Proceedings of the 9th International Workshop on **Declarative Agent Languages and Technologies**

Workshop Chairs:
Chiaki Sakama (Wakayama University, Japan)
Sebastian Sardina (RMIT University, Australia)
Wamberto Vasconcelos (University of Aberdeen, UK)
Michael Winikoff (University of Otago, New Zealand)

http://www.cs.rmit.edu.au/~ssardina/DALT2011

This page intentionally left blank.

Contents

Committees

Programme Committee

Thomas Agotnes, Bergen University College, Norway
Marco Alberti, Universidade Nova de Lisboa, Portugal
Natasha Alechina, University of Nottingham, UK
Cristina Baroglio, University of Torino, Italy
Rafael Bordini, Federal University of Rio Grande do Sul, Brazil
Jan Broersen, University of Utrecht, The Netherlands
Federico Chesani, University of Bologna, Italy
Amit Chopra, University of Trento, Italy
Francesco M. Donini, University of Tuscia, Italy
James Harland, RMIT University, Australia
Andreas Herzig, Paul Sabatier University, France
Koen Hindriks, Delft University of Technology, The Netherlands
Joao Leite, Universidade Nova de Lisboa, Portugal
Yves Lesperance, York University, Canada
Viviana Mascardi, University of Genova, Italy
Nicolas Maudet, University of Paris-Dauphine, France
John-Jules Meyer, University of Utrecht, The Netherlands
Peter Novak, Czech Technical University, Czech Republic
Fabio Patrizi, University of Rome, Italy,
Enrico Pontelli, New Mexico State University, USA
David Pym, University of Aberdeen, UK
Michael Rovatsos, University of Edinburgh, UK
Flavio Correa da Silva, Universidade de Sao Paulo, Brazil
Guillermo Simari, Universidad Nacional del Sur, Argentina
Tran Cao Son, New Mexico State University, USA
Marina De Vos, University of Bath, UK

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)
Pinar Yolum (Bogazici University, Turkey)
Michael Winikoff (University of Otago, New Zealand)

Plan Indexing for State-Based Plans

Louise A. Dennis[1]

Department of Computer Science, University of Liverpool, UK
L.A.Dennis@liverpool.ac.uk

Abstract. We consider the issue of indexing plans (or rules) in the implementation of BDI languages. In particular we look at the issue of plans which are not triggered by the occurence of specific events. The selection of a plan from such a set represents one of the major bottle-necks in the execution of BDI programs. This bottle-neck is particularly obvious when attempting to use program model checkers to reason about such languages.
This paper describes the problem and examines one possible indexing scheme. It evaluates the scheme experimentally and concludes that it is only of benefit in fairly specific circumstances. It then discusses ways the indexing mechanism could be improved to provide wider benefits.

1 Introduction

The implementation of the theory of Beliefs, Desires and Intentions [10] as programming languages has led to a family of languages with many similarities to resolution based logic programming languages and resolution based first-order theorem provers.

A key component of programs written in these languages is the plan or rule base, consisting of programmer designed procedures for achieving intentions. For simplicity we will here refer to these procedures as *plans* and the set of such procedures as the *plan library*.

At given points in the execution of a BDI agent's reasoning cycle the plan library will be accessed in order to determine which plans are applicable given the agent's current set of beliefs and intentions. There are two types of plans used in these languages: *triggered plans* are activated by the occurence of some event (normally the acquisition of a belief or a goal) while *state-based plans* may become active at any time a particular set of beliefs and goals are held. Both types of plans typically have a *guard* – a set of beliefs and goals – that the agent must either believe or intend before the plan is deemed applicable. Both triggers and guards may (and indeed commonly do) contain free variables which are instantiated by unification against the current events, beliefs and goals. A naive implementation of plan selection involves accessing all the plans in the library and then checking each one in turn to see if its trigger event has occurred (in the case of triggered plans) and its guard is satisfied by the agent's state. The time involved in doing this, especially in the presence of large plan libraries, represents a significant bottle-neck in agent execution.

This paper investigates an indexing mechanism for plans in the hope this will reduce the time spent checking guards for applicability. A preliminary implementation is presented and the results of testing this implementation are discussed. The results reveal that there are complex tradeoffs and attention needs to be paid to the efficiency of retrieval from the index if the process is to be of use outside a small number of situations.

1.1 Plans in BDI Languages

The design of plans in BDI languages owes much to logic programming and many implementations are similar to guarded horn clauses. The guards on plans are checked against the agent's *belief base*, Σ, and, in some cases, also against the *goal base*, Γ. In the tradition of logic programming these guards are expressed as first-order predicates with free variables which are instantiated by unification with the goal and belief bases.

In some languages the guards may also be more complex and contain logical formulae constructed using negation, conjuction and disjunction. There may even be deductive rules that operate on the belief and goal bases allowing the agent to conclude that it has some derived belief or goal from the explicit ones stored in its database.

Notation: In what follows we will write triggered plans as $trigger : \{guard\} \leftarrow body$ and state-based plans as $\{guard\} \leftarrow body$. We will refer to the individual forumulae contained in guards as *guard statements*. Where a guard statement states that something is *not* believed by the agent or is *not* a goal of the agent we will refer to this as a *negative guard statement*. Where a guard statement can be deduced using rules we will refer to it as a *deductive guard statement*. All other guard statements, i.e. those that can be inferred directly by inspection of Σ or Γ we will refer to as *explicit guard statements*. Our interest here is primarily in the use of explicit guard statements as a filter on plan selection.

A naive implemention of plan selection retrieves all plans from the agent's plan library and then iterates over this set checking each trigger and guard in turn for applicability. This involves the construction of unifiers and, in some cases, logical deduction, typically in a PROLOG-style. This represents a significant bottle-neck in execution of the agent program. Figure 1 shows profile information generated using the JProfiler tool [1] from running a program written in the GOAL language [5, 8] as implemented in the AIL toolkit [2]. The procedure matchPlans selects all plans and then checks their guards in turn. In the example shown, this procedure is taking up 40% of the execution time. Of the constituent calls within matchPlans most are involved in the checking of guards (ail.syntax.Guards$1.hasNext). We performed similar profiling on all the examples in the AIL GOAL distribution (fewer than half a dozen, sadly). The percentage of time spent on plan selection shown in figure 1 is typical of the programs we examined.

In many agent programs the time taken for plan selection is not of major concern. It typically only becomes a problem in the presence of an extremely large plan library and there are relatively few examples of the use of BDI-programs in

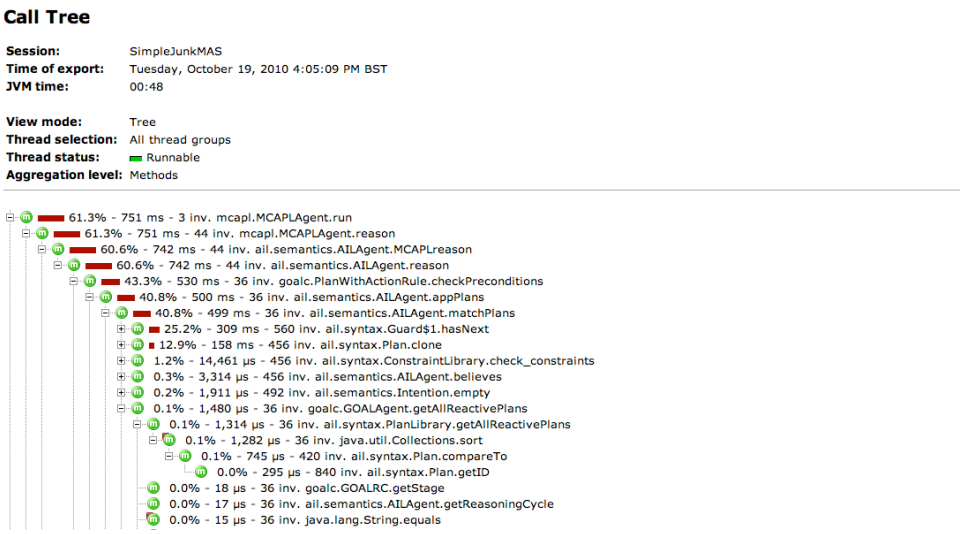


Fig. 1. Profiling the Execution of a GOAL Program

such cases. However there is considerable interest in the community in the use of program model checking for BDI programs [9, 3]. A program model checker uses the actual program as the model which is checked against some property. This causes individual segments of code to be executed many times as the model checker investigates all execution paths and exacerbates the effects of any inefficiences in the program or the underlying language interpreter.

1.2 Indexing

An indexing system allows the fast retrieval of data from a large set by organising the data in some way. For instance, the data can be stored in tuples where some subset of the data is indexed by a key. Data is retrieved by deriving a *query key* (or key set) from the current problem and using that to traverse the index, returning only that data associated with the key in a tuple.

Clearly an index is only of value if the cost of storing data in the index, computing query keys and retrieving data using those keys is lower than the cost of examining the entire set of data and computing the relevance of each item on the fly.

2 Related Work

2.1 Plan Indexing in *Jason*

The *Jason* implementation of AgentSpeak [4] uses triggered plans. Each event is represented by a predicate and the *Jason* implementation generates a *predi-*

cate indicator from these predicates. The predicate indicator is the name of the predicate plus the number of arguments it takes represented as a string.

Consider, for instance the plan: $see(X) : \{garbage(X)\} \leftarrow remove(X)$. This states that if an agent sees some garbage then it removes the garbage. This plan would be indexed by the predicate indicator `see/1` (predicate `see` with 1 argument). *Jason* stores all its plans in tuples of the trigger predicate indicator and a list of all the plans with that trigger. When plans are retrieved the list of predicate indicators alone is searched (using string matching) and then only the plans in the associated tuple are returned.

By indexing plans by their triggers *Jason* is able to considerably speed up the plan selection process. The guards are checked only for those plans whose trigger matches the current event. In our example unifiers for X are only determined and the plan's guard checked, when the agent has just acquired a belief that it can see something, not when any other event occurs.

Jason gains considerable speed up in the plan selection process by this means, and this indexing style is used in many implementations of languages that have triggered plans (e.g. GWENDOLEN [6]).

Unfortunately not all languages use triggered plans. GOAL, for instance, has *state-based plans*, called *conditional actions* which have no trigger and consist of a guard and a plan body alone. The absense of a trigger means that these can not be indexed in the same way.

Aside: It is of note that *Jason* also indexes its belief base with predicate indicators allowing the rapid filtering out of irrelevant beliefs during the checking of plan guards. This technique is trivially applicable in the case of state-based plans and, indeed, the implementation of GOAL discussed in section 4 uses indexing of the belief base in this style.

2.2 Term Indexing

First order theorem provers have long investigated a similar problem, indexing horn clauses for retrieval against a given literal in a variety of situations [11]. This is a similar problem to that addressed by *Jason*. Theorem provers typically work with horn clauses with a single head literal which must be matched to a single literal in the problem at hand.

In our case we are considering a set of literals (the guard) all of which need to match a set of literals in the belief base, but the belief base may also contain literals irrelevant to the current problem.

Theorem provers also deal with a far larger number of clauses which need to be retrieved and much of the research in theorem proving has focused on efficient term representations in order to minimize time searching the index. In situations where we consider agents which may have plans numbering in the tens of thousands the advanced techniques for term indexing developed in the theorem proving field may have a contribution to make to the problem of plan indexing, particularly in languages which have triggered plans.

3 Data Structures

We have chosen to index our plans using a tree-based indexing scheme. Plans are stored as sets at the leaves of a tree data structure and the tree is traversed with reference to the current agent state in order to retrieve plans.

We do not generate an explicit query key before traversing the tree but instead refer to indexing information stored in the agent's belief base. However there is no reason why a query key should not be generated and, indeed, the pseudo-code in the appendices assumes this.

3.1 Plan Keys

We slightly extend the notion of a predicate indicator from *Jason*'s plan indexing algorithm to that of a *plan key*. A plan key is a tuple, (pi, τ) of a predicate indicator, pi, and a type, τ, which refers to the type of entity the associated guard refers to – in the examples we have considered these are either beliefs or goals.

Plan guards are thus associated with a list of plan keys. This list contains a plan key for each explicit guard statement. Negative and deductive guard statements are ignored. We call this list the guard's *index list*.

3.2 Plan Index Trees

We store all our plans in a tree data structure. Each node in the tree is labelled with a plan key and has two subtrees. One subtree contains all plans which have the plan key in their index list, the *must have branch*, and the other subtree contains all the plans which do not, the *don't care branch*. More complex logical formulae and predicates which can be deduced are ignored[1] partly for simplicity and partly because plan keys alone do not provide sufficient information to tell that some guard statement is *not* true in the current agent state. For ease of indexing the plan keys are ordered as the levels in the tree descend [2]. The leaves of the tree are populated with the set of plans which exist on that tree branch.

Say for instance we have three plans:

$$\textbf{plan1} : \{\mathcal{B}a, \mathcal{B}b\} \leftarrow body$$
$$\textbf{plan2} : \{\mathcal{B}b, \mathcal{B}c\} \leftarrow body$$
$$\textbf{plan3} : \{\mathcal{B}c\} \leftarrow body$$

Where a statement $\mathcal{B}b$ means that the agent must believe b for the plan to apply. These would be stored as shown in figure 2.

[1] In the languages we consider deduction is performed by resolution using horn clauses stored in a *rule base* so we simply exclude all predicates that appear in the head of any of the listed horn clauses. It may be that in some languages it is harder to identify and exclude these predicates.

[2] Details of this are discussed in appendix A.

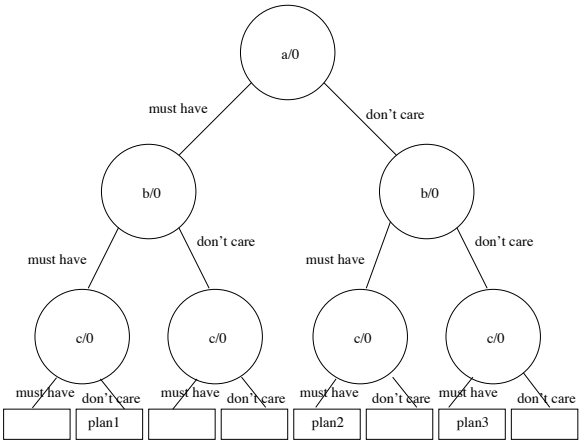


Fig. 2. Example of a Plan Tree

In order to select plans, the program traverses the tree. At each node it checks Σ or Γ (as relevant) for the presence of the appropriate plan key. If it exists then both plans that require that element and plans that do not require that element may be applicable to the current state. As a result, the program searches both sub-trees for relevant plans. If the plan key doesn't exist in the relevant set then only plans that do not require that guard statement will be applicable and the program *only* searches the don't care branch of that node.

So if the belief base contained both a and c then plan 3 would be returned and the branches of the tree highlighted in figure 3 would be explored.

We include pseudo-code for the algorithms to insert plans into the tree and look plans up from the tree in the appendices.

4 Results

We implemented two versions of our plan indexing algorithm in the AIL-based implementation of the GOAL language[3]. The code used in the implementation is available from the author and via the MCAPL sourceforge distribution [4].

The first version indexed by predicate indicators alone and considered only guard statements that referred to beliefs. The second version used plan keys and considered also guard statements referring to goals. We then conducted some simple experiments with the system to see whether the overhead associated with storing and accessing plans from the tree data structure was off-set by the gains in time reduced spent checking the plan guards. In all the experiments the only

[3] AIL is, among other things, a prototyping tool for BDI-languages [2]. The version of GOAL used was based on the semantics described in [8].

[4] http://mcapl.sourceforge.net

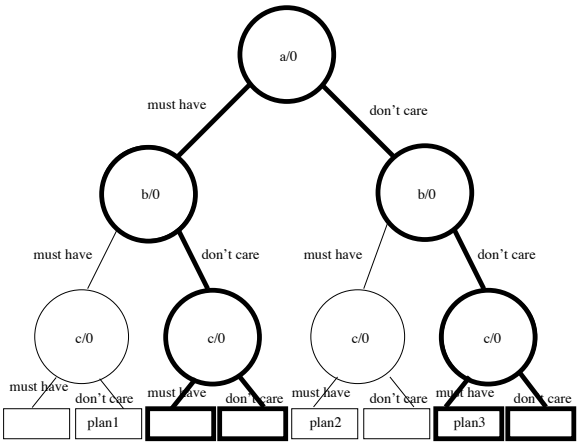


Fig. 3. Example of a Plan Tree Lookup

differences between the code run was the way in which the plans in the library were stored and retrieved, all other parts of the system were identical.

4.1 Experiment 1: Junk Code

In the first example we studied a simple program in which a lead agent communicated with two others simply to ascertain their existence. When they received a query they responded with a simple "ping" message and the program was done. To this were added plans for a Dining Philosopher program which were irrelevant to achieving the goal at hand and these "junk" plans were duplicated n times. The average run time of the system was then plotted against the number of duplications of the redundant code in the system.

We ran each version of the code 100 times and averaged the time taken in order to allow for differences in running time caused by different scheduling behaviour between the agents.

Results The graph in figure 4 shows the result of running the program with up to 24 duplications of the redundant code.

The graph shows that the fastest performance is achieved by the system that organises its plan library as a tree indexed by plan keys that refer to both beliefs and goals and that the performance gain increases as the number of plans increase.

4.2 Experiment 2: Generic Contract Net with Many Goals

The second example we considered was a contract net style program with three agents. One agent wished to achieve a number of goals, none of which it could

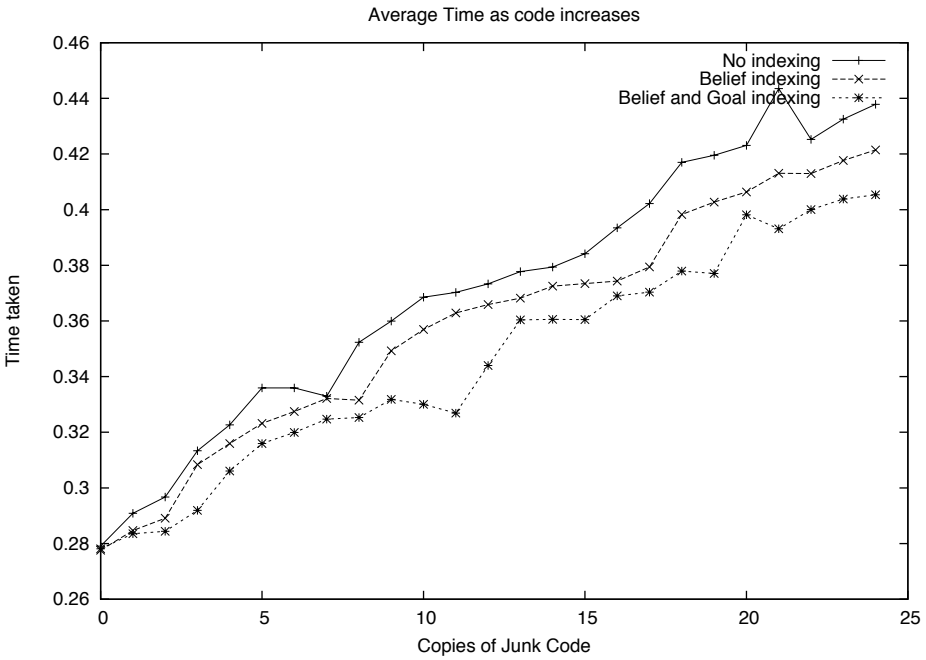


Fig. 4. Plan Indexing Performance in the presence of Redundant Plans

do on its own. The other two agents could both achieve these goals and would "bid" for them. Whichever bid was received first was awarded the "contract" for the goal.

Results Figure 5 shows the results, averaged over 100 runs of the program, as the number of goals to be achieved increases.

In this case it can be seen that the traditional approach of testing all plans is working considerably better than the plan indexing variety and indeed, that as the number of goals increases the efficiency gap between the two methods is going to get significantly worse. This result is obviously disappointing.

4.3 Discussion of the Results

The results are obviously disappointing but it is of interest to consider the differences between the two experimental set ups. Clearly as the guards on plans contain more statements, especially if those statements require further deduction to check, then the system slows down. At the same time as the number of plan keys in the plan tree increases[5] the computation required to traverse the tree also increases and, again, the system slows down.

[5] by this we mean the size of the plan tree increases from experiment to experiment, not within any particular run of the program. That said, if plans were to be added

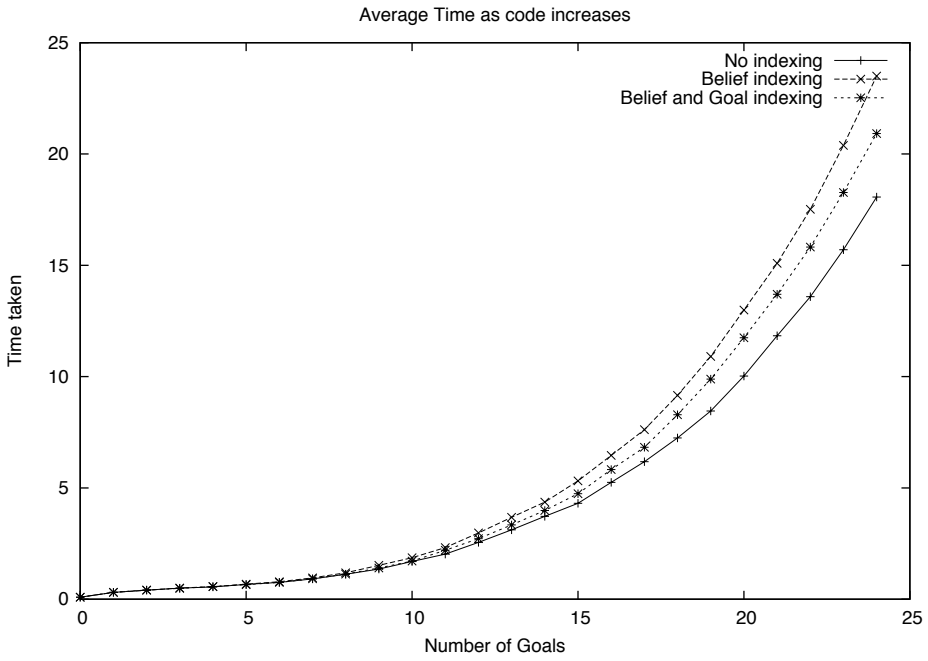


Fig. 5. Plan Indexing Performance in the presence of Multiple Goals

In the first experiment the system contained an increasing number of plans that were never applicable in the agent's current state. Since the plans were duplicated a relatively small number of plan keys were involved in the plan tree. In the second example each new goal introduced into the problem introduced a new plan key into the plan tree, and the increased traversal overhead clearly more than offset any gains being made by filtering out the plans before the guards were considered.

Tentatively, therefore, it seems that plan indexing in this style may be of benefit in programs were there are a large number of plans which refer to a comparatively small number of predicates.

5 Further Work

This paper represents a preliminary investigation into the indexing of state-based plans.

The programs we investigated contained a number of the plan guards involving negative and deductive guard statements. The plan index ignored such guards when indexing the plans. It would be desirable to have a quick way to

dynamically to the agent then dynamic changes to the indexing tree would also be necessary.

eliminate plans based on these types of guards, particularly because they are more complex to check than explicit guard statements. Deductive rules require the (sometimes recursive) checking of the truth of their bodies, while negative guard statements require exhaustive checking of either Σ or Γ.

An obvious extension to the indexing proposed would be to investigate ways to incorporate consideration of these guards into the indexing scheme. A problem with both is that plan keys, alone, are in general not sufficient to provide a quick check of the applicability of the guard. For instance in the case of negative guard statements, simply knowing that there *is* a predicate in the belief base does not necessarily imply *for certain* that the guard statement is true since there may be no unifier that satisfies both it and any other guard statements with which it shares free variables. However it might be possible to perform some limited indexing of negative guards using predicate indicators so long as the guard, itself, had no parameters. Similarly, in the case of ground guards, it might be possible to match directly against the belief base.

Similarly it might be possible to represent belief rules in such a way (assuming the rule bodies do not contain any negative literals) that a judgement could be quickly drawn that the rule *was not* applicable (e.g. as a list of the plan keys that referred to explicit predicates appearing within the rule and which would need to hold for the rule to apply).

A further extension would be to look at techniques from term indexing for sharing subterms and, in particular, free variables. Some of these techniques allow unifiers to be constructed as part of the term lookup process. This would allow plan trees to return *only* plans that matched, together with relevant unifiers and so remove the need for checking the guards at all once the list were found.

Obviously all these approaches run the risk that the plan lookup process becomes even more inefficient when compared to simply iterating over the list of all plans in order to check the guards. Another important aspect of further work is improving the data structure and lookup process currently used for storing the plans. An adaptation of the RETE algorithm [7] would appear to be a promising approach in this direction.

6 Conclusions

This work represents an initial approach to the indexing of plans for retrieval in the plan selection phase of a BDI interpreter. The proposed scheme represents efficiency gains in situations where an agent's plan library contains a large number of plans referring to a small number of predicates. However the scheme is less efficient in situations where many predicates are used. Therefore some care should be taken before deciding to implement such an indexing method.

It seems plausible that the indexing of state-based plans can be improved, even if the approach presented here has not yielded good result. Such improvements would supply gains both in terms of the efficiency of BDI-programs in large scale settings, and in terms of the model checking of such programs.

References

1. JProfiler. http://www.ej-technologies.com/products/jprofiler/overview.html.
2. R. H. Bordini, L. A. Dennis, B. Farwer, and M. Fisher. Automated Verification of Multi-Agent Programs. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 69–78, L'Aquila, Italy, September 2008.
3. R. H. Bordini, L. A. Dennis, B. Farwer, and M. Fisher. Automated Verification of Multi-Agent Programs. In *Proc. 23rd Int. Conf. Automated Software Engineering (ASE)*, pages 69–78. IEEE CS Press, 2008.
4. R. H. Bordini, J. F. Hübner, and R. Vieira. Jason and the Golden Fleece of Agent-Oriented Programming. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*, chapter 1, pages 3–37. Springer-Verlag, 2005.
5. F. S. de Boer, K. V. Hindriks, W. van der Hoek, and J.-J. C. Meyer. A Verification Framework for Agent Programming with Declarative Goals. *J. Applied Logic*, 5(2):277–302, 2007.
6. L. A. Dennis and B. Farwer. Gwendolen: A BDI Language for Verifiable Agents. In B. Löwe, editor, *Logic and the Simulation of Interaction and Reasoning*, Aberdeen, 2008. AISB. AISB'08 Workshop.
7. C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.
8. K. V. Hindriks and M. B. van Riemsdijk. A computational semantics for communicating rational agents based on mental models. In L. Braubach, J.-P. Briot, and J. Thangarajah, editors, *Programming Multi-Agent Systems - 7th International Workshop, ProMAS 2009, Budapest, Hungary, May 10-15, 2009. Revised Selected Papers*, volume 5919 of *Lecture Notes in Computer Science*. Springer, 2010.
9. S.-S. T. Q. Jongmans, K. V. Hindriks, and M. B. van Riemsdijk. Model Checking Agent Programs by Using the Program Interpreter. In *Proc. 11th International Workshop on Computational Logic in Multi-Agent Systems (CLIMA)*, volume 6245 of *LNCS*, pages 219–237. Springer, 2010.
10. A. S. Rao and M. P. Georgeff. BDI agents: From theory to practice. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS)*, pages 312–319, San Francisco, USA, June 1995.
11. R. Sekar, I. V. Ramakrishnan, and A. Voronkov. *Handbook of Automated Reasoning*, volume 2, chapter Term Indexing, pages 1853–1964. North Holland, 2001.

A Insertion Code

Notation: Both the algorithms presented in these appendices recurse through a tree structure. Each node in this tree contains a plan key and two subtrees, the must have branch and the don't care branch. The leaves of the plan tree contain a list of plans. We will treat both nodes and leaves as plan trees. We abuse object oriented notation and refer to the plan key of a plan tree, pt, as pt.pk, the must have branch as pt.musthave, the don't have branch as pt.donthave and the list of plans as pt.plans.

The insertion code recurses through a list of plan keys generated from the guard of a plan, p, and inserts the plan into a pre-existing plan tree (which could

be empty). However where the plan contains a plan key that does not already exist in the tree the tree must be modified with new nodes for that plan key.

The algorithm takes as inputs the list of plan keys associated with the guard of plan, p, and the pre-existing plan tree into which the plan is to be inserted.

Code fragment 1.1 Insert a Plan into an Index Tree

```
addPlan(PlanKey List pks, Plan p, PlanTree pt)                    1
  if (pks is empty)                                               2
    if (pt is a leaf)                                             3
       add p to pt.plans                                          4
       return pt                                                  5
    else                                                          6
       replace pt.dontcare with addPlan(pks, p, pt.dontcare)      7
       return pt                                                  8
  else                                                            9
    if (pt is a leaf)                                             10
       create a new plan tree node, n,  where                     11
         n.pk equals the head of pks                              12
         n.dontcare is a plantree leaf where                      13
                n.plans = pt.plans                                14
         n.musthave is the result of                              15
                addPlan(tail of pks, p, new plantree leaf)        16
       return n                                                   17
    else                                                          18
       if pt.pk equals the head of pks                            19
          replace pt.musthave with                                20
             addPlan(tail of pks, p, pt.musthave)                 21
          return pt                                               22
       else if the head of pks is ordered after pt.pk             23
          replace pt.dontcare with                                24
             addPlan(pks, p, pt.dontcare)                         25
          return pt                                               26
       else                                                       27
         create a new plan tree node, n,  where                   28
            n.pk equals the head of pks                           29
            n.dontcare is pt                                      30
            n.musthave is the result of                           31
                addPlan(tail of pks, p, pt.musthave)              32
         return n                                                 33
```

B Lookup Code

Notation: The notation used in this code is explained in appendix A.

The algorithm takes a list of plan keys (generated from the agent's belief and goal bases) and which have been ordered according to some ordering on plan keys. The algorithm recurses through the tree comparing the plan key at each node against the supplied list of plan keys.

Code fragment 2.1 Look up plans in the Index

```
lookup(PlanKey List pks, Plan Tree pt)
  if (pt is a leaf)
     return pt.plans
  else
     if pt.pk is in pks
        return
          lookup(pks, pt.musthave)
          AND
          lookup(pks, pt.dontcare)
     else
        return lookup(pks, pt.dontcare)
```

Detecting Conflicts in Commitments

Akın Günay and Pınar Yolum

Department of Computer Engineering
Boğaziçi University
34342, Bebek, İstanbul, Turkey
[akin.gunay,pinar.yolum]@boun.edu.tr

Abstract. Commitments are being used widely to specify interaction among autonomous agents in multiagent systems. While various formalizations for commitments and its life cycle exist, there has been little work that studies commitments in relation to each other. However, in many situations, the content and state of one commitment may render another commitment useless or even worse create conflicts. This paper studies commitments in relation to each other. Following and extending an earlier formalization by Chesani *et al.*, we identify key conflict relations among commitments. The conflict detection can be used to detect violation of commitments before the actual violation occurs during agent interaction (run-time) and this knowledge can be used to guide an agent to avoid the violation. It can also be used during creation of multiagent contracts to identify conflicts in the contracts (compile-time). We implement our method in $\mathcal{REC}$ and present a case study to demonstrate the benefit of our method.

1 Introduction

A commitment is a contract from one agent to another to bring about a certain property [2, 11]. For instance a merchant and a customer may have a contract, in which the customer agrees to pay to the merchant in return of the delivery of a good. This contract can be represented as a commitment, in which the merchant will be committed to the customer to deliver a good, if it is paid. In this commitment, the merchant is the debtor, the customer is the creditor, delivery of the good is the property and payment is the condition of the commitment.

Commitments are dynamic entities and they evolve over time according to the occurrence of events in the environment they exist. To represent the dynamic nature of a commitment, the commitment is associated with a state and transitions between states are defined over a set of operations. These states and operations are called the life cycle of a commitment. Previous work has studied the life cycle of individual commitments in detail [5, 10, 12–14]. However, individual life cycle of a commitment provides limited information to manage and monitor commitments in a multiagent system.

Example 1 Consider the two commitments: The first commitment is between a merchant and a customer, which states, if the customer pays for some goods, then

the merchant will be committed to deliver the goods within the next day. The second commitment is between the merchant and a delivery company, which states, if the merchant pays for the delivery of some goods, then the delivery company will be committed to deliver these goods to the customer within three to five days.

If we examine the commitments in Example 1 individually according to the life cycle of a commitment, then we detect no problem, since both commitments are valid. However if we examine the two commitments together, then it is obvious that the first commitment is going to be violated. This is because the merchant commits to the customer to deliver the goods in the next day, but the commitment with the delivery company cannot deliver before three days. This example demonstrates that if we examine commitments in a multiagent system together, instead of examining them individually, we can detect possible problems as early as the commitments created.

The above example demonstrates the benefit of examining commitments together in order to detect possible problems in advance at run-time. However, the same idea can also be used in an offline manner to detect inconsistencies in multiagent contracts. A multiagent contract is simply a set of related commitments. A major issue in a contract is the consistency between the commitments of the contract. If there are inconsistencies between commitments, then one commitment may not be satisfied without violating other commitment(s), which causes a participating agent to find itself in a problematic situation. In order to avoid such situations we should examine the commitments in a contract together and eliminate the inconsistencies before creating the contract.

In this paper, we present a method that examines commitments in a multiagent system together in order to capture commitment pairs such that one commitment cannot be satisfied without violating the other commitment. The major benefit of our approach is capturing such situations in advance before a commitment is actually violated. Hence, it makes it possible to take early action to avoid future problems. We use an extended version of event calculus formalization of commitments proposed by Chesani *et al.* [3]. We extend this formalization by introducing the axioms for conditional commitments, which are essential to fully model commitments. On top of this formalization, we identify and develop a set of axioms to reason about inconsistencies between commitments. To achieve this, we define a *conflict* relation between commitments. The conflict relation indicates that a commitment cannot be satisfied without violating another. Our formalism and approach to capture inconsistencies in contracts is executable in $\mathcal{REC}$ [3], which is a tool for reasoning based on reactive event calculus.

Our major contributions in this paper are (1) extending the previous event calculus formalization of commitments by introducing conditional commitments; (2) introducing new axioms to define a conflict relation between commitments; (3) use of conflicts of commitments to detect violation of a commitment in advance before the commitment is actually violated.

The rest of the paper is structured as follows. Section 2 reviews commitments and describes the extended formal model of commitments in event calculus.

Section 3 describes the conflict relations of commitments and how they can be used to capture inconsistencies in contracts. Section 4 examines our approach over a running example. Finally, Section 5 concludes the paper with further discussion.

2 Background: Commitments

A commitment is made from one agent to another to bring about a certain property [2, 11]. By participating in a commitment, the participating agents put themselves under the obligation to satisfy the requirements of the commitment. A commitment is represented as $C(x, y, q, p)$, which states that the *debtor* agent x will be committed to the *creditor* agent y to bring about the *property* p, if the *condition* q is satisfied.

In order to represent real world situations more precisely, the condition and the property of a commitment may be associated with temporal quantifiers, which defines when and how the condition and the property must be satisfied. In general there are two types of temporal quantifiers [3, 7]. Existential temporal quantifier states that the associated property must be satisfied at least at one moment within a given interval of moments. Universal temporal quantifier states that the associated property must be satisfied at all moments within a given interval of moments.

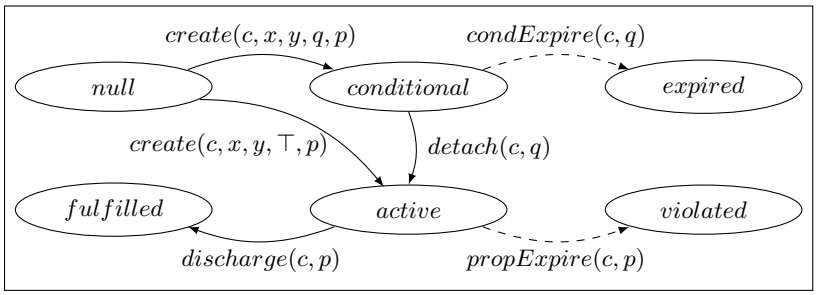


Fig. 1. Life cycle of a commitment.

A commitment is a dynamic entity and has a life cycle. Each commitment has a state that evolves over time. The state of a commitment changes according to a set of operations that can be performed by the participating agents of the commitment. The state of a commitment also changes, when the condition or the property of the commitment is not satisfied according to the associated temporal quantifier. In this paper we use the commitment life cycle that we present in Figure 1. In this life cycle we skip operations such as delegate and cancel, which are used in previous work, for simplicity.

The following operations can be performed on a commitment.

– $create(c, x, y, q, p)$: Creates a new commitment c, in which x is the debtor, y is the creditor, q is the condition and p is the property of the commitment. This operation can only be performed by the debtor x.
– $detach(c, q)$: Detaches the condition q from the commitment c. This operation can only be performed by the creditor y.
– $discharge(c, p)$: Discharges the commitment c, when the property p is satisfied. This operation can only be performed by the debtor x.

$condExpire(c, q)$ and $propExpire(c, p)$ are meta operations that show that the condition q and property p of the commitment c are violated according to the associated temporal quantifier, respectively. A commitment can be in one of the following states.

– *null*: A dummy state, which is assigned to a commitment before its creation.
– *conditional*: The condition of the commitment is not satisfied yet. This is like an offer and neither the debtor nor the creditor is under the obligation of the commitment.
– *expired*: The condition of the commitment is violated considering to the associated temporal quantifier. Hence, the commitment expires. This usually corresponds to the rejection of an offer.
– *active*: The debtor is under the obligation of the commitment to satisfy the property of the commitment. Otherwise, the debtor may be punished depending on the properties of the underlying environment.
– $fulfilled$: The property of the commitment is satisfied and the debtor fulfilled its commitment. The debtor is no more under the obligation of the commitment.
– *violated*: The property of the commitment is not satisfied and the debtor violates its commitment.

2.1 Event Calculus

Event calculus is a formalism to reason about events and their effects. An *event* in event calculus initiates or terminates a *fluent*. A fluent is a property whose value is subject to change over time. A fluent starts to hold after an event that initiates it and ceases to hold after an event that terminates it. Event calculus was introduced by Kowalski and Sergot [6] and extended by Shanahan [9].

In the following, $\mathcal{E}$ is a sort of events (variables $E, E_1, E_2, ...$), $\mathcal{F}$ is a sort of fluents (variables $F, F_1, F_2, ...$) and $\mathcal{T}$ is a sort for integer time moments (variables $T, T_1, T_2, ...$), which are ordered by the $<$ relation that is transitive and asymmetric. Variables are universally quantified, unless otherwise specified.

The event calculus predicates are as follows [9].

– $initiates(E, F, T)$: Fluent F starts to hold after event E at time T.
– $terminates(E, F, T)$: Fluent F ceases to hold after event E at time T.
– $initially(F)$: Fluent F holds from time 0.
– $happens(E, T)$: Event E occurs at time T.

– $holdsAt(F, T)$: Fluent F holds at time T.
– $clipped(F, T_1, T_2)$: Fluent F is terminated between times T_1 and T_2.

In the following we present the axiomatisation of the event calculus predicates.

Axiom 1

$$holdsAt(F, T) \leftarrow$$
$$initially(F) \wedge \neg clipped(F, 0, T) \blacksquare$$

Axiom 1 states that the fluent F holds at time T, if it held at time 0 and has not been terminated between 0 and T.

Axiom 2

$$holdsAt(F, T_2) \leftarrow$$
$$happens(E, T_1) \wedge initiates(E, F, T_1) \wedge \neg clipped(F, T_1, T_2) \wedge T_1 < T_2 \blacksquare$$

Axiom 2 states that the fluent F holds at time T, if the fluent F is initiated by an event E at some time T_1 before T_2 and the fluent F has not been terminated between T_1 and T_2.

Axiom 3

$$clipped(F, T_1, T_2) \leftrightarrow$$
$$\exists E, T[happens(E, T) \wedge terminates(E, F, T) \wedge T_1 < T < T_2] \blacksquare$$

Axiom 3 states that fluent F is clipped between T_1 and T_2, if and only if there is an event E happens between T_1 and T_2 and terminates the fluent F .

2.2 Formalizing Commitments in Event Calculus

In the rest of this section, we present the event calculus axioms that represent the life cycle of a commitment. These axioms extend the axioms introduced by Chesani *et al.* [3] by introducing the conditional commitment, which is not present in the axioms of Chesani *et al.*. The conditional commitment is essential to represent a complete life cycle of a commitment.

In the following, we use $\mathcal{A}$ as a sort of agents (variables $A, A_1, A_2, ...$), $\mathcal{P}$ as the set of properties (variables $P, P_1, ..., Q, Q_1, ...$), $\mathcal{C}$ as the set of commitments (variables $C, C_1, C_2, ...$) and $\mathcal{S}$ as the set of commitment states. We represent an existentially quantified moment interval as $e(T_1, T_2)$, a universally quantified moment interval as $u(T_1, T_2)$ and a property as $prop(\mathcal{Q}(T_1, T_2), F)$, in which $\mathcal{Q} = \{e, u\}$. We represent a commitment as $comm(A_1, A_2, Q, P)$. The state of a commitment is represented by the fluent $status(C, S)$. We also use predicates $conditional(C, T)$, $expired(C, T)$, $active(C, T)$, $violated(C, T)$ and $fulfilled(C, T)$ to represent that at moment T the commitment C is in conditional, expired, active, violated and fulfilled state, respectively.

Axiom 4 (Creating active commitment)
The $create(E, A, C, T)$ operation performed by the debtor A through the occurrence of event E at moment T creates the commitment C in active state.

$$initiates(E, status(comm(A_1, A_2, \top, P), active), T) \leftarrow$$
$$\quad create(E, A_1, comm(A_1, A_2, \top, P), T) \blacksquare$$

Axiom 5 (Creating conditional commitment)
The $condCreate(E, A, C, T)$ operation performed by the debtor A through the occurrence of event E at moment T creates the commitment C in conditional state.

$$initiates(E, status(comm(A_1, A_2, Q, P), conditional), T) \leftarrow$$
$$\quad condCreate(E, A_1, comm(A_1, A_2, Q, P), T) \blacksquare$$

Note that, while creating a commitment in active state in Axiom 4, we use a variable Q that corresponds to a condition. We use this variable just as a place holder in order not to introduce a different syntax for active and conditional commitments. This variable is not used by the axioms that deal with active commitments, hence its value has no effect on the life cycle of the commitment.

Axiom 6 (Expiration of conditional commitment)
The state of a commitment changes from conditional to expired, when the condition of the commitment is not detached by the creditor within the corresponding moment interval.

$$initiates(E, status(comm(A_1, A_2, Q, P), expired), T) \leftarrow$$
$$\quad condExpire(E, comm(A_1, A_2, Q, P), T)$$

$$terminates(E, status(comm(A_1, A_2, Q, P), conditional), T) \leftarrow$$
$$\quad condExpire(E, comm(A_1, A_2, Q, P), T)$$

A commitment in conditional state with an existentially quantified condition expires, when the commitment is still in conditional state after the corresponding moment interval.

$$condExpire(E, comm(A_1, A_2, prop(e(T_1, T_2), F), P), T) \leftarrow$$
$$\quad conditional(comm(A_1, A_2, prop(e(T_1, T_2), F), P), T) \wedge T > T_2$$

A commitment in conditional state with a universally quantified condition expires, when the condition does not hold at any moment within the corresponding moment interval.

$$condExpire(E, comm(A_1, A_2, prop(u(T_1, T_2), F), P), T) \leftarrow$$
$$\quad conditional(comm(A_1, A_2, property(u(T_1, T_2), F), P), T) \wedge$$
$$\quad \neg holdsAt(F, T) \wedge T_1 \leq T \wedge T \leq T_2 \blacksquare$$

Axiom 7 (Detaching conditional commitment)
The state of a commitment changes from conditional to active, when the commitment is detached by the creditor through the occurrence of event E at moment T.

$$initiates(E, status(comm(A_1, A_2, Q, P), active), T) \leftarrow$$
$$\quad detach(E, A_2, comm(A_1, A_2, Q, P), T)$$

$$terminates(E, status(comm(A_1, A_2, Q, P), conditional), T) \leftarrow$$
$$\quad detach(E, A_2, comm(A_1, A_2, Q, P), T)$$

A commitment in conditional state with an existentially quantified condition is detached, when the event E initiates the fluent F of the condition within the corresponding moment interval.

$$detach(E, A_2, comm(A_1, A_2, prop(e(T_1, T_2), F), P), T) \leftarrow$$
$$\quad conditional(comm(A_1, A_2, prop(e(T_1, T_2), F), P), T) \wedge$$
$$\quad initiates(E, F, T) \wedge T_1 \leq T \wedge T \leq T_2$$

A commitment in conditional state with a universally quantified condition is detached, when the commitment is still in conditional state after the end of the corresponding moment interval of the condition.

$$detach(E, A_2, comm(A_1, A_2, prop(u(T_1, T_2), F), P), T) \leftarrow$$
$$\quad conditional(comm(A_1, A_2, prop(u(T_1, T_2), F), P), T) \wedge T > T_2 \blacksquare$$

Axiom 8 (Violating active commitment)
The state of a commitment changes from active to violated, when the property of the commitment is not discharged by the debtor within the corresponding time interval.

$$initiates(E, status(comm(A_1, A_2, Q, P), violated), T) \leftarrow$$
$$\quad propExpire(E, comm(A_1, A_2, Q, P), T)$$

$$terminates(E, status(comm(A_1, A_2, Q, P), active), T) \leftarrow$$
$$\quad propExpire(E, comm(A_1, A_2, Q, P), T)$$

A commitment in active state with an existentially quantified property expires, when the commitment is still in active state after the corresponding moment interval.

$$propExpire(E, comm(A_1, A_2, Q, prop(e(T_1, T_2), F)), T) \leftarrow$$
$$\quad active(comm(A_1, A_2, Q, prop(e(T_1, T_2), F)), T) \wedge T > T_2$$

A commitment in conditional state with a universally quantified property expires, when the property does not hold at any moment within the corresponding moment interval.

$$propExpire(E, comm(A_1, A_2, Q, prop(u(T_1, T_2), F)), T) \leftarrow$$
$$active(comm(A_1, A_2, Q, prop(u(T_1, T_2), F)), T)$$
$$\neg holdsAt(F, T) \wedge T_1 \leq T \wedge T \leq T_2 \blacksquare$$

Axiom 9 (Discharging active commitment)
The state of a commitment changes from active to fulfilled, when the commitment is discharged by the debtor through the occurrence of event E at moment T.

$$initiates(E, status(comm(A_1, A_2, Q, P), fulfilled), T) \leftarrow$$
$$discharge(E, A_1, comm(A_1, A_2, Q, P), T)$$

$$terminates(E, status(comm(A_1, A_2, Q, P), active), T) \leftarrow$$
$$discharge(E, A_1, comm(A_1, A_2, Q, P), T)$$

A commitment in active state with an existentially quantified property is discharged, when the event E initiates the fluent F of the property within the corresponding moment interval.

$$discharge(E, A_1, comm(A_1, A_2, Q, prop(e(T_1, T_2), F)), T) \leftarrow$$
$$active(comm(A_1, A_2, Q, prop(e(T_1, T_2), F)), T) \wedge$$
$$initiates(E, F, T) \wedge T_1 \leq T \wedge T \leq T_2$$

A commitment in active state with a universally quantified property is discharged, when the commitment is still in active state after the end of the corresponding moment interval of property.

$$discharge(E, A_1, comm(A_1, A_2, Q, prop(u(T_1, T_2), F), T) \leftarrow$$
$$active(comm(A_1, A_2, Q, prop(u(T_1, T_2), F)), T) \wedge T > T_2 \blacksquare$$

3 Conflicting Commitments

Our aim in this paper is to develop a method to capture commitment pairs, such that one of the commitments cannot be satisfied without violating the other commitment. We call such commitment pairs as conflicting commitments. Since satisfaction and violation of a commitment is determined according to the satisfaction and violation of its committed property, in order to capture conflicting commitments, we should first define conflicting properties. Similar to the conflicting commitments, two properties conflict with each other if one of the properties cannot be satisfied without violating the other. The idea of our method is, if properties of two commitments conflict with each other, then the commitments also conflict with each other.

3.1 Conflicting Properties

In order to define a conflict between two properties we have to know the meaning of the fluents involved by these properties in the intended domain of the

underlying multiagent system. This is necessary since without such a domain knowledge, fluents are meaningless. In order to formalize this situation, we use a *fluent conflict* relation. Two fluents conflict with each other if it is not possible to hold both fluents at the same time in a given domain.

Definition 1. *Fluents F_1 and F_2 in a given domain $\mathcal{D}$ are in a fluent conflict, if it is not possible to hold both fluents at the same moment in the domain $\mathcal{D}$. The predicate $fluentConf(F_1, F_2, \mathcal{D})$ indicates the fluent conflict between the fluents F_1 and F_2 in domain $\mathcal{D}$.*

Example 2 Consider the fluent $carRented(C, P)$, which means the car C is rented to the person P. Now consider the same fluent with two different set of grounded values, $carRented(herbie, sally)$ and $carRented(herbie, linus)$. The first fluent states that the car *herbie* is rented to *sally* and the second fluent states that the car *herbie* is rented to *linus*. As a domain knowledge, we know that the same car cannot be rented to two different person at the same time. Hence, we also know $carRented(herbie, sally)$ and $carRented(herbie, linus)$ cannot hold at the same moment. As result these two fluents conflict with each other.

The above case can be represented as the following rule in domain D.

$$fluentConf(carRented(C, P_1), carRented(C, P_2), D) \leftarrow$$
$$isCar(C) \wedge isPerson(P_1) \wedge isPerson(P_2) \wedge P_1 \neq P_2$$

In the rest of the paper we assume that the domain dependent fluent conflict knowledge is already present.

Definition 2. *Properties P_1 and P_2 are in a property conflict relation if it is not possible to satisfy one property without violating the other. The predicate $propConf(P_1, P_2)$ indicates a conflict between properties P_1 and P_2.*

Occurrence of a property conflict depends on the existence of a fluent conflict between the fluents of the properties as we discussed above and the temporal quantifiers of the properties. There are three possible cases considering temporal quantifiers of the properties as we present below.

Existential-Existential A property conflict relation between two existentially quantified properties occurs if and only if fluents of these properties are in fluent conflict relation and both properties must be satisfied at a common moment.

Axiom 10

$$propConf(prop(e, (T_1, T_2), F_1), prop(e, (T_3, T_4), F_2) \leftarrow$$
$$fluentConf(F_1, F_2, D) \wedge T_1 = T_2 = T_3 = T_4 \blacksquare$$

Example 3 Consider the properties $prop(e(1,1), isRented(herbie, sally))$ and $prop(e(1,1), isRented(herbie, linus))$. The first property is satisfied if the car

herbie is rented by *sally* exactly at moment 1 and the second property is satisfied if the car *herbie* is rented by *linus* exactly at moment 1. As domain knowledge we know that the fluents $isRented(herbie, sally)$ and $isRented(herbie, linus)$ have a fluent conflict, which means the car *herbie* cannot be rented both by *sally* and *linus* at moment 1. Thus, it is not possible to satisfy one of these properties without violating the other, therefore these two properties are in property conflict.

Note that, in order to have a property conflict in the case of existentially quantified properties, the moment intervals of the properties must refer exactly to the same moment. If the moment interval is not just on a moment, it is possible to satisfy both properties, even if the fluents of the properties have a fluent conflict.

Existential-Universal A property conflict between an existentially and a universally quantified property occurs if there is a fluent conflict between the fluents of the properties and the moment interval of the universally quantified property covers the moment interval of the existentially quantified property.

Axiom 11

$$propConf(prop(e(T_1, T_2), F_1), prop(u(T_3, T_4), F_2) \leftarrow \\ fluentConf(F_1, F_2, D) \wedge T_3 \leq T_1 \wedge T_2 \leq T_4 \blacksquare$$

Example 4 Consider the properties $prop(e(1,3), isRented(herbie, sally))$ and $prop(u(1,5), isRented(herbie, linus))$. The first property is satisfied if the car *herbie* is rented by *sally* at least at one moment between 1 and 3 and the second property is satisfied if the car *herbie* is rented by *linus* at all moments between 1 and 5. As domain knowledge we know that the fluents $isRented(herbie, sally)$ and $isRented(herbie, linus)$ have a fluent conflict, which means the car *herbie* cannot be rented both by *sally* and *linus* between moments 1 and 3. If *herbie* is rented to *sally* at any moment between 1 and 3 to satisfy the first property, then it is not possible to satisfy the second property. If *herbie* is rented to *linus* at all moment between 1 and 5 to satisfy the second property, then it is not possible to satisfy the first property. Thus, it is not possible to satisfy one of these properties without violating the other and these two properties are in property conflict.

Universal-Universal A property conflict between two universally quantified properties occurs if there is a fluent conflict between the fluents of the properties and the moment intervals of the properties overlap with each other.

Axiom 12

$$propConf(prop(u(T_1, T_2), F_1), prop(u(T_3, T_4), F_2)) \leftarrow \\ fluentConf(D, F_1, F_2) \wedge \\ [T_1 \leq T_3 \wedge T_3 \leq T_2 \vee T_3 \leq T_1 \wedge T_1 \leq T_4] \blacksquare$$

Example 5 Consider the properties $prop(u(1,5), isRented(herbie, sally))$ and $prop(u(3,7), isRented(herbie, linus))$. The first property is satisfied if the car *herbie* is rented by *sally* at all moments between 1 and 5 and the second property is satisfied if the car *herbie* is rented by *linus* at all moments between 3 and 7. As domain knowledge we know that the fluents $isRented(herbie, sally)$ and $isRented(herbie, linus)$ have a fluent conflict, which means the car *herbie* cannot be rented both by *sally* and *linus* between moments 3 and 5. If *herbie* is rented to *sally* at all moments between 1 and 5 to satisfy the first property, then it is not possible to satisfy the second property. If *herbie* is rented to *linus* at all moment between 3 and 7 to satisfy the second property, then it is not possible to satisfy the first property. Thus, it is not possible to satisfy one of these properties without violating the other and these two properties are in property conflict.

3.2 Conflict Relations between Commitments

Now we define the first type of conflict relation between two commitments using the property conflict relation that we defined before. A *commitment conflict* relation may occur between two active commitments, which indicates that one of the commitments cannot be satisfied without violating the other.

Definition 3. *Given the two commitments C_1 and C_2 with properties P_1 and P_2, respectively, there is a commitment conflict between commitments C_1 and C_2, if the properties P_1 and P_2 have a property conflict and commitments C_1 and C_2 are in active state. The fluent $commConf(C_1, C_2)$ indicates a conflict between commitments C_1 and C_2.*

Axiom 13 (Commitment conflict)

$$initiates(E, commConf(comm(\_,\_,Q_1,P_1), comm(\_,\_,Q_2,P_2)), T) \leftarrow$$
$$active(comm(\_,\_,Q_1,P_1), T) \wedge active(comm(\_,\_,Q_2,P_2), T) \wedge$$
$$propConf(P_1, P_2) \blacksquare$$

Example 6 Consider the commitments $comm(charlie, sally, \top, prop(u(1,5), isRented(herbie, sally)))$ and $comm(charlie, linus, \top, prop(u(3,7), isRented(herbie, linus)))$. The first commitment states that *charlie* is committed to *sally* to rent *herbie* at all moments between 1 and 5 and the second commitment states that *charlie* is committed to *linus* to rent *herbie* at all moments between 3 and 7. We know that there is a property conflict between the properties of these two commitments. Hence it is not possible to satisfy both properties. As result, it is also not possible to satisfy one commitment without violating the other. If *charlie* rents *herbie* to *sally* and satisfies his commitment to *sally*, then he violates his commitment to *linus*. On the other hand, if *charlie* rents *herbie* to *linus* and satisfies his commitment to *linus*, then he violates his commitment to *sally*.

Note that, the debtors and the creditors are actually irrelevant while capturing commitment conflicts. The only relevant factors are the property conflict between the properties of the commitments and the states of the commitments.

The commitment conflict relation that we discuss above points out to a definite violation of at least one commitment. This happens, since both commitments are in active state. However, this is not the case if at least one of the commitments are not in active but conditional state. In this case, occurrence of a commitment conflict and violation of the commitment depends on the satisfaction of the condition(s) of the commitment(s). In the following we define another relation, which we call *conditional commitment conflict* relation between commitments to reflect such situations.

Definition 4. *Given the two commitments C_1 and C_2 with conditions Q_1 and Q_2, and properties P_1 and P_2, respectively, there is a conditional commitment conflict between commitments C_1 and C_2 if the properties P_1 and P_2 have a property conflict and at least one of the commitments C_1 or C_2 is in conditional state, if not in active state. The fluent $condCommConf(C_1, C_2)$ indicates a conditional commitment conflict between commitments C_1 and C_2.*

Axiom 14 defines the conditional commitment conflict, where one commitment is in active state and the other commitment is in conditional state. Note that, if the condition of the commitment in conditional state is satisfied, then the conditional conflict relation between the commitments is terminated and the commitment conflict relation is initiated.

Axiom 14 (Conditional commitment conflict (active-conditional))

$$initiates(E, condCommConf(comm(\_, \_, Q_1, P_1), comm(\_, \_, Q_2, P_2)), T) \leftarrow active(comm(\_, \_, Q_1, P_1), T) \wedge conditional(comm(\_, \_, Q_2, P_2), T) \wedge propConf(P_1, P_2) \blacksquare$$

Axiom 15 defines the conditional commitment conflict, where both commitments are in conditional state. Note that, if one of the conditions is satisfied, then Axiom 14 applies.

Axiom 15 (Conditional commitment conflict (conditional-conditional))

$$initiates(E, condCommConf(comm(\_, \_, Q_1, P_1), comm(\_, \_, Q_2, P_2)), T) \leftarrow conditional(comm(\_, \_, Q_1, P_1), T) \wedge conditional(comm(\_, \_, Q_2, P_2), T) \wedge propConf(P_1, P_2) \blacksquare$$

Example 7 Consider the commitments $comm(charlie, sally, prop(e(1,3), isPaid(sally)), prop(u(3,5), isRented(herbie, sally)))$ and $comm(charlie, linus, \top, prop(u(3,7), isRented(herbie, linus)))$. The first commitment is in conditional state, which means that *charlie* will be committed to *sally* to rent *herbie* at all moments between 3 and 5, if *sally* pays the rent between moments 1 and 3 and the second commitment is in active state, which means that *charlie* is committed to *linus* rent *herbie* at all moments between 3 and 7. We know that there is a property conflict between the properties of these two commitments. In this case, occurrence of a commitment conflict depends on the satisfaction of the condition of the first commitment. If *sally* does not pay, then the first commitment expires and the conflict is resolved automatically.

4 A Commitment Conflict Scenario

We present a scenario to demonstrate how the conflict relations that we define can be used to capture violation of a commitment in a multiagent system, before the violation actually occurs. We also implement this scenario in the REC tool and we present the trace of the execution.

Scenario Description There are two customers Sally and Linus and a car rental agent Charlie. Charlie has a commitment in conditional state to Sally, which states, if Sally pays the rent between days one and three, then Charlie will be committed to Sally to rent a car to her between days four and seven. Charlie has also a commitment in conditional state to Linus, which states, if Linus uses a promotion ticket between days one and five, then Charlie will be committed to Linus to rent a car to him for days six and seven for a cheaper price. We also know that Charlie has just one car, namely Herbie, available for rent for the next seven days.

In this scenario, it is obvious that Charlie will get into trouble if both Sally and Linus satisfy the conditions of their own commitments. If this happens, Charlie will have active commitments to both of them to rent a car at the same dates. However, Charlie has only one car to rent at that dates, hence he cannot satisfy one of these commitments without violating the other. This situation cannot be captured at run time by considering these two commitments individually, at least until one of the commitments is actually violated. However, if we consider these commitments together, we can capture that there is a potential problem, immediately when the two commitments are created.

Let us first define the fluents, the events and effects of the events on the fluents in our scenario.

Fluents:
- $rentPaid(C, Car, A)$: The customer C paid the rent for the car Car to the agency A.
- $promeUsed(C, Car, A)$: The customer C used a promotion ticket for the car Car to the agency A.
- $rented(Car, C)$: The car Car is rented to the customer C.

Events:
- $payRent(C, Car, A)$: The customer C pays the rent for the car Car to the agency A.
- $usePromo(C, Car, A)$: The customer C gives the promotion ticket for the car Car to the agency A.
- $rent(A, Car, C)$: The agency A rents the car Car to the customer C.

Effects of events on fluents:
- $initiates(payRent(C, Car, A), rentPaid(C, Car, A), T)$
- $initiates(usePromo(C, Car, A), promoUsed(C, Car, A), T)$
- $initiates(rent(A, Car, C), rented(Car, C), T)$

Finally we define the creation of commitments as result of the event as follows.

$$ccreate(offer(A, Car, C)), A, comm(A, C,\\ prop(e(T, T_2), rentPaid(C, Car, A)),\\ prop(u(T_3, T_4), rented(Car, C))), T) \leftarrow\\ T_2\ is\ T + 2 \wedge T_3\ is\ T + 3 \wedge T_4\ is\ T_3 + 3$$

$$ccreate(promote(A, Car, C)), A, comm(A, C,\\ prop(e(T, T_2), promoUsed(C, Car, A)),\\ prop(u(T_3, T_4), rented(Car, C))), T) \leftarrow\\ T_2\ is\ T + 4 \wedge T_3\ is\ T + 5 \wedge T_4\ is\ T_3 + 1$$

Assume that the domain dependent fluent conflicts are already defined and following list of happens statement shows the execution of the system.

$$happens(offer(charlie, herbie, sally), 1)$$
$$happens(promote(charlie, herbie, linus), 1)$$
$$happens(payRent(sally, herbie, charlie), 2)$$
$$happens(rent(charlie, herbie, sally), 3)$$
$$happens(usePromo(linus, herbie, charlie), 4)$$

Let us trace the execution. At moment 1, *charlie* creates two conditional commitments as described in the scenario. Let us call the commitment between *charlie* and *sally* as C_S and the commitment between *charlie* and *linus* as C_L. The property of the commitment C_S is $prop(u(4, 7), rented(herbie, sally))$ and the property of the commitment C_L is $prop(u(6, 7), rented(herbie, linus))$. Assuming that there is a fluent conflict between the two grounded *rented* fluent, by using the Axiom 12 we can conclude that there is a property conflict between the properties of the commitment C_S and C_L. Detection of this property conflict further causes the condition of the Axiom 15 to hold, which allows us to conclude that there is a conditional commitment conflict between the commitments C_S and C_L. Hence, we immediately capture that depending on the satisfaction of the conditions of these two commitments C_S and C_L, one of these commitments cannot be satisfied without violating the other. At moment 2, *sally* pays the rent and satisfies the condition of her own commitment C_S and the state of C_S changes to active. At that moment using the Axiom 14, we can deduce that we have still a conditional commitment conflict, which depends on the satisfaction of the condition of the commitment C_L. At moment 3, *charlie* rents *herbie* to *sally* to satisfy the commitment C_S. Finally, at moment 4 by using the promotion ticket, *linus* satisfies the condition of the commitment C_L and the state of this commitment changes to active. Accordingly the condition of the Axiom 13 holds and we conclude that there is a commitment conflict. At that moment, we definitely know that at least one of the commitments C_S and C_L is going to be violated. Note that, if we examine the commitments using only the life cycle axioms, we cannot capture the violation of one of these commitments not before moment 6.

5 Discussion

In this paper we introduce the conflict relation between two commitments. A conflict relation indicates one of the commitments in this relation cannot be satisfied without violating the other commitment. To formalize this conflict relation we first extend the existing event calculus formalization of the commitments with conditional commitments and then introduce a set of new axioms to capture conflicts between commitments. We implement our axioms using the REC tool and evaluate them on a multiagent scenario. In our future work we plan to apply our method to capture inconsistencies in multiagent contracts [4]. We also left detection of conflicts between more than two commitments and handling of conflicting commitments as future work.

The first formalization of commitments in event calculus is introduced by Yolum and Singh [14]. In their formalization they do not use an explicit state definition for commitments. They discuss how a multiagent protocol can be represented in a flexible way by using the event calculus formalization of commitments and they also show how agents can reason about commitments on the execution of the protocol. In a series of papers Torroni and his colleagues develop another event calculus based monitoring framework for commitments, which uses SCIFF abductive logic programming proof-procedure [1, 3]. Their framework is capable of efficiently monitoring evolution of commitments in a multiagent system at run-time. We use their framework as a basis for our work. We extend their commitment formalization with conditional commitments and on top of this formalization we build our axioms to define conflict relation of commitments.

Singh discusses semantics of dialectical and practical commitments [12]. In his work, Singh provides a unified temporal logic based semantics for dialectical and practical commitments. Our main motivation in this paper is to deal with practical commitments and we do not discuss dialectical commitments. Singh also provides some reasoning postulates related to the ones we discuss here. These are named consistency and strong consistency, which states an agent cannot commit to false and an agent cannot commit to a negation of previously committed property, respectively. Especially, the strong consistency postulate corresponds to our commitment conflict relation. However, the postulates introduced by Singh acts as a constraint and restrict existence of such commitments. We do not put any restrictions on commitments, instead our aim is to just detect such situations and let dealing with the situation to the underlying multiagent system.

Mallya *et al.* discuss resolvability of commitments [8]. They use a variant of CTL to formalize commitments and provide a set of definitions about when a commitment is resolvable using the same temporal quantifiers for properties of commitments. Their discussion concentrates on the resolvability of individual commitments. In our work we assume that all commitments are individually resolvable as defined by Mallya *et al.*. However, our work can be used in order to capture resolvability of multiple commitments, considering individual resolvability.

Acknowledgment

This research is partially supported by Boğaziçi University Research Fund under grant BAP5694, and the Turkish State Planning Organization (DPT) under the TAM Project, number 2007K120610. Akın Günay is partially supported by TÜBİTAK National PhD Scholarship (2211).

References

1. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Verifiable agent interaction in abductive logic programming: The sciff framework. ACM Transactions on Computational Logic 9, 1–43 (August 2008)
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. Chesani, F., Mello, P., Montali, M., Torroni, P.: Commitment Tracking via the Reactive Event Calculus. In: Proceedings of the 21st International Joint Conference on Artifical Intelligence. pp. 91–96. Morgan Kaufmann Publishers Inc. (2009)
4. Desai, N., Narendra, N.C., Singh, M.P.: Checking Correctness of Business Contracts via Commitments. In: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems - Volume 2. pp. 787–794. AAMAS '08 (2008)
5. Fornara, N., Colombetti, M.: Operational Specification of a Commitment-Based Agent Communication Language. In: AAMAS'02: Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems. pp. 536–542. ACM (2002)
6. Kowalski, R., Sergot, M.: A Logic-based Calculus of Events. New Generation Computing 4, 67–95 (January 1986)
7. Mallya, A.U., Huhns, M.N.: Commitments Among Agents. IEEE Internet Computing 7, 90–93 (July 2003)
8. Mallya, A.U., Yolum, P., Singh, M.P.: Resolving Commitments Among Autonomous Agents. In: Dignum, F. (ed.) Advances in Agent Communication. Lecture Notes in Artificial Intelligence, vol. 2922, pp. 166–182. Springer-Verlag (2003)
9. Shanahan, M.: The Event Calculus Explained. In: Wooldridge, M.J., Veloso, M. (eds.) Artificial Intelligence Today, pp. 409–430. Springer-Verlag (1999)
10. Singh, M.P.: Agent Communication Languages: Rethinking the Principles. Computer 31(12), 40–47 (1998)
11. Singh, M.P.: An Ontology for Commitments in Multiagent Systems. Artificial Intelligence and Law 7(1), 97–113 (1999)
12. Singh, M.P.: Semantical Considerations on Dialectical and Practical Commitments. In: AAAI'08: Proceedings of the 23rd International Conference on Artificial Intelligence. pp. 176–181. AAAI Press (2008)
13. 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)
14. Yolum, P., Singh, M.P.: Flexible Protocol Specification and Execution: Applying Event Calculus Planning using Commitments. In: AAMAS '02: Proceedings of the first International Joint Conference on Autonomous Agents and Multiagent Systems. pp. 527–534. ACM (2002)

Formalizing Commitments Using Action Languages

Tran Cao Son[1], Enrico Pontelli[1], and Chiaki Sakama[2]

[1] Dept. Computer Science, New Mexico State University, tson|epontell@cs.nmsu.edu
[2] Computer and Comm. Sciences, Wakayama Univ., sakama@sys.wakayama-u.ac.jp

Abstract. This paper investigates the use of high-level action languages for representing and reasoning about commitments in multi-agent domains. We introduce the language $\mathcal{L}^{mt}$, an extension of the language $\mathcal{L}$, with new features motivated by the problem of representing and reasoning about commitments. The paper demonstrates how features and properties of commitments can be described in this action language. We show how $\mathcal{L}^{mt}$ can handle both simple commitment actions as well as complex commitment protocols. Furthermore, the semantics of $\mathcal{L}^{mt}$ provides a uniform solution to different problems in reasoning about commitments such as the problem of (*i*) verifying whether an agent fails (or succeeds) to deliver on its commitments; (*ii*) identifying pending commitments; and (*iii*) suggesting ways to satisfy pending commitments.

1 Introduction and Motivation

Commitments are an integral part of societies of agents. Modeling commitments has been an intensive topic of research in autonomous agents. The focus has often been on the development of ontologies for commitments [6, 15], on the identification of requirements for formalisms to represent commitments [13], and the development of formalisms for specifying and verifying protocols or tracking commitments [7, 18, 11].

Commitments are strongly related to agents' behavior and capabilities, and they are often associated with time constraints, such as a specific time (or time interval) in the future. For example, a customer will not pay for the promised goods if the goods have not been delivered; a client will have to wait for her cheque if the insurance agent does not keep her promise of entering her claim into the system; or an on-line shopper needs to pay for the order within 10 minutes after clicking the 'Check Out' button. Thus, any formalization of *commitments* should be considered in conjunction with a formalization of *actions* and *changes*, which allows us to reason about narratives in presence of (quantitative) time constraints, actions with durations, etc.

Action languages (e.g., $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ [10]), with their English like syntax and simple transition function based semantics, provide an easy and compact way for describing dynamic systems. Unlike event calculus —an action description formalism often used in the literature for reasoning about commitments—action languages can elegantly deal with indirect effects of actions and static laws. Furthermore, off-the-shelf implementations of various action languages are available. Research has provided various avenues to extend the basic action languages with advanced features, such as resources, deadlines, and preferences. Existing action languages, on the other hand, do not provide means for expressing statements like *"I will make some sandwiches"* or *"I will come at 7pm."* Both statements are about achieving a certain state of the world without specifying how. The first statement does not indicate a specific time in the future while the

second does. Moreover, with a few exceptions, action languages have been developed mostly for single-agent environments. Action languages have been successfully used in specifying and reasoning about narratives (e.g., [2, 4]). Some attempts to use action languages in formalizing commitments have been made [8, 9]. However, these attempts do not consider time constraints and actions with durations.

In this paper we answer the question of whether action languages, like $\mathcal{B}$ or $\mathcal{L}$, can be enriched with adequate features to enable the representation of domains where agents can interact through commitments, maintaining the desirable features of having a clear semantics and a declarative representation. In particular, we develop an action language, called $\mathcal{L}^{mt}$, to perform this activity. $\mathcal{L}^{mt}$, an extension of the action language $\mathcal{L}$ [2–4], is a language for multi-agent domains with features related to time, observations, and delayed effects. We show that several tasks related to reasoning with commitments, such as identifying satisfied, pending, and unsatisfied commitments, can be expressed as *queries* in $\mathcal{L}^{mt}$. Furthermore, the problem of finding a way to satisfy pending commitments can be directly addressed using planning. The language also provides a natural means for specifying, verifying, and reasoning about protocols among agents.

2 The Language $\mathcal{L}^{mt}$

2.1 $\mathcal{L}^{m}$: Concurrency and Multi-agency

In this section, we introduce an action language which supports concurrency and multi-agency; the language is an extension of the language $\mathcal{L}$ [3, 4].

The signature of the language is $\langle \mathcal{AG}, \{\mathcal{F}_i, \mathcal{A}_i\}_{i\in\mathcal{AG}} \rangle$ where $\mathcal{AG}$ is a (finite) set of agent identifiers and $\mathcal{F}_i$ and $\mathcal{A}_i$ are the sets of *fluents* and the set of *actions* of the agent i, respectively. We assume that $\mathcal{A}_i \cap \mathcal{A}_j = \emptyset$ for any two distinct $i, j \in \mathcal{AG}$. Observe also that $\bigcap_{i\in S} \mathcal{F}_i$ may be not empty for some $S \subseteq \mathcal{AG}$. This represents the fact that fluents in $\bigcap_{i\in S} \mathcal{F}_i$ are relevant to all the agents in S. A *fluent literal* (or *literal*) is either a fluent or a fluent preceded by $\neg$. Given a literal ℓ, we denote with $\bar{\ell}$ its complement. A *fluent formula* is a propositional formula constructed from literals.

A multi-agent domain specification is a set of axioms of the following forms:

a **causes** ℓ **if** ψ (1) $\qquad$ φ **if** ψ (2)

impossible A **if** ψ (3) $\qquad$ **initially** ℓ (4)

where $a \in \bigcup_{i\in\mathcal{AG}} \mathcal{A}_i$ is an action, ℓ is a fluent literal, ψ and φ are sets of fluent literals (interpreted as conjunctions), and $A \subseteq \bigcup_{i\in\mathcal{AG}} \mathcal{A}_i$ is a set of actions.

Axioms of type (1), (2), and (3) are referred to as *dynamic laws*, *static laws* (or *state constraints*), and *non-executability laws*, respectively. Intuitively, a dynamic law describes the direct effects of execution of one action (possibly concurrently to other actions), static laws describe integrity constraints on states of the world, and non-executability laws describe conditions that prevent the (concurrent) execution of groups of actions. Statements of type (4) are employed to describe the initial state of the world.

The semantics of a multi-agent domain is defined by the transition function Φ_D, which maps a set of actions and a state to a set of states, where $D = \bigcup_{i\in\mathcal{AG}} D_i$ is the domain description defined over the set of fluents $\bigcup_{i\in\mathcal{AG}} \mathcal{F}_i$ and the set of actions $\bigcup_{i\in\mathcal{AG}} \mathcal{A}_i$. For later use, we define an *action snapshot* as a set $\{a_i\}_{i\in\mathcal{AG}}$ where $a_i \in \mathcal{A}_i \cup \{\texttt{noop}\}$. Intuitively, each action snapshot encodes the set of actions that the agents in $\mathcal{AG}$ concurrently execute in a state. Intuitively, given an action snapshot A and a state

s, the transition function Φ_D defines the set of states that may be reached after executing A in state s. If $\Phi_D(A, s)$ is the empty set, then A is not executable in s.

An *interpretation* I of the fluents in D is a maximal consistent set of fluent literals drawn from $\mathcal{F}$. A fluent f is said to be true (resp. false) in I iff $f \in I$ (resp. $\neg f \in I$). The truth value of a fluent formula in I is defined recursively over the propositional connectives in the usual way. We say that I satisfies φ ($I \models \varphi$) if φ is true in I.

Let I be a set of fluent literals. We say that I is closed under D if for every rule (φ **if** ψ) in D, if $I \models \psi$ then $I \models \varphi$. By $Cl_D(I)$ we denote the smallest superset of I which is closed under D. A *state* of D is an interpretation that is closed under the set of static causal laws of D.

A set of actions B is *prohibited (not executable)* in a state s if there exists an executability condition of the form (3) in D such that $A \subseteq B$ and $s \models \psi$.

The *effect of an action a* in a state s of D is the set of formulae $e_A(s) = \{\ell \mid D$ contains a law a **causes** ℓ **if** ψ, $a \in A$, and $s \models \psi\}$.

Given the domain description D, if A is prohibited in s, then $\Phi_D(A, s) = \emptyset$, otherwise $\Phi_D(A, s) = \{s' \mid s' = Cl_D((s \cap s') \cup e_A(s))$ *and* s' *is a state*$\}$. The function Φ_D is extended to define $\widehat{\Phi_D}$ for reasoning about the effects of sequences of action snapshots as follows. For a state s and a sequence of action snapshots $\alpha = [A_1, \ldots, A_n]$, let $\alpha_{n-1} = [A_1, \ldots, A_{n-1}]$, we define

$$\widehat{\Phi_D}(\alpha, s) = \begin{cases} \{s\} & \text{if } n = 0 \\ \emptyset & \text{if } \widehat{\Phi_D}(\alpha_{n-1}, s) = \emptyset \vee \exists s'.[s' \in \widehat{\Phi_D}(\alpha_{n-1}, s) \wedge \Phi_D(A_n, s') = \emptyset] \\ \bigcup_{s' \in \widehat{\Phi_D}(\alpha_{n-1}, s)} \Phi_D(A_n, s') & \text{otherwise} \end{cases}$$

An *initial state* is a state s_0 such that, for each statement of type (4) in D we have that $s_0 \models \ell$. We will assume from now on that there exists at least one initial state. A trajectory is a sequence $s_0\beta_0 s_1\beta_1 \ldots \beta_{n-1}s_n$ such that each β_j is a snapshot, s_0 is an initial state, and $s_i \in \Phi_D(s_{i-1}, \beta_{i-1})$ for $1 \leq i \leq n$.

We allow *queries* to be composed, of the form: φ **after** α, where α is a sequence of action snapshots. A query q is true w.r.t. an initial state s_0, denoted $s_0 \models q$, if $\widehat{\Phi_D}(\alpha, s_0) \neq \emptyset$ and $\forall s \in \widehat{\Phi_D}(\alpha, s_0)$ we have that $s \models \varphi$. A query q is entailed by D ($D \models q$) if for each initial state s_0 of D we have $s_0 \models q$.

2.2 Considering Time: The Action Language $\mathcal{L}^{mt}$

The language proposed so far does not allow for the specification of durative actions. In particular, we wish to be able to model actions with delayed effects and actions whose effects can be overridden by the execution of another action. For example, pumping gasoline into the tank causes the tank to be full after 5 minutes; drilling a hole in the tank takes only 1 minute and will cause the tank never to be full. The execution of drilling 1 minute after initiating the pumping action will cause the tank to never become full. Thus, the execution of the action drill makes the tank no longer full and this effect cannot be reversed by other actions. To address the first issue, we introduce the notion of *annotated fluents*, i.e., fluents associated to relative time points, and use annotated fluents in axioms of the form (1)-(3). To deal with the second issue, we introduce the notions of *irreversible* and *reversible* processes.

The signature of $\mathcal{L}^{mt}$ extends the signature of $\mathcal{L}^{m}$ with a countable set of *process names* $\mathcal{P}$. An annotated literal is a formulae of the form ℓ^t, where ℓ is a fluent literal and

$t > 0$ is an integer, representing a future point in time. We also allow annotations of the form $\ell^{\vee[t_1,t_2]}$, denoting $\ell^{t_1} \vee \cdots \vee \ell^{t_2}$ for $t_1 \leq t_2$. Annotated formulae are propositional formulae that use annotated literals. Given a fluent formula φ (i.e., where fluents are not annotated), φ^t ($\varphi^{\vee[t_1,t_2]}$) is the formula obtained by replacing each literal ℓ in φ with the annotated literal ℓ^t ($\ell^{\vee[t_1,t_2]}$). An annotated formula is *single time* if it is of the form $\varphi^{\vee[t_1,t_2]}$ for some non-annotated formula φ. An annotated formula is *actual* if no literal in the formula is annotated. For an annotated formula φ, φ^{+t} is the formula obtained by replacing each ℓ^r in φ with ℓ^{r+t}.

A multi-agent domain specification is a collection of laws of the form (1)-(3) and laws of following forms:

$$\varphi \textbf{ starts } process\_id \text{ [}\textbf{reversible}|\textbf{irreversible}\text{]} \ \ell^{\hat{t}} \quad (5)$$

$$\varphi \textbf{ stops } process\_id \quad (6)$$

$$a \textbf{ starts } process\_id \text{ [}\textbf{reversible}|\textbf{irreversible}\text{]} \ \ell^{\hat{r}} \textbf{ if } \varphi \quad (7)$$

$$a \textbf{ stops } process\_id \textbf{ if } \varphi \quad (8)$$

where the φ's are sets of fluent literals, $a \in \cup_{i \in \mathcal{AG}} \mathcal{A}_i$, $\ell^{\hat{t}}$ and $\ell^{\hat{r}}$ are time annotated literals, of the form $\vee[t_1, t_2]$ with $1 \leq t_1 \leq t_2$ and $\vee[r_1, r_2]$ with $0 \leq r_1 \leq r_2$,[1] and $process\_id$ belongs to $\mathcal{P}$. The main novelty is the introduction of the notion of *process*. A process is associate to a delayed effect, denoted by $\ell^{\hat{t}}$, and the time interval $\hat{t}$ indicates when the process will produce its effect. A process can be started by an action or a property. Each **reversible** process can be interrupted by a **stops** action/condition before materializing its effects, while **irreversible** processes cannot be interrupted.

The notion of a state in an $\mathcal{L}^{mt}$ domain D is similar to a state in $\mathcal{L}$ domain, in that it is an interpretation of the fluents in D and needs to satisfy the constraints imposed by static laws in D. In presence of processes, a state of the world needs to account for changes that will occur only in the future, when a process reaches its completion. For example, an action $sendPayment$ may state that the action starts a process named *payment_process* whose effect is to make $paid$ true 3, 4, or 5 units of time after the execution of the action. For this reason, we introduce the notion of an *extended state* as a triple (s, IR, RE) where s is a state and IR and RE are sets of pairs of future effects, each of the form $(x : \ell^{\hat{t}})$, where x is a process name and $\ell^{\hat{t}}$ is an annotated fluent. s encodes the *current* state of the world, while IR and RE contain the irreversible and reversible processes, respectively. (s, IR, RE) is *complete* if $IR = \emptyset$ and $ER = \emptyset$.

In presence of future effects encoded by the processes, the world changes due to (*i*) the completion of a process; or (*ii*) action occurrences. Let us consider an extended state $(s, \{(x : p^1)\}, \emptyset)$ with $(x : p^1)$ as a process whose effect is p. Intuitively, if nothing happens, we would expect that p would be true in the world state one unit of time from the current time. This results in the new extended state of the world $(s \backslash \{\neg p\} \cup \{p\}, \emptyset, \emptyset)$. If instead we perform in the initial extended state an action a, whose effect is to make q true in the next moment of time, then the next state will be $(s \setminus \{\neg p, \neg q\} \cup \{p, q\}, \emptyset, \emptyset)$. Thus, in order to define the semantics of $\mathcal{L}^{mt}$ domains, we need two steps. First, we specify an update function, which computes the extended state which is t units of time from the current state assuming that no action occurs during this time span. Second, we define the transition function that takes into consideration the action occurrences.

[1] For simplicity, we do not consider $\wedge[t_1, t_2]$. This is because a law with the annotation $\wedge[t_1, t_2]$ can be replaced by a set of laws whose annotation is $\vee[t_i, t_i]$ for $t_1 \leq t_i \leq t_2$.

The *update* of an extended state (s, IR, RE) is used to move forward by one time step; the time of the annotated fluents is decreased by one. Fluents that have become actual are used to update the state—in such a case we need to ensure that irreversible changes prevail over reversible ones. Formally, for $\hat{s} = (s, IR, RE)$, the set of literals that should be used in updating s in the next moment of time is

$$\tau(\hat{s}) = \{\ell \mid (x : \ell^1) \in IR\} \cup \{\ell \mid (x : \ell^1) \in RE \text{ such that } \nexists (z : \bar{\ell}^1) \in IR\}.$$

For a state s, the set of processes started and stopped by s in the next moment of time is $IR_1(s) = \{(process\_id : \hat{\ell^t}) \mid$ there exists a law of the form (5) with the option **irreversible** such that $s \models \varphi\}$, $RE_1(s) = \{(process\_id : \hat{\ell^t}) \mid$ there exists a law of the form (5) with the option **reversible** such that $s \models \varphi\}$, and $P_2(s) = \{process\_id \mid$ there exists a law of the form (6) such that $s \models \varphi\}$. For a set of process names N and a set of future effects X, let $X \setminus N = X \setminus \{(x : \ell^t) \mid x \in N, (x : \ell^t) \in X\})$.

The update of $\hat{s}$ by one unit of time is a set of extended states defined as follows:

$$update(\hat{s}) = \{(s', I(IR, s'), R(ER, s') \mid s' = Cl_D(\tau(\hat{s}) \cup (s \cap s')) \text{ and } s' \text{ is a state}\}$$

where, $I(IR, s') = (IR{-}1) \cup IR_1(s')$ and $R(ER, s') = ((RE{-}1) \cup RE_1(s')) \setminus P_2(s')$, and for a set of future effects X, we have $X - d = \{(x : \ell^{t-d}) \mid (x : \ell^t) \in X\}$. Intuitively, s' is a state that satisfies the effects that need to be true one unit from the current state. For $t > 0$, let $\hat{s} + t = \bigcup_{\hat{u} \in update(\hat{s}+t-1)} update(\hat{u})$ where $\hat{s} + 0 = \hat{s}$.

Given an extended state $\hat{s} = (s, IR, ER)$ and an annotated literal ℓ^t, we say that ℓ^t holds in $\hat{s}$, denoted $\hat{s} \models \ell^t$, if, for $t = 0$, $\hat{s} \models \ell^t$ if $s \models \ell$, and, for $t > 0$, $\hat{s} \models \ell^t$ if $\hat{u} \models \ell$ for every $\hat{u} \in \hat{s} + t$.

Let us now consider the case where an action snapshot $A = \{a_i\}_{i \in \mathcal{AG}}$ is executed in the extended state $\hat{s}$. Intuitively, there are two possible types of effects: the direct effect of the actions ($e_A(s)$) and the processes that are created by the actions. We know that $e_A(s)$ must be satisfied in the next time point. The effects of the processes starting by A in s, denoted by $procs_A(s)$, is a set of pairs (IR', RE') where:

- For each (a_i **starts** p_{id} **irreversible** $\ell^{\vee[t_1,t_2]}$ **if** φ) in D, with $a_i \in A$ and $s \models \varphi$, we have that IR' contains $(p_{id} : \ell^t)$ for some t s.t. $t_1 \leq t \leq t_2$;
- For each (a_i **starts** p_{id} **reversible** $\ell^{\vee[t_1,t_2]}$ **if** φ) in D, with $a_i \in A$, and $s \models \varphi$, we have that RE' contains $(p_{id} : \ell^t)$ for some t s.t. $t_1 \leq t \leq t_2$.

In addition, the set of processes stopped by A in s is defined as $stop_A(s) = \{p_{id} \mid$ (a_i **stops** p_{id} **if** φ) $\in D, s \models \varphi\}$. Intuitively, each (IR', RE') encodes a possible set of effects that the snapshot A can create given the current state of the world is s. $stop_A(s)$ is the set of processes that need to be stopped.

We are now ready to define transition function Φ_D^t for $\mathcal{L}^{mt}$ domains which maps extended states and action snapshots to sets of extended states. We assume that $\top$ is a special process name in $\mathcal{P}$ that does not appear in any laws of D. For a set of literals L, we define $\oplus(L) = \{(\top : \ell^1) \mid \ell \in L\}$. Given an extended state $\hat{s} = (s, IR, RE)$, a fluent literal ℓ holds in $\hat{s}$ if ℓ holds in s. The notion of executability of a set of actions can be carried over to $\mathcal{L}^{mt}$ domains without changes as it only considers the current state of the world. The transition function Φ_D^t is:

$$\Phi_D^t(A, \hat{s}) = \bigcup_{(I,R) \in procs_A(s)} update((s, IR \cup I \cup \oplus(e_A(s)), (RE \cup R) \setminus stop_A(s))$$

if A is executable in s, and $\Phi^t_D(\hat{s}, A) = \emptyset$ otherwise. Intuitively, $\Phi^t_D(\hat{s}, A)$ encodes the possible trajectories of the world given that A is executed in $\hat{s}$. We extend Φ^t_D to $\widehat{\Phi^t_D}$ which operates on sequences of action snapshots in the same way as done for Φ_D.

In presence of time, we might be interested in the states of the world given that A is executed t units of time from the current state of the world. We overload Φ^t_D and define

$$\Phi^t_D(\hat{s}, A, t) = \widehat{\Phi^t_D}(\hat{s}, [\underbrace{\{\texttt{noop}\}_{i\in\mathcal{AG}}, \ldots, \{\texttt{noop}\}_{i\in\mathcal{AG}}}_{t}] \circ [\, A \,])$$

We also write $\Phi^t_D(\hat{s}, A, t)+t_1$ to denote

$$\Phi^t_D(\hat{s}, A, t) + t_1 = \bigcup_{\hat{s'}\in\Phi^t_D(\hat{s},A,t)} \widehat{\Phi^t_D}(\hat{s'}, [\underbrace{\{\texttt{noop}\}_{i\in\mathcal{AG}}, \ldots, \{\texttt{noop}\}_{i\in\mathcal{AG}}}_{t_1}])$$

Intuitively, a member of $\Phi^t_D(\hat{s}, A, t) + t_1$ is a possible extended state after t_1 time steps from the execution of A, which in turn was executed t time steps from $\hat{s}$.

Let us define a *timed action snapshot* to be a pair (A, t) where A is an action snapshot and t is a time reference. $\widehat{\Phi^t_D}$ can also be extended to a transition function that operates on sequences of timed action snapshots $\alpha = [(A_1, t_1), \ldots, (A_n, t_n)]$ where $t_1 < t_2 < \ldots < t_n$ and A_i's are action snapshots as follows:

- For $n = 0$: $\widehat{\Phi^t_D}(\hat{s}, \alpha) = \hat{s}$; and
- For $n > 0$: $\widehat{\Phi^t_D}(\hat{s}, \alpha) = \bigcup_{\hat{u}\in\Phi^t_D(\hat{s},A_1,t_1)} \widehat{\Phi^t_D}(\hat{u}, \beta)$
  where β=$[(A_2, t_2-t_1), \ldots, (A_n, t_n-t_1)]$ if $\widehat{\Phi^t_D}(\hat{u}, \beta) \neq \emptyset$ for every $\hat{u}$∈$\Phi^t_D(\hat{s}, A_1, t_1)$; otherwise, $\widehat{\Phi^t_D}(\hat{s}, \alpha) = \emptyset$.

For a state s and a sequence of timed action snapshot α, $\widehat{\Phi^t_D}(s, \alpha) = \widehat{\Phi^t_D}((s, \emptyset, \emptyset), \alpha)$.

Example 1. Let us consider a slight modification of the the popular Netbill example [13]. Let us assume that every action takes one day to complete but the action of sending the payment might take 3 to 5 days for its effects to materialize. Also, as long as the payment has not been made, the customer can still cancel the payment. We envision $\mathcal{AG} = \{$*merchant*, *customer*$\}$. Both the *merchant* and the *customer* use the set of fluents $\mathcal{F} = \{$*request*, *paid*, *goods*, *receipt*, *quote*, *accept*$\}$; the agents use the sets of actions:

$$\mathcal{A}_{merc} = \{\textit{sendQuote}, \textit{sendGoods}, \textit{sendReceipt}\}$$
$$\mathcal{A}_{cust} = \{\textit{sendRequest}, \textit{sendAccept}, \textit{sendPayment}\}$$

The domain specification D_n consists of the following axioms ($\mathcal{P} = \{pmt\}$):

Customer	**Merchant**
sendRequest **causes** $request$	*sendGoods* **causes** $goods$
sendAccept **causes** $accept$	*sendReceipt* **causes** $receipt$
sendPayment **starts** pmt **reversible** $paid^{\vee[3,5]}$	*sendQuote* **causes** $quote$
cancelPayment **stops** pmt	**impossible** $\{$*sendReceipt*$\}$ **if** $\neg paid$
impossible $\{$*sendAccept*$\}$ **if** $\neg quote$	**impossible** $\{$*sendGoods*$\}$ **if** $\neg accept$
impossible $\{cancelPayment\}$ **if** $paid$	

The last two laws state that the *Merchant* cannot execute the action $sendReceipt$ if $\neg paid$ is true (the *Customer* has not paid yet); he cannot execute the action $sendGoods$ if $\neg accept$ is true (the *Customer* has not accepted the offer). On the other hand, the *Customer* cannot execute the action $sendAccept$ if he has not received the quote.

Let $s_0 = \{request, quote, accept, \neg paid, \neg receipt, \neg goods\}$, and $\alpha_1 = \{\texttt{noop}, sendGoods\}$. α_1 is executable in s_0 and $\Phi^t_{D_n}((s_0, \emptyset, \emptyset), \alpha_1) = \{(s'_0, \emptyset, \emptyset)\}$, where

$$s'_0 = \{request, quote, accept, \neg paid, \neg receipt, goods\}.$$

Let $\hat{u} = (s'_0, \emptyset, \emptyset)$ and $\alpha_2 = \{sendPayment, \texttt{noop}\}$. It is easy to see that

$$\Phi^t_D(\hat{u}, \alpha_2) = \{update((s'_0, \emptyset, \{(pmt : paid^i)\})) \mid i = 3, 4, 5\}$$

Thus, $\Phi^t_D(\hat{u}, \alpha_2) + 3 = \{(u', \emptyset, \emptyset)\} \cup \{update((s'_0, \emptyset, \{(pmt : paid^i)\})) \mid i = 1, 2\}$ where $u' = \{request, quote, accept, paid, \neg receipt, goods\}$. We can see that $\Phi^t_D(\hat{u}, \alpha_2) + 5 = \{(u', \emptyset, \emptyset)\}$. □

3 Basic Commitments in $\mathcal{L}^{mt}$

We demonstrate that $\mathcal{L}^{mt}$ is adequate to encode commitments and their manipulation. Commitments are encoded as a new class of fluents and are manipulated by *commitment actions*. Due to the lack of space, we present our study on unconditional commitments [15]. We observe that the treatment of conditional commitments can be done similarly.

A *commitment* is of the form $c(x, y, \varphi, t_1, t_2)$, where $x, y \in \mathcal{AG}$, $0 < t_1 \leq t_2$, and φ is formula. This states that the debtor x agrees to establish φ between t_1 and t_2 for the creditor y. For example, the statement "A commits to visit B in three hours," conveys the commitment $c(A, B, arrived, 3, 3)$. A commitment where we do not care when the property is made true can be expressed using a disjunctive annotation.

Observe that we can think of commitment fluents as propositions, i.e., $c(x, y, \varphi)$ is a syntactic sugar for $c\_x\_y\_name(\varphi)$ where $name(\varphi)$ is a propositional variable representing the name of the formula φ. We assume that the various propositions $c(x, y, \varphi)$ are in $\bigcap_{i \in \mathcal{AG}} \mathcal{F}_i$. We also assume that, to enable communication, if $c(x, y, \varphi)$ is a commitment fluent, then φ is a fluent formula which uses fluents from $\mathcal{F}_x \cup \mathcal{F}_y$.

The following operations are used to manipulate commitments:

- *Creation:* $create(x, y, \varphi, t_1, t_2)$ describes the fact that agent x creates a commitment towards agent y in the period between t_1 and t_2. We assume that each created commitment is associated to a unique identifier;
- *Discharge:* $discharge(x, y, \varphi)$ indicates that agent x discharges a commitment towards agent y (by satisfying the request);
- *Release:* $release(x, y, \varphi)$ indicates that agent y releases x from its obligation;
- *Assignment:* $assign(x, y, k, \varphi, t_1, t_2)$ indicates that agent y transfers the commitment to a different creditor (with a new time frame);
- *Delegation:* $delegate(x, y, k, \varphi, t_1, t_2)$ indicates that agent x delegates the commitment to another debtor (with a new time frame);
- *Cancel:* $cancel(x, y, \varphi, \psi, t_1, t_2)$ indicates that x modifies the terms of the commitment (by canceling the previous one and generating a new one).

These manipulations of commitments are the consequence of actions performed by the agents or conditions occurring in the state of the world. We consider two types of enabling statements, called *trigger statements*, for commitment manipulation

$$[\varphi | a] \ \textbf{triggers} \ c\_activity$$

where φ is a fluent formula, $a \in \mathcal{A}$, and $c\_activity$ is one of the activities (or commitment actions). They indicate that the commitment activity $c\_activity$ should be executed whenever φ holds or a is executed. An example of the first type of statement is

$$paid \ \textbf{triggers} \ create(m, c, receipt, 1, 3) \tag{9}$$

which encodes the fact that the merchant agrees to send the customer the receipt between 1 and 3 units of time since receiving the payment. The statement

$$sendAccept \ \textbf{triggers} \ create(c, m, paid, 1, 5) \tag{10}$$

states that the customer agrees to pay for the goods between 1 to 5 units of time after sending the acceptance notification. A more complicated trigger statement is the following, taken from an example in [7],

$$broken \ \textbf{triggers} \ create(s, c, (broken \Rightarrow paid\_10), k, k)$$

for $k \geq 3$, which represents the agreement between the service provider (s) and a customer (c) that, if the printer is broken, the service provider needs to fix it within three days or faces the consequence of paying $10 each day the printer is not fixed.

A *domain with commitments* is a pair (D, C) where D is a domain specification in $\mathcal{L}^{mt}$ and C is a collection of trigger statements. Intuitively, a domain with commitments is an action theory enriched with a set of (social or contractual) agreements between agents in the domain which are expressed by the set of trigger statements.

In the following, we will define the semantics of a domain with commitments (D, C) by translating it into a $\mathcal{L}^{mt}$ domain D' where D' consists of D and a collection of dynamic laws and static laws originating from C.

Action Triggers: a **triggers** $c\_activity$ belongs to C: in this case,

- if $c\_activity = create(x, y, \varphi, t_1, t_2)$, then the laws
  a **causes** $c(x, y, \varphi)$ and a **starts** $c(x, y, \varphi)$ **reversible** $done(x, y, \varphi)^{\vee[t_1, t_2]}$
  are added to D'. The dynamic law records the fact that the commitment $c(x, y, \varphi)$ has been made by the execution of the action a. The second law starts a process which indicates that the commitment must be satisfied between t_1 and t_2.
- if $c\_activity = discharge(x, y, \varphi)$ then D' contains
  a **stops** $c(x, y, \varphi)$ **if** $c(x, y, \varphi)$ a **causes** $\neg c(x, y, \varphi)$ **if** $c(x, y, \varphi)$
  a **starts** $discharging(x, y, \varphi)$ **irreversible** φ **if** $c(x, y, \varphi)$
  Here, the action a stops the commitment process $c(x, y, \varphi)$ by starting a process of achieving φ. It also records the fact that the commitment $c(x, y, \varphi)$ has been satisfied.
- if $c\_activity = release(x, y, \varphi)$ then
  a **stops** $c(x, y, \varphi)$ **if** $c(x, y, \varphi)$ and a **causes** $\neg c(x, y, \varphi)$ **if** $c(x, y, \varphi)$
  belongs to D'. The action stops the commitment process and records that the commitment has been removed.
- if $c\_activity = assign(x, y, k, \varphi, t_1, t_2)$ then D' contains
  a **stops** $c(x, y, \varphi)$ **if** $c(x, y, \varphi)$ a **causes** $\neg c(x, y, \varphi)$ **if** $c(x, y, \varphi)$
  a **causes** $c(x, k, \varphi)$ a **starts** $c(x, k, \varphi)$ **reversible** $done(x, k, \varphi)^{\vee[t_1, t_2]}$
  The action stops the commitment process $c(x, y, \varphi)$ and starts the commitment process $c(x, k, \varphi)$. It also releases the process $c(x, y, \varphi)$.
- if $c\_activity = delegate(x, y, k, \varphi, t_1, t_2)$ then D' contains
  a **stops** $c(x, y, \varphi)$ **if** $c(x, y, \varphi)$ a **causes** $\neg c(x, y, \varphi)$ **if** $c(x, y, \varphi)$
  a **causes** $c(k, y, \varphi)$ a **starts** $c(k, y, \varphi)$ **reversible** $done(k, y, \varphi)^{\vee[t_1, t_2]}$
  This is similar to the case of $release$, only with different debtor.
- if $c\_activity = cancel(x, y, \varphi, \psi, t_1, t_2)$ then D' contains
  a **stops** $c(x, y, \varphi)$ **if** $c(x, y, \varphi)$ a **causes** $\neg c(x, y, \varphi)$ **if** $c(x, y, \varphi)$
  a **causes** $c(x, y, \psi)$ a **starts** $c(x, y, \psi)$ **reversible** $done(x, y, \psi)^{\vee[t_1, t_2]}$

The action stops the commitment $c(x, y, \varphi)$ and starts a new one $c(x, y, \psi)$.

The translation of fluent triggers is similar. For each fluent trigger ψ **triggers** $c\_activity$, the translation is obtained from the corresponding action trigger one by:

- replacing a dynamic law of the form (a **causes** φ **if** λ) with (φ **if** λ, ψ);
- replacing a law of the form (a **starts** p_{id} [**reversible**|**irreversible**] φ **if** λ) with the law (ψ **starts** p_{id} [**reversible**|**irreversible**] φ **if** λ); and
- replacing a law of the form (a **stops** p_{id} **if** λ) with the law (ψ **stops** p_{id} **if** λ).

We further need to include some additional static laws: if $c(x, y, \varphi)$ is present and φ is true, then the commitment can be released: $\neg c(x, y, \varphi)$ **if** $\varphi, done(x, y, \varphi)$.

Let $\mathcal{M} = (D, C)$ be a domain with commitments. We denote with $\tau(C)$ the collection of axioms generated from the translation process mentioned above; with a slight abuse of notation, we denote $\tau(\mathcal{M}) = D \cup \tau(C)$. By definition, the domain $\tau(\mathcal{M})$ defines a transition function $\Phi^t_{\tau(\mathcal{M})}$ which determines the possible evolutions of the world given a state and the sequence of timed action snapshots $[(\alpha_1, t_1), \ldots, (\alpha_n, t_n)]$. The function $\Phi^t_{\tau(\mathcal{M})}$ can be used to specify the transition function for $\mathcal{M}$, i.e., the transition function $\Phi_{\mathcal{M}}$ for $\mathcal{M}$ is defined to be the function $\Phi^t_{\tau(\mathcal{M})}$. Observe that each state of $\tau(\mathcal{M})$ consists of fluent literals in D and commitments which appear in $\tau(C)$. In the definition of $\Phi^t_{\tau(\mathcal{M})}$, this is treated as any normal fluent. The presence of $c(x, y, \varphi)$ in a state indicates that the commitment $c(x, y, \varphi)$ has been made. $done(x, y, \varphi)$ encodes the fact that the commitment $c(x, y, \varphi)$ needs to be realized by the debtor.

Example 2. Consider the domain with commitments $\mathcal{M}_1 = (D_n, C_2)$, where D_n is the domain description described in Example 1 and C_2 is the set of statements consisting of (9), (10), and the following statements

$request$ **triggers** $create(m, c, quote, 1, 1)$ $\quad$ $accept$ **triggers** $create(m, c, goods, 1, 1)$.

So, the set of fluents in $\tau(\mathcal{M}_1)$, denoted by $\mathcal{F}_1$, consists of $\mathcal{F}$ (the set of fluents of D_1) and the commitment fluents such as $c(m, c, receipt)$, $c(c, m, paid)$, $c(m, c, quote)$, and $c(m, c, goods)$, and fluents of the form $done(x, y, \varphi)$ which are introduced by the translation from $\mathcal{M}_1$ to $\tau(\mathcal{M}_1)$. Let $s_0 = \{\neg f \mid f \in \mathcal{F}_1\}$, we have that

$$\Phi^t_{\tau(\mathcal{M}_1)}(s_0, \{sendRequest\}) = \{[s_0, u, v]\}$$

where $u = s_0 \setminus \{\overline{request}, \overline{c(m, c, quote)}\} \cup \{request, c(m, c, quote)\}$ and $v = u \setminus \{\overline{done(m, c, quote)}\} \cup \{done(m, c, quote)\}$. The presence of $c(m, c, quote)$ and $done(m, c, quote)$ in u and v is due to the laws $c(x, y, quote)$ **if** $request$ and

$$request \textbf{ starts } c(x, y, quote) \textbf{ reversible } done(x, y, \varphi)^1$$

respectively, both are the result of the translation to laws in $\tau(\mathcal{M}_1)$ of the statement

$$request \textbf{ triggers } create(m, c, quote, 1, 1). \qquad \square$$

Let $\mathcal{M} = (D, C)$ be a domain with commitments and $\gamma = [s_0, \ldots, s_n]$ be a sequence of states in $\tau(\mathcal{M})$. Let $c(x, y, \varphi)$ be a commitment fluent appearing in γ. We say that $c(x, y, \varphi)$ is

- *satisfied* in γ if $s_n \models \neg c(x, y, \varphi)$;
- *violated* in γ if $s_n \models c(x, y, \varphi) \wedge done(x, y, \varphi)$; or
- *pending* in γ if $s_n \models c(x, y, \varphi)$ and $s_n \not\models done(x, y, \varphi)$.

The reasoning about commitments given the execution of a sequence of action snapshots can then be defined as follows. Let $\mathcal{M} = (D, C)$ be a domain with commitments,

s_0 be a state in D, and $A = [(\alpha_1, t_1), \ldots, (\alpha_n, t_n)]$ be a sequence of timed action snapshots. We say that a commitment $c(x, y, \varphi)$ is *factual* during the execution of A in s if there exists a sequence of states $\gamma = [s_0, \ldots, s_m]$ in $\widehat{\Phi^t}_{\tau(\mathcal{M})}(s_0, A)$ and $c(x, y, \varphi)$ appears in γ. A factual commitment $c(x, y, \varphi)$ is

- *satisfied* after the execution of A in s_0 if it is satisfied in every sequence of states belonging to $\widehat{\Phi^t}_{\tau(\mathcal{M})}(s_0, A)$.
- *strongly violated* after the execution of A in s_0 if it is violated in every sequence of states belonging to $\widehat{\Phi^t}_{\tau(\mathcal{M})}(s_0, A)$.
- *weakly violated* after the execution of A in s_0 if it is violated in some sequence of states belonging to $\widehat{\Phi^t}_{\tau(\mathcal{M})}(s_0, A)$.
- *pending* after the execution of A in s_0 if it is not violated in any sequence of states and not satisfied in some sequences of states belonging to $\widehat{\Phi^t}_{\tau(\mathcal{M})}(s_0, A)$.

Example 3. Consider the domain $\mathcal{M}_1$ and the state s_0 in Ex. 2. We have that $c(m, c, quote)$ is violated after the execution of $sendRequest$ at s_0, since $\Phi^t_{\tau(\mathcal{M}_1)}(s_0, \{sendRequest\}) = \{[s_0, u, v]\}$. It is easy to verify that for $A = [(sendRequest, 0), (sendQuote, 1)]$, $\Phi^t_{\tau(\mathcal{M}_1)}(s_0, A) = \{[s_0, u, v']\}$ where $v' = u \backslash \{\neg done(m, c, quote), \neg quote, c(m, c, quote)\} \cup \{done(m, c, quote), quote, \neg c(m, c, quote)\}$. This implies that the commitment $c(m, c, quote)$ is satisfied after the execution of A in s_0. □

4 Observations and Narratives

4.1 Observation Language

We consider an extension of the action language by enabling the representation of *observations*. We extend the signature of the language $\mathcal{L}^{mt}$ with a set of *situation constants* $\mathbf{S}$, containing two special constants, s_0 and s_c, denoting the initial situation and the current situation. Observations are axioms of the forms:

φ **at** s (11) $\quad \alpha$ **occurs_at** s (12) $\quad \mathsf{s}$ **at** t (13)

α **between** $\mathsf{s}_1, \mathsf{s}_2$ (14) $\quad \mathsf{s}_1 \prec \mathsf{s}_2$ (15)

where φ is a fluent formula, α is a (possibly empty) sequence of timed action snapshots, and $\mathsf{s}, \mathsf{s}_1, \mathsf{s}_2$ are situation constants which differ from s_c. Axioms of the forms (11) and (15) are called *fluent facts* and *precedence facts*, respectively. (11) states that φ is true in the situation s. (15) says that s_1 occurs before s_2. Axioms of the forms (14) and (12) are referred to as *occurrence facts*. (12) indicates that α starts its execution in the situation s. On the other hand, (14) states that α starts and completes its execution in s_1 and s_2, respectively. Axioms of the form (13) link situations to time points.

A *narrative of a multi-agent system* (a *narrative*, for short) is a pair (D, Γ) where D is a domain description and Γ is a set of observations of the form (11)-(15) such that $\{\mathsf{s}_0 \prec \mathsf{s},\ \mathsf{s} \prec \mathsf{s}_c \mid \mathsf{s} \in \mathbf{S}\} \subseteq \Gamma$.

Observations are interpreted with respect to a domain description. While a domain description defines a transition function that characterizes what states *may* be reached when an action is executed in a state, a narrative consisting of a domain description together with a set of observations defines the possible situation histories of the system. This characterization is achieved by two functions, Σ and Ψ. While Σ maps situation constants to sequences of sets of actions, Ψ picks one among the various transitions given by $\Phi_D(A, s)$ and maps sequences of sets of actions to a unique state.

More formally, let (D, Γ) be a narrative. A *causal interpretation* of (D, Γ) is a partial function Ψ from action snapshots sequences to extended states, whose domain is nonempty and prefix-closed.[2] By $Dom(\Psi)$ we denote the domain of a causal interpretation Ψ. Notice that $[] \in Dom(\Psi)$ for every causal interpretation Ψ. A *causal model* of D is a causal interpretation Ψ such that $\Psi([])$ is an extended state of D and, for every $\alpha \circ [A] \in Dom(\Psi)$, $\Psi(\alpha \circ [A]) \in \Phi_D(A, \Psi(\alpha))$.

A *situation assignment* of $\mathbf{S}$ with respect to D is a mapping Σ from $\mathbf{S}$ into the set of sequences of action snapshots of D that satisfy the following properties: $\Sigma(\mathsf{s}_0) = []$ and, for every $\mathsf{s} \in \mathbf{S}$, $\Sigma(\mathsf{s})$ is a prefix of $\Sigma(\mathsf{s}_c)$.

An *interpretation M* of (D, Γ) is a triple (Ψ, Σ, Δ), where Ψ is a causal model of D, Σ is a situation assignment of $\mathbf{S}$ such that and $\Sigma(\mathsf{s}_c)$ belongs to the domain of Ψ, and Δ is a *time assignment* which maps prefixes of $\Sigma(\mathsf{s}_c)$ to the set of non-negative numbers, with the following restrictions: $\Delta([\,]) = 0$ and $\Delta(\beta) \leq \Delta(\gamma)$ for every $\beta \sqsubseteq \gamma \sqsubseteq \Sigma(\mathsf{s}_c)$. Additionally, for every α, β s.t. $\beta \circ \alpha \sqsubseteq \Sigma(\mathsf{s}_c)$, $\Psi(\beta \circ \alpha)$ belongs to $\widehat{\Phi^t_D}(\Psi([\,]), (\beta, 0) \circ (\alpha, \Delta(\beta)))$.

For an interpretation $M = (\Psi, \Sigma, \Delta)$ of (D, Γ):

(i) α **occurs_at** s is true in M if the sequence $\Sigma(\mathsf{s}) \circ \alpha$ is a prefix of $\Sigma(\mathsf{s}_c)$;
(ii) α **between** $\mathsf{s}_1, \mathsf{s}_2$ is true in M if $\Sigma(\mathsf{s}_1) \circ \alpha = \Sigma(\mathsf{s}_2)$;
(iii) φ **at** s is true in M if φ holds in $\Psi(\Sigma(\mathsf{s}))$;
(iv) $\mathsf{s}_1 \prec \mathsf{s}_2$ is true in M if $\Sigma(\mathsf{s}_1)$ is a prefix of $\Sigma(\mathsf{s}_2)$;
(v) s **at** t is true in M if $\Delta(\Sigma(\mathsf{s})) = t$.

Given two sequences of sets of actions $\alpha = [A_1, \ldots, A_n]$ and $\alpha' = [B_1, \ldots, B_m]$, we say that α is a subsequence of α', denoted by $\alpha \ll \alpha'$, if α can be obtained from α' by (*i*) deleting some B_i from α'; and (*ii*) replacing some action $a \in \mathbf{A}$ in the remaining B_i by $\mathtt{noop}$. An interpretation $M = (\Psi, \Sigma, \Delta)$ is a *model* of a narrative (D, Γ) if all facts in Γ are true in M, and there is no other interpretation $M' = (\Psi, \Sigma', \Delta')$ such that M' satisfies condition (i) above and $\Sigma'(\mathsf{s}_c)$ is a subsequence of $\Sigma(\mathsf{s}_c)$. These models are minimal, as they exclude extraneous actions. A narrative is *consistent* if it has a model.

We can also envision an extension of the query language by allowing queries of the form φ **after** α **at** s, where the testing of the entailment starts from the states in $\Psi(\Sigma(\mathsf{s}))$. In the presence of time, given a narrative (D, Γ) and a fluent formula φ, we are also interested in knowing whether φ^t is true (resp. false) in a situation s for some $t_1 \leq t \leq t_2$. This is expressed using a query of the form

$$\varphi^{\vee[t_1,t_2]} \textbf{ at } \mathsf{s} \qquad (16)$$

We say that a query q of form (16) holds w.r.t. (D, Γ) (i.e., $(D, \Gamma) \models q$) if, for every model $M = (\Psi, \Sigma, \Delta)$ of (D, Γ), there exists $t_1 \leq t \leq t_2$ s.t. φ is true in $\Psi(\Sigma(\mathsf{s})) + t$.

4.2 Narratives and Commitments

A *narrative with commitments* is a triple (D, Γ, C) where (D, C) is a domain with commitments and Γ is a collection of observations. The semantics of a narrative with commitments (D, Γ, C) is defined by (*i*) translating it to the narrative $(\tau(\mathcal{M}), \Gamma)$ in $\mathcal{L}^{mt}$ where $\mathcal{M} = (D, C)$; and (*ii*) specifying models of $(\tau(\mathcal{M}), \Gamma)$ to be models of (D, Γ, C). To save space, we omit the specific details on the semantics of narratives

[2] A set X of action sequences is prefix-closed if for every sequence $\alpha \in X$, every prefix of α is also in X. The symbol $\circ$ denotes list concatenation.

with commitments. Let $N = (D, \Gamma, C)$ be a narrative and M be a model of N. We say that a commitment $c(x, y, \varphi)$ is:

- *satisfied* by M if $M \models \neg c(x, y, \varphi)$ **at** $\mathsf{s_c}$.
- *violated* by M if $M \models (done(c, y, \varphi) \wedge c(x, y, \varphi))$ **at** $\mathsf{s_c}$.
- *pending* w.r.t. M if $M \models \neg done(c, y, \varphi) \wedge c(x, y, \varphi)$ **at** $\mathsf{s_c}$.

Given a narrative N, we will say that a commitment is satisfied if it is satisfied in all models of N; it is strongly violated if it is violated in all models of N; and it is weakly violated if it is violated in some models of N.

Example 4. Consider the narrative $N_1 = (D_n, \Gamma, C_2)$ where $\mathcal{M}_1 = (D_n, C_2)$ is the domain description in Ex. 2 and Γ consists of the precedence facts $\mathsf{s}_0 \prec \mathsf{s}_1 \prec \mathsf{s}_2 \prec \mathsf{s}_3 \prec \mathsf{s_c}$ and the following observations:

$$\neg paid \wedge \neg accept \wedge \neg quote \wedge \neg goods \ \textbf{at} \ \mathsf{s}_0$$
$$sendRequest \ \textbf{occurs\_at} \ \mathsf{s}_0 \quad \text{and} \quad sendAccept \ \textbf{occurs\_at} \ \mathsf{s}_2$$

where s_0, s_1, s_2, s_3, $\mathsf{s_c}$ are situation constants.
A model $M = (\Psi, \Sigma, \Delta)$ for this narrative can be built as follows:

- The sequences of actions leading to the various situations are $\Sigma(\mathsf{s}_0)$=[ ], $\Sigma(\mathsf{s}_1)$=[$\{sendRequest\}$], $\Sigma(\mathsf{s}_2)$=[$\{sendRequest\}, \{sendQuote\}$], and $\Sigma(\mathsf{s}_3)$=$\Sigma(\mathsf{s_c})$=[$\{sendRequest\}, \{sendQuote\}, \{sendAccept\}$].
- $\Psi([\,])$ is the state where all fluents are false and $\Psi(\mathsf{s}_i) = \widehat{\Phi^t}_{\mathcal{M}_1}(\Sigma(\mathsf{s}_i), \Psi([\,]))$.
- The time assignment for situation constants is given by $\Delta(\mathsf{s}_i) = i$ for each i and $\Delta(\mathsf{s_c}) = 3$. This is because each action only takes one unit of time to accomplish.

The presence of the action $sendQuote$ can be explained by the fact that $quote$ is the precondition for $sendAccept$. We can show that M is a model of the narrative N_1.

The minimality condition of models of a narrative also allows us to prove that, for every model $(\Psi', \Sigma', \Delta')$ of $\mathcal{M}_1$, the situation assignment Σ' is identical to Σ and $\Psi'([\,])$ must satisfy $\{\neg paid, \neg accept, \neg quote, \neg goods\}$. This allows us to conclude that $N_1 \models (\neg paid \ \textbf{at} \ \mathsf{s})$ for $\mathsf{s} \in \mathbf{S}$ and $N_1 \models c(c, m, paid) \wedge \neg done(c, m, paid) \ \textbf{at} \ \mathsf{s_c}$. We can show that the commitment $c(m, c, quote)$ is satisfied, the commitment $c(c, m, paid)$ is pending, and there are no violated commitments. □

5 Complex Commitments and Protocols

A basic commitment represents a promise made by an agent to another one, but without specifying a precise procedure to accomplish the commitment. Basic commitments also do not describe complex dependencies among "promises".

A *protocol* is a pair (P_{id}, P) where P_{id} is a unique identifier and P is of the form:

1. a set $\{a_i\}_{i \in \mathcal{AG}}$, where $a_i \in \mathcal{A}_i \cup \{any\}$;
2. $?\varphi$ where φ is a formula;
3. $p_1; \ldots; p_n$ where p_i's are protocols;
4. $p_1 | \ldots | p_n$ where p_i's are protocols;
5. **if** φ **then** p_1 **else** p_2 where p_1, p_2 are protocols and φ is a formula;
6. **while** φ **do** p where p is a protocol and φ is a formula;
7. $p_1 < p_2$ where p_1 and p_2 are protocols.

Intuitively, Case (1) describes a request for execution of certain specific actions by certain agents (*any* indicates that we do not care about what that agent is doing); Case (2)

is a test action, which tests for the condition φ in the world state; Case (3) sequentially composes protocols, i.e., it requires first to meet the requirements of p_1, then those of p_2, etc.; Case (4) requires any of the protocols $p_1, \ldots, p_n$ to be satisfied, i.e., it represents a non-deterministic choice; Cases (5) and (6) are the usual conditional selection and iteration over protocols; Case (7) is a partial ordering among protocols, indicating that p_1 must be completed sometime before the execution of p_2. According to this definition, $(p_0, \textit{sendGoods} < \textit{sendPayment} < \textit{sendReceipt})$ is a protocol.

The language can be extended to allow statements that trigger complex commitments, analogously to the case of basic commitments:

$$[a \mid \varphi] \quad \textbf{triggers} \quad \textit{complex commitment}$$

A narrative can be extended with the following type of observation:

$$P_{id} \quad \textbf{at} \quad \mathsf{s} \qquad (17)$$

where P_{id} is a protocol identifier. This observation states that the protocol referred to by P_{id} has started execution at situation s. A narrative is a triple (D, Γ, C) where Γ can contain also protocol observations.

For a trajectory $h = s_0\alpha_1 s_1 \ldots \alpha_k s_k$, s_0 is called the start of h and is denoted by $start(h)$. $h[i, j]$ denotes the sub-trajectory $s_i\alpha_{i+1} \ldots \alpha_j s_j$. For every state s, $traj(s)$ denotes a set of trajectories whose start state is s. Given a protocol P and a trajectory $h = s_0\alpha_1 \ldots \alpha_k s_k$, we say that h is an *instance* of (P_{id}, P) if

- If $P = \{a_i\}_{i \in \mathcal{AG}}$ then $k = 1$ and, if $\alpha_1 = \{a_i^1\}_{i \in \mathcal{AG}}$, then for each $a_i \neq \textit{any}$ we have $a_i = a_i^1$.
- If $P = \varphi$ then $k = 0$ and $s_0 \models \varphi$.
- If $P = p_1; \ldots; p_n$ then there exists some sequence of indices $i_0 = 0 \leq i_1 \leq \ldots \leq i_n \leq i_{n+1} = k$ such that $h[i_{i_t}, i_{i_{t+1}}]$ is an instance of p_t.
- If $P = p_1 | \ldots | p_n$ then there exists some $1 \leq i \leq n$ such that h is an instance of p_i.
- If $P =$ **if** φ **then** p_1 **else** p_2 and $s_0 \models \varphi$ then h is an instance of P if it is an instance of p_1; otherwise, h must be an instance of p_2.
- If $P =$ **while** φ **do** p and $s_0 \not\models \varphi$ then h is an instance of P if $k = 0$; otherwise, there is an index $0 \leq i \leq k$ s.t. $h[0, i]$ is an instance of p and $h[i, k]$ is an instance of P.
- If $P = p_1 < p_2$ then there exists $0 \leq i \leq j \leq k$ such that $h[0, i]$ is an instance of p_1 and $h[j, k]$ is an instance of p_2.

$(P_{id}, P) \models h$ denotes that h is an instance of (P_{id}, P).

We will now complete the definition of a model of a narrative with protocols. The notion of interpretation and the entailment relation between interpretations and observations, except for the observations of type (17), are defined as in the previous section. For an interpretation $M = (\Psi, \Sigma, \Delta)$ of a narrative (D, Γ, C) and a protocol observation $(P_{id}$ **at** $\mathsf{s}) \in C$, we say that $M \models (P_{id}$ **at** $\mathsf{s})$ if there exists some instance $s_0\alpha_1 s_1 \ldots \alpha_k s_k$ of (P_{id}, P) where: $s_0 = \Psi(\Sigma(\mathsf{s}))$, $\Sigma(\mathsf{s}) \circ [\alpha_1, \ldots, \alpha_k]$ is a prefix of $\Sigma(\mathsf{s_c})$;[3] and For every $1 \leq j \leq k$, $\Psi(\Sigma(\mathsf{s}) \circ [\alpha_1, \ldots, \alpha_j]) = s_j$. The remaining definitions related to narratives can be used unchanged for narratives with protocols.

Example 5. Let $N_2 = (D_n, \Gamma, C_2)$ where D_n is defined as in Exp. 4, C_2 is defined as in Exp. 4 with the addition of the protocol $(p_0, \textit{sendGoods} < \textit{sendPayment} <$

[3] We use $\circ$ to denote concatenation of lists.

sendReceipt) and Γ consists of the precedence facts $\mathsf{s}_0 \prec \mathsf{s}_c$ and the single observation p_0 **at** s_0. Observe that any instance of p_0 contains the actions *sendGoods*, *sendPayment*, and *sendReceipt*, in this order. The executability condition of *sendGoods* implies that $accept$ has to be true at the time it is executed. Together with the minimality condition of models of N_2, we have that for every model $M = (\Psi, \Sigma, \Delta)$ of N_2, $\Psi(\mathsf{s}_0) \models accept$. We construct one model as follows:

- $\Sigma(\mathsf{s}_0) = [\,]$ and $\Sigma(\mathsf{s}_c) = [\{\textit{sendGoods}\}, \{\textit{sendPayment}\}, \{\textit{sendReceipt}\}]$;
- $\Psi(\mathsf{s}_0) = s_0$ where $accept \in s_0$, and $\Psi(\mathsf{s}_c) \in \widehat{\Phi^t_{\tau(\mathcal{M}_2)}}(s_0, \Sigma(\mathsf{s}_c))$;
- $\Delta(\mathsf{s}_0) = 0$ and $\Delta(\mathsf{s}_c) = 3$.

Observe that we can also infer that, in the above model, the customer must have paid right after he/she received the goods (at time 1), since (*i*) $paid$ must be true for *sendReceipt* to be executed; and (*ii*) *sendReceipt* is executed at time 2. □

6 Related Works

Our proposal is related to several works on reasoning with commitments. The main differences between our work and previous works lie in our use of an action language and in our formulation of various problems as a query in our language; this also allows the use of planning to satisfy pending commitments. The treatment of commitments and the ontology for commitments adopted in this paper is largely inspired by [13, 15]. Space limitations allow us to highlight only some representative cases.

With respect to [18], our formalization of basic commitments embedded in a domain with commitments and in a narrative of a multi-agent system allows also for a protocol specification that subsumes that of [18]. Similar differences are present w.r.t. [11], which builds on dynamic temporal logic.

Our approach has some relations to [7]; using a reactive event calculus, they provide a notion similar to narratives. Besides being different from each other in the use of an action language, our approach considers protocols and [7] does not. The same authors, in [16], propose a new language for modeling commitments in which existential quantifier of time points are used. The use of disjunctive time specification in annotating fluent formulas in our work allows us to avoid the issues raised in [12, 16].

[8, 9] also makes use of an action language in dealing with commitments and protocols. While we focus on formalizing commitments, the works [8, 9] use C+ in specifying protocols. A protocol in our definition is similar to a protocol defined in [8, 9] in that it restricts the evolution of the system to a certain sets of trajectories. In this sense, our definition of protocols provides the machineries for off-line verification of properties of protocols [17]. By introducing the observation of the from "P_{id} **at** s" we allow for the possible executions of a protocol in different states and hence different contexts. However, we do not have the notion of a transformer as in [8] and the ability to handle nested commitments as in [9].

The use of complex protocols in commitments has also been explored in [1].

The language $\mathcal{L}^{mt}$ is an evolution of a classical action languages, drawing features like static causal laws from $\mathcal{B}$ [10], narrative and observations from $\mathcal{L}$ [3, 4], and time and deadlines from $\mathcal{ADC}$ [5]. To the best of our knowledge, $\mathcal{L}^{mt}$ is the first action language with all these features, *embedded in the context of modeling multi-agent domains*. $\mathcal{L}^{mt}$ has similarities to the language PDDL 2.1 in that it can describe systems with du-

rative actions and delayed effects. $\mathcal{L}^{mt}$ has a transition function based semantics and considers observations, ir/reversible processes and multiple agents, while PDDL 2.1 does not. It should also be mentioned that $\mathcal{L}^{mt}$ differs from the event calculus in that it allows representing and reasoning with static causal laws and considers ir/reversible fluents while event calculus does not. These are also the differences between $\mathcal{L}^{mt}$ and situation calculus based approaches to dealing with duration [14].

7 Discussion and Conclusion

In this paper, we show how various problems in reasoning about commitments can be described by a suitable instantiation of commitment actions in the language $\mathcal{L}^{mt}$. In particular, we show how the problem of verifying commitments or identifying pending commitments can be posed as queries to a narrative with commitments. We show how the language can also be easily extended to consider commitment protocols.

Since our framework provides a way to identify pending, violated, and satisfiable commitments given a narrative (D, Γ, C), a natural question that arises is what should the agents do to satisfy the pending commitments. The semantics of domains with commitments suggests that we can view the problem of identifying a possible course of actions for the agents to satisfy the pending commitments as an instance of the planning problem and thus can be solved by planning techniques. An investigation of the application of multi-agent planning techniques in generating plans to satisfy pending commitments is one of our main goals in this research in the near future.

References

1. M. Baldoni et al. Commitment-based Protocols with Behavioral Rules and Correctness. *DALT*, Springer, 2010.
2. M. Balduccini and M. Gelfond. Diagnostic Reasoning with A-Prolog. *TPLP*, 3(4,5), 2003.
3. C. Baral et al. Representing Actions: Laws, Observations and Hypothesis. *JLP*, 31(1-3).
4. C. Baral et al. Formulating diagnostic problem solving using an action language with narratives and sensing. *KR*, 311–322, 2000.
5. C. Baral et al. A transition function based characterization of actions with delayed and continuous effects. *KR*, 291–302. 2002.
6. C. Castelfranchi. Commitments: From individual intentions to groups and organizations. *Int. Conf. on Multiagent Systems*, pages 41–48. The MIT Press, 1995.
7. F. Chesani et al. Commitment tracking via the reactive event calculus. *IJCAI*, 2009.
8. A.K. Chopra and M.P. Singh. Contextualizing commitment protocol. *AAMAS*, pages 1345–1352. ACM, 2006.
9. N. Desai et al. Representing and reasoning about commitments in business processes. *AAAI*, 1328–1333. 2007.
10. M. Gelfond and V. Lifschitz. Action languages. *ETAI*, 3(6), 1998.
11. L. Giordano et al. Specifying and Verifying Interaction Protocols in a Temporal Action Logic. *Journal App. Logic*, 5(2), 2007.
12. A. Mallya et al. Resolving Commitments Among Autonomous Agents. *ACL*, 2003.
13. A. Mallya and M. Huhns. Commitments among agents. *IEEE Internet Comp.*, 7(4), 2003.
14. R. Reiter. Knowledge in Actions. MIT Press. 2001.
15. M.P. Singh. An ontology for commitments in multiagent systems. *Artif. Int. Law*, 7(1), 1999.
16. P. Torroni et al. Social Commitments in Time: Satisfied or Compensated. In *DALT*, 2009.
17. P. Torroni et al. Modelling interactions via commitments and expectations. *Handbook of Research on Multi-Agent Systems*, 263–284. IGI Global, 2009.
18. P. Yolum and M.P. Singh. Flexible protocol specification and execution: applying event calculus planning using commitments. *AAMAS*, pages 527–534. ACM, 2002.

An Integrated Formal Framework for Reasoning about Goal Interactions

Michael Winikoff*

Department of Information Science
University of Otago
Dunedin, New Zealand
michael.winikoff@otago.ac.nz

Abstract. One of the defining characteristics of intelligent software agents is their ability to pursue goals in a flexible and reliable manner, and many modern agent platforms provide some form of goal construct. However, these platforms are surprisingly naive in their handling of *interactions* between goals. Most provide no support for detecting that two goals interact, which allows an agent to interfere with itself, for example by simultaneously pursuing conflicting goals. Previous work has provided representations and reasoning mechanisms to identify and react appropriately to various sorts of interactions. However, previous work has not provided a framework for reasoning about goal interactions that is generic, extensible, formally described, and that covers a range of interaction types. This paper provides such a framework.

1 Introduction

One of the defining characteristics of intelligent software agents is their ability to pursue goals in a flexible and reliable manner, and many modern agent platforms provide some form of goal construct [1]. However, these platforms are surprisingly naive in their handling of *interactions* between goals in that few implemented agent platforms provide support for reasoning about interactions between goals. Platforms such as Jason, JACK, 2APL and many others don't make any attempt to detect interactions between goals, which means that agents may behave irrationally. Empirical evaluation [2] has shown that this can be a serious issue, and that the cost of introducing limited reasoning to prevent certain forms of irrational behaviour is low, and consistent with bounded reasoning.

There has been work on providing means for an agent to detect various forms of interaction between its goals, such as resource contention [3], and interactions involving logical conditions, both positive and negative (e.g. [4]). However, this strand of work has not integrated the various forms of reasoning into a single framework: each form of interaction is treated separately. Although more recent work by Shaw and Bordini [5] does integrate a range of interaction reasoning mechanisms, it does so indirectly, by translation to Petri nets, which makes it difficult to extend, to determine whether the

* This work was partly done while the author was employed by RMIT University.

reasoning being done is correct, or to relate the reasoning back to the agent's goals and plans (traceability).

This paper provides a framework for extending BDI platforms with the ability to reason about interactions between goals. The framework developed improves on previous work by being generic and by being formally presented. The sorts of goal interactions that we want to be able to model and reason about include the following.

Resources: goals may have resource requirements, including both reusable resources such as communication channels, and consumable resources such as fuel or money. Given a number of goals it is possible that their combined resource requirements exceed the available resources. In this case the agent should realise this, and only commit to pursuing some of its goals or, for reusable resources, schedule the goals so as to use the resources appropriately (if possible). Furthermore, should there be a change in either the available resources or the estimated resource requirements of its goals, the agent should be able to respond by reconsidering its commitments. For example, if a Mars rover updates its estimate of the fuel required to visit a site of interest (it may have found a shorter route), then the rover should consider whether any of its suspended goals may be reactivated.

Conditions: goals affect the state of the agent and of its environment, and may also at various points require certain properties of the agent and/or its environment. An agent should be aware of interactions between goals such as:

- After moving to a location in order to perform some experiment, avoid moving elsewhere until the experiment has been completed.
- If two goals involve being at the same location, schedule them so as to avoid travelling to the location twice.

In summary, the challenge is to provide mechanisms that allow for:

- Specification of the dependencies between goals/plans and resources/conditions. To be practical, dependencies must be specified in a local and modular fashion where each goal or plan only needs to specify the resources/conditions that it is directly affected by.
- Reasoning about conditions and resources so as to detect situations where there is interaction between goals.
- Having a means of specifying suitable responses to detected interactions. Possible responses include suspending or aborting a goal, changing the means by which a goal is achieved (e.g. travelling by train rather than plane to save money), and scheduling goals (e.g. to avoid double-booking a reusable resource).

Section 2, reviews the goal framework and agent notation that we will build on. Section 3 presents our framework for reasoning about goal interactions, and Section 4 completes the framework by extending the agent notation. We finally return to the motivating scenarios (Section 5) and then conclude (Section 6).

2 Conceptual Agent Notation with Generic Goals

We now briefly present the conceptual agent notation (CAN) [6]. CAN is used as a representative for a whole class of BDI agent languages which define agent execution

in terms of event-triggered plans, where multiple plans may be relevant to handle a given event. It is similar to AgentSpeak(L) [7], if somewhat more expressive (it provides additional constructs).

In order to extend goals into "interaction-aware goals" that are able to detect and respond to interactions with other goals we will use a variant of CAN which uses the generic goal construct of van Riemsdijk *et al.* [1]. Their framework defines a goal type with certain default life-cycle transitions, and provides a mechanism for adding additional life-cycle transitions. A goal type is defined in terms of a set C of condition-response pairs $\langle c, S\rangle$ where c is a condition to be checked that, if true, changes the goal's state to S. For example, a goal to achieve p includes $\langle p, \text{DROPPED}\rangle$ which specifies that when p becomes true the goal should be dropped. Condition-response pairs come in two flavours: "continuous", checked at all times, and "end", checked only at the start/end of plan execution. A goal instance $\mathsf{g}(C, \pi_0, S, \pi)$ specifies a set of condition-response pairs C, an initial plan π_0, a current state S (e.g. ACTIVE, DROPPED, SUSPENDED), and a current plan π.

The default goal life-cycle of van Riemsdijk *et al.* [1] is that goals are adopted into a suspended state, and they are then repeatedly activated and suspended until they are dropped. Active goals are subjected to means-end reasoning to find an abstract plan for pursuing the goal, and this plan is then executed (as long as the goal remains active).

We integrate this generic goal construct into CAN, replacing its more limited goal construct. The resulting language defines an agent in terms of a set Π of plans of the form $e : c \leftarrow \pi$ where e is the triggering event, c is a context condition (a logical formula over the agent's beliefs), and π is a plan body (we will sometimes refer to plan bodies as "plans"):

$$\pi ::= \epsilon \mid a \mid e \mid \pi_1; \pi_2 \mid \pi_1 \| \pi_2$$

We denote the empty plan body by ϵ, and an event is written as e. For simplicity we define a generic action construct, a, which has a pre condition pre_a and post-condition defined in terms of non-overlapping addition and deletion sets add_a and del_a. A number of previously defined CAN constructs can be viewed as special cases of this, for example $+b$ can be defined as an action with $pre_{+b} = true$, $add_{+b} = \{b\}$ and $del_{+b} = \emptyset$. Similarly, $-b$ has $pre_{-b} = \{b\}$, $add_{-b} = \emptyset$, $del_{-b} = \{b\}$ and $?c$ has $pre_{?c} = \{c\}$ and $add_{?c} = del_{?c} = \emptyset$. We assume that events e and actions a can be distinguished. An agent configuration is a pair $\langle B, G\rangle$ where B is the agent's current beliefs, and G is a set of goals.

Figures 1 and 2 provide formal semantics for this language (based on previously presented semantics for CAN [1, 6, 8]) in structured operational semantics style [9] where the premise (above the line) gives the conditions under which the transition below the line may take place. We define a number of different transition types. Firstly, $\rightarrow$ as being a transition over a *set* of goals (i.e. $\langle B, G\rangle$), and $\Rightarrow$ is defined as being a transition over a *single* goal/plan (i.e. $\langle B, g\rangle$ where $g \in G$). Furthermore, we use letters to denote particular transition types (e for execute, u for update) and a superscript asterisk ($*$) denotes "zero or more" as is usual. For conciseness we abbreviate $\langle B, g\rangle$ by just g, for example the bottom of rule 9 abbreviates $\langle B, e\rangle \stackrel{e}{\Rightarrow} \langle B', (\!|\Gamma|\!)\rangle$, and similarly for rules 1-3.

$$\frac{S = \textsc{Active}}{\mathsf{g}(C,\pi_0,S,\epsilon) \overset{e}{\Rightarrow} \mathsf{g}(C,\pi_0,S,\pi_0)}\ 1 \qquad \frac{\pi \overset{e}{\Rightarrow} \pi' \quad S = \textsc{Active}}{\mathsf{g}(C,\pi_0,S,\pi) \overset{e}{\Rightarrow} \mathsf{g}(C,\pi_0,S,\pi')}\ 2$$

$$\frac{\langle c, S', f\rangle \in C \quad B \models c \quad S \neq S' \quad ok(f,\pi)}{\mathsf{g}(C,\pi_0,S,\pi) \overset{u}{\Rightarrow} \mathsf{g}(C,\pi_0,S',\pi)}\ 3$$

$$\frac{g \in G \quad \langle B, g\rangle \overset{e}{\Rightarrow} \langle B', g'\rangle}{\langle B, G\rangle \overset{e}{\rightarrow} \langle B', (G \setminus \{g\}) \cup \{g'\}\rangle}\ 4 \qquad \frac{g \in G \quad g \overset{u}{\Rightarrow} g'}{\langle B, G\rangle \overset{u}{\rightarrow} \langle B, (G \setminus \{g\}) \cup \{g'\}\rangle}\ 5$$

$$\frac{\langle B, G\rangle \overset{u}{\rightarrow}{}^{*} \langle B, G'\rangle \quad \langle B, G'\rangle \overset{u}{\not\rightarrow} \langle B, G''\rangle}{\langle B, G\rangle \overset{u}{\twoheadrightarrow} \langle B, \{g | g \in G' \wedge g \neq \mathsf{g}(C,\pi_0,\textsc{Dropped},\pi)\}\rangle}\ 6$$

$$\frac{\langle B, G\rangle \overset{u}{\twoheadrightarrow} \langle B, G''\rangle \quad \langle B, G''\rangle \overset{e}{\rightarrow} \langle B', G'\rangle}{\langle B, G\rangle \rightarrow \langle B', G'\rangle}\ 7$$

Fig. 1. Formal semantics for CAN with generic goals

Figure 1 defines the semantics of goals. The first two rules specify that an active goal can be executed by replacing an empty plan with the initial plan π_0 (rule 1) or by executing the goal's plan (rule 2) which makes use of the rules for plan execution (Figure 2). The next rule (3) defines a single goal update: if an update condition holds, update the goal's state, subject to two conditions: firstly, the new state should be different ($S \neq S'$), secondly, the condition c should be active given the f tag[1] and the plan π; formally $ok(f,\pi) \equiv ((f = end \wedge \pi = \epsilon) \vee (f = mid \wedge \pi \neq \epsilon) \vee f = all)$. Rules 4 and 5 define respectively execution (rule 4) and update (rule 5) of a set of goals by selecting a single goal and respectively executing it or updating it. Rule 6 defines a complete update cycle $\overset{u}{\twoheadrightarrow}$ which performs all possible updates, and deletes goals with a state of "DROPPED". Rule 7 defines a single top-level transition step of a set of goals: first perform all possible updates ($\overset{u}{\twoheadrightarrow}$) and then perform a single execution step ($\overset{e}{\rightarrow}$). We require that all possible updates are done in order to avoid ever executing a goal that has a pending update to a non-active state.

Figure 2 defines a single execution step ($\overset{e}{\Rightarrow}$) for various CAN plan constructs. Rule 8 defines how an action a is executed in terms of its precondition and add/delete sets. Rule 9 defines how an event is replaced by the set of guarded relevant plan instances ($\langle\!|\Gamma|\!\rangle$). Rule 10 selects an applicable plan instance from a set of plans, using the auxiliary construct $\triangleright$ to indicate "try π, but if it fails, use the set of (remaining) relevant plans". Rule 11 simply defines the semantics of sequential execution ";", rules 12 and 13 define parallel execution "$\|$", and rules 14 and 15 define "try-else" ($\triangleright$). The function denoted by an overline (e.g. $\overline{\pi_1;\pi_2}$) cleans up by removing empty plan bodies: $\overline{\epsilon;\pi} = \overline{\epsilon\|\pi} = \overline{\pi\|\epsilon} = \pi$, and $\overline{\epsilon \triangleright \pi} = \epsilon$, otherwise $\overline{\pi} = \pi$.

[1] We have compressed the two sets C and E of van Riemsdijk *et al.* [1] into a single set of triples $\langle c, S, f\rangle$ where f is a flag specifying when the condition should be checked: when the plan is empty (end), when the plan is non-empty, i.e. during execution (mid) or at all times (all). E.g. $\langle c, S\rangle \in C$ in their framework translates to $\langle c, S, all\rangle$.

$$\frac{B \models pre_a}{\langle B, a\rangle \overset{e}{\Rightarrow} \langle (B \cup add_a) \setminus del_a, \epsilon\rangle}\ 8 \qquad \frac{\Gamma = \{c\theta{:}\pi\theta \mid (e'{:}c{\leftarrow}\pi) \in \Pi \wedge \theta = \mathsf{mgu}(e, e')\}}{e \overset{e}{\Rightarrow} (\!|\Gamma|\!)}\ 9$$

$$\frac{(c_i{:}\pi_i) \in \Gamma \quad B \models c_i\theta \quad \pi_i\theta \overset{e}{\Rightarrow} \pi'}{(\!|\Gamma|\!) \overset{e}{\Rightarrow} \pi_i\theta \triangleright (\!|\Gamma \setminus \{c_i{:}\pi_i\}|\!)}\ 10 \qquad \frac{P_1 \overset{e}{\Rightarrow} P'}{P_1; P_2 \overset{e}{\Rightarrow} \overline{P'; P_2}}\ 11 \qquad \frac{P_1 \overset{e}{\Rightarrow} P'}{P_1 \| P_2 \overset{e}{\Rightarrow} \overline{P' \| P_2}}\ 12$$

$$\frac{P_2 \overset{e}{\Rightarrow} P'}{P_1 \| P_2 \overset{e}{\Rightarrow} \overline{P_1 \| P'}}\ 13 \qquad \frac{P_1 \overset{e}{\Rightarrow} P'}{P_1 \triangleright P_2 \overset{e}{\Rightarrow} \overline{P' \triangleright P_2}}\ 14 \qquad \frac{P_1 \overset{e}{\not\Rightarrow} P' \quad P_2 \overset{e}{\Rightarrow} P_2'}{P_1 \triangleright P_2 \overset{e}{\Rightarrow} P_2}\ 15$$

Fig. 2. Formal semantics for CAN plan constructs

Note that the semantics model failure as an inability to progress, i.e. a failed plan body π is one where $\pi \not\Rightarrow \pi'$. This simplifies the semantics at the cost of losing the distinction between failure and suspension, and creating a slight anomaly with parallelism where given $\pi_1 \| \pi_2$ we can continue to execute π_2 even if π_1 has "failed". Both these issues are easily repaired by modelling failure separately (as is done by Winikoff *et al.* [6]), but this makes the semantics considerably more verbose.

We can now define a (very!) simple Mars rover that performs a range of experiments at different locations on the Martian surface. The first plan below for performing an experiment of type X at location L firstly moves to the appropriate location L, then collects a sample using the appropriate measuring apparatus.

$$exp(L, X) \ :\ \neg locn(L) \ \leftarrow\ goto(L)\ ;\ sample(X)$$
$$exp(L, X) \ :\ locn(L) \quad \leftarrow\ sample(X)$$

We assume for simplicity of exposition that $goto(L)$, and $sample(X)$ are primitive actions, but they could also be defined as events that trigger further plans. The action $goto(L)$ has precondition $\neg locn(L)$ and add set $\{locn(L)\}$ and delete set $\{locn(x)\}$ where x is the current location.

3 Reasoning about Interactions

We provide reasoning about interactions between goals by:

1. Extending the language to allow goal requirements (resources, conditions to be maintained etc.) to be specified (Section 3.1).
2. Providing a mechanism to reason about these requirements, specifically by aggregating requirements and propagating them (Section 3.2).
3. Defining new conditions that can be used to specify goal state transitions, and adding additional state transition types that allow responses to detected interactions to be specified. These are then used to extend CAN with interaction-aware goals (Section 4).

3.1 Specifying Requirements

There are a number of ways of specifying requirements. Perhaps the simplest is to require each primitive action to specify its requirements. Unfortunately this is less flexible since it does not allow the user to indicate that a plan, perhaps combining a number of actions, has certain requirements. We thus extend the language with a construct $\tau(\pi, R)$ which indicates that the plan π is tagged ("τ") with requirements R. It is still possible to annotate actions directly, $\tau(a, R)$, but it is no longer the only place where requirements may be noted.

However, in some cases, the requirements of a goal or plan can only be determined in context. For example, the fuel consumed in moving to a location depends on the location, but also on the current location, which is not known ahead of time. We thus provide a second mechanism for dynamic tagging where the requirements of a goal/plan are provided in terms of a procedure that computes the requirements, and a condition that indicates when the procedure should be re-run. This is denoted $\tau(\pi, f, c)$ where f is a function that uses the agent's beliefs to compute the requirements, and c is a re-computation condition. Once the requirements have been propagated (see next section) this becomes $T(\pi, R, f, c)$ (the difference between τ and T is discussed in Section 3.2) we need to retain f and c so the requirements can be re-computed (if c becomes true). Otherwise $T(\pi, R, f, c)$ behaves just like $T(\pi, R)$.

We define R as being a pair of two sets, $\langle L, U\rangle$, representing a lower and upper bound. For convenience, where a requirement R is written as a set $R = \{\ldots\}$ then it is taken to denote the pair $\langle R, R\rangle$. Each of the sets can be defined in many ways, depending on the needs of the domain and application. Here we define each set as containing a number of the following requirement statements:

- $re(r/c, t, n)$ where the first argument in the term is either r or c, denoting a reusable or consumable resource, t is a type (e.g. fuel), and n is the required amount of the resource.
- $pr(c)$ where c is a condition that must be true at the *start* of execution (i.e. a pre-condition)
- $in(c)$ where c is a condition that must be true *during* the *whole* of execution (including at the start). For the computation of summaries we also define a variant in_s which means that c must be true *somewhere* during the execution but not necessarily during the whole execution.

In the Mars rover example we have the following requirements:

1. $goto(L)$ computes its requirements based on the distance between the destination and current location. This needs to be re-computed after each goto. We thus specify the requirements of the $goto(L)$ action as $\tau(goto(L), f(L), c)$ where $f(L)$ looks up the current location $locn$ in the belief base, and then computes the distance between it and L; and where $c = \Delta locn(x)$ (informally, Δc means that the belief base has changed in a way that affects the condition c; formally, if B is the old belief base and B' the updated belief base, then $\Delta c \equiv \neg(B \models c \Leftrightarrow B' \models c)$);
2. $sample(X)$ requires that the rover remains at the desired location, hence we specify an in-condition (in) that the location ($locn$) remains L: $\tau(sample(X), \{in(locn(L))\})$.

We thus provide requirements by specifying the following plan body (for the first plan): $\tau(goto(L), f(L), c); \tau(sample(X), \{in(locn(L))\})$

3.2 Propagating Requirements

We define a function Σ that takes a plan body and tags it with requirements by propagating and aggregating given requirements. We use ε to denote the empty requirement. The function returns a modified plan which contains tags of the form $T(\pi, R)$: this is different from $\tau(\pi, R)$ in that τ is used by the user to provide requirements for a plan, not including the requirements of the plan's sub-plans, but T *does* include the requirements of sub-plans. Observe that Σ is defined compositionally over the plan, and that computing it is not expensive in the absence of recursive plans [2].

$\Sigma(\epsilon) = T(\epsilon, \varepsilon)$
$\Sigma(a) = T(a, \{pr(pre_a)\})$
$\Sigma(e) = T(e, \langle L_1 \sqcap \ldots \sqcap L_n, U_1 \sqcup \ldots \sqcup U_n \rangle)$, where $T(\pi'_i, \langle L_i, U_i \rangle) = \Sigma(\pi_i)$
and $\pi_1 \ldots \pi_n$ are the plans relevant for e.
$\Sigma(\pi_1; \pi_2) = T(\pi'_1; \pi'_2, \langle L_1 \fatsemi L_2, U_1 \fatsemi U_2 \rangle)$, where $T(\pi'_i, \langle L_i, U_i \rangle) = \Sigma(\pi_i)$
$\Sigma(\pi_1 || \pi_2) = T(\pi'_1 || \pi'_2, \langle L_1 \,[\!]\, L_2, U_1 \,[\!]\, U_2 \rangle)$, where $T(\pi'_i, \langle L_i, U_i \rangle) = \Sigma(\pi_i)$
$\Sigma(\pi_1 \triangleright \pi_2) = T(\pi'_1 \triangleright \pi'_2, \langle L_1, U_1 \fatsemi U_2 \rangle)$, where $T(\pi'_i, \langle L_i, U_i \rangle) = \Sigma(\pi_i)$
$\Sigma(\langle\!|\Gamma|\!\rangle) = T(\langle\!|\Gamma'|\!\rangle, \langle L_1 \sqcap \ldots \sqcap L_n, U_1 \sqcup \ldots \sqcup U_n \rangle)$, where $T(\pi'_i, \langle L_i, U_i \rangle) = \Sigma(\pi_i)$
and $\Gamma = \{b_1{:}\pi_1, \ldots, b_n{:}\pi_n\}$ and $\Gamma' = \{b_1{:}\pi'_1 \ldots, b_n{:}\pi'_n\}$.
$\Sigma(\tau(\pi, \langle L, U \rangle)) = T(\pi', \langle L' \oplus L, U' \oplus U \rangle)$, where $T(\pi', \langle L', U' \rangle) = \Sigma(\pi)$
$\Sigma(\tau(\pi, f, c)) = \Sigma(T(\pi, f(B), f, c))$, where B is the agent's beliefs.
$\Sigma(T(\pi, R)) =$ if $\pi' = T(\pi'', \varepsilon)$ then $T(\pi'', R)$ else π', where $\pi' = \Sigma(\pi)$

The requirements of an action are simply its pre-condition. The requirements of an event are computed by taking the requirements of the set of relevant plans and combining them: the best case is the minimum of the available plans ($\sqcap$), and in the worse case (represented by the upper bound) we may need to execute all of the plans and so we take the (unspecified sequential) maximum of the requirements of the available plans using $\sqcup$. The requirements for $\pi_1; \pi_2$ and $\pi_1 \| \pi_2$ are determined by computing the requirements of π_1 and of π_2 and then combining them appropriately with auxiliary functions $\fatsemi$ and $[\!]$ which are both variants on $\sqcup$: $\fatsemi$ treats pre-conditions of the first requirement set as pre-conditions, since we *do* know that they occur at the start of execution; and $[\!]$ is like $\sqcup$ except that, because execution is in parallel, we cannot reuse resources. The lower bound requirements for $\pi_1 \triangleright \pi_2$ are just the (lower bound) requirements for π_1, since, if all goes well, there will be no need to execute π_2. However, in the worse case (upper bound) both π_1 and π_2 will need to be executed (sequentially, hence $\fatsemi$). The requirements for a user-tagged plan, $\tau(\pi, R)$ are determined by computing the requirements of π and then adding ($\oplus$) this to the provided R. Finally, when re-computing the requirements of $T(\pi, R)$ we simply replace R with the newly computed requirements.

The case for events (e) is interesting: in the worst case, we may need to execute all of the available plans. In the best case, only one plan will be executed. However, when computing the initial estimate of requirements we don't know which plans are applicable, so we overestimate by using all relevant plans, and later update the estimate (see below).

The function Σ is defined in terms of a number of auxiliary functions: $\oplus$, $\sqcup$, $\sqcap$, $\fatsemi$ and $[]$. By defining what can appear in R as well as these functions the agent designer can create their own types of reasoning. We have defined an example R in the previous section, and briefly and informally given the intended meaning of the auxiliary functions above. The appendix contains precise formal definitions of these auxiliary functions.

We integrate requirements propagation into the operational semantics of CAN by defining operational semantics for the $T(\pi, R)$ construct which captures the process of updating requirements:

$$\frac{\pi \Rightarrow \pi' \quad \pi \neq \epsilon \quad R \neq \varepsilon}{T(\pi, R) \Rightarrow \Sigma(\pi')} \qquad \frac{\pi \neq \epsilon}{T(\pi, \varepsilon) \Rightarrow \pi} \qquad \frac{}{T(\epsilon, R) \Rightarrow \epsilon}$$

The first rule is the general case: if π executes one step to π' then $T(\pi, R)$ can also be stepped to $\Sigma(\pi')$. The next rule specifies that tagging with an empty requirement set can be deleted. The final rule allows an empty plan body with requirements to be resolved to the empty plan body. Finally, we modify the goal initialisation rule (first rule in figure 1) to compute requirements by replacing the right-most π_0 with $\Sigma(\pi_0)$:

$$\frac{S = \text{ACTIVE}}{\mathsf{g}(C, \pi_0, S, \epsilon) \stackrel{e}{\Rightarrow} \mathsf{g}(C, \pi_0, S, \Sigma(\pi_0))}$$

Alternatively, as is done by Thangarajah *et al.* [3], we could modify the agent's plan set by replacing π with $\Sigma(\pi)$ at compile time.

Returning to the Mars rover, let $\pi = \tau(goto(L), f, c); \tau(sample(X), \{in(locn(L))\})$ then the following requirements are computed (recall that $T(\pi, R)$ where R is a set is short for $T(\pi, \langle R, R \rangle)$, and we assume that f returns 20 for the fuel requirement of reaching L from the starting location):

$$\begin{aligned}
\Sigma(\pi) &= T(\pi_2; \pi_3, \{re(c, \mathit{fuel}, 20), in_s(locn(L)), in_s(\neg locn(L))\}) \\
\pi_2 &= T(goto(L), \{re(c, \mathit{fuel}, 20), pr(\neg locn(L))\}, f, c) \\
\pi_3 &= T(sample(X), \{in(locn(L))\})
\end{aligned}$$

After the first action (π_2) has completed we have:
$\Sigma(\pi') = T(\pi_3, \{in(locn(L)), pr(\neg locn(L))\})$.

4 Using Requirements to Deal with Interactions

The work of the previous sections allows us to specify requirements, and to compute and update them. In this section we consider how this information can be used to avoid undesirable interactions and (attempt to) ensure desirable interactions. For goals, we do this by defining a new goal type, an "interaction-aware goal", which has additional condition-response triples. But first, we define a number of new conditions, and new responses.

4.1 Conditions

The language of conditions is extended with new constructs: rok ("resources are ok"), $culprit$, and $interfere$

The new condition $rok(G)$ means that there are enough resources for all of the goals in G. Informally we define $rok(G)$ by computing the resource requirements of the active goals in G and comparing it with the available resources. If the available resources exceed the resource requirements of the active goals, then clearly $rok(G)$ is true. If not, then we need to work out which goals should be suspended. We define the condition $culprit(g)$ to indicate that the goal g is responsible for a lack of sufficient resources. Informally, $culprit(g)$ is true if removing g from G makes things better[2] i.e., $culprit(g) \equiv rok(G \setminus \{g\}) \wedge \neg rok(G)$. See the appendix (definitions 4 and 5) for the (correct) formal definitions of rok and $culprit$.

The new condition $interfere(g)$ is true if g is about to do something that interferes with another goal. Informally, this is the case if one of the actions that g may do next (denoted $na(g)$, defined in the appendix) has an effect that is inconsistent with another goal's in-condition (where both goals are active). Formally $interfere(g) \equiv \exists g' \in (G \setminus \{g\}), c \in getin(g'), a \in na(g)\ .\ g.S = g'.S = \textsc{Active} \wedge \mathit{eff}_a \supset \neg c$ where G is the agent's goals, we use $g.S$ to refer to the state of the goal g, we use $\supset$ to denote logical implication (to avoid confusion with transitions), and we define $getin$ to return the in-conditions of a goal, plan or requirements set:

$$\begin{aligned} getin(\mathsf{g}(C, \pi_0, S, \pi)) &= getin(\pi) \\ getin(T(\pi, \langle L, U\rangle)) &= \{c \mid in(c) \in L\} \\ getin(\pi) &= getin(\Sigma(\pi)), \text{ if } \pi \neq T(\pi', R) \end{aligned}$$

We also define eff_a to be a logical formula combining add_a and del_a as a conjunction of atoms in add_a and the negations of atoms in del_a. For example, for $goto(L)$ we have $\mathit{eff}_{goto(L)} = locn(L) \wedge \neg locn(x)$.

We can also define a similar condition that detects interference with pre-conditions. In order to avoid suspending goals unnecessarily, we only consider interference to be real if the pre-condition being affected currently holds. In other words, if the precondition c of goal g' does *not* currently hold, then there is not a strong reason to suspend goal g which makes c false because c is *already* false. This gives $interfere_{pre}(g) \equiv \exists g' \in (G \setminus \{g\}), c \in getpre(g'), a \in na(g)\ .\ B \models c \wedge g.S = g'.S = \textsc{Active} \wedge \mathit{eff}_a \supset \neg c$ where $getpre$ retrieves the pre-conditions, similarly to $getin$ (see appendix definition 2).

4.2 Responses

Responses to interactions can be either "subtle": influencing existing choices, but not changing the semantics, i.e. "subtle" responses can be viewed as refining the semantics by reducing non-determinism. Alternatively, responses can be "blunt" responses which change the semantics.

[2] In fact, as discussed in the appendix, this isn't entirely correct.

So-called "subtle" responses apply where there is a choice to be made in the execution. This is the case in the following places: when selecting which (top-level) goal to execute, when selecting which plan to use from a set of alternatives ($(\!|\Gamma|\!)$), and when selecting which parallel plan to execute ($\pi_1 \| \pi_2$). Note that only the first case involves goals: the second and third involve plan bodies.

Influencing the choice of goal can be done by a range of means, including suspending goals and giving certain goals higher priority. Suspending goals can be done using the generic goal mechanism. In order to allow a goal to be given a higher priority we define a new *response* (not goal state) PICKME (below).

Influencing the selection of a plan from a set of possible plans ($(\!|\Gamma|\!)$) can be done by modifying the selection rule (it can't be done using the generic goal mechanism because plan selection occurs within a single goal). For example, we could require that an applicable plan is not selected if a cheaper plan exists. This can be formalised by adding to the rule for plan selection the following additional condition (the relation $\prec$ and function $getres$ are defined in the appendix in definitions 1 and 3):

$$\frac{(c_i{:}\pi_i) \in \Gamma \quad B \models c_i\theta \quad \neg\exists (c_j{:}\pi_j) \in \Gamma . getres(\pi_j) \prec getres(\pi_i)}{(\!|\Gamma|\!) \stackrel{e}{\Rightarrow} \pi_i\theta \triangleright (\!|\Gamma \setminus \{c_i{:}\pi_i\}|\!)}$$

However, we do not consider plan selection to be particularly useful in preventing resource issues, because the set of applicable plans will typically not contain a wide range of options.

The third case, influencing the scheduling of parallel plans ($\pi_1 \| \pi_2$) we consider to be less useful and leave it for future work.

Turning now to the so-called "blunt" responses we have a number of possible responses including: (a) dropping a goal, and (b) adding a new goal. The former may be used to permanently eliminate a goal that cannot be achieved (although suspension may be a more sensible response). The second may be used to create a new goal (or plan), for example, if a resource shortage is detected, a plan may be created to obtain more of the resource (e.g. re-fuelling).

We thus define the following additional responses:

- $!\pi$ which executes π (we can define synchronous and asynchronous variants of this)
- PICKME which specifies that this goal should be given priority when selecting which goal to execute (but, since more than one goal may be flagged as PICKME, cannot guarantee that the goal will be selected next). More generally, we could have a priority mechanism and have responses that raise/lower the priority of the goal.

These are defined formally as follows. Although they appear in condition-response triples, the semantics of these two constructs aren't just changing the state of the goal, and so we revise the existing rule so it does not apply to these two responses:

$$\frac{\langle c, S', f\rangle \in C \quad S' \in asd \quad B \models c \quad S \neq S' \quad ok(f, \pi)}{\mathsf{g}(C, \pi_0, S, \pi) \stackrel{u}{\Rightarrow} \mathsf{g}(C, \pi_0, S', \pi)}\ 3'$$

where $asd = \{\text{ACTIVE}, \text{SUSPENDED}, \text{DROPPED}\}$.

Because we want a PICKME to only last while the corresponding condition is true, we do not update the goal's state to PICKME, but instead modify the selection rule (rule 4) by adding the following additional condition (premise, where we use $\supset$ to denote logical implication) which requires that if any active goals are prioritised, then the selected goal must be a prioritised one: ($(\exists \mathsf{g}(C', \pi_0', \text{ACTIVE}, \pi') \in G \,.\, \langle c', \text{PICKME}\rangle \in C' \wedge B \models c') \supset (\langle c, \text{PICKME}\rangle \in g.C \wedge B \models c)$). Where g is the goal being selected, and where we use $g.C$ to denote the C set of g (i.e. $g = \mathsf{g}(C, \pi_0, S, \pi)$).

We now turn to $!\pi$. A response of the form $!\pi$ transforms the goal from $\mathsf{g}(C, \pi_0, S, \pi')$ to the variant $\mathsf{g}_\pi(C, \pi_0, S, \pi')$:

$$\frac{\langle c, !\pi\rangle \in C \quad B \models c}{\mathsf{g}(C, \pi_0, S, \pi') \overset{u}{\Rightarrow} \mathsf{g}_\pi(C, \pi_0, S, \pi')} \; 16$$

We then define the semantics of this as follows:

$$\frac{\pi \overset{e}{\Rightarrow} \pi_1}{\mathsf{g}_\pi(C, \pi_0, S, \pi') \overset{e}{\Rightarrow} \overline{\mathsf{g}_{\pi_1}(C, \pi_0, S, \pi')}} \; 17$$

where $\overline{\mathsf{g}_\epsilon(C, \pi_0, S, \pi)} = \mathsf{g}(C, \pi_0, S, \pi)$, and for $g = \mathsf{g}_\pi(\ldots)$ with $\pi \neq \epsilon$ we have $\overline{g} = g$.

4.3 Interaction-Aware Goals

Finally, we are in a position to define a new goal type which uses the conditions and responses defined, along with the underlying infrastructure for specifying and propagating requirements, in order to deal with interactions as part of the agent's goal reasoning process.

We extend goals into interaction-aware goals by simply adding to their C set the following condition-response triples, where $culprit$ is short for $culprit(g)$ with g being the current goal, and similarly for $interfere$. The condition $notculprit$ differs from $\neg culprit$ in that it includes the current goal g in the computation of resources (whereas $culprit$ treats it as not having any resource requirements, since it is suspended). Formally $notculprit(\mathsf{g}(C, \pi_0, \text{SUSPENDED}, \pi)) \equiv \neg culprit(\mathsf{g}(C, \pi_0, \text{ACTIVE}, \pi))$.

$$\begin{aligned}\mathcal{I} = \{&\langle culprit, \text{SUSPENDED}, all\rangle, \langle notculprit, \text{ACTIVE}, all\rangle,\\ &\langle interfere, \text{SUSPENDED}, all\rangle, \langle \neg interfere, \text{ACTIVE}, all\rangle\}\end{aligned}$$

An alternative, if there is a plan π_r which obtains more of a needed resource, is to use it instead of suspending: $\mathcal{I}' = \{\langle culprit, !\pi_r, all\rangle, \ldots\}$.

5 Motivating Scenarios Revisited

We now consider how the different forms of reasoning discussed at the outset can be supported. We define

$$gexp(l, x) \equiv g(\mathcal{I} \cup \{\langle locn(l), \text{PICKME}, all\rangle\}, exp(l, x))$$

that is, $gexp(l, x)$ is an interaction-aware goal which uses the initial plan body (which is actually just an event) $exp(l, x)$. Finally, we suppose that the Mars rover has been asked to perform three experiments: experiment 1 of type T_1 at location L_A (i.e. $g_1 = gexp(L_A, T_1)$) experiment 2 of type T_1 at location L_B (i.e. $g_2 = gexp(L_B, T_1)$), and experiment 3 of type T_2 at location L_A (i.e. $g_3 = gexp(L_A, T_2)$).

Let us now briefly consider how the Mars rover deals with the following cases of interaction:

1. **A lack of resources causes a goal to be suspended, and, when resources are sufficient, resumed:** since the goals are interaction-aware, suspension and resumption will occur as a result of the conditions-responses in $\mathcal{I}$. Specifically, should the resources available be insufficient to achieve all goals, then some of goals will be suspended by the $\langle culprit, \text{SUSPENDED}, all\rangle$ condition-response triple. Note that since updates are performed one at a time, this will only suspend as many goals as are needed to resolve the resource issue.
   If further resources are obtained, then the suspended goals will be re-activated ($\langle notculprit, \text{ACTIVE}, all\rangle$). In the case of reusable resources, the suspension/resumption mechanism will realise scheduling of the reusable resources amongst goals: once a goal has completed and releases the (reusable) resources it has been using, another goal that requires these resources can then resume.
2. **A lack of resources, instead of suspending, may trigger a plan to obtain more resources:** if the goals are defined using $\mathcal{I}'$ rather than $\mathcal{I}$, then a lack of resources will cause a plan body π_r to be used to obtain more resources. In this domain, where the main resource is fuel, a sensible choice for π_r would be to re-fuel.
3. **Once the Mars rover has moved to location L_A, it avoids moving again until the sampling at L_A has completed:** once goal g_1 has executed $goto(L_A)$ then, as discussed at the end of Section 3.2, its requirement is updated to include the in-condition $locn(L_A)$. Should goal g_2 get to the point of being about to execute its action $goto(L_B)$, then this next action interferes with the in-condition, and goal g_2 will then be suspended, using the condition-response triple $\langle interfere, \text{SUSPENDED}, all\rangle$, preventing the execution of $goto(L_B)$. Once g_1 has concluded the experiment, then it no longer has $locn(L_A)$ as an in-condition, and at this point g_2 will be re-activated ($\langle \neg interfere, \text{ACTIVE}, all\rangle$).
4. **Once it has moved to location L_A, the rover also performs g_3 before moving elsewhere:** when it reaches L_A the PICKME response of g_3 (and g_1) is triggered which prioritises selecting these goals over g_2, and thus the rover will remain at L_A until g_1 and g_3 are both completed.

As can be seen, interaction-aware goals — which are defined in terms of the additional condition and response types, which themselves rest on the resource specification and propagation mechanism defined in Section 3 — are able to deal with a range of goal-interaction scenarios.

6 Discussion

We have provided a framework for reasoning about goal interactions that is: **generic**, i.e. can be customised to provide the reasoning that is needed for the application at hand;

presented formally, and hence precisely, avoiding the ambiguity of natural language; and that **integrates** different reasoning types into one framework. We have also defined a wider range of conditions and responses than previous work.

Our work can be seen as a rational reconstruction of earlier work [3, 4] which formalises and makes precise the English presentation in these papers. However, we do more than just formalise existing work: we provide a generic framework that allows for other forms of reasoning to be added, and for the existing forms to be integrated.

In addition to work on reasoning about interactions between an agent's goals, there has also been work on reasoning about interactions between the goals of different agents [10, 11]. This work has a somewhat different flavour in that it is concerned with the cost of communication between agents. However, in some other aspects, such as the use of requirements summaries, it is similar to the single agent case.

Also related is the work by Horty and Pollack [12] which looked at the cost of plans in context (i.e. taking into account the agent's other plans). Although the paper is ostensibly concerned with cost, they do also define various notions of compatibility between plans. However, their plans are composed only of primitive actions.

Thangarajah *et al.* [13] consider the goal adoption part of goal deliberation: should a candidate goal (roughly speaking, a desire) be added to the agent's set of adopted goals? They embed the goal adoption problem in a BDI setting into a soft constraint optimisation problem model and discuss a range of factors that can be taken into account in making decisions. However, while promising, this is early work: the presentation is informal and a precise definition of the mapping to soft constraint optimisation problems is not given.

There are three main directions for future work that we would like to pursue: implementation, evaluation, and extending to further interaction scenarios.

What this paper presents can be seen as an extended BDI programming language with interaction-aware goals. One area for future work is how to implement this extended language using a standard BDI platform (such as Jason, Jadex, JACK etc.) that doesn't have a generic goal construct, or resource/condition management. One possibility is to transform the agent program, Π, into a variant that uses existing constructs (such as maintenance goals) to realise the desired behaviour. Another possibility, if the platform provides an API for manipulating the state of goals, is to realise generic goals by two parallel goals: one that executes the plan π, and another (with higher priority) that monitors for conditions and updates the first goal's state. An implementation would allow for an evaluation to be done in order to assess the benefits, and also the real practical computational cost.

An interesting scenario which we have not yet investigated is "achieve then maintain", where a particular condition is achieved (e.g. booking a hotel), but then for some period of time (e.g. until the travel dates) the condition is maintained and updated should certain changes take place (e.g. budget reductions or changes to travel dates).

References

1. van Riemsdijk, M.B., Dastani, M., Winikoff, M.: Goals in agent systems: A unifying framework. In: Proceedings of the Seventh International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS). (2008) 713–720

2. Thangarajah, J., Padgham, L.: Computationally effective reasoning about goal interactions. Journal of Automated Reasoning (2010) 1–40
3. Thangarajah, J., Winikoff, M., Padgham, L., Fischer, K.: Avoiding resource conflicts in intelligent agents. In van Harmelen, F., ed.: Proceedings of the 15th European Conference on Artificial Intelligence, IOS Press (2002) 18–22
4. Thangarajah, J., Padgham, L., Winikoff, M.: Detecting and avoiding interference between goals in intelligent agents. In: Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI). (2003) 721–726
5. Shaw, P.H., Bordini, R.H.: Towards alternative approaches to reasoning about goals. In Baldoni, M., Son, T.C., van Riemsdijk, M.B., Winikoff, M., eds.: Declarative Agent Languages and Technologies (DALT). (2007) 164–181
6. Winikoff, M., Padgham, L., Harland, J., Thangarajah, J.: Declarative & procedural goals in intelligent agent systems. In: Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR2002), Toulouse, France (2002) 470–481
7. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In de Velde, W.V., Perrame, J., eds.: Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96), Springer Verlag (1996) 42–55 LNAI, Volume 1038.
8. Sardina, S., Padgham, L.: Goals in the context of BDI plan failure and planning. In: Proceedings of the Sixth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS). (2007) 16–23
9. Plotkin, G.: Structural operational semantics (lecture notes). Technical Report DAIMI FN-19, Aarhus University (1981 (reprinted 1991))
10. Clement, B.J., Durfee, E.H.: Identifying and resolving conflicts among agents with hierarchical plans. In: AAAI Workshop on Negotiation: Settling Conflicts and Identifying Opportunities, Technical Report WS-99-12. (1999)
11. Clement, B.J., Durfee, E.H.: Theory for coordinating concurrent hierarchical planning agents using summary information. In: Proceedings of the Sixteenth National Conference on Artificial Intelligence. (1999) 495–502
12. Horty, J.F., Pollack, M.E.: Evaluating new options in the context of existing plans. Artificial Intelligence **127**(2) (2001) 199–220
13. Thangarajah, J., Harland, J., Yorke-Smith, N.: A soft COP model for goal deliberation in a BDI agent. In: Proceedings of the Sixth International Workshop on Constraint Modelling and Reformulation (ModRef). (September 2007)

A Definitions

Definition 1 ($\prec$). *We define an ordering on requirement sets as follows. We say that R_1 is less than R_2 ($R_1 \preceq R_2$) if, intuitively, R_2 requires more than R_1. Formally, we define this by recognising that for a given condition c we have that $in_s(c) \preceq pr(c) \preceq in(c)$, i.e. a requirement that a condition hold for some unspecified part of the execution is less demanding than insisting that it hold at the start, which in turn is less demanding than insisting that it hold during the whole of execution (including at the start). We thus define $R_1 \preceq R_2$ to hold iff:*

- $re(f, t, n_1) \in R_1 \supset (r(f, t, n_2) \in R_2 \wedge n_1 \leq n_2)$
- $in(c) \in R_1 \supset (in(c') \in R_2 \wedge c' \supset c)$

- $pr(c) \in R_1 \supset ((pr(c') \in R_2 \vee in(c') \in R_2) \wedge c' \supset c)$
- $in_s(c) \in R_1 \supset ((in_s(c') \in R_2 \vee pr(c') \in R_2 \vee in(c') \in R_2) \wedge c' \supset c)$

We next define na ("next action") which takes a plan body and returns a set of possible next actions. Note that na is an approximation: it doesn't attempt to predict which actions might result from a set of plans $(\!|\Gamma|\!)$. A more accurate approach is to wait until an action is about to be executed before checking for interference.

$$\begin{aligned} na(a) &= \{a\} \\ na(\pi_1; \pi_2) &= na(\pi_1) \\ na(\pi_1 \| \pi_2) &= na(\pi_1) \cup na(\pi_2) \\ na(\pi_1 \triangleright \pi_2) &= na(\pi_1) \\ na(e) &= \emptyset \\ na((\!|\Gamma|\!)) &= \emptyset \end{aligned}$$

Definition 2 ($getpre$)**.** *$getpre$ returns the pre-condition of a goal/plan.*

$$\begin{aligned} getpre(\mathsf{g}(C, \pi_0, S, \pi)) &= getpre(\pi) \\ getpre(T(\pi, \langle L, U \rangle)) &= \{c \mid pr(c) \in L\} \\ getpre(\pi) &= getpre(\Sigma(\pi)), \text{ if } \pi \neq T(\pi', R) \end{aligned}$$

Definition 3 ($getres$)**.** *Calculating resource requirements only uses active goals, we ignore goals that are suspended or are executing responses triggered by $!\pi$.*

$$\begin{aligned} getres(\mathsf{g}(C, \pi_0, S, \pi)) &= getres(\pi), \text{ if } S = \textsc{Active} \\ getres(\mathsf{g}(C, \pi_0, S, \pi)) &= \varepsilon, \text{ if } S \neq \textsc{Active} \\ getres(\mathsf{g}_\pi(C, \pi_0, S, \pi)) &= \varepsilon \\ getres(T(\pi, \langle L, U \rangle)) &= \{re(f, t, n) \mid re(f, t, n) \in U\} \\ getres(\pi) &= getres(\Sigma(\pi)), \text{ if } \pi \neq T(\pi', R) \end{aligned}$$

Definition 4 (rok)**.** *In defining $rok(G)$ we need to sum the resource requirements of the set of goals, and then check whether the available resources are sufficient. As discussed by Thangarajah et al. [3], there are actually a number of different cases. Here, for illustrative purposes, we just consider the case where there are sufficient resources to execute the goals freely as being an rok situation. We thus define the collected resource requirements of a goal set $G = \{g_1, \ldots, g_n\}$ as being $getres(G) = U_1 [\!] \ldots [\!] U_n$ where $U_i = getres(g_i)$. Finally, we define $rok(G) \equiv getres(G) \preceq \mathcal{R}$ where $\mathcal{R}$ is the available resources.*

Definition 5 ($culprit$)**.** *In defining $culprit(g)$ one situation to be aware of is where removing a single goal is not enough. In this situation the definition given in the body of the paper will fail to identify any goals to suspend. To cover this case we need a slightly more complex definition. Informally, the previous definition is correct except where there does not exist a single goal that can be removed to fix the resource issue ($\neg\exists g \in G.rok(G \setminus \{g\})$). In this case we consider $culprit(g)$ to be true if removing g and one other goal will fix the problem. This generalises to the situation where one must remove n goals to fix a resource issue:*

$$culprit(g) \equiv \exists n \,.\, (\ (\exists G' \subseteq G. |G'| = n \wedge rok(G \setminus G') \wedge \neg rok(G) \wedge g \in G')$$
$$\wedge\ (\neg \exists G'' \subseteq G. |G''| < n \wedge rok(G \setminus G'') \wedge \neg rok(G)))$$

We now turn to defining the various auxiliary functions that are needed. We assume that requirements definitions, R_i, are *normalised*, i.e. that they contain (a) exactly one $re(f, t, n)$ for each resource type t that is of interest (where n may be 0); and (b) exactly one in, one in_s and one pr. We also assume that resource reusability is consistent, i.e. that a resource type t is not indicated in one place as being consumable and in another as being reusable.

The intended meaning of the auxiliary functions (based on where they are used in the definition of Σ) is as follows: $\oplus$ adds resources without changing the intervals; $\sqcup$ is used to collect the upper bound for a set of plans which are executed sequentially in an unknown order; $\sqcap$ computes the minimal (lower bound) requirements of a set of alternative plans; $\fatsemi$ corresponds to a sequential join of two intervals, and $[\!]$ corresponds to the parallel composition of two intervals. Formally, they are defined as follows:

$$R_1 \oplus R_2 =$$
$$\{re(f, t, n_1 + n_2) \mid re(f, t, n_1) \in R_1 \wedge\ re(f, t, n_2) \in R_2\} \cup$$
$$\{in(c_1 \wedge c_2) \mid in(c_1) \in R_1 \wedge in(c_2) \in R_2\} \cup$$
$$\{in_s(c_1 \wedge c_2) \mid in_s(c_1) \in R_1 \wedge in_s(c_2) \in R_2\} \cup$$
$$\{pr(c_1 \wedge c_2) \mid pr(c_1) \in R_1 \wedge pr(c_2) \in R_2\}$$

$$R_1 \sqcup R_2 =$$
$$\{re(r, t, \max(n_1, n_2)) \mid re(r, t, n_1) \in R_1 \wedge\ re(r, t, n_2) \in R_2\} \cup$$
$$\{re(c, t, n_1 + n_2) \mid re(c, t, n_1) \in R_1 \wedge\ re(c, t, n_2) \in R_2\} \cup$$
$$\{in_s(c_1 \wedge c_2 \wedge c_3 \wedge c_4 \wedge c_5 \wedge c_6) \mid in(c_1) \in R_1$$
$$\wedge\ in(c_2) \in R_2 \wedge in_s(c_3) \in R_1 \wedge in_s(c_4) \in R_2$$
$$\wedge\ pr(c_5) \in R_1 \wedge pr(c_6) \in R_2\}$$

$$R_1 \sqcap R_2 =$$
$$\{re(f, t, \min(n_1, n_2)) \mid re(f, t, n_1) \in R_1 \wedge\ re(f, t, n_2) \in R_2\} \cup$$
$$\{in(c_1 \vee c_2) \mid in(c_1) \in R_1 \wedge in(c_2) \in R_2\} \cup$$
$$\{in_s(c_1 \vee c_2) \mid in_s(c_1) \in R_1 \wedge in_s(c_2) \in R_2\} \cup$$
$$\{pr(c_1 \vee c_2) \mid pr(c_1) \in R_1 \wedge pr(c_2) \in R_2\}$$

$$R_1 \fatsemi R_2 =$$
$$\{re(r, t, \max(n_1, n_2)) \mid re(r, t, n_1) \in R_1 \wedge\ re(r, t, n_2) \in R_2\} \cup$$
$$\{re(c, t, n_1 + n_2) \mid re(c, t, n_1) \in R_1 \wedge\ re(c, t, n_2) \in R_2\} \cup$$
$$\{in_s(c_1 \wedge c_2 \wedge c_3 \wedge c_4 \wedge c_5) \mid in(c_1) \in R_1 \wedge\ in(c_2) \in R_2\ \wedge in_s(c_3) \in R_1$$
$$\wedge\ in_s(c_4) \in R_2 \wedge pr(c_5) \in R_2\}\ \cup \{pr(c) \mid pr(c) \in R_1\}$$

$$R_1 [\!] R_2 =$$
$$\{re(f, t, n_1 + n_2) \mid re(f, t, n_1) \in R_1 \wedge\ re(f, t, n_2) \in R_2\} \cup$$
$$\{in_s(c_1 \wedge c_2 \wedge c_3 \wedge c_4 \wedge c_5 \wedge c_6) \mid in(c_1) \in R_1$$
$$\wedge\ in(c_2) \in R_2 \wedge in_s(c_3) \in R_1 \wedge\ in_s(c_4) \in R_2 \wedge pr(c_5) \in R_1 \wedge pr(c_6) \in R_2\}$$

A Distributed Treatment of Exceptions in Multiagent Contracts (Preliminary Report)

Özgür Kafalı and Pınar Yolum

Department of Computer Engineering
Boğaziçi University
34342, Bebek, İstanbul, Turkey
ozgurkafali@gmail.com, pinar.yolum@boun.edu.tr

Abstract. Commitments are key to contract-based multiagent systems. When agents enter a contract, they project the outcome of the contract based on its content as well as their past experiences and the current world state. We model an agent's projections as individual world states, e.g., a satisfactory state. If the contract is indeed executed to satisfy these projections, then the agent is said to complete the contract successfully. If not, we expect the agent to take proper action. This paper formalizes the notion of projection and its relations, to contract execution. Accordingly, we propose a *satisfiability* relation to check if an agent's state complies with its projections. We then relate satisfiability to the occurrence of exceptions.

1 Introduction

Business interactions among parties are governed by contracts. Contracts are (desirably) unambiguous specifications that precisely define business rules among interacting parties under different circumstances. Commitments [13, 18, 6, 16, 17] are proven to be a significant element for modeling contracts among agents. When agents enter contracts, they can make informed projections about the future. If interactions do not evolve as projected, agents can go back to their commitments to examine what has gone wrong and in principle find ways to correct the execution [2, 9].

We are interested in a distributed multiagent system, where each agent has a local view of the environment. Each agent represents this local view through a state that captures the agent's perception of its environment through commitments and propositions. Similar to these local views, each agent has a projected state based on the commitments it has as well as its local view of the world. These projected states represent an agent's expectations from the future. Given these representations, an immediate question is whether it is possible to ever end up in these projected states. This question is important because a negative answer to this question implies that the world cannot evolve as the agent projects as it will, and hence an exception on the agent's side will take place. Example 1 demonstrates a common delivery scenario from e-commerce.

Example 1 Two agents, a customer and a merchant, have the following conditional commitment; if the customer pays for an item, then the merchant will deliver the item within three business days. Assume that the customer pays on Monday, so the merchant is committed to deliver by Thursday.

Following this example, a typical understanding of an exception is that if by Thursday the item is not delivered to the customer, an exception occurs. This typical interpretation of an exception corresponds to contract violation. While this is certainly important, an exception is not always a synonym for violation. Hence, there could be other cases where the contract is not violated but an exception occurs and vice versa. Let us illustrate these points following Example 1.

- Assume that there is a bad snow storm. The customer predicts that the roads will be closed due to the storm and that the delivery will be late. Hence, even though it is Friday (and the contract is violated), the customer does not signal an exception and does not take any actions.
- Assume that the customer is well aware that the merchant always delivers one or two days earlier than promised. Hence, even though it is only Wednesday (and the contract is not violated), the customer signals an exception and takes an action to handle the exception, e.g., contacts the merchant.

Hence, it is important to capture agents' projections about the outcome of contracts and to compare the reality to these projections. This paper develops a principled approach for verifying whether the commitments the agent is currently involved in satisfy the agent's projections about its future. In order to relate agents' states with their projections, we propose a *satisfiability* relation. This relation compares states by comparing its commitment and proposition elements pairwise. Consider the first case described in the above examples: since there is a snow storm, and the customer learns about this, she makes a projection (let us say on Thursday) that her contract for delivery will be violated on Friday. When it is Friday, the customer will compare her state with her projection. Although the contract is violated, her state will still satisfy her projection, because, she was not expecting delivery anyway. Hence, even though there is a contract violation, the customer will not immediately take action. A similar reasoning applies for the second case. We show how this and other cases are handled in practice to check whether the agent's state is in compliance with its projections.

The rest of the paper is organized as follows: Section 2 describes our formal model. Section 3 describes the *satisfiability* relation that is used to compare agent states. Section 4 introduces *exceptions* in terms of satisfiability. Finally, Section 5 concludes the paper.

2 Formal Model

Our formal model is based on the description of the world through states and the evolution of the states through agents' commitments.

2.1 Describing the World

We propose the multiagent system to progress as a set of states that describe the world in different time points. Each agent views a part of the world since the execution of the multiagent system is distributed among the agents. First, we define a time point.

Definition 1 A time point T is a discrete measure of time, which can be used to totally order time; $t_2 < t_3$, $t_5 < t_{12}$, etc. ▮

Throughout the paper, t_1, t_2, ..., t_n are used to denote time points. Next, we describe the agents' world.

Definition 2 The real world is described by a global state $\mathcal{S}_G^T = \langle \Phi_G, \mathcal{C}_G \rangle$ that consists of atomic propositions (Φ_G) and commitments ($\mathcal{C}_G$) that hold at a time point. This global state demonstrates a global view of the multiagent system. ▮

The global state is not meant to be known or processed by any of the entities in the multiagent system. Instead, each agent has a local state (i.e., its local world model). In a distributed execution, agents perceive the real world from different view-points. Thus, their states may differ from each other based on their observations.

Definition 3 A state $\mathcal{S}_A^T = \langle \Phi, \mathcal{C} \rangle$ for agent A at time T is a subset of the real world ($\mathcal{S}_G^T$), which represents A's view at T, where

- Φ is a finite set of atomic propositions that hold at T,
- $\mathcal{C}$ is a finite set of commitments that exist at T.

▮

Definition 3 describes the model of the agent's world. It consists of propositions that are known to be true at that time point and commitments that are represented via their states. Propositions tell what has happened in the system so far (i.e., facts) to the agent's perception. Commitments, on the other hand, provide both facts and expectations about the agent's future states[1]. Table 1(a) shows the state of the $customer$ agent from Example 1 at time t_2, which includes one proposition.

Definition 4 A satisfactory state $\mathcal{ES}_A^{T_i,T_j} = \langle \Phi_\mathcal{E}, \mathcal{C}_\mathcal{E} \rangle$ represents a projection of A for the world at T_j projected as of time T_i, where

- $\Phi_\mathcal{E}$ is a finite set of atomic propositions that A projects at T_i to hold at T_j,
- $\mathcal{C}_\mathcal{E}$ is a finite set of commitments that A projects at T_i to exist at T_j,
- $T_i < T_j$.

▮

Definition 4 describes a future projection of the agent. It is a representation of what the agent will consider satisfactory when time has evolved to that point. Similar to the real world model, it includes propositions and commitments. The propositions represent the facts that are assumed to hold at that time point. The commitments represent the states of the future contracts that the agent is assumed to be involved in. Recall that the agent always makes the projection at an earlier time point ($T_i < T_j$). Table 1(b) shows a satisfactory state of the $customer$ agent for time t_3 which includes one proposition. Notice the projection is made at time t_2.

A rational agent tries to perform actions that will enable the satisfactory state to be reached (e.g., via a plan). However, note that a projected state may not be realized solely by the agent itself since it may contain propositions or commitments that can only be realized by others.

[1] Commitments will be discussed in detail in Section 2.2.

(a) $\mathcal{S}^{t_2}_{customer}$ = $\langle\{paid\}, \{\}\rangle$
(b) $\mathcal{ES}^{t_2,t_3}_{customer}$ = $\langle\{delivered\}, \{\}\rangle$

Table 1. Examples of states

2.2 Formalizing the Interactions

We model the agents' interactions via commitments [13, 18, 6]. A commitment represents a contract from a debtor agent towards a creditor agent about a specific property. Definition 5 defines a commitment formally. Below, A_i and A_j denote agents; Ant and Con are propositions.

Definition 5 A commitment $C^{St}_{A_i,A_j}(Ant, Con)$ denotes the commitment between the agents A_i and A_j, with St being its state[2]. This is a conditional commitment; if the antecedent Ant is satisfied, then the debtor A_i becomes committed to the creditor A_j for satisfying the consequent Con. If Ant is $True$ (denoted $\top$), then this is a base-level commitment; A_i is committed to A_j for satisfying Con unconditionally. ∎

We follow the idea and notation of [14, 5, 3] to represent commitments (i.e., every commitment is conditional). A base-level commitment is simply a commitment with its condition being true. We also separate the agents from the commitment properties (i.e., the antecedent and the consequent). This way, we can omit the agents from the commitment description whenever they are not significant. We have the following grammar for the commitment properties:

- $Ant, Con \rightarrow P$.
- $P \rightarrow \phi \mid P \wedge P$.

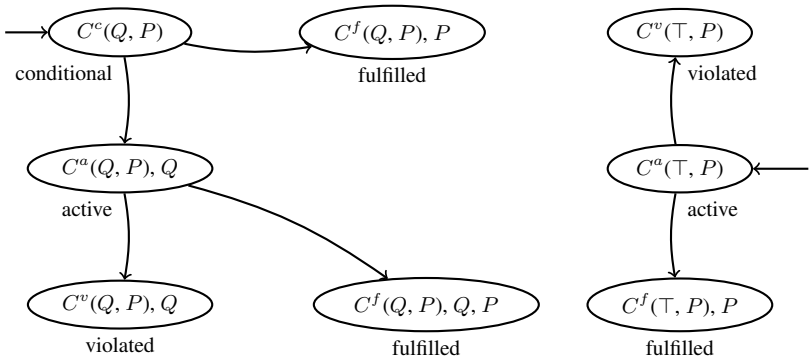
Above, ϕ is an atomic proposition. Currently, we do not support negation or nested commitments (e.g., commitments for the antecedent or the consequent).

Commitments are live objects; we always consider a commitment with its state [18, 6]. Next, we describe each commitment state with respect to the corresponding world states and the transitions in between[3].

Conditional: When the commitment $C^{St}(Q, P) \in \mathcal{C}$ for state $\mathcal{S}^T = \langle\Phi, \mathcal{C}\rangle$ is in conditional commitment state, denoted $C^c(Q, P)$, then $Q, P \notin \Phi$. In other words, the commitment $C^c(Q, P)$ cannot coexist in the same state with its antecedent or consequent:

- If the commitment's antecedent already holds, then the commitment is no longer *conditional* (i.e., its condition is satisfied), and it will become *active*.

[2] Note that the commitment's state differs from the agent's state as described in Definition 3.
[3] We omit the agents both from the commitments and from the states.



(a) Conditional (b) Base-level

Fig. 1. Commitment states

– If the commitment's consequent already holds, then the commitment is no longer *conditional*, and it will become *fulfilled*. Afterwards, it is not significant whether the antecedent is also satisfied or not.

Active: When the commitment $C^{St}(\top, P) \in \mathcal{C}$ for state $\mathcal{S}^T = \langle \Phi, \mathcal{C} \rangle$ is in active commitment state, denoted $C^a(\top, P)$, then $P \notin \Phi$. In other words, the commitment $C^a(\top, P)$ cannot coexist in the same state with its consequent:

– If the commitment's consequent already holds, then the commitment is no longer *active*, and it will become *fulfilled*. Once the consequent is satisfied, then the commitment's life-cycle ends.

Additionally, a commitment can become active from its conditional state. $C^{St}(\top, P) \in \mathcal{C}_T$ for state $\mathcal{S}^T = \langle \Phi_T, \mathcal{C}_T \rangle$ is active, when $C^c(Q, P) \in \mathcal{C}_{T-1}$ for state $\mathcal{S}^{T-1} = \langle \Phi_{T-1}, \mathcal{C}_{T-1} \rangle$ and $Q \in \Phi_T$. That is, if the state that includes the commitment $C^c(Q, P)$ makes a transition to a state where the commitment's antecedent holds, then the commitment will become *active*.

Fulfilled (conditional): The commitment $C^{St}(Q, P) \in \mathcal{C}_T$ for state $\mathcal{S}^T = \langle \Phi_T, \mathcal{C}_T \rangle$ is in fulfilled commitment state, denoted $C^f(Q, P)$, when $C^c(Q, P) \in \mathcal{C}_{T-1}$ for state $\mathcal{S}^{T-1} = \langle \Phi_{T-1}, \mathcal{C}_{T-1} \rangle$ and $P \in \Phi_T$. That is, if the state that includes the commitment $C^c(Q, P)$ makes a transition to a state where the commitment's consequent holds, then the commitment will become *fulfilled*. The conditional commitment's life-cycle ends with this state.

Fulfilled (base-level): The commitment $C^{St}(\top, P) \in \mathcal{C}_T$ for state $\mathcal{S}^T = \langle \Phi_T, \mathcal{C}_T \rangle$ is in fulfilled commitment state, denoted $C^f(\top, P)$, when $C^a(\top, P) \in \mathcal{C}_{T-1}$ for state $\mathcal{S}^{T-1} = \langle \Phi_{T-1}, \mathcal{C}_{T-1} \rangle$ and $P \in \Phi_T$. That is, if the state that includes the commitment $C^c(\top, P)$ makes a transition to a state where the commitment's consequent holds, then

the commitment will be in the fulfilled state. The base-level commitment's life-cycle ends with this state.

Violated: When the commitment $C^{St}(\top, P) \in \mathcal{C}_T$ for state $\mathcal{S}^T = \langle \Phi_T, \mathcal{C}_T \rangle$ is in violated commitment state, denoted $C^v(\top, P)$, then $C^a(\top, P) \in \mathcal{C}_{T_i}$ for a state $\mathcal{S}^{T_i} = \langle \Phi_{T_i}, \mathcal{C}_{T_i} \rangle$, where $T_i < T$ and $P \notin \Phi_T$. That is, a violated commitment should be active in a previous state. Again, the base-level commitment's life-cycle ends with this state.

Figure 1 summarizes the commitment states; 1(a) for conditional commitments and 1(b) for base-level commitments.

3 Satisfiability

We capture the projections of an agent through satisfactory states. On one hand, a satisfactory state is similar to a temporal achievement goal [12], where the agent plans to reach some properties or be involved in some commitments at a certain time point. On the other hand, satisfactory states do not necessarily represent goals. They also model the agent's expectations about the future. That is, the agent may assume certain properties to hold in the future even though it does not wish so, e.g., the customer may expect a late delivery if she is informed of a traffic jam covering her territory.

It is important that an agent can compare its actual and satisfactory states. For that purpose, we propose the satisfiability relation. It compares two states and tells if one satisfies the other. This is a strong relation in the sense that once a state satisfies another, then the two states are either equal or the former can replace the latter. This captures our intuition that if the current state of the world is equivalent or better than the projected world state, the execution is in order and no action needs to be taken. However, if the comparison yields that the current state does not satisfy a projected state, then there is an exception and it should be handled by the agent.

First, we make some preliminary definitions.

Definition 6 A term $\mathcal{T}$ is either an atomic proposition ϕ or a commitment C. ∎

Definition 7 A formula $\mathcal{F}$ is a conjunction of atomic propositions $\phi_1 \wedge \phi_2 \wedge ... \wedge \phi_n$. ∎

Satisfiability, denoted by $X \Vdash Y$, is read as "Y is satisfiable by X". Note that Y here represents the minimum satisfactory condition. If X includes more than the necessary propositions or more beneficial commitments than Y, then Y is again satisfiable. We try to explain this in detail throughout the section via several axioms. We begin by describing the preliminary axioms that are necessary to describe *satisfiability* over states.

Axiom 1 Proposition ϕ is satisfiable by formula $\mathcal{F}$, denoted $\mathcal{F} \Vdash \phi$, iff $\mathcal{F} \models \phi$. ∎

Axiom 1 follows directly from logical entailment. Figure 2 summarizes the satisfiability relation with two propositions; *paid* and *delivered*, together with their combination, as well as the base-level and conditional commitments that include them[4]. We will refer to the figure while describing the axioms, whenever necessary. For example, $paid \wedge delivered \Vdash paid$ is a trivial result of Axiom 1.

[4] The agents are omitted from the commitments to simplify the demonstration. Arrows show the direction of satisfiability.

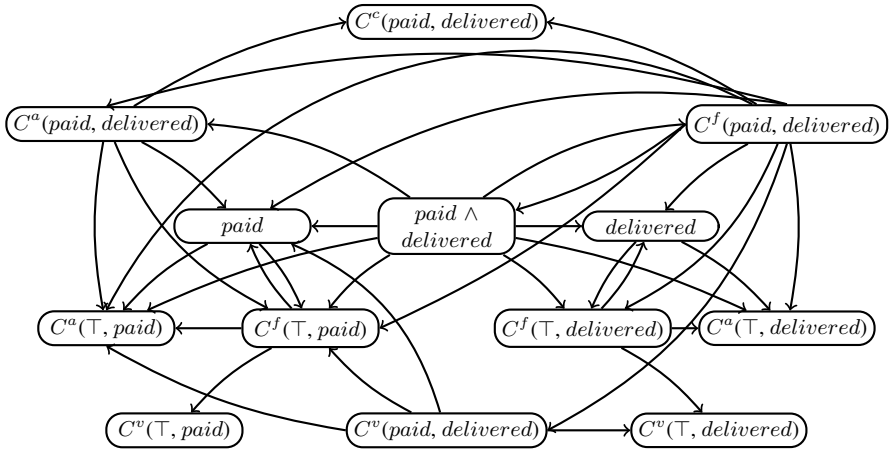


Fig. 2. Satisfiability network

Axiom 2 Base-level commitment $\mathcal{C}^S_{A_i,A_j}(\top, Con)$ is satisfiable by formula $\mathcal{F}$, denoted $\mathcal{F} \Vdash \mathcal{C}^S_{A_i,A_j}(\top, Con)$, iff $\mathcal{F} \models Con$ and $S \in \{fulfilled, active, violated\}$. ∎

When commitments are involved, we are no longer limited with logic entailment. According to Axiom 2, if the formula entails the commitment's consequent, then the commitment is satisfiable whatever its state is:

- A formula entailing a fulfilled commitment's consequent is equivalent to the commitment. For example, $delivered \Vdash C^f(\top, delivered)$. Assume that the customer has a projection that her commitment towards delivery will be fulfilled. If the customer perceives that the delivery is performed, then her projection is satisfied.
- A formula entailing an active commitment's consequent is more beneficial than the commitment itself. For example, $delivered \Vdash C^a(\top, delivered)$. Assume that the customer now has a projection that her commitment will be active. However, the customer perceives that the delivery is performed. This is indeed a better situation for the customer.
- A formula entailing a violated commitment's consequent is more beneficial than the commitment itself. Similar to above, $delivered \Vdash C^v(\top, delivered)$. Now, if the customer expects that her commitment will be violated, then she will be satisfied when she actually sees that delivery is performed.

Axiom 3 Proposition ϕ is satisfiable by base-level commitment $\mathcal{C}^S_{A_i,A_j}(\top, Con)$, denoted $\mathcal{C}^S_{A_i,A_j}(\top, Con) \Vdash \phi$, iff $Con \models \phi$ and $S \in \{fulfilled\}$. ∎

Similar to Axiom 2, Axiom 3 states that if the commitment's consequent entails the formula, then the proposition is only satisfiable if the commitment is fulfilled. For example, $C^f(\top, delivered) \Vdash delivered$. A fulfilled commitment towards delivery means delivery has occurred. Indeed, this is equivalent to the proposition $delivered$.

Axiom 4 Base-level commitment $\mathcal{C}^{S_2}_{A_k,A_l}$ ($\top$, Con_2) is satisfiable by base-level commitment $\mathcal{C}^{S_1}_{A_i,A_j}$ ($\top$, Con_1), denoted $\mathcal{C}^{S_1}_{A_i,A_j}$ ($\top$, Con_1) $\Vdash\!\!\vdash \mathcal{C}^{S_2}_{A_k,A_l}$ ($\top$, Con_2), iff

- $Con_1 \models Con_2$, $S_1 \in \{fulfilled\}$, and $S_2 \in \{fulfilled\}$, or
- $Con_1 \models Con_2$, $S_1 \in \{fulfilled, active\}$, and $S_2 \in \{active\}$, or
- $Con_1 \models Con_2$, $S_1 \in \{fulfilled, violated\}$, and $S_2 \in \{violated\}$.

∎

Axiom 4 follows from Axioms 2 and 3. Let us review each case:

- A projection of a fulfilled commitment can only be satisfied by another fulfilled commitment. If in reality, the commitment is active or violated, this will not be good enough.
- A projection of an active commitment can be satisfied by another active or fulfilled commitment. For example, $C^f(\top, delivered) \Vdash\!\!\vdash C^a(\top, delivered)$.
- A projection of a violated commitment can be satisfied by another violated or fulfilled commitment. For example, $C^f(\top, delivered) \Vdash\!\!\vdash C^v(\top, delivered)$. That is, if the expectation was that the commitment would have been violated, nothing worse can happen. Hence, any commitment state is as good as the violated state.

Axiom 5 Proposition ϕ is satisfiable by conditional commitment $\mathcal{C}^{S}_{A_i,A_j}$ (Ant, Con), denoted $\mathcal{C}^{S}_{A_i,A_j}$ (Ant, Con) $\Vdash\!\!\vdash \phi$, iff $Ant \models \phi$ and $S \in \{fulfilled, active, violated\}$.
∎

Axiom 6 Proposition ϕ is satisfiable by conditional commitment $\mathcal{C}^{S}_{A_i,A_j}$ (Ant, Con), denoted $\mathcal{C}^{S}_{A_i,A_j}$ (Ant, Con) $\Vdash\!\!\vdash \phi$, iff $Con \models \phi$ and $S \in \{fulfilled\}$. ∎

Axiom 5 tells that a once activated conditional commitment (active, fulfilled or violated) satisfies its antecedent. The motivation for this is that if the commitment is activated at one point, then its antecedent must have been true. Even if the consequent never becomes true (i.e., the commitment is violated), the commitment would still satisfy the antecedent; hence for example, $C^v(paid, delivered) \Vdash\!\!\vdash paid$. Similarly, Axiom 6 tells that a fulfilled conditional commitment satisfies its consequent. A similar example would be $C^f(paid, delivered) \Vdash\!\!\vdash delivered$.

Axiom 7 Base-level commitment $\mathcal{C}^{S_2}_{A_k,A_l}$ ($\top$, Con_2) is satisfiable by conditional commitment $\mathcal{C}^{S_1}_{A_i,A_j}$ (Ant, Con_1), denoted $\mathcal{C}^{S_1}_{A_i,A_j}$ (Ant_1, Con_1) $\Vdash\!\!\vdash \mathcal{C}^{S_2}_{A_k,A_l}$ ($\top$, Con_2), iff

- $Con_1 \models Con_2$, $S_1 \in \{fulfilled\}$, and $S_2 \in \{fulfilled\}$, or
- $Con_1 \models Con_2$, $S_1 \in \{fulfilled, active\}$, and $S_2 \in \{active\}$, or
- $Con_1 \models Con_2$, $S_1 \in \{fulfilled, violated\}$, and $S_2 \in \{violated\}$.

∎

Axiom 8 Conditional commitment $\mathcal{C}^{S_2}_{A_k,A_l}$ (Ant_2, Con_2) is satisfiable by conditional commitment $\mathcal{C}^{S_1}_{A_i,A_j}$ (Ant_1, Con_1), denoted $\mathcal{C}^{S_1}_{A_i,A_j}$ (Ant_1, Con_1) $\Vdash\!\vdash \mathcal{C}^{S_2}_{A_k,A_l}$ (Ant_2, Con_2), iff

- $Ant_1 \models Ant_2, Con_1 \models Con_2, S_1 \in \{fulfilled\}$, and $S_2 \in \{fulfilled\}$, or
- $Ant_1 \models Ant_2, Con_1 \models Con_2, S_1 \in \{fulfilled, active\}$, and $S_2 \in \{active\}$, or
- $Ant_1 \models Ant_2, Con_1 \models Con_2, S_1 \in \{fulfilled, violated\}$, and $S_2 \in \{violated\}$, or
- $Ant_1 \models Ant_2, Con_1 \models Con_2, S_1 \in \{fulfilled, active, conditional\}$, and $S_2 \in \{conditional\}$.

∎

Axioms 7 and 8 apply a similar reasoning to Axiom 4 for conditional commitments. For example, $C^f(paid, delivered) \Vdash\!\vdash C^f(\top, delivered)$. Here both commitments are fulfilled, and the conditional commitment entails $paid$ as well as $delivered$. Moreover, $C^f(paid, delivered) \Vdash\!\vdash C^a(paid, delivered)$. Here, the fulfilled commitment entails both $paid$ and $delivered$, whereas the active commitment only entails $paid$. Next, we give some important remarks regarding the axioms above.

Remark 1. $\alpha \Vdash\!\vdash \alpha$ (reflexive).

Remark 2. It is not necessarily true that $\beta \Vdash\!\vdash \alpha$ or $\beta \nVdash \alpha$ if $\alpha \Vdash\!\vdash \beta$ (not symmetric).

It is trivial that $\Vdash\!\vdash$ is reflexive and not symmetric.

Remark 3. $\alpha \Vdash\!\vdash \gamma$ if $\alpha \Vdash\!\vdash \beta$ and $\beta \Vdash\!\vdash \gamma$ (transitive).

Proof. Assume α is a formula, β is an atomic proposition, and γ is a base-level commitment. Other combinations follow similarly. Now, α entails β since $\alpha \Vdash\!\vdash \beta$ and β entails the consequent of γ since $\beta \Vdash\!\vdash \gamma$. Using the fact that logic entailment is transitive, α entails the consequent of γ. Thus, $\alpha \Vdash\!\vdash \gamma$.

Now, we are ready to describe how a state is satisfiable by another state. We do this transitively though a formula. That is, first we describe how a formula is satisfiable by a state, then we describe state satisfiability via the formula.

Axiom 9 Formula $\mathcal{F} = \phi_1 \wedge \phi_2 \wedge \ldots \wedge \phi_n$ is satisfiable by state $\mathcal{S}^T_A$, denoted $\mathcal{S}^T_A \Vdash\!\vdash \mathcal{F}$, iff $\forall\ \phi_i\ \exists\ \mathcal{F}' = \mathcal{T}_1 \wedge \mathcal{T}_2 \wedge \ldots \wedge \mathcal{T}_n$: $\mathcal{T}_i \in \mathcal{S}^T_A$ and $\mathcal{F}' \Vdash\!\vdash \phi_i$. ∎

Axiom 9 tells that a formula is satisfiable by a state if every proposition in the formula is satisfiable by a conjunction of terms that can be constructed from the state.

Axiom 10 Term $\mathcal{T}$ is satisfiable by state $\mathcal{S}^T_A$, denoted $\mathcal{S}^{T_k}_{A_i} \Vdash\!\vdash \mathcal{T}$, iff $\exists\ \mathcal{F}$: $\mathcal{S}^T_A \Vdash\!\vdash \mathcal{F}$ and $\mathcal{F} \Vdash\!\vdash \mathcal{T}$. ∎

Note that a term is satisfiable by a state via a formula according to Axiom 9.

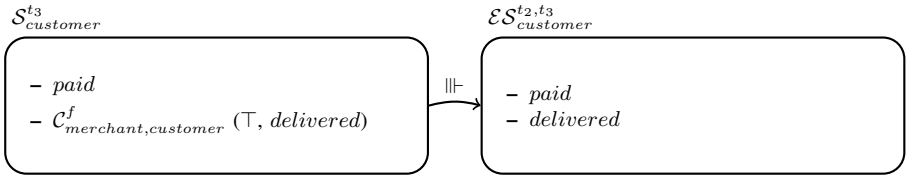


Fig. 3. State satisfiability

Definition 8 State $\mathcal{S}^{T_l}_{A_j}$ is satisfiable by state $\mathcal{S}^{T_k}_{A_i}$, denoted $\mathcal{S}^{T_k}_{A_i} \Vdash \mathcal{S}^{T_l}_{A_j}$, iff $\forall\, \mathcal{T} \in \mathcal{S}^{T_l}_{A_j}$: $\mathcal{S}^{T_k}_{A_i} \Vdash \mathcal{T}$. ∎

According to Definition 8, a state is satisfiable by another state if every term in the former state is satisfiable by the latter[5]. Figure 3 shows an example; the term $paid$ in state $\mathcal{ES}^{t_2,t_3}_{customer}$ is satisfiable by the formula $\mathcal{F}_1 = paid$, which is in turn satisfiable by state $\mathcal{S}^{t_3}_{customer}$. Similarly, the term $delivered$ in state $\mathcal{ES}^{t_2,t_3}_{customer}$ is satisfiable by the formula $\mathcal{F}_2 = delivered$ which is satisfiable by state $\mathcal{S}^{t_3}_{customer}$.

4 Exceptions

Definition 9 describes an exception based on the satisfiability relation. If the agent's satisfactory state is not satisfiable by its corresponding state (i.e., for the same time point), then there is an exception. Note that this definition of an exception is subjective in the sense that one agent may identify a particular situation as an exception while another agent might not.

Definition 9 An exception occurs for agent A_i iff $\exists$ satisfactory state $\mathcal{ES}^{T_i,T_j}_{A_i}$: $\mathcal{S}^{T_j}_{A_i} \nVdash \mathcal{ES}^{T_i,T_j}_{A_i}$. ∎

Now, let us review several cases to demonstrate the usage of the satisfiability relation for detecting exceptions.

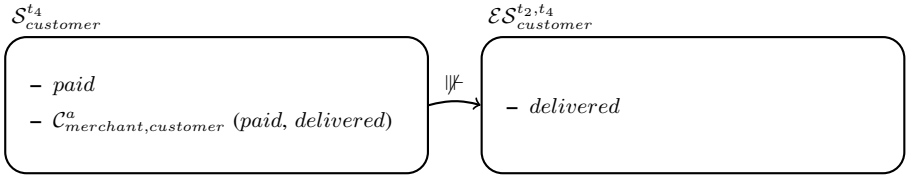


Fig. 4. Exception: merchant fails to deliver

Exception for customer: Figure 4 demonstrates a simple case. The left box shows the customer's state at time t_4, while the right box shows her satisfactory state for the same

[5] Note that the states in Definition 8 can also be satisfactory states.

time point. Here, the merchant currently has an active commitment to the customer for delivery. However, the customer expects delivery to be completed. Recall that for a state to be satisfiable, all its terms should be satisfiable (Definition 8). There is only one term in state $\mathcal{ES}^{t_2,t_4}_{customer}$, which is the proposition *delivered*. However, none of the Axioms for satisfiability (Section 3) can be applied to satisfy *delivered* from the terms in state $\mathcal{S}^{t_4}_{customer}$. Thus, the customer's satisfactory state is not satisfiable. This causes an exception for the customer according to Definition 9.

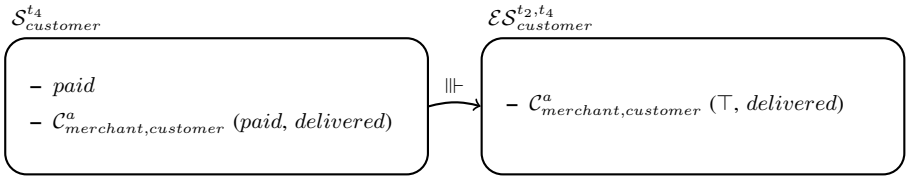


Fig. 5. No exception: customer does not expect delivery!

No exception for customer: Figure 5 demonstrates a slightly different case. The customer's state is the same as before. However, this time, she does not expect delivery to be fulfilled at time t_4. According to Axiom 7, $\mathcal{C}^a_{merchant,customer}$ ($\top$, $delivered$) is satisfiable by $\mathcal{C}^a_{merchant,customer}$ ($paid$, $delivered$). Thus, no exception occurs for the customer. Note that, we would normally signal an exception when a commitment is still active when it should be fulfilled. However, by letting the satisfactory state to be constructed according to the agent's projections, an insignificant exception is avoided (e.g., the customer is already aware that the delivery will be delayed).

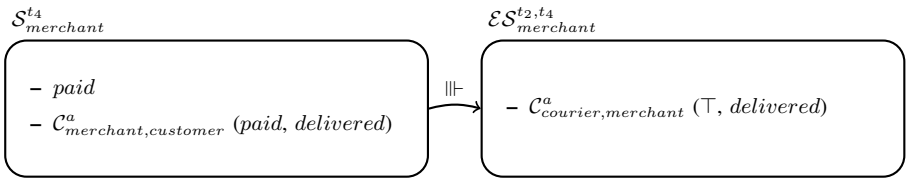


Fig. 6. Customer and merchant's projections do not match

Exception for customer, no exception for merchant: Consider Figures 4 and 6 together. There is an exception for the customer as she expects delivery at time t_4. However, the merchant's projection for t_4 is that he has delegated its commitment to the courier, and that commitment is still active. His satisfactory state is satisfiable by his actual state. Thus, there is no exception for the merchant. Note that, in a centralized environment, this would never happen. That is, a central monitor either signals an exception or not. Here, on the other hand, we allow autonomous agents to have their own

projections about the future. Accordingly, an exception for one agent might just be an expected situation for another.

5 Discussion

Commitments and expectations are two well-structured ways of modeling multiagent interactions [16, 1]. In this work, we use commitments since we take a state-oriented perspective and try to capture the relations among agents' states. Moreover, we capture an agent's projections about the future using satisfactory states. This is somewhat different from the concept of an expectation as described in the literature. We do not explicitly provide expected events. Rather, we model propositions and commitments that are projected by the agent to hold at certain time points in the future. We are not concerned with how those projections are formed. However, we use them to verify smooth execution.

A variety of commitment types and relations among them have been proposed in the literature [10, 9], as well as obligations and prohibitions that can also be related with social commitments [7]. Here, we consider contract-based multiagent systems. Thus, commitments that refer to achievement type goals or projections are more suitable for our discussion. Indeed, commitments with achievement type properties can be modeled with atomic or conjunctive propositions, while negated propositions can be used to model maintenance type properties [8], e.g., a car rental company committing to a customer for the car not breaking down during the rental period. We leave out maintenance type (e.g., negated) commitment properties for this work. We plan to investigate the relations among them in the future.

Previously, there has been work in the literature on comparing states. One similar work is that of Mallya and Singh [11], which focuses on similarity relations for protocol runs. In parallel with our work, a state is described as a set of propositions and commitments, and several similarity relations are given to compare states. The idea is to compare protocol runs in terms of states. One major difference between their similarity relations and our *satisfiability* relation is that their relations are equivalence relations (e.g., supports symmetry) while ours is not. This is mainly related to the motivation behind those relations. That is, they aim to merge smaller protocols into larger ones by comparing protocol states. We, on the other hand, provide a one-way relation (e.g., not necessarily symmetric) to compare states. Because, our main motivation is to ensure that an agent's state will comply with its projections, the inverse direction is not significant at all (e.g., whether expectations support actual states). Moreover, it is not necessary to ensure that both states are identical.

Commitments account for the constitutive specification of a protocol [4, 15]. They provide flexible execution for the agents as long as their commitments are satisfied. When an exception occurs, there may be several reasons behind it. One such reason for exceptions in constitutively regulated protocols is the misalignment of agents' commitments. That is, the debtor and the creditor have different understandings for the same commitment [5, 9]. The state satisfiability relation we have proposed helps detect such exceptions.

In this paper, we have proposed a satisfiability relation that can be used to compare agents' states. When used to compare an agent's state with its satisfactory state, the outcome tells whether there is an exception for the agent or not. That is, if the agent's satisfactory state is not satisfiable by its current state, then the agent signals an exception. In addition, the agent may verify its compliance to the protocol it is executing by consistently comparing its current state to its satisfactory states. This way, the agent can identify at which point of the protocol, there has been a problem. Consider the merchant in the delivery example. If he expects to receive the payment before he actually delivers the item, then he will project payment in an earlier satisfactory state than delivery. Now, if the latter is satisfied, but the former is not, he can conclude that there is a problem with the customer's side of the contract.

Apart from satisfiability, another relation among the agent's states can be *reachability*, which is used to verify whether a state is reachable by another state. This relation can be utilized in yet another phase of exception handling; *prediction*. In a smooth execution, an agent should predict its current probability of facing an exception in the future. The investigation of reachability relation is left for future work. We plan to consider different levels of reachability, and connect them with satisfiability.

Acknowledgement

This research is supported by Boğaziçi University Research Fund under grant BAP5694, and the Turkish State Planning Organization (DPT) under the TAM Project, number 2007K120610.

References

1. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Verifiable agent interaction in abductive logic programming: The SCIFF framework. ACM Transactions on Computational Logic 9(4), 1–43 (2008)
2. Chesani, F., Mello, P., Montali, M., Torroni, P.: Commitment tracking via the reactive event calculus. In: IJCAI '09: Proceedings of the 21st International Joint Conference on Artifical Intelligence. pp. 91–96 (2009)
3. Chopra, A.K., Dalpiaz, F., Giorgini, P., Mylopoulos, J.: Reasoning about agents and protocols via goals and commitments. In: AAMAS '10: Proceedings of The 9th International Conference on Autonomous Agents and Multiagent Systems. pp. 457–464 (2010)
4. Chopra, A.K., Singh, M.P.: Constitutive interoperability. In: AAMAS '08: Proceedings of the 7th International Conference on Autonomous Agents and Multiagent Systems. pp. 797–804 (2008)
5. Chopra, A.K., Singh, M.P.: Multiagent commitment alignment. In: AAMAS '09: Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems. pp. 937–944 (2009)
6. Fornara, N., Colombetti, M.: Defining interaction protocols using a commitment-based agent communication language. In: AAMAS '03: Proceedings of the 2nd International Conference on Autonomous agents and multiagent systems. pp. 520–527 (2003)
7. Fornara, N., Colombetti, M.: Ontology and time evolution of obligations and prohibitions using semantic web technology. In: 7th International Workshop on Declarative Agent Languages and Technologies (DALT). pp. 101–118 (2009)

8. Hindriks, K.V., van Riemsdijk, M.B.: Satisfying maintenance goals. In: 6th International Workshop on Declarative Agent Languages and Technologies (DALT). pp. 86–103 (2008)
9. Kafalı, Ö., Chesani, F., Torroni, P.: What happened to my commitment? Exception diagnosis among misalignment and misbehavior. In: Computational Logic in Multi-Agent Systems. Lecture Notes in Computer Science, vol. 6245, pp. 82–98 (2010)
10. Letia, I.A., Groza, A.: Agreeing on defeasible commitments. In: 4th International Workshop on Declarative Agent Languages and Technologies (DALT). pp. 156–173 (2006)
11. Mallya, A.U., Singh, M.P.: An algebra for commitment protocols. Autonomous Agents and Multi-Agent Systems 14(2), 143–163 (2007)
12. van Riemsdijk, M.B., Dastani, M., Winikoff, M.: Goals in agent systems: a unifying framework. In: AAMAS '08: Proceedings of the 7th International Conference on Autonomous Agents and Multiagent Systems. pp. 713–720 (2008)
13. Singh, M.P.: An ontology for commitments in multiagent systems: Toward a unification of normative concepts. Artificial Intelligence and Law 7, 97–113 (1999)
14. Singh, M.P.: Semantical considerations on dialectical and practical commitments. In: AAAI'08: Proceedings of the 23rd National Conference on Artificial Intelligence. pp. 176–181. AAAI Press (2008)
15. Singh, M.P., Chopra, A.K.: Correctness properties for multiagent systems. In: 7th International Workshop on Declarative Agent Languages and Technologies (DALT). pp. 192–207 (2009)
16. Torroni, P., Yolum, P., Singh, M.P., Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P.: Modelling interactions via commitments and expectations. In: Handbook of Research on Multi-Agent Systems: Semantics and Dynamics of Organizational Models. pp. 263–284 (2009)
17. Torroni, P., Chesani, F., Mello, P., Montali, M.: Social commitments in time: Satisfied or compensated. In: DALT. Lecture Notes in Computer Science, vol. 5948, pp. 228–243. Springer (2009)
18. Yolum, P., Singh, M.P.: Flexible protocol specification and execution: applying event calculus planning using commitments. In: AAMAS '02: Proceedings of the 1st International Conference on Autonomous Agents and Multiagent Systems. pp. 527–534 (2002)

An Operational Semantics for AgentSpeak(RT) (Preliminary Report)

Konstantin Vikhorev[1], Natasha Alechina[1], Rafael H. Bordini[2], and Brian Logan[1]

[1] School of Computer Science
University of Nottingham
Nottingham NG8 1BB UK
{kxv,nza,bsl}@cs.nott.ac.uk
[2] Institute of Informatics
Federal University of Rio Grande do Sul
Porto Alegre, Brazil
r.bordini@inf.ufrgs.br

Abstract. In this paper we give an operational semantics for the real-time agent programming language AgentSpeak(RT). AgentSpeak(RT) was introduced in [21], and extends AgentSpeak(L) with deadlines and priorities for intentions. The version of AgentSpeak(RT) presented in this paper differs in certain aspects from that in [21], mainly to incorporate both hard and soft deadlines, and allow for the concurrent execution of intentions.

1 Introduction

In this paper we give an operational semantics for the AgentSpeak(RT) real-time agent programming language introduced in [21]. In AgentSpeak(RT), an agent's intentions have priorities and deadlines. In a dynamic environment, a real-time BDI agent that has more tasks that it can feasibly accomplish by their deadlines should make a rational choice regarding which tasks to commit to: that is, it should try to accomplish higher priority tasks, but also try to execute tasks so that they are accomplished by their deadlines. An AgentSpeak(RT) agent commits to a set of intentions that are 'maximally feasible': no more intentions can be added to the schedule if the scheduled intentions are to remain feasible at the specified confidence level, and moreover, intentions which are dropped are incompatible with some scheduled higher priority intention(s).

In the version of AgentSpeak(RT) introduced in [21], only hard deadlines were supported: that is, intentions were dropped if it was impossible to execute them by their deadlines. This is reasonable for many tasks and environments, for example writing conference papers, sending bids to auctions, or catching trains. However, some tasks have soft deadlines: it may be desirable to finish a certain task by its deadline, but the work still has to be done after the deadline has passed. For example, the agent may need to charge its battery every 24 hours, but if it is delayed for some reason in reaching the charging station and the time passed since the last charge is 24 hours and one second, it is still important to reach the charging station. For this reason, we introduce *soft deadlines* in the version of AgentSpeak(RT) we consider here. Another important difference from [21] is that we assume that tasks may be executed concurrently. For

example, an agent may be simultaneously moving to a new location and communicating with other agents. However, some intentions need to be executed atomically to prevent undesired interactions between different actions. For this reason, in the version of AgentSpeak(RT) we consider here, we introduce the notion of *atomic* plans (which cannot be executed simultaneously with other atomic plans).

The rest of this paper is organised as follows. In section 2, we briefly describe the AgentSpeak(RT) architecture and the modifications relative to [21]. In section 3, we define the operational semantics of AgentSpeak(RT). We survey related work in section 4 and conclude in section 5.

2 The AgentSpeak(RT) Architecture

In this section we introduce the AgentSpeak(RT) agent programming language and its associated interpreter. Note that the version of the language presented here differs from the one in [21].

We assume that an AgentSpeak(RT) agent operates in a real-time task environment, that is, top-level goals may optionally specify a deadline and/or a priority. An AgentSpeak(RT) agent responds to events by adopting and executing intentions. A developer can specify the required level of confidence for the successful execution of intentions in terms of a probability α. An AgentSpeak(RT) agent should schedule its intentions so as to ensure that the probability that intentions are completed by their deadlines is at least α. If not all intentions can be executed with the required level of confidence due to lack of time, the agent favours intentions responding to high priority events.

The syntax and semantics of AgentSpeak(RT) with various minor modifications is based on AgentSpeak(L) [18]. To illustrate the syntax of AgentSpeak(RT) we use a simple running example of an agent which removes litter from a parking lot. Each evening, the agent is given a set of goals to achieve, each of which specifies the removal of a particular item of litter from particular parking space. In addition, the agent may detect additional litter while moving around the lot. There is a deadline for the removal of litter e.g., before the barrier is opened in the morning (we assume the agent can't cope with parking cars), and it is more important to remove some types of litter (e.g., broken glass) than others (e.g., paper).

The AgentSpeak(RT) architecture consists of five main components: a belief base, a set of events, a plan library, an intention structure, and an interpreter.

2.1 Beliefs and Goals

The agent's beliefs represent the agent's information about its environment, e.g., sensory input, information about other agents, etc. Beliefs are represented as ground atomic formulas. For example, the agent may believe that it is in space1 and there is some litter in space2:

```
at(robot,space1)
litter(paper,space2)
```

A belief atom or its negation is called a *belief literal*. A ground belief atom is called a *base belief*, and the agent's belief base is a conjunction of base beliefs.

A goal is a state the agent wishes to bring about or a query to be evaluated. An achievement goal, written $!g(t_1, \ldots, t_n)$ where $t_i, \ldots, t_n$ are terms, specifies that the agent wishes to achieve a state in which $g(t_1, \ldots, t_n)$ is a true belief. A test goal, written $?g(t_1, \ldots, t_n)$, specifies that the agent wishes to determine if $g(t_1, \ldots, t_n)$ is a true belief. For example, the goals

```
!remove(paper,space2)
?parked(X,space2)
```

indicate that the agent wants to remove the paper in space2, and determine if there is a car parked in space2.[3]

2.2 Events

Events correspond to changes in the agent's beliefs or the acquisition of new achievement goals. An addition event, denoted by $+$, indicates the addition of a base belief or an achievement goal. A deletion event, denoted by $-$, indicates the retraction of a base belief.[4] Events can be internal or external. External events originate outside the agent, while internal events result from the execution of the agent's program. As in AgentSpeak(L), all belief change events are external (originating in the agent's environment), while goal change events may be external (goals originated by a user or another agent) or internal (subgoals generated by the agent's program in response to an external event).

To allow the specification of real-time tasks, external goal addition events may optionally specify a deadline and a priority. A *deadline* specifies the time by which a goal should be achieved. Deadlines are expressed as real time values in some appropriate units, e.g, a user may specify a deadline for a goal as "4pm on Friday". Deadlines in AgentSpeak(RT) may be hard or soft. For a hard deadline it is assumed that there is no value in achieving a goal after the deadline has passed. For a soft deadline, it may still make sense to continue trying to achieve the goal after the deadline has passed, provided that this does not interfere with higher priority goals. A *priority* specifies the relative importance of achieving the goal. Priorities define a partial order over events and are expressed as non-negative integer values, with larger values taken to indicate higher priority. For example, the event

```
+!remove(paper,space2)[8am,10]
```

indicates the acquisition of a goal to remove some paper from space2 with deadline 8am and priority 10. By default the deadline is equal to infinity and the priority is equal to zero.

[3] As in Prolog, constants are written in lower case and variables in upper case, and all negations must be ground when evaluated.

[4] In the interests of brevity, we do not consider goal deletion events.

2.3 Plans

Plans specify sequences of actions and subgoals an agent can use to achieve its goals or respond to changes in its beliefs. The head of a plan consists of a triggering event which specifies the kind of event the plan can be used to respond to, and a belief context which specifies the beliefs that must be true for the plan to be applicable. The body of a plan specifies a sequence of actions which need to be executed and subgoals which need to be achieved in response to the triggering event.

Actions are the basic operations an agent can perform to change its environment in order to achieve its goals. Actions are denoted by *action symbols* and are written $a(t_1, \ldots, t_n)$ where a is an action symbol and $t_1, \ldots, t_n$ are the (ground) arguments to the action. For example, the action

```
move(trashcan)
```

will cause the agent to move from a parking space to the trashcan.

Plans may also contain achievement and test (sub)goals. Achievement subgoals allow an agent to choose a course of action as part of a larger plan on the basis of its current beliefs. An achievement subgoal $!\,g(t_1, \ldots, t_n)$ gives rise to a internal goal addition event $+!\,g(t_1, \ldots, t_n)$ which may in turn trigger subplans at the next execution cycle. Test goals are evaluated against the agent's belief base, possibly binding variables in the plan. For example, the plan

```
+litter(L,S) : at(robot,S1) & not parked(C,S) <-
  move(S); pickup(L); move(trashcan); deposit(L).
```

causes the agent to remove litter from the parking space the agent is in if there is no car parked in the space.

The BNF for plans is given below:

belief-event	::= "+" *atomic-formula* | "−" *atomic-formula*
goal-event	::= "+!" *atomic-formula* [*realtime-spec*]
belief-plan	::= "@" *label* ["`atomic`"] *belief-event* ["`:`" *context*] "<−" (*body* | "`!`" *atomic-formula realtime-spec*) "`.`"
goal-plan	::= "@" *label* ["`atomic`"] *goal-event* ["`:`" *context*] "<−" *body* "`.`"
context	::= *true* | *literal* ("`&`" *literal*)*
literal	::= *atomic-formula* | "`not`" *atomic-formula*
atomic-formula	::= $p(t_1, \ldots, t_n)$
realtime-spec	::= "[" (($hd(time)$ | $sd(time)$) "," *number*) | ($hd(time)$ | $sd(time)$) | *number* "]"
body	::= *true* | *step* ("`;`" *step*)*
step	::= $a(t_1, \ldots, t_n)$ | "`!`" *atomic-formula* | "`?`" *atomic-formula*

where *label* is a string uniquely identifying a plan, p and a are respectively predicate and action symbols of arity $n \geq 0$, and $t_1, \ldots, t_n$ are terms.

AgentSpeak(RT) allows a potentially unbounded number of plans to execute concurrently (assuming actions are not executing on the same CPU as the interpreter).

However, plans may be declared as requiring exclusive access to a single 'lock'. Intentions which do not contain an atomic plan may execute concurrently. If two plans are mutually exclusive, the execution of the intentions containing the plans must be serialised, as explained below.

The concurrent execution of intentions in AgentSpeak(RT) is similar to capabilities provided by atomic plans in Jason [1] and 2APL [6]. An atomic plan is a plan which should be executed ensuring that its execution is not interleaved with the execution of the goals and actions of other plans of the same agent. The resulting agent system is more expressive than Jason and 2APL in one sense, as Jason and 2APL cannot run non-atomic plans in parallel with an atomic one. However, it is less expressive in another sense, as in Jason and 2APL a non-atomic plan can have an atomic subplan.

In order to determine whether a plan can achieve a goal by a deadline with a given level of confidence, each action and plan has an associated *execution time profile* which specifies the probability that the action or plan will terminate successfully as a function of execution time. The expected execution time for an action or plan ϕ at confidence level α is given by $et(\phi, \alpha)$. The execution time profile will typically be influenced by the characteristics of the environment in which the agent will operate. For example, the probability of a plan to move to a location terminating successfully within a given time may be lower in environments with many obstacles than in environments with fewer obstacles.

Execution time profiles can be derived from an analysis of the agent's actions, plans and environment, or using automated techniques, e.g., stochastic simulation. In the simple case of a plan consisting of a sequence of actions, the execution time profile for the plan can computed from the execution time profiles of its constituent actions. However for plans which contain subgoals, the execution time will depend on the relative frequency with which the alternative plans for a subgoal are selected in the agent's task environment.

2.4 Intentions

Plans triggered by changes in beliefs or the acquisition of an external (top-level) achievement goal give rise to new intentions. Plans triggered by the processing of an achievement subgoal in an already intended plan are pushed onto the intention containing the subgoal. Each intention consists of a stack of partially executed plans, a set of substitutions for plan variables, a set of shared resources, a deadline and priority. The set of variable substitutions for each plan in an intention results from matching the belief context of the plan and any test goals it contains against the agent's belief base. The deadline and priority of an intention are determined by the triggering event of the root plan.

Each intention can be in one of two states: *executing* and *executable*. An intention is executing if the first action in the topmost plan in the stack of partially executed plans which forms the intention is currently executing. If the first step in the topmost plan is a goal or an action which is not currently executing, the intention is said to be executable.

Algorithm 1 AgentSpeak(RT) Interpreter Cycle

$E := E \cup G \cup \mathit{belief\text{-}events}(B, P)$
$B := \mathit{update\text{-}beliefs}(B, P)$
for all $(e, \tau) \in E$ **do**
  $O_e := \{\pi\theta \mid \theta$ is an applicable unifier for e and plan $\pi\}$
  $\pi\theta := S_O(O_e)$
  if $\pi\theta \neq \emptyset$ **and** $\tau \notin I$ **then**
    $I := I \cup \pi\theta$
  else if $\pi\theta \neq \emptyset$ **and** $\tau \in I$ **then**
    $I := (I \setminus \tau) \cup \mathit{push}(\pi\theta\sigma, \tau)$ where σ is an mgu for $\pi\theta$ and τ
  else if $\pi\theta = \emptyset$ **and** $\tau \in I$ **then**
    $I := I \setminus \tau$
  end if
end for
$I :=$ SCHEDULE(I)
for $\tau \in I$ **do**
  if $s(\tau) = \mathit{now} \wedge \mathit{executable}(\tau)$ **then**
    if $\mathit{completed}(\mathit{first}(\mathit{body}(\mathit{top}(\tau))))$ **then**
      $\pi := \mathit{pop}(\tau)$
      $\mathit{push}(\mathit{head}(\pi) \leftarrow \mathit{rest}(\mathit{body}(\pi)), \tau)$
    end if
    if $\mathit{first}(\mathit{body}(\mathit{top}(\tau))) = \mathit{true}$ **then**
      $\pi\theta := \mathit{pop}(\tau)$, $\pi' := \mathit{pop}(\tau)$
      $\mathit{push}((\mathit{head}(\pi') \leftarrow \mathit{rest}(\mathit{body}(\pi')))\theta', \tau)$
      where θ' is a restriction of θ to variables in $\mathit{head}(\pi)$.
    else if $\mathit{first}(\mathit{body}(\mathit{top}(\tau))) = !g(t_1, \ldots, t_n)$ **then**
      $E := \{(+!g(t_1, \ldots, t_n), \tau)\}$
    else if $\mathit{first}(\mathit{body}(\mathit{top}(\tau))) = ?g(t_1, \ldots, t_n)$ **then**
      if $?g(t_1, \ldots, t_n)\theta$ is an answer substitution **then**
        $\pi := \mathit{pop}(\tau)$
        $\mathit{push}((\mathit{head}(\pi) \leftarrow \mathit{rest}(\mathit{body}(\pi)))\theta, \tau)$
      else
        $I := I \setminus \tau$
      end if
    else if $\mathit{first}(\mathit{body}(\mathit{top}(\tau))) = a(t_1, \ldots, t_n)$ **then**
      $\mathit{execute}(a(t_1, \ldots, t_n))$
    end if
    break
  end if
end for

2.5 The AgentSpeak(RT) Interpreter

The interpreter is the main component of the agent. It manipulates the agent's belief base, event queue and intention structure, deliberates about which plan to select in response to belief and goal change events, and schedules and executes intentions.

The interpreter code is shown in Algorithm 1. B is the agent's belief base, E is the set of events, I is a partially ordered set of intentions. The functions *head* and *body* return the head and body of an intended plan, and *first* and *rest* are used to return the first and all but the first elements of a sequence. The function *top* returns the topmost plan in an intention. The function *pop* removes and returns the topmost plan of an intention and the function *push* takes a plan (and any substitution) and an intention and pushes the plan onto the top of the intention. The function *executable* takes an intention and returns true if the intention is executable and the function *completed* returns true if the first step in an executable intention is an action that has completed execution. The function *execute* initiates the execution of an action in a separate thread.

In contrast to AgentSpeak(L) which processes a single event at each interpreter cycle, to ensure reactivity, AgentSpeak(RT) iterates through the set of events E, and, for each event $e \in E$, generates a set of applicable plans O_e. A plan is *relevant* if its triggering event can be unified with e and a relevant plan is *applicable* if its belief context is true in B'. In general, there may be many applicable plans or options for each event. A selection function S_O chooses one of these plans for each event to give a set of options $O = \{S_O(O_e) \mid e \in E\}$. S_O is a partial function, i.e., it is not defined if O_e is empty. If the event was triggered by a subgoal of an existing intention, failure to find a applicable plan for the subgoal, i.e., if $O_e = \emptyset$, aborts the intention which posted the subgoal and the intention is removed from I. For each plan π in the set of applicable plans, if the triggering event for π was internal, the plan is pushed on top of the existing intention in I that generated the triggering event. If the triggering event for π was external, a new intention τ is created and added to I.

The scheduling algorithm is applied to I and returns a priority-maximal set of feasible intentions together with their start times. Finally, an executable intention is chosen from I for execution. An intention is executable if the first step in the topmost plan in the stack of partially executed plans that forms the intention is a goal or an action which is not currently executing (i.e., it has either completed executing or has yet to begin execution). If the first step in an executable intention is an action which has completed execution, the completed action is removed from the plan. Execution then proceeds from the next step of the topmost plan in the intention.

Executing an executable intention involves executing the first goal or action of the body of the topmost plan in the stack of partially executed plans which forms the intention. Executing an achievement goal adds a corresponding internal goal addition event to E'. Executing a test goal involves finding a unifying substitution for the goal and the agent's base beliefs. If a substitution is found, the test goal is removed from the body of the plan and the substitution is applied to rest of the body of plan. If no such substitution exists, the intention is dropped and removed from I. Executing an action results in the invocation of the Java code that implements the action and changes the state of the intention from executable to executing. We assume that action execution is performed in a separate thread, and execution of the AgentSpeak(RT) interpreter resumes immediately

after initiating the action. Reaching the end of a plan (denoted by *true* below) causes the plan to be popped from the intention and any substitutions for variables appearing in the head of the popped plan are applied to the topmost plan in the intention.

The AgentSpeak(RT) Scheduler A schedule is a priority-maximal set of feasible intentions together with their start times. A set of intentions $\{\tau_1, \ldots, \tau_n\}$ is *feasible* if

1. each intention will complete execution before its deadline with probability at least α, that is, for each scheduled intention τ_i

$$s(\tau_i) + et(\tau_i, \alpha) - ex(\tau_i) \leq d(\tau_i)$$

   where $s(\tau_i)$ is the time at which τ_i will next execute, $ex(\tau_i)$ is the time τ_i has spent executing up to this point, and $d(\tau_i)$ is the deadline for τ_i; and
2. if τ_i is an atomic intention, no intention τ_j scheduled to execute concurrently with τ_i, namely no element of the set $\{\tau_j \mid s(\tau_i) < s(\tau_j) + et(\tau_j, \alpha) \wedge s(\tau_j) < s(\tau_i) + et(\tau_i, \alpha)\}$, is atomic.

A set of intentions is priority-maximal if no more intentions can be added to the schedule if the scheduled intentions are to remain feasible at the specified confidence level, and intentions which are dropped are incompatible with some scheduled higher priority intention(s).

Scheduling in AgentSpeak(RT) is pre-emptive in that the adoption of a new high-priority intention τ may prevent previously scheduled intentions with priority lower than τ (including currently executing intentions) being added to the new schedule. Intentions which exceed their expected execution time and/or their deadline may or may not be dropped, depending on whether the deadline is hard or soft and the amount of uncommitted or 'slack' time in the schedule. If an intention has a hard deadline that has been exceeded, the intention is dropped. If an intention has a soft deadline that has been exceeded, its deadline is reset to ∞ and its priority to 0. The agent will continue to purse the intention if it can be executed concurrently with other, higher priority intentions. However if the intention is atomic, it will be scheduled after all other atomic intentions. An intention τ which has exceeded its expected execution time but not its deadline has its priority reduced to 0 and its expected execution time reset to $ex(\tau) + \delta_a$, where δ_a is the expected time required to execute the next step in the intention. τ will only be scheduled if, after scheduling all higher priority intentions, there is sufficient slack in the schedule to execute at least one step in τ before its deadline. Given sufficient slack in the schedule, τ can therefore still complete successfully. It will be however dropped if it exceeds its deadline

The scheduling algorithm is shown in Algorithm 2. We assume that the deadlines, priorities and expected execution times of the input intentions I are adjusted as described above. The set of candidate intentions is processed in descending order of priority. For each intention τ, if the intention is atomic it is added to the schedule if it can be inserted into the schedule in deadline order while meeting its own and all currently scheduled deadlines. If the intention is not atomic an attempt is made to schedule it at $s(\tau) := now$. Intentions which are not feasible in the context of the current schedule are dropped. The resulting schedule can be computed in polynomial time (in fact, quadratic time) in the size of the set I, and is priority-maximal (see [21]).

Algorithm 2 Scheduling Algorithm

```
function SCHEDULE(I)
    Γ_s := ∅, Γ_p := ∅
    for all τ ∈ I in descending order of priority do
        if ¬atomic(τ) then
            s(τ) := now
            if Γ_p ∪ {τ} is feasible then
                Γ_p := Γ_p ∪ {τ}
            end if
        else
            t := now
            Γ'_s := ∅
            for all τ' ∈ Γ_s do
                if d(τ') ≤ d(τ) then
                    Γ'_s := Γ'_s ∪ {τ'}
                    t := s(τ') + et(τ', α) − ex(τ')
                else
                    s(τ') := s(τ') + et(τ, α) − ex(τ)
                    Γ'_s := Γ'_s ∪ {τ'}
                end if
            end for
            s(τ) := t
            if Γ'_s ∪ {τ} is feasible then
                Γ_s = Γ'_s ∪ {τ}
            end if
        end if
    end for
    return Γ_p ∪ Γ_s
end function
```

3 Operational Semantics

This section gives semantics to AgentSpeak(RT) based on the operational semantics for AgentSpeak, by showing which rules have to be changed to new ones that are specific to AgentSpeak(RT). An earlier version of the operational semantics for AgentSpeak appeared in [4]. The semantic rules for communication appeared in [14] and were later improved and extended in [19]. The latter version (but without communication rules) forms the basis for this section.[5]

The operational semantics is given by a set of rules that define a transition relation between configurations $\langle ag, C, T, s\rangle$ where:

- An agent program ag is formed by a set of beliefs bs and a set of plans ps (as defined by the BNF in section 2.3 above).
- An agent's circumstance C is a tuple $\langle I, E, A\rangle$ where:

[5] Note that in the rules below, the notation for events, plans and intentions has been modified to be consistent with that in [21].

- I is a set of *intentions* $\{\tau, \tau', \ldots\}$; each intention i is a stack of partially instantiated plans.
- E is a set of *events* $\{(e, \tau), (e', \tau'), \ldots\}$. Each event is a pair (e, τ), where e is a triggering event and τ is an intention (a stack of plans in case of an internal event, or the empty intention T in case of an external event).
- A is a set of *actions* to be performed in the environment.

– T is a tuple $\langle R, Ap, \iota, \varepsilon, \rho \rangle$ which keeps track of temporary information that is required in subsequent stages within a single reasoning cycle. Note that structure of each of these components has been changed from the original semantics because AgentSpeak(RT) handles all outstanding events in a single reasoning cycle, which is a significant change from original AgentSpeak. The components of T are:
  - R for the mapping from each of the events to the set of its *relevant plans*.
  - Ap for the sets of *applicable plans* (the relevant plans whose contexts are true), again a set for each of the events currently in the set of events.
  - ι, ε, and ρ record, in the original semantics, a particular intention, event, and applicable plan (respectively) being considered along the execution of one reasoning cycle; ι is not used here, and ε and ρ have been changed to be respectively a set rather than a single event and a mapping from each of the outstanding events to the selected applicable plan (i.e., intended means) to handle it.
– The current step s within an agent's reasoning cycle is symbolically annotated by $s \in \{\mathsf{SelEv}, \mathsf{RelPl}, \mathsf{ApplPl}, \mathsf{SelAppl}, \mathsf{AddIM}, \mathsf{SelInt}, \mathsf{ExecInt}, \mathsf{ClrInt}\}$, which stands for: selecting a set of events, retrieving all relevant plans, checking which of those are applicable, selecting applicable plans (the intended means), adding the new intended means to the set of intentions, selecting an intention, executing the selected intention, and clearing an intention or intended means that may have finished in the previous step.

In the interests of readability, we adopt the following notational conventions in our semantic rules:

– If C is an AgentSpeak agent circumstance, we write C_E to make reference to the component E of C. Similarly for all the other components of a configuration.
– We write $\tau[\pi]$ to denote the intention that has plan π on top of intention τ.

New Rules for Event Selection

As AgentSpeak(RT) typically handles all outstanding events, the **SelEv** rules have been changed as follows.

$$\frac{S_E(C_E) = SEs}{\langle ag, C, T, \mathsf{SelEv} \rangle \longrightarrow \langle ag, C', T', \mathsf{RelPl} \rangle} \quad \textbf{(SelEv}_\mathbf{1}\textbf{)}$$

$$\textit{where: } \begin{array}{l} C'_E = C_E \setminus SEs \\ T'_\varepsilon = SEs \end{array}$$

Above, SEs is a set of events, those that have been selected by S_E; by default in AgentSpeak(RT), S_E selects all the events in the set of events.

Rule **SelEv$_2$** skips to the intention execution part of the cycle, in case there is no event to handle. It is the same as in the original AgentSpeak semantics:

$$\frac{C_E = \{\}}{\langle ag, C, T, \mathsf{SelEv}\rangle \longrightarrow \langle ag, C, T, \mathsf{SelInt}\rangle} \qquad \textbf{(SelEv}_\mathbf{2}\textbf{)}$$

New Rules for Relevant Plans

In contrast to the original AgentSpeak, we now have to keep track of the set of relevant plans not for one chosen event but for a set of events (typically all currently outstanding events). T_ε therefore keeps track of this set of events. For each of the events in T_ε, we find a set of relevant plans for it and keep track of that in T_R.

$$\frac{T_\varepsilon = \{(e,\tau)\} \cup RES \qquad \mathsf{RelPlans}(ag_{ps}, e) \neq \{\}}{\langle ag, C, T, \mathsf{RelPl}\rangle \longrightarrow \langle ag, C, T', \mathsf{RelPl}\rangle} \qquad \textbf{(Rel}_\mathbf{1}\textbf{)}$$

$$\textit{where: } T'_R = T_R \cup \{(e,\tau) \mapsto \mathsf{RelPlans}(ag_{ps}, e)\} \quad T'_\varepsilon = RES$$

Events with no relevant plans are ignored, as shown in the rule below.

$$\frac{T_\varepsilon = \{(e,\tau)\} \cup RES \qquad \mathsf{RelPlans}(ag_{ps}, e) = \{\}}{\langle ag, C, T, \mathsf{RelPl}\rangle \longrightarrow \langle ag, C, T', \mathsf{RelPl}\rangle} \qquad \textbf{(Rel}_\mathbf{2}\textbf{)}$$

$$\textit{where: } T'_\varepsilon = RES$$

Finally, we need an additional rule that is used to go to the next stage of the reasoning cycle when relevant plans have been found for all previously selected events.

$$\frac{T_\varepsilon = \{\}}{\langle ag, C, T, \mathsf{RelPl}\rangle \longrightarrow \langle ag, C, T, \mathsf{ApplPl}\rangle} \qquad \textbf{(Rel}_\mathbf{3}\textbf{)}$$

New Rules for Applicable Plans

Again we need a couple of rules to handle each of the mappings from events to a set of relevant plans, filtering them to keep only the applicable ones (in T_{Ap}). Rule **Appl$_1$** handles the normal case (i.e., where applicable plans are found); **Appl$_2$** says that events with no applicable plans are ignored;[6] **Appl$_3$** deals with the case where we have updated all mappings and there are events with sets of applicable plans for which intended means need to be selected; finally, **Appl$_4$** handles the case where no new intended means will result in this reasoning cycle.

[6] Note that we do not consider here the plan failure handling mechanism introduced in some of the extensions of AgentSpeak.

$$\frac{T_R = \{ev \mapsto RPs\} \cup ERs \qquad \mathsf{AppPlans}(ag_{bs}, RPs) \neq \{\}}{\langle ag, C, T, \mathsf{ApplPl}\rangle \longrightarrow \langle ag, C, T', \mathsf{ApplPl}\rangle} \quad \textbf{(Appl}_\mathbf{1}\textbf{)}$$

$$\textit{where: } T'_{Ap} = T_{Ap} \cup \{ev \mapsto \mathsf{AppPlans}(ag_{bs}, ER)\}$$
$$T'_R = ERs$$

$$\frac{T_R = \{ER\} \cup ERs \qquad \mathsf{AppPlans}(ag_{bs}, ER) = \{\}}{\langle ag, C, T, \mathsf{ApplPl}\rangle \longrightarrow \langle ag, C, T', \mathsf{ApplPl}\rangle} \quad \textbf{(Appl}_\mathbf{2}\textbf{)}$$

$$\textit{where: } T'_R = ERs$$

$$\frac{T_R = \{\} \qquad T_{Ap} \neq \{\}}{\langle ag, C, T, \mathsf{ApplPl}\rangle \longrightarrow \langle ag, C, T, \mathsf{SelAppl}\rangle} \quad \textbf{(Appl}_\mathbf{3}\textbf{)}$$

$$\frac{T_R = \{\} \qquad T_{Ap} = \{\}}{\langle ag, C, T, \mathsf{ApplPl}\rangle \longrightarrow \langle ag, C, T, \mathsf{SelInt}\rangle} \quad \textbf{(Appl}_\mathbf{4}\textbf{)}$$

New Rules for Selecting an Applicable Plan

As before, we need two rules: one to handle a particular mapping from an event to applicable plans and one for when all mappings have been processed.

$$\frac{T_{Ap} = \{ev \mapsto APs\} \cup EAs \qquad \mathcal{S}_{\mathcal{O}}(APs) = (\pi, \theta)}{\langle ag, C, T, \mathsf{SelAppl}\rangle \longrightarrow \langle ag, C, T', \mathsf{SelAppl}\rangle} \quad \textbf{(SelAppl}_\mathbf{1}\textbf{)}$$

$$\textit{where: } T'_\rho = T_\rho \cup \{ev \mapsto (\pi, \theta)\}$$
$$T'_{Ap} = EAs$$

$$\frac{T_{Ap} = \{\}}{\langle ag, C, T, \mathsf{SelAppl}\rangle \longrightarrow \langle ag, C, T, \mathsf{AddIM}\rangle} \quad \textbf{(SelAppl}_\mathbf{2}\textbf{)}$$

New Rules for Adding an Intended Means to the Set of Intentions

For each mapping of an event to the chosen intended means, we need to move it to the set of intentions; as before this requires two rules, depeding on whether the particular event was internal or external. Then we need rule **EndIM** for when all mappings have been processed.

It is important to note that the set C_I updated here is subsequently *ordered* by the scheduling algorithm presented in section 2, as stated in rules **EndIM**. In future work, we aim to formalize the scheduling of intentions within the operational semantics, by giving further rules that describe the schedulling of the intentions based on the real-time criteria.

$$\frac{T_\rho = \{(e, \mathsf{T}) \mapsto (\pi, \theta)\} \cup EIs}{\langle ag, C, T, \mathsf{AddIM}\rangle \longrightarrow \langle ag, C', T', \mathsf{AddIM}\rangle} \quad \textbf{(ExtEv)}$$

$$\textit{where: } C'_I = C_I \cup \{\ [\pi\theta]\ \} \qquad T'_\rho = EIs$$

$$\frac{T_\rho = \{(e, \tau) \mapsto (\pi, \theta)\} \cup EIs}{\langle ag, C, T, \mathsf{AddIM}\rangle \longrightarrow \langle ag, C', T', \mathsf{AddIM}\rangle} \quad \textbf{(IntEv)}$$

$$\textit{where: } C'_I = C_I \setminus \{\tau\} \cup \{\ \tau[(\pi\theta)]\ \} \qquad T'_\rho = EIs$$

$$\frac{T_\rho = \{\}}{\langle ag, C, T, \mathsf{AddIM}\rangle \longrightarrow \langle ag, C', T, \mathsf{SelInt}\rangle} \quad \textbf{(EndIM)}$$

$$\textit{where: } C'_I = \textsc{schedule}(C_I)$$

4 Related Work

Two strands of work on agent programming languages are related to work reported in this paper.

One strand is work on agent programming languages designed for developing agents with real-time capabilities. For example, the Procedural Reasoning System (PRS) [10] and PRS-like systems, e.g., JAM [12] and SPARK [15], have features such as meta-level reasoning which facilitate the development of agents for real time environments. However, to guarantee real time behaviour, these systems have to be programmed for each particular task environment—there are no general methods or tools which allow the agent developer to specify that a particular goal should be achieved by a specified time or that an action should be performed within a particular interval of an event occurring. In contrast, AgentSpeak(RT) provides a high-level programmatic interface to a standardised real-time reasoning mechanism for tasks with different priorities and deadlines.

More closely related to real-time aspects of AgentSpeak(RT) are architectures such as the Soft Real-Time Agent Architecture [22] and AgentSpeak(XL) [2]. These architectures use the TÆMS (Task Analysis, Environment Modelling, and Simulation) framework [7] together with Design-To-Criteria scheduling [23] to schedule intentions. TÆMS provides a high-level framework for specifying the expected quality, cost and duration of methods (actions) and relationships between tasks (plans). Like AgentSpeak(RT), methods and tasks can have deadlines, and TÆMS assumes the availability of probability distributions over expected execution times (and quality and costs). DTC decides which tasks to perform, how to perform them, and the order in which they should be performed, so as to satisfy hard constraints (e.g., deadlines) and maximise the agent's objective function. In comparison to AgentSpeak(RT), TÆMS allows the specification of more complex interactions between tasks. However the view of 'real-time' used in these systems is different from that taken by AgentSpeak(RT), for example in considering only soft deadlines (all tasks still have value after their deadline).

As mentioned in the Introduction, AgentSpeak(RT) was first introduced in [21]. An earlier version of this work (extending PRS rather than AgentSpeak with priorities and deadlines) was reported in [20].

Another strand of related work is research on the formal semantics of agent programming languages. Many agent-oriented programming languages have been formalised using the operational semantics approach, for example, AgentSpeak[19], GOAL [11], 2APL [6], and CAN [24], as well as the AIL effort towards unifying semantics [8]. However, to the best of our knowledge this work has not dealt with issues relating to real-time agency, and to this extent the work presented here is novel. There are, of course, various other approaches in the literature that have logical semantics, for example, MINERVA [13], and others, such as CLAIM [9] that have a formal model based on process algebra, but the operational semantics approach seems more popular in the agent programming language community. There are also important agent programming languages and platforms that have no formal semantics, such as JADEX [17] and SPARK [16], for example.

5 Conclusion

In this paper we described a modified version of AgentSpeak(RT) with parallel execution of intentions and with soft as well as hard deadlines. It is intended to be more programmer-friendly and flexible compared to the original version introduced in [21] which was designed to provide provable probabilistic guarantees of real-time behaviour. We provide an operational semantics for the language in order to make it precise and facilitate analysis of programs written in the language.

References

1. Bordini, R.H., Hübner, J.F., Vieira, R.: Jason and the Golden Fleece of agent-oriented programming. In: Bordini, R.H., Dastani, M., Dix, J., Seghrouchn, A.E.F. (eds.) Multi-agent programming : languages, platforms and applications, pp. 3–37. Springer (2005)
2. Bordini, R., Bazzan, A.L.C., Jannone, R.d.O., Basso, D.M., Vicari, R.M., Lesser, V.R.: AgentSpeak(XL): Efficient intention selection in BDI agents via decision-theoretic task scheduling. In: Proceedings of the First International Conference on Autonomous Agents and Multiagent Systems (AAMAS'02). pp. 1294–1302 (2002)
3. Bordini, R.H., Dastani, M., Dix, J., Fallah-Seghrouchni, A.E. (eds.): Multi-Agent Programming: Languages, Platforms and Applications, Multiagent Systems, Artificial Societies, and Simulated Organizations, vol. 15. Springer (2005)
4. Bordini, R.H., Moreira, Á.F.: Proving BDI properties of agent-oriented programming languages: The asymmetry thesis principles in AgentSpeak(L). Annals of Mathematics and Artificial Intelligence 42(1–3), 197–226 (Sep 2004), special Issue on Computational Logic in Multi-Agent Systems
5. Dastani, M., Fallah-Seghrouchni, A.E., Ricci, A., Winikoff, M. (eds.): Programming Multi-Agent Systems, 5th International Workshop, ProMAS 2007, Honolulu, HI, USA, May 15, 2007, Revised and Invited Papers, Lecture Notes in Computer Science, vol. 4908. Springer (2008)

6. Dastani, M., Hobo, D., Meyer, J.J.C.: Practical Extensions in Agent Programming Languages. In: Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '07). pp. 1–3. ACM, New York, NY, USA (2007), www.cs.uu.nl/docs/vakken/map/2aplposter.pdf
7. Decker, K.S., Lesser, V.R.: Quantitative modeling of complex environments. International Journal of Intelligent Systems in Accounting, Finance and Management 2, 215–234 (1993)
8. Dennis, L.A., Farwer, B., Bordini, R.H., Fisher, M., Wooldridge, M.: A common semantic basis for BDI languages. In: Dastani et al. [5], pp. 124–139
9. Fallah-Seghrouchni, A.E., Suna, A.: Claim and sympa: A programming environment for intelligent and mobile agents. In: Bordini et al. [3], pp. 95–122
10. Georgeff, M.P., Lansky, A.L.: Procedural knowledge. Proceedings of the IEEE, Special Issue on Knowledge Representation 74(10), 1383–1398 (1986)
11. Hindriks, K.V.: Modules as policy-based intentions: Modular agent programming in goal. In: Dastani et al. [5], pp. 156–171
12. Huber, M.J.: JAM: a BDI-theoretic mobile agent architecture. In: Proceedings of the Third Annual Conference on Autonomous Agents (AGENTS'99). pp. 236–243 (1999)
13. Leite, J.A., Alferes, J.J., Pereira, L.M.: Minerva - a dynamic logic programming agent architecture. In: Meyer, J.J.C., Tambe, M. (eds.) ATAL. Lecture Notes in Computer Science, vol. 2333, pp. 141–157. Springer (2001)
14. Moreira, Á.F., Vieira, R., Bordini, R.H.: Extending the operational semantics of a BDI agent-oriented programming language for introducing speech-act based communication. In: Leite, J., Omicini, A., Sterling, L., Torroni, P. (eds.) Declarative Agent Languages and Technologies, Proc. of the First Int. Workshop (DALT-03), held with AAMAS-03, 15 July, 2003, Melbourne, Australia. pp. 135–154. No. 2990 in LNAI, Springer-Verlag, Berlin (2004)
15. Morley, D., Myers, K.: The SPARK agent framework. In: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'04). pp. 714–721 (2004)
16. Morley, D.N., Myers, K.L.: The spark agent framework. In: AAMAS. pp. 714–721. IEEE Computer Society (2004)
17. Pokahr, A., Braubach, L., Lamersdorf, W.: Jadex: A BDI reasoning engine. In: Bordini et al. [3], pp. 149–174
18. Rao, A.S.: Agentspeak(l): BDI agents speak out in a logical computable language. In: MAAMAW'96: Proceedings of the 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World: Agents Breaking Away. pp. 42–55 (1996)
19. Vieira, R., Moreira, Á.F., Wooldridge, M., Bordini, R.H.: On the formal semantics of speech-act based communication in an agent-oriented programming language. J. Artif. Intell. Res. (JAIR) 29, 221–267 (2007)
20. Vikhorev, K., Alechina, N., Logan, B.: The ARTS real-time agent architecture. In: Dastani, M., El Fallah Segrouchni, A., Leite, J., Torroni, P. (eds.) Languages, Methodologies, and Development Tools for Multi-Agent Systems, Second International Workshop, LADS 2009, Torino, Italy, September 7-9, 2009, Revised Selected Papers, Lecture Notes in Computer Science, vol. 6039, pp. 1–15. Springer Berlin / Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-13338-1_1
21. Vikhorev, K., Alechina, N., Logan, B.: Agent programming with priorities and deadlines. In: Proceedings AAMAS 2011 (2011)
22. Vincent, R., Horling, B., Lesser, V., Wagner, T.: Implementing soft real-time agent control. In: Proceedings of the Fifth International Conference on Autonomous Agents (AGENTS'01). pp. 355–362 (2001)
23. Wagner, T., Garvey, A., Lesser, V.: Criteria-directed heuristic task scheduling. International Journal of Approximate Reasoning 19, 91–118 (1998)

24. Winikoff, M., Padgham, L., Harland, J., Thangarajah, J.: Declarative & procedural goals in intelligent agent systems. In: Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR-02). pp. 470–481 (2002)

Probing Attacks on Multi-agent Systems using Electronic Institutions

(Preliminary Report)

Shahriar Bijani [1,2], David Robertson[1] and David Aspinall[1]

[1] Informatics School, University of Edinburgh. 10 Crichton St. Edinburgh, UK.
[2] Computer Science Dept., Shahed University, Persian Gulf Highway, Tehran, Iran.

S.Bijani@ed.ac.uk , dr@inf.ed.ac.uk, David.Aspinall@ed.ac.uk.

Abstract. In open multi-agent systems, electronic institutions are used to form the interaction environment by defining social norms for group behaviour. However, as this paper shows, electronic institutions can be turned against agents to breach their security in a variety of ways. We focus our attention on probing attacks using electronic institutions specified in the Lightweight Coordination Calculus (LCC) language. LCC is an orchestration language used to define electronic institutions in agent systems. A probing attack is an attack against the confidentiality of information systems. In this paper, we redefine the probing attack in conventional network security to be applicable in a multi-agent system domain, governed by electronic institutions. We introduce different probing attacks against LCC interaction models and suggest a secrecy analysis framework for these interactions. We also propose some countermeasures against probing attacks in LCC.

Keywords: Multi-Agent Systems, Electronic Institutions, Interaction Models, Security, Probing Attack, Lightweight Coordination Calculus (LCC).

1 Introduction

One way to build large-scale multi-agent systems is to develop open architectures in which agents are not pre-engineered to work together and in which agents themselves determine the social norms that govern collective behaviour. Open multi-agent systems have growing popularity in the Multi-agent Systems community and are predicted to have many applications in the future [4]. A major practical limitation to such systems is security because the openness of such systems negates many traditional security solutions.

An electronic institution [10] is an organisation model for multi-agent systems that provides a framework to describe, specify and deploy agents' interaction environments [15]. It is a formalism which defines agents' interaction rules and their permitted and prohibited actions. Lightweight Coordination Calculus, LCC [18;19], is a declarative language to execute electronic institutions in a peer to peer style. In LCC, electronic institutions are called *interaction models*. While electronic

institutions can be used to implement security requirements of a multi-agent system, they also can be turned against agents to breach their security in a variety of ways, as this paper shows.

Although openness in open multi-agent systems makes them attractive for different new applications, new problems emerge, among which security is a key. This is because we can make only minimum guarantees about identity and behaviour of agents. The more these systems are used in the real world, the more the necessity of their security will be obvious to users and system designers. Unfortunately there remain many potential gaps in the security of open multi-agent systems and relying on security of low level network communications is not enough to prevent many attacks on multi-agent systems. Furthermore, traditional security mechanisms resist use in multi-agent systems directly, because of the social nature of them. Confidentiality is one of the main features of a secure system and there are various attacks against it. In this paper, we focus our attention on probing attacks from agents on agents using electronic institutions specified in the LCC language.

Most work on security of multi-agent systems directly or indirectly focus on mobile agents and many of the solutions have been proposed for threats from agents to hosts or from hosts to agents (e.g.[9;23;24]). But not much research has been done on attacks from agents on agents in open multi-agent systems. A survey of possible attacks on multi-agent systems and existing solutions for attack prevention and detection can be found in [6]. None of these solutions address the probing attacks introduced in this paper.

Xiao et al. [26] have proposed multilevel secure LCC interaction models for health care multi-agent systems. A security architecture for the *HealthAgents* system and a security policy set using LCC have been suggested in [25]. Hu et al. [13] have developed a system to support data integration and decision making in the breast cancer domain using LCC and briefly addressed some security issues. They have all used constraints and message passing in LCC interaction models to implement security solutions for access control and secure data transfer, but they have not addressed inference of private data based on our defined probing attack.

In this paper we introduce a new attack against the confidentiality of agents' local knowledge, inspired by the concept of probing attack in conventional computer networks. We introduce an attack detection method by proposing a conceptual representation of LCC interaction models and adapting an inference system from credential-based authorisation policies [5] to electronic institutions. We also suggest countermeasures to prevent probing attacks on the systems using the LCC language.

2 Lightweight Coordination Calculus (LCC)

LCC is a compact executable specification to describe the notion of social norms [20]. It is a choreography language based on π-calculus [17] and logic programming. We use LCC to implement interaction models for agent communication. An interaction model (an electronic institution) in LCC is defined as a set of clauses, each of which specifies a role and its process of execution and message passing. The LCC syntax is shown in Fig. 1.

```
Interaction Model := {Clause,...}
Clause := Role::Def
Role := a(Type, Id)
Def := Role | Message | Def then Def| Def or Def | null ← Constraint
Message:= M ⇒ Role | M ⇒ Role ← Constraint | M ⇐ Role |
     M ⇐ Role ← Constraint
Constraint:= Constant | P(Term,...)| not(Constraint) | Constraint and
    Constraint | Constraint or Constraint
Type := Term
Id := Constant | Variable
M := Term
Term := Constant | Variable | P(Term,...)
Constant := lower case character sequence or number
Variable := upper case character sequence or number
```

Fig. 1. LCC language syntax; principal operators are: outgoing and incoming messages (⇒ *and* ⇐), conditional (←), sequence (then) and committed choice (or). Variable names in a clause are local.

An interaction model in LCC is a set of clauses each of the form *Role :: Def*, where *Role* denotes the role in the interaction and *Def* is the definition of the role. Roles are of the form a(Type, Id), where Type gives the type of role and Id is an identifier for the individual peer undertaking that role. The definition of performance of a role is constructed using combinations of the sequence operator (then) or choice operator (or) to connect messages and changes of role. Messages are either outgoing to another peer in a given role (⇒) or incoming from another peer in a given role (⇐). Message input/output or change of role can be governed by constraints (connected by the "←" operator) which may be conjunctive or disjunctive. Constraints can be satisfied via shared components registered with a website (e.g. www.openk.org), so that complex (possibly interactive) solving methods can be shared along with interaction models; or they can be calls to services with private data and reasoning methods. Variables begin with upper case characters.

Role definitions in LCC can be recursive and the language supports structured terms in addition to variables and constants so that, although its syntax is simple, it can represent sophisticated interactions. Notice also that role definitions are "stand alone" in the sense that each role definition specifies all the information needed to complete that role. This means that definitions for roles can be distributed across a network of computers and (assuming the LCC definition is well engineered) will synchronise through message passing while otherwise operating independently. Matching of output messages from one peer to input messages of another is achieved by simple pattern matching, since (although operating independently) the roles were originally defined to work together. More sophisticated forms of input/output matching have been defined for LCC (to allow for more sophisticated ontology

matching) but these are not the subject of this paper. For a more detailed introduction to LCC, see [18].

For different applications, agents may use their own interaction model or download an existing one. When an agent selects an interaction model and a role in it, its behaviour in that interaction is then determined by the constraints attached to message sending/receiving events specified in the definition of that role. Agents may be involved in any number of interactions (specified by interaction models) simultaneously.

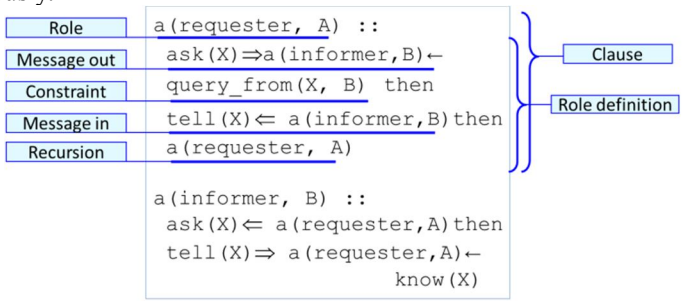


Fig. 2. An example of an interaction model in LCC

Fig. 2 illustrates an example of an interaction model for a simple communication in LCC. There are two roles (clauses) in this interaction model: *requester* and *informer*. In the first clause, a *requester* asks about something from an *informer*, then gets an answer from it and then continues as a *requester*. In the second clause, an *informer* is asked by a *requester* and then it should tell the *requester* if it knows the answer.

3 Probing Attack in Multi-agent Systems

We redefine the probing attack [3] in conventional network security to be applicable in multi-agent systems. A probing attack in network security is an attack based on injecting traffic into the victim's network and analysing the results [28]. It is a sort of active traffic analysis, which is a popular attack in cryptography [14;21] and is a basis for other attacks in computer network systems {Ahmad, 2009 5 /id}.

In our case, an adversary, who plays a role in an interaction model, could infer information not only from the interaction model itself, but also from the local knowledge of other agents. An adversary could control other agents' behaviour in an interaction, by publishing a malicious interaction model. Furthermore, it could access the private local knowledge (e.g. decision rules and policies) of the victim agents by injection of facts to the agent knowledge-base, asking queries and analysing the queries result.

We can define four types of probing attacks on open multi-agent systems: (1) explicit query attack, (2) implicit query attack, (3) injection attack and (4) indirect query. In explicit query probing attack, the idea is to make several direct queries to an agent via messages (Fig. 3-a). It may seem an elementary attack, but there can be

sophisticated versions of it, such as gathering provenance information [7;27] by an attacker or accessing all the information in a semi-open ontology by asking intelligent questions from different parts of it. An example of a semi-open ontology is ontology of a service provider company which is open to customer's questions, but where extensive knowledge of the whole ontology is a commercial confidential asset [6].

In Fig. 3, two simple examples, which are modified versions of a proteomics lab [1] interaction model used in one of the testbeds of the OpenKowledge project [22], illustrate the first type of probing attack. In these examples, an adversary (in the role of *researcher*) could ask explicit and implicit queries from a proteomics lab agent (*omicslab*). In Fig. 3-a, in the *omicslab* clause (lines 9 to 13), when a proteomics lab agent *O* receives an *ask(X)* message, it directly sends X, which is a private annotation of a specific protein, to the *researcher*.

```
1.a(researcher(LabList), R) ::
2.( ask(X)=>a(omicslab, H)<-
3.          LabList=[H|T]   then
4.  tell(X)<=a(omicslab, H) then
5.  null <- processResult(X, H)
6.  then a( researcher(T), R)
7. ) or
8. null <- LabList = []

9. a(omicslab, O) ::
10. ask(X)<= a(researcher,R) then
11. tell(X)=>a(researcher,R)
12.                    <-know(X)
13. then a(omicslab, O)
```

(a)

```
1.a(researcher(LabList), R) ::
2.( ask(X)=> a(omicslab, H) <-
3.           LabList=[H|T] then
4.  tell(Y)<= a(omicslab,H) then
5.  null <- processResult(X,Y,H)
6.  then a( researcher(T), R)
7. ) or
8. null <- LabList = []

9.a(omicslab, O) ::
10.ask(X)<= a(researcher,R) then
11.tell(Y)=>a(researcher,R)<-
12.                  Combine(X,Y)
13. then a(omicslab, O)
```

(b)

Fig. 3. Two examples of type 1 probing attack, in which a malicious *researcher* could ask explicit and implicit queries from a proteomics lab agent (*omicslab* clause, lines 11,12). (a) A direct query example asking X from the *omicslab*. (b) An indirect query, *know(X),* as a constraint in the *omicslab* clause.

The second type of probing attack is asking an implicit query on confidential information. An adversary often might not be interested ask a query explicitly, for various reasons; e.g. a direct question from confidential information may be forbidden or might attract the attention of the victim. An indirect query could be asked by placing a query as a constraint in LCC, rather than sending a message. In other words, an adversary could not only infer information from a received message, but also from analysing the constraints in an interaction model. An example of confidential information in proteomics lab could be the combination (binding potential) of two publicly known proteins that activate a particular gene. In this example, the relation between two pieces of public information is private. In Fig. 3-b, X and Y are not confidential but a malicious *researcher, R,* could recognise whether proteins X and Y could combine not by asking a direct question, but by putting a *combine(X,Y)* constraint in line 12. When *O* sends the non-confidential *tell(Y)* message to *R* it will indirectly inform *R* that *X* and *Y* could combine together because R knows that *combine(X, Y)* had to be satisfied before the *tell(Y)* message could be sent.

The third type of probing attack happens by injection of some facts into the system and asking queries before and after the injection. Arguably, the whole interaction model that has been designed by an adversary could be considered as injected information for agents using it. But the purpose of the injection is to introduce the constraints in the victim's interaction model. In this type of attack, the assumption is that the injection affects decisions of the victim. This attack is similar to the implicit query attack and in some cases might be considered as compound implicit queries. We illustrate a sample attack in Fig. 4 and Fig. 5 inspired from an example of a probing attack in authorisation languages by Gurevich and Neeman [12].

```
1.  a(vendor, V)::
2.   null <- ask(S)<= a(customer,C) then
3.   null <- (not(want(C,S)) or   payFor(C,S) ) then  /*injection */
4.   null <- (not(SupplyFrom(X)) or want(C,S) ) then /* injection */
5.   ok => a(customer,C) <- agree(C,S)    then  /* implicit query */
6.    ...
7.   then a(vendor, V)
```

Fig. 4. A fragment of a selling interaction model shows an example of type three probing attack

Fig. 4 shows one clause of a selling interaction model that could be used for a probing attack by injection. The attack begins when an agent selects a *vendor* role of this malevolent interaction model, which has been created and published by an adversary. The adversary (*C*) plays the role of *customer* and initiates the interaction by sending the *ask(S)* message to the vendor. The goal of the adversary is to discover the confidential fact whether X is the supplier of the *vendor (SupplyFrom(X))*. The first two constraints (line 3) tell the *vendor* that the *customer pays* for S or does not *want* S. The next two constraints (line 4) inject the facts that *X* is not the vendor's *supplier* or the customer *want*s S. In other words, these constraints are added information to the knowledge-base of the *vendor* agent and could shape its decisions. The subsequent implicit query asking if the *vendor* agrees with the deal is sent (line 5) to signal to the attacker that the complex constraint was satisfied. These injections and the agent's response to the query are not still enough for the attacker to infer the validity of *SupplyFrom(X)*. Then the adversary terminates this interaction and initiates two other interactions with the victim (Fig. 5).

```
1. a(vendor2, V)::
2.  null<-want(C,S) then
                  /*injection */
3.  ok=> a(customer,C) <-
   agree(C,S)  /*query*/
       ...
```

(a)

```
1.  a(vendor3, V)::
2.  null <- payFor(C,S) then
                  /*injection */
3.  ok=> a(customer,C) <-
   agree(C,S)  /*query*/
       ...
```

(b)

Fig. 5. Definitions of the vendor roles in two malicious interaction models as parts of a probing attack scenario

Each new interaction model injects only one part of the previous injections and asks the same implicit query. If the answer to the first query is positive (an *ok*

message) and to the next two queries are negative, after some analyses (see section 5), the adversary could infer the confidential fact that X is the supplier of the *vendor*.

Indirect query is the fourth type of probing attack, in which an adversary tries to access confidential information of the victim agent via a third party for reasons similar to the implicit attack. Indirect attack is a modification of the explicit query attack and could also be combined with the other types of probing attacks. A modified fragment of an interaction model in MIAKT project [13], which aims to support multidisciplinary meetings for the diagnosis and management of breast cancers, is illustrated in Fig. 6. The *dataHandler* retrieves patient's private data based on the request submitted by an authorised domain specialist (Fig. 6-a: line 3), but an illegitimate *nurse* has open access to without any authorisation check (Fig. 6-b: line 7).

```
1. a(dataHandler,H) ::
2.  patient_record(Patient) <=  a(specialist,E) then
3.  inform(Patient) =>  a(specialist,E) <- is_authorised(E,ID)and
   get_patient_id(Patient,ID) then
4.  ...
```

(a)

```
1. a(specialist,E) ::
2.   patient_record(Patient) =>  a(dataHandler,H) then
5.   process(Patient) <- inform(Patient) <=  a(dataHandler,H) then ...
6.   patient_record(Patient) <=  a(nurse, N) then
7.   inform(Patient) =>  a(nurse, N)
3.   ...
```

(b)

Fig. 6. A fragment of an interaction model to support multidisciplinary meetings for the diagnosis and management of breast cancers. **(a)** the data handler role [13]. **(b)** the specialist role.

4 Conceptual Representation of Interaction Models

The first step in our security analysis is converting interaction models to simpler logical representations in order to illustrate only the related parts of the LCC code to the security evaluation. Although LCC resembles a type of logic programming language, conversion of an LCC specification to first order logic expressions is not straightforward because its semantics must be understood in terms of temporal beliefs assigned to agents. What we need for our conceptual representation is a more minimal interpretation of LCC, which reflects information leaks or helps to find knowledge leakage.

The conceptual representation links the notion of electronic institutions with the idea of information flow analysis. It could vary in different scenarios and from various stakeholders' points of view. For example when an adversary has designed and published the interaction model herself / himself, and plays one or more roles in it, she/he might be only interested to analyse clauses related to other roles. We should interpret interaction models for each scenario differently, to be able to discover information leaks and consequently to achieve more accurate secrecy analyses.

We now introduce two conceptual representations of interaction models. They are to some extent similar, but the main differences are derived from the way an adversary exploits the interaction model and what the interaction model could add to the knowledge of an agent. In both representations, if we use non-temporal logic for the conceptual representations, the *then* operator in LCC will be equivalent to a logical conjunction. That is because we analyse the interaction model ahead of time, so we can ignore the effect of the actions' sequence on the inferred information inferred by the adversary. We also interpret the choice operator *or* in LCC as logical disjunction and message passing operators as *send* and *receive* functions. We can legitimately do this because we are not defining the semantics of the LCC specification but, instead, we are describing the (constraint-based) information that can be inferred to be true if the definition is satisfied (i.e. it has completed in the interaction).

In the first version, the conditional operator (<–) in LCC is interpreted as a material conditional in logic. For example the *informer* clause in Fig. 3-a simply could be represented by two first order logic expressions as:

```
receive(R,X),
know(X) → send(R, X)
```

This representation could be used for analysing explicit or indirect query attacks, but is not useful for implicit or injection probing attacks.

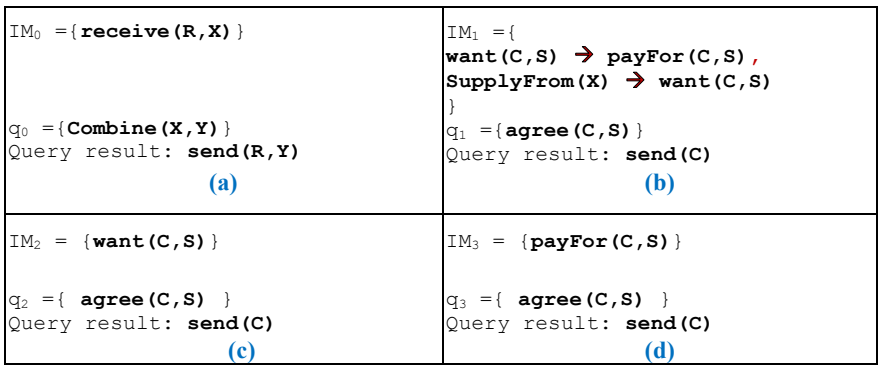


Fig. 7. In these conceptual representations, q is a query. (a) Implicit query attack example in Fig. 3-b. (b) Injection attack example in Fig. 4. (c) Injection attack example in Fig. 5-a. (d) Injection attack example in Fig. 5-b.

In the second representation of interaction models, constraints are interpreted as queries or injection from the counterpart agent (an adversary). Hence the conditional operator (<-) does not mean a logical condition anymore and the sent message in the left of <- (if it exists), is an answer to the query. The received message's parameters are also considered new information for the receiver agent. So the equivalent representation of Fig. 3-b and Fig. 4 would be as shown in Fig. 7.

5 Attack Detection

After the conceptual representation of interaction models, in which injections and queries are defined, we could analyse them to detect any possibility of a probing attack. A probing attack happens when a malicious agent could infer anything about its counterpart's local knowledge. We use Becker's inference system[5] to detect probing attacks from interaction models' conceptual representations. Becker has introduced an inference system for *detectability* [8][1] of a specific property in Datalog-based policy languages. Although this inference system has been created for credential-based authorisation policies with some modifications it could also be used to detect probing attacks on multi-agent systems. We want to know when an adversary injects expressions into the agent's private knowledge-base and asks a query, what else the adversary could infer from the knowledge-base. To answer this question we use the inference system in Fig. 8.

$$(\text{PEEK})\ \frac{IM \vdash q}{\Pi, IM \Vdash q}\ q \text{ is monotonic},\ IM \text{ is ground} \qquad (WEAK)\ \frac{\Pi, IM \Vdash q \quad q \Rightarrow q'}{\Pi, IM \Vdash q'}$$

$$(\text{POKE1})\ \frac{A_0 \cup IM \vdash q \quad (IM, q) \in P}{\Pi, IM \Vdash q} \qquad (\text{POKE2})\ \frac{A_0 \cup IM \nvdash q \quad (IM, q) \in P}{\Pi, IM \Vdash \neg q}$$

$$(\text{MONO1})\ \frac{\Pi, IM \Vdash q \quad IM' \succcurlyeq IM}{q \text{ is monotonic} \quad IM' \text{is ground}} \qquad (\text{MONO2})\ \frac{\Pi, IM \Vdash q \quad IM' \preccurlyeq IM}{\neg q \text{ is monotonic} \quad IM' \text{is ground}}$$

$$(\text{CONJ})\ \frac{\Pi, IM \Vdash q_1 \quad \Pi, IM \Vdash q_2}{\Pi, IM \Vdash q_1 \wedge q_2} \qquad (\text{DIFF})\ \frac{\Pi,\ A_0 \cup IM \Vdash q}{\Pi, A_0 \Vdash q \vee \bigvee_{a \in IM} fired(IM, head(a))}$$

Fig. 8. The inference system of Becker [5] (with a few changes) to be used to detect information leaks

In this inference system, IM is the set of injected predicates and q is the query in the conceptual representation defined in the previous section. The inference system assumes that the injection is ground and the query is monotonic (without negation). A_0 is the confidential local knowledge set of the under attack agent, P is the set of all injections and queries in a probing attack and the probing environment Π is (A_0, P). IM'⩾IM in the axiom (MONO1) means all facts that are entailed from IM ∪ S, could also be entailed from IM' ∪ S where S is all sets of ground atoms. (DIFF) is the most important part of the inference system and tells us what can be inferred from A_0 when we inject IM to A_0.

The ***fired*** operator [5] in (DIFF) is ¬ ∧⋁$_{\Delta\in}$ ⋀Δ, where S is a set of explanations of why any ground atom f is inferred from the injection (IM) and it could be computed by standard abduction [16] method. The intuition behind the ***fired*** operator is that when an adversary injects some expressions into the agent's knowledge-base

[1] Detectability (or non-opacity) is an information flow property that shows the ability to infer a specific predicate from a set of rules.

(A_0) and receives a result, at least one of the injected expressions, which was not held in A_0, has the main role in shaping the result.

We must convert the injections and queries to ground expressions to be able to use this inference system. Finding a ground substitution for these expressions is not hard and does not cause loss of generality. So, we formulate the injection attack example in Fig. 7-b to 7-d with the expressions in Fig. 9. Injections and queries in an interaction model are illustrated as $A_0 \cup IM_i \vdash q$. We assume that the adversary's query is successful the first time and unsuccessful the second and third times.

$IM_1 = \{w \rightarrow p, s \rightarrow w\}$ and $q_1 = g$: $A_0 \cup IM_1 \vdash g$,

$IM_2 = \{w\}$, $q_2 = g$ $A_0 \cup IM_2 \nvdash g$,

$IM_3 = \{p\}$, $q_3 = g$ $A_0 \cup IM_3 \nvdash g$.

$P = \{(IM_1,g), (IM_2,g), (IM_3,g)\}$

Fig. 9. The ground version of the probing attack example in Fig. 7-b to 3-d . w=want(C,S), p=payFor(C,S), s=SupplyFrom(X) and g=agree(C,S).

The sequence of inference rules in Fig. 10shows what the adversary could infer from the local knowledge of the victim agent. As a result of this analysis, the adversary finds that the target agent knows: ¬ agree(C,S) ∧ ¬ want(C,S) ∧ ¬ payFor(C,S) ∧ SupplyFrom(X). All the inferred facts might be important but in this example, the goal of the adversary was to find the private information about the supplier of the target vendor, so it was a successful attack. In [5], it is shown that ground finite detectability is fully decidable and this inference system is sound but it's completeness is still an open problem.

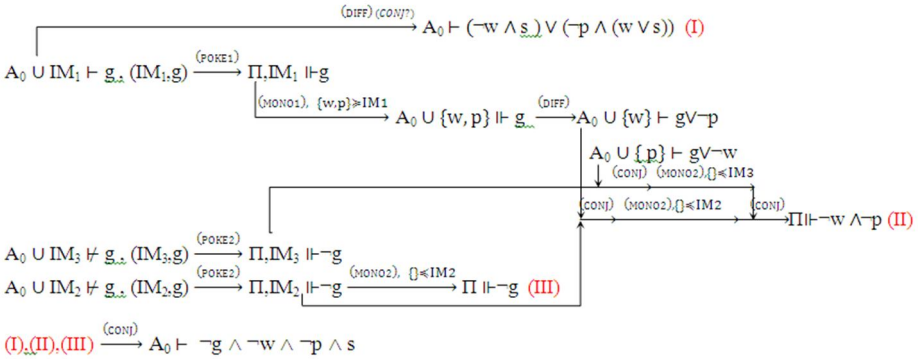


Fig. 10. Steps of using the inference system to detect the possibility of probing attack using the example in Fig. 7-b to 7-d. It shows what an adversary could infer from the local knowledge (A0) of the victim agent using those interaction models.

6 Countermeasures

Two reasons that security problems might lead to probing attacks are (1) no distinguishing notion of private and public data in LCC and (2) no mechanism for information leakage control in an interaction. Hence, two countermeasures to these problems are adding some access control features in LCC and secrecy analysis of interaction models. The first solution is to label information in LCC. Variables, constants and constraints are ultimately the most elementary causes of the described information leak, so when each peer receives an interaction model, it could annotate it to reflect the confidentiality level of the information. Fig. 11 suggests an added syntax for LCC with four levels of confidential terms.

```
Term  := Constant | Variable | P(Term,...) | pTerm
pTerm := Term {Label}
Label := l | L | h | H
```

Fig. 11. Added LCC syntax to support private terms

The rules that the LCC interpreter should support to prevent explicit, implicit and indirect probing attacks are:

- Terms without privacy labels are public,
- Labels of confidential levels are as follows: public < l < L < h < H,
- Sending messages containing higher level terms than the receiver's level is not allowed,
- Sending messages containing lower level terms than its corresponding constraint is not allowed,
- The confidential level of a compound expression is the maximum level of all parts.

As LCC can be interpreted and executed by agents in distributed peer to peer networks and each clause of an interaction model might be run separately, considering the above rules in one clause is not enough to prevent some probing attacks. So, all clauses of an interaction model should be analysed together. Although labeling information and considering the above rules are a solution for some probing attacks, it cannot prevent or detect injecting attacks.

The second solution for probing attacks is secrecy analysis of interaction models using techniques such as using the introduced inference system to detect injection attacks. This analysis could be implemented a constraint at the beginning of each interaction model, or as a separate interaction model that receives other interaction models and after extracting the corresponding logical representation, check possibility of information leak using the inference system.

Information flow analysis is one of the main techniques for studying confidentiality [11]. The first solution uses *non-interference* and the second solution exploits *detectability* and both properties are popular tools in information flow analysis. Hence, the suggested countermeasures are promising enough to preserve the secrecy of interaction models against many probing attacks.

7 Conclusion

In this paper, we have introduced probing attacks on multi-agent systems governed by electronic institutions and developed a secrecy analysis model for the interaction models used in LCC to describe electronic institutions. We have proposed four types of probing attacks, namely, explicit query, implicit query, injection and indirect query attacks on LCC interaction models. To analyse information leaks in these agent systems, we have suggested two conceptual (logical) representations of interaction models and adapted the Becker's inference system, which shows the possibility of private information disclosure by an adversary. Finally we have presented two solutions to prevent and detect probing attacks in LCC interaction models. The first solution adds a labelling capability to the LCC language to have various levels of private information and the second solution analyses the secrecy of the agent system at the interaction level. To generalise our work to other electronic institution languages besides LCC, it is enough to adapt the conceptual representation module for each language.

Acknowledgement
We would like to thank the anonymous reviewers for providing useful comments. This work is supported under the ISPRF research funding, which is sponsored by the Ministry of Science, Research and Technology of Iran.

References

[1] J. Abian, M. Atencia, P. Besana, L. Bernacchioni, D. Gerloff, S. Leung, J. Magasin, A.P. de Pinninck, X. Quan, D. Robertson. OpenKnowledge Deliverable 6.3: Bioinformatics Interaction Models. 2008.

[2] I. Ahmad, A.B. Abdullah, A.S. Alghamdi. Application of artificial neural network in detection of probing attacks. Industrial Electronics & Applications, 2009. (ISIEA 2009), 557-562. 4-10-2009.

[3] R. Anderson, M. Kuhn. Tamper Resistance: A Cautionary Note. Proceedings of the Second USENIX Workshop on Electronic Commerce 2, 1-11. 1996. USENIX Association.

[4] A. Artikis, M. Sergot, J. Pitt, Specifying Norm-Governed Computational Societies. Acm Transactions on Computational Logic 10 (2009).

[5] M.Y. Becker. Information Flow in Credential Systems. Computer Security Foundations Symposium (CSF), 2010 23rd IEEE , 171-185. 2010. IEEE.

[6] S. Bijani, D. Robertson, A Review of Attacks and Security Approaches in Open Multi-agent Systems. Artificial Intelligence Review2011).

[7] U. Braun, A. Shinnar, M. Seltzer. Securing provenance. Proceedings of the 3rd conference on Hot topics in security , 1-5. 2008. USENIX Association.

[8] J.W. Bryans, M. Koutny, L. Mazare, P.Y.A. Ryan, Opacity generalised to transition systems. International Journal of Information Security 7 (2008) 421-435.

[9] R. Endsuleit, A. Wagner. Possible attacks on and countermeasures for secure multi-agent computation. Arabnia.H.R., Aissi, S., and Mun, Y. International Conference on Security and Management (SAM '04) , 221-227. 2004. CSREA Press.

[10] M. Esteva, D. De La Cruz, B. Rosell, J.L. Arcos, J.A. Rodriguez-Aguilar, G. Cuní. Engineering open multi-agent systems as electronic institutions. Procedings of the National Conference on Artificial Intelligence (AAA '04) , 1010-1011. 2004. AAAI Press.

[11] R. Gorrieri, F. Martinelli, I. Matteucci, Towards information flow properties for distributed systems. Electronic Notes in Theoretical Computer Science 236 (2009) 65-84.

[12] Y. Gurevich, I. Neeman. DKAL: Distributed-knowledge authorization language. Computer Security Foundations Symposium, 2008.CSF'08.IEEE 21st , 149-162. 2008. IEEE.

[13] B. Hu, S. Dasmahapatra, P. Lewis, D. Dupplaw, N. Shadbolt, Facilitating Knowledge Management in Pervasive Health Care Systems. Networked Knowledge-Networked Media 221 (2009) 285-304.

[14] Y. Ishai, A. Sahai, D. Wagner, Private circuits: Securing hardware against probing attacks. Advances in Cryptology-Crypto 2003, Proceedings 2729 (2003) 463-481.

[15] S. Joseph, A.P. de Pinninck, D. Robertson, C. Sierra, C. Walton. OpenKnowledge Deliverable 1.1: Interaction Model Language Definition. 2006.

[16] A.C. Kakas, R.A. Kowalski, F. Toni The Role of Abduction in Logic Programming. in: D. M. Gabbay, C. J. Hogger, and J. A. Robinson (Eds.), Handbook of Logic in Artificial Intelligence and Logic Programming: Logic programming 5, Oxford University Press, USA, 1998, pp. 235-324.

[17] R. Milner, J. Parrow, D. Walker, A Calculus of Mobile Processes .1. Information and Computation 100 (1992) 1-40.

[18] D. Robertson. Multi-agent coordination as distributed logic programming. Logic Programming, Proceedings 3132, 416-430. 2004. Lecture Notes in Computer Science.

[19] D. Robertson. Multi-agent coordination as distributed logic programming. International Conference on Logic Programming , 77-96. 2004. Springer.

[20] D. Robertson, A lightweight coordination calculus for agent systems. Declarative Agent Languages and Technologies Ii 3476 (2005) 183-197.

[21] J.-M. Schmidt, C. Kim A Probing Attack on AES. in: K. I. Chung, K. Sohn, and M. Yung (Eds.), Information Security Applications, Springer Berlin / Heidelberg, 2009, pp. 256-265.

[22] R. Siebes, D. Dupplaw, S. Kotoulas, A.P. de Pinninck, F. Van Harmelen, D. Robertson. The openknowledge system: an interaction-centered approach to knowledge sharing. Proceedings of the 15th International Conference on Cooperative information systems (CoopIS) , 381-390. 2007. Springer-Verlag.

[23] G.J. Van't Noordende, B.J. Overeinder, R.J. Timmer, F.M.T. Brazier, Constructing secure mobile agent systems using the agent operating system. International Journal of Intelligent Information and Database Systems 3 (2009) 363-381.

[24] S. Venkatesan, C. Chellappan. Protection of Mobile Agent Platform through Attack Identification Scanner (AIS) by Malicious Identification Police (MIP). First International Conference on Emerging Trends in Engineering and Technology , 1228-1231. 2008. IEEE.

[25] L. Xiao, S. Dasmahapatra, P. Lewis, A. Peet, A. Gibb, D. Dupplaw, M. Croitoru, B. Hu, F. Estanyol, J. Martinez-Miranda, H. Gonzalez-Velez, M. Lluch i Ariet, The design and implementation of a novel security model for HealthAgents. Knowledge Engineering Review 26 (2011).

[26] L. Xiao, P. Lewis, S. Dasmahapatra. Secure Interaction Models for the HealthAgents System. 27th International Conference on Computer Safety, Reliability, and Security (SAFECOMP). Lecture Notes in Computer Science 5219, 167-180. 2008. Springer.

[27] S. Xu, Q. Ni, E. Bertino, R. Sandhu. A characterization of the problem of secure provenance management. IEEE International Conference on Intelligence and Security Informatics, ISI09. 310-314. 2009. IEEE.

[28] J. Zheng, M.Z. Hu, Intrusion detection of DoS/DDoS and probing attacks for web services. Advances in Web-Age Information Management, Proceedings 3739 (2005) 333-344.