent systems. In *Proc. of the 11th WOA Workshop*. CEUR-WS.org, 2010.

[1] Private communication of one of the authors of this paper with the authors of the A&A model.

3. F. Bellifemine, A. Poggi, and G. Rimassa. Developing multi-agent systems with JADE. In *Intelligent Agents VII*, pages 89–103. Springer Verlag, 2001. LNAI 1986.
4. T. Berners-Lee. An parenthetical discussion to the web architecture at 50,000 feet and the semantic web roadmap. Online at `www.w3.org/DesignIssues/RDFnot.html`. Accessed on 2011-10-10, 1998.
5. T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, pages 29–37, May 2001.
6. K. L. Clark and F. G. McCabe. Go! a multi-paradigm programming language for implementing multi-threaded agents. *Ann. Math. Artif. Intell.*, 41:171–206, 2004.
7. K. L. Clark and F. G. McCabe. Ontology schema for an agent belief store. *Int. J. Hum.-Comput. Stud.*, 65:640–658, July 2007.
8. J. Euzenat and P. Shvaiko. *Ontology Matching*. Springer, 2007.
9. Foundation for Intelligent Physical Agents. FIPA ACL message structure specification. Approved for standard, Dec. 6^{th}, 2002.
10. J. A. Hendler. Agents and the semantic web. *IEEE Intelligent Systems*, 16(2):30–37, 2001.
11. S.-S. T. Q. Jongmans, K. V. Hindriks, and M. B. van Riemsdijk. Model checking agent programs by using the program interpreter. In *Proc. of the 11th Int. Workshop on Computational Logic in Multi-Agent Systems*, pages 219–237. Springer, 2010.
12. T. Klapiscak and R. Bordini. JASDL: A practical programming approach combining agent and semantic web technologies. In *Proc. of the 6th Int. Workshop on Declarative Agent Languages and Technologies*, volume 5397 of *LNCS*, pages 91–110. Springer Verlag, 2009.
13. J. Krupansky. Richness of semantic infrastructure, 2011. Online `http://semanticabyss.blogspot.com/2011/06/richness-of-semantic-infrastructure.html`. Accessed on 2011-10-10.
14. V. Mascardi, D. Ancona, R. H. Bordini, and A. Ricci. CooL-AgentSpeak: Enhancing AgentSpeak-DL agents with plan exchange and ontology services. In *Proc. of the Int. Conf. on Intelligent Agent Technology*, pages 109–116. IEEE Computer Society, 2011.
15. J. Mayfield, Y. Labrou, and T. Finin. Evaluation of KQML as an agent communication language. In *Proc. of the 2nd Int. ATAL Workshop*, pages 347–360. Springer Verlag, 1995.
16. Á. F. Moreira, R. Vieira, R. H. Bordini, and J. F. Hübner. Agent-oriented programming with underlying ontological reasoning. In *Proc. of the 3rd Int. Workshop on Declarative Agent Languages and Technologies*, volume 3904 of *LNCS*, pages 155–170. Springer Verlag, 2006.
17. A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Agents Breaking Away*, pages 42–55. Springer Verlag, 1996. LNAI 1038.
18. A. Ricci, M. Viroli, and A. Omicini. Programming MAS with artifacts. In *Programming Multi-Agent Systems*, volume 3862 of *LNCS*, pages 206–221. Springer Verlag, 2006.
19. B. L. Smith, V. A. M. Tamma, and M. Wooldridge. An ontology for coordination. *Applied Artificial Intelligence*, 25(3):235–265, 2011.
20. Q.-N. N. Tran and G. Low. Mobmas: A methodology for ontology-based multi-agent systems development. *Information & Software Technology*, 50(7-8):697–722, 2008.

Designing and Implementing a Framework for BDI-style Communicating Agents in Haskell (Position Paper)

Alessandro Solimando and Riccardo Traverso*

Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, Italy
{alessandro.solimando,riccardo.traverso}@disi.unige.it

Abstract. In this position paper we present the design and prototypical implementation of a framework for BDI-style agents defined as Haskell functions, supporting both the explicit representation of beliefs and backtracking (at the level of individual agents), and asynchronous communication via message passing. The communication layer is separated from the layers implementing the features of individual agents through different stacked monads, while beliefs are represented through atomic or structured values depending on the user's needs. Our long-term goal is to develop a framework for purely functional BDI agents, which is currently missing, in order to take advantage of the features of the functional paradigm, combined with the flexibility of an agent-oriented approach.

1 Introduction

The Belief-Desire-Intention (BDI) model is a well-known software model for programming intelligent rational agents [10]. Only a few frameworks that implement the BDI approach are developed directly on top of logical languages [7], but most of those exploiting imperative or object oriented languages, such as Jason [2] that exploits Java's inheritance and overriding to define selection functions and the environment in a very convenient and flexible way, have to re-implement many features that are natively available in the logic programming paradigm. For example, in Jason unification is needed to find plans relevant to a triggering event, and to resolve logical goals in order to verify that the plan context is a logical consequence of the belief base. BDI-style agents are usually described in a declarative way, no matter how the language interpreter is implemented. The functional paradigm supports pattern matching for free and gives all the advantages of declarativeness; moreover, the use of types for typing communication channels may provide great benefits to guarantee correctness properties both a priori, and during the execution. Nevertheless, to the best of our knowledge no functional frameworks for BDI-style communicating agents have been proposed so far.

* Both authors of this paper are Ph. D. students at the University of Genova, Italy.

In order to fill this gap, we propose a framework for functional agents taking inspiration from the BDI model (although not implementing all of its features), and supporting communication and backtracking. A generic and easily composable architecture should partition the agents' functionalities into several well-separated layers, and in functional programming monads are a powerful abstraction to satisfy these needs. Intuitively, in our solution agents are monadic actions provided with local backtracking features and point-to-point message passing primitives. Their local belief base is stored within variables that are passed down through execution steps. Goals are defined with functions from beliefs to booleans. When it comes to monadic computations, Haskell [3], being strongly based on them, is the best fit.

Our work is a generalization of [11], where the authors describe a single-agent monadic BDI implementation relying on CHR [6]; we share with [11] the idea of a BDI architecture based on monads, but instead of relying on CHR to represent beliefs and their evolution, the aim of our work is to provide a better integration with the language by handling them directly as Haskell values and expressions.

In [13] agents executing abstract actions relative to deontic specifications (prohibition, permission, and obligation) are simulated in Haskell. Although close to our approach up to some extent, that work does not take the BDI model into account. We are not aware of other proposals using functional languages to represent BDI-style agents.

2 Our Framework

In this section we first provide a very brief overview of Haskell's syntax [8], to allow the reader to understand our design choices.

The keyword `data` is used to declare new, possibly polymorphic, data types. A new generic type may be, e.g., `data MyType a b = MyType a a b`: `a` and `b` are two type variables, and the constructor for new values takes (in the order) two `a` arguments and one `b`. A concrete type for `MyType` could be, e.g., `MyType Int String`. A type signature for `f` is written `f :: a`, where `a` is a type expression; an arrow `→` is a right-associative infix operator for defining domain and codomain of functions. A type class is a sort of interface or abstract class that data types may support by declaring an `instance` for it. A special type `()`, called unit, acts as a tuple with arity 0; its only value is also written `()`.

In our framework, we split the definition of the capabilities of the agents in different layers by means of monads. The innermost one, `Agent`, provides support for the reasoning that an agent may accomplish in isolation from the rest of the system, that is without any need to communicate. On top of it we build another monad `CAgent` for communicating agents that provides basic message-passing features.

```
data Agent s a = Agent (s → (s,a))
instance Monad (Agent s) where {- omitted -}
```

The declaration of `Agent` follows the definition of the well-known state monad [4]. It is parameterized on two types: the state `s` of the agent, containing its

current beliefs, and the return type `a` of the action in the monad. Each action is a function from the current state to the (possibly modified) new one, together with the return value.

At this layer it is safe to introduce goal-directed backtracking support, because computations are local to the agent and no interaction is involved. In Haskell, one could provide a basic backtracking mechanism for a monad `m` by defining an instance of the `MonadPlus` type class. `MonadPlus m` instances must define two methods, `mzero :: m a` and `mplus :: m a → m a → m a`, that respectively represent failure and choice. Infinite computations, i.e. with an infinite number of solutions, can not be safely combined within `MonadPlus` because the program could diverge. In order to address this problem the authors of [5] propose a similar type class – along with a comparison between different implementations – where its operators behave fairly, e.g. solutions from different choices are selected with a round robin policy. In our work we plan to exploit their solutions to give `Agent` the possibility to handle backtracking even in such scenarios. Goals can be defined as predicates `pred :: Agent s Bool` to be used in guards that may stop the computation returning `mzero` whenever the current state does not satisfies `pred`. It is worth noting how this concept of goals fits well into Haskell: such guards are the standard, natural way to use `MonadPlus`.

```
type AgentId = String
data Message a = Message AgentId AgentId a
data AgentChan a = {- omitted -}
```

Another building block for our MAS architecture is the FIFO channel `AgentChan`. We omit the full definition for the sake of brevity: it is sufficient to know that messages have headers identifying sender and receiver agents and a payload of arbitrary type `a`.

```
data CAgentState a = CAgentState AgentId (AgentChan a)
data CAgent s a b = CAgent (CAgentState a → Agent s (CAgentState a, b))
instance Monad (CAgent s a) where {- omitted -}
```

A `CAgent` is, just like before, defined by means of a state monad. It only needs to know its unique identifier and the communication channel to be used for interacting with other agents. This is why, unlike before, the type that holds the state is fixed as `CAgentState`. The function wrapped by `CAgent`, thanks to its codomain `Agent s (CAgentState a, b)`, is able to merge an agent computation within a communicating agent. Intuitively, a `CAgent` can be executed by taking in input the initial `CAgentState` and beliefs base `s`, producing at each intermediate step a value `b` and the new `CAgent` and `Agent` states. The execution flow of a `CAgent` may use functionalities from `Agent`; once the computation moves to the inner monad we gain access to the beliefs base, goals, and backtracking, but all the interaction capabilities are lost until the execution reaches `CAgent` again. Both monads may be concisely defined through the use of the Monad Transformer Library [4], thus many type class instances and utility functions are already given.

A `CAgent` may interact using point-to-point message exchange. The communication interface is summarized below; all functions are blocking and asynchronous, with the exception of `tryRecvMsg` that is non-blocking.

```
myId       :: CAgent s a AgentId
sendMsg    :: AgentId → a → CAgent s a ()
recvMsg    :: CAgent s a (Message a)
tryRecvMsg :: CAgent s a (Maybe (Message a))
```

Given a set of communicating agents, it is straightforward to define a simple module that manages the threads and the synchronization between them.

3 Conclusion and Future Work

We presented a basic architecture based on monads for MAS composed of Haskell agents. Similarly to other solutions, our system provides backtracking capabilities, even if they are limited to the decisions taken between two communication acts.

We have been able to show how the concepts behind MAS can be naturally instantiated in a purely functional language without any particular influence from other paradigms or solutions that may undermine the integration of the framework with the Haskell standard library.

This is still a preliminary work, as the architecture may change to better address the objectives and the prototype of this framework needs to be developed further in order to provide full support for all the described features. Some ideas for future extensions are (1) integrating the backtracking capabilities described in [5], (2) supporting event-based selection of plans, (3) adding communication primitives (e.g. broadcast, multicast), and (4) enriching the communication model with session types [12] in order to check the correctness of ongoing communication along the lines of [1] and [9].

References

1. D. Ancona, S Drossopoulou, and V Mascardi. Automatic generation of self-monitoring MASs from multiparty global session types in Jason. In *In this volume: Proceedings of DALT 2012*, 2012.
2. R.H. Bordini, J.F. Hübner, and M. Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*, volume 8. Wiley-Interscience, 2008.
3. P. Hudak, J. Hughes, S.P. Jones, and P. Wadler. A history of Haskell: being lazy with class. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1, 2007.
4. M. Jones. Functional programming with overloading and higher-order polymorphism. *Advanced Functional Programming*, pages 97–136, 1995.
5. O. Kiselyov, C. Shan, D.P. Friedman, and A. Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, ICFP '05, pages 192–203, New York, NY, USA, 2005. ACM.

6. E.S.L. Lam and M. Sulzmann. Towards agent programming in CHR. *CHR*, 6:17–31, 2006.
7. V. Mascardi, D. Demergasso, and D. Ancona. Languages for programming BDI-style agents: an overview. In *Proceedings of WOA 2005*, pages 9–15. Pitagora Editrice Bologna, 2005.
8. B. O'Sullivan, D.B. Stewart, and J. Goerzen. *Real World Haskell*. O'Reilly Media, 2009.
9. R. Pucella and J.A. Tov. Haskell session types with (almost) no class. In *Haskell*, pages 25–36, 2008.
10. A.S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world : Agents Breaking Away*, MAAMAW '96, pages 42–55, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.
11. M. Sulzmann and E.S.L. Lam. Specifying and Controlling Agents in Haskell.
12. K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *PARLE*, pages 398–413, 1994.
13. A.Z. Wyner. A functional program for agents, actions, and deontic specifications. In Matteo Baldoni and Ulle Endriss, editors, *DALT*, volume 4327 of *Lecture Notes in Computer Science*, pages 239–256. Springer, 2006.

Author Index