João A. Leite Andrea Omicini Leon Sterling
Paolo Torroni (eds.)

Declarative Agent Languages and Technologies

First International Workshop, DALT 2003
Melbourne, Victoria, July 15th, 2003
Workshop Notes

DALT 2003 Home Page:
http://centria.di.fct.unl.pt/~jleite/dalt03/index.htm

Preface

Agent metaphors and technologies are increasingly adopted to harness and govern the complexity of today's systems. As a consequence, the growing complexity of agent systems calls for models and technologies that promote system predictability, and enable feature discovery and verification.

Formal methods and declarative technologies have recently witnessed a growing interest as a means to address such issues.

The aim of this workshop's call was to foster a discussion forum to export such techniques into the broader community of agent researchers and practitioners and, on the other hand, to bring in the issues of real-world, complex, and possibly large-scale agent system design in the perspective of formal methods and declarative technologies.

The workshop received 17 submissions, of which 11 were selected to be included in this volume, after careful review. We grouped the contributions in three parts: (*i*) software engineering and MAS prototyping, (*ii*) social aspects of MAS, and (*iii*) agent reasoning, BDI logics and extensions. The workshop discussion will be organized around these three main topics, and it will include both paper presentations and free discussions on workshop proposals and results, as well as on the open issues and perspectives of the workshop's themes. Since DALT focuses on a hot, emergent research topic, our idea is to reserve a considerable amount of the total workshop time open to discussions.

After the workshop, accepted papers will be further extended to incorporate workshop discussion, and reviewed for inclusion in the DALT Post-Proceedings, to be published as a volume in the LNCS/LNAI series by Springer-Verlag

We want to take this opportunity to thank the authors who answered our call with high quality contributions, and the members of the program committee for ensuring the quality of the workshop program by kindly offering their time and expertise so that each paper could undergo quadruple reviewing.

May 23rd, 2003

Joao A. Leite
Andrea Omicini
Leon Sterling
Paolo Torroni

DALT 2003 Program committee

Rafael H. Bordini, The University of Liverpool, UK
Jeff Bradshaw, The University of West Florida, FL, USA
Antonio Brogi, Università di Pisa, Italy
Stefania Costantini, Università degli Studi di L'Aquila, Italy
Yves Demazeau, Institut IMAG, Grenoble, France
Jürgen Dix , The University of Manchester, UK
Toru Ishida, Kyoto University, Japan
Catholijn Jonker, Vrije Universiteit Amsterdam, The Netherlands
Antonis Kakas, University of Cyprus, Cyprus
Daniel Kudenko, University of York, UK
Alessio Lomuscio, King's College, London, UK
Viviana Mascardi, Università degli Studi di Genova, Italy
Paola Mello, Università di Bologna, Italy
John Jules Ch. Meyer, Universiteit Utrecht, The Netherlands
Charles L. Ortiz, SRI International, Menlo Park, CA, USA
Sascha Ossowski, Universidad Rey Juan Carlos, Madrid, Spain
Luís Moniz Pereira, Universidade Nova de Lisboa, Portugal
Jeremy Pitt, Imperial College, London, UK
Ken Satoh, National Institute of Informatics, Tokyo, Japan
Michael Schroeder, City University, London, UK
Onn Shehory, IBM Research Lab in Haifa, Israel
Carles Sierra, Spanish Research Council, Barcelona, Spain
V.S. Subrahmanian, University of Maryland, MD, USA
Francesca Toni, Imperial College, London, UK
Wiebe van der Hoek, The University of Liverpool, UK
Franco Zambonelli, Università di Modena e Reggio Emilia, Italy

DALT 2003 Organizers

João A. Leite, Universidade Nova de Lisboa, Portugal
Andrea Omicini, Università di Bologna / Cesena, Italy
Leon Sterling, University of Melbourne
Paolo Torroni, Università di Bologna, Italy

Table of Contents

Software Engineering and MAS Prototyping

Social Aspects of MAS

Agent Reasoning, BDI Logics and Extensions

An Agent-based Domain Specific Framework for Rapid Prototyping of Applications in Evolutionary Biology

T.C. Son[1] E. Pontelli[1] D. Ranjan[1] B. Milligan[2] G. Gupta[3]

[1] Department of Computer Science
New Mexico State University
{tson,epontell,dranjan}@cs.nmsu.edu

[2] Department of Biology
New Mexico State University
brook@biology.nmsu.edu

[3] Department of Computer Science
University of Texas at Dallas
gupta@utdallas.edu

Abstract. In this paper we present a brief overview of the ΦLOG project, aimed at the development of a domain specific framework for the rapid prototyping of applications in evolutionary biology. This includes the development of a domain specific language, called ΦLOG, and an agent-based implementation for the monitoring and execution of ΦLOG's programs. A ΦLOG program—representing an intended application from an evolutionary biologist—is a specification of *what to do* to achieve her/his goal. The execution and monitoring component of our system will automatically figure out *how to do* it. We achieve that by viewing the available bioinformatic tools and data repositories as *web services* and casting the problem of execution of a sequence of bioinformatic services (possibly with loops, branches, and conditionals, specified by biologists) as the web services composition problem.

1 Introduction and Motivation

In many fields of science, data is accumulating much faster than our ability to convert it into meaningful knowledge. This is perhaps nowhere more true than in the *biological sciences* where the Human Genome Project and related activities have flooded our databases with molecular data. The size of the DNA sequence database (e.g., at NCBI), for example, has surpassed 15 million sequences and 17 billion nucleotides, and is growing rapidly. Our modeling tools are woefully inadequate for the task of integrating all that information into the rest of biology, preventing scientists to effectively take advantage of these data in drawing meaningful biological inferences. Thus, one of the major challenges faced by computer scientists and biologists *together* is the enhancement of information technology suitable for modeling a diversity of biological relationships and processes, leading to a greater *understanding* from the influx of data. Instead of allowing the direct expression of high-level concepts natural to a scientific discipline, current software development techniques require mastery of computer science and access to very low level aspects of software development in order to construct significantly complex applications. Even in places where attempts to introduce domain-specific concepts have been made—e.g., design of database formats—scientists are hampered in their efforts by complex issues of interoperation. As a result, currently only biologists

with strong quantitative skills and high computer literacy can realistically be expected to undertake the task of transforming the massive amounts of available data into real knowledge. Very few scientists (domain experts) have such computing skills; even if they do, their skills are better utilized in dealing with high-level scientific models than low-level programming issues. To enable scientists to effectively use computers, we need a well-developed methodology, that allows a domain expert (e.g., a biologist) to solve a problem on a computer by developing and programming solutions at the same level of abstraction they are used to think and reason, thus moving the task of programming from software professionals to the domain experts, the end-users of information technology. This approach to software engineering is commonly referred to as *Domain Specific Languages* and it has been advocated by many researchers over the years [29]. The relevance of domain-specific approaches to bioinformatics has been underlined by many recent proposals (both in computer science as well as in biology) [14, 8, 3, 15, 2]. Domain-specific languages like ΦLOG offer biologists with work-benches for the rapid exploration of ideas and experiments, without the burden of low-level coding of data and processes and interoperation between existing software tools.

In this project we investigate the *design*, *development*, and *application* of a *Domain Specific Language (DSL)*, called ΦLOG, for rapid prototyping of bioinformatic applications in the area of phylogenetic inference and evolutionary biology. Phylogenetic inference involves study of evolutionary change of traits (genomic sequences, morphology, physiology, behavior, etc.) in the context of biological entities (genes, individuals, species, higher taxa, etc.) related to each other by a phylogenetic tree or genealogy depicting the set of common ancestors. It finds important applications in areas such as study of ecology and dynamics of viruses. To be attractive to biologists, an effective DSL should provide: (1) descriptions of the concepts and operations naturally associated with biology (e.g., data sources, types of data, transformations of the data), (2) mechanisms allowing users to manipulate those concepts in a compact, intuitive manner at a high level of abstraction, (3) models specialized enough to reflect the real biological processes of interest, and (4) efficient execution mechanisms that do not require extensive intervention and programming by the end user. Furthermore, a large class of biological models integrates information on relationships among organisms, homology of traits, and specifications of evolutionary change in traits; this commonality can be used to advantage in designing the structure of domain-specific representations and transformations of biological data. Information technology based on the major commonality evident in problems explicitly involving relationships, homology, and trait evolution can readily be expanded to incorporate a much broader range of biological models. To date no software development environment or methodology available to biologists has identified all of these elements and explicitly designed uniform solutions incorporating them. Instead, there exist a large array of mostly ad hoc technologies. Existing tools provide monolithic interfaces (rather than libraries encapsulating the basic computational elements from which larger constructions can be built) and a black box structure, that does not provide access to the underline mechanisms and heuristics [28] used to solve biological problems.

ΦLOG is part of a comprehensive computational framework, based on agent technology, capable of harnessing local, national, and international data repositories and

computational resources to make the required modeling activities feasible. Solving a typical problem in phylogenetic inference requires the use of a number of different bioinformatic tools, the execution of a number of manual steps (e.g., judging which sequence alignment for two genes is the "best"), and extra low-level coding to glue everything together (e.g., low-level scripting). An important characteristic of the ΦLOG framework is its ability to *interoperate* with the existing biological databases and bioinformatic tools commonly used in phylogenetic processing, e.g., CLUSTAL W, BLAST, PHYLIP, PAUP. These existing tools and data repositories are treated as *semantic Web services*, automatically accessed by ΦLOG to develop the solution requested by the domain expert. From a semantic point of view, these existing components are regarded as semantic algebras [23], used for defining the valuation predicates in the denotational specification of the DSL. Thus, ΦLOG provides a uniform language through which biologists can perform complex computations (involving one or more of these software systems) without much effort. In absence of such a DSL, biologists are required to perform significant manual efforts (e.g., locating and accessing tools, determine adequate input/output data formats) and to write considerable amount of glue code.

The execution model of ΦLOG is built on an agent infrastructure, capable of transparently determining the bioinformatic services needed to solve the problem and the data transformations that are required to seamlessly stream the data between such components during the execution of a ΦLOG program. Bioinformatic services are viewed as actions in situation calculus, and the problem of deriving a correct sequence of service invocations is reduced to the problem of deriving a successful plan. The agent infrastructure relies on a *service broker* for the discovery of bioinformatic services, and each agent makes use of logic-based planner for composition and monitoring of services. The framework implements typical agents' behaviors, including planning, interaction, and interoperation.

2 The ΦLOG System

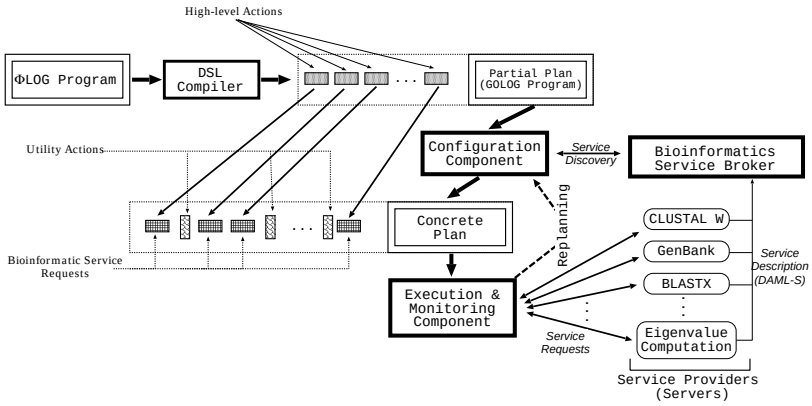

Fig. 1. System Organization

ΦLOG is based on a comprehensive agent-based platform, illustrated in Figure 1.

The higher level is represented by the ΦLOG language, a DSL specifically designed for evolutionary biologists—described in Section 3.

The execution of each ΦLOG program is supported by a *DSL compiler*—described in Section 4—and an *Execution Agent*—described in Section 5. The execution agent is, in turn, composed of a *configuration component* and an *execution/monitoring component*. In this framework, bioinformatic services are viewed as *actions*, and execution of ΦLOG programs as an instance of the planning problem. The compiler translates each ΦLOG program into a *partial plan*—specifically a GOLOG program [13]; this describes the steps required to execute the ΦLOG program in terms of high-level actions and their sequencing. The plan is considered partial for various reasons: *(i)* each high-level action has to be resolved into invocation of actual bioinformatic software tools and data sources; *(ii)* interaction between successive steps in the plan may require the introduction of intermediate low-level actions (e.g., interoperation between existing tools). The actual execution requires transformation of the partial plan into a *concrete plan*—whose (low-level) actions are actual accesses to the data repositories and execution of bioinformatic tools. This transformation is accomplished by the configuration component of the agent, via a planning process. This planning process is performed in cooperation with a *service broker*, which supplies description and location of the existing data sources and software tools. The configuration agent makes use of these services descriptions to develop the action theory needed to generate the concrete plan.

The execution of the concrete plan is carried out by the execution/monitoring component of the agent. Execution involves contacting data sources and software tools and requesting the appropriate execution steps. Monitoring is required to validate progress of the execution and re-enter the planning phase to repair eventual execution failures. In the successive sections we highlight the relevant aspects of the various components and the research challenges to be tackled.

3 The Design of the ΦLOG Language

In this section we propose a preliminary design of the ΦLOG language. More details regarding this initial design can be found in [20]. We use problems from biology as motivating examples. These examples relate to one of the most challenging problems in biology: that of determining the evolution of species. The evolution of a set of species can naturally be represented by an *evolutionary tree* with leaf nodes representing sampled species, interior nodes representing ancestors, and edges representing the ancestor-descendant relationship in the usual fashion. Given a set of related species there is a multitude of possibilities as to how they evolved. The goal is to use the biological data to determine the most likely evolutionary history. One way to determine how a set of related species evolved is based on modeling DNA sequence data using a stochastic model of evolutionary change [28]. The evolutionary tree that "best" fits this model and data is adopted as the most likely evolutionary history. The starting point of this process is represented by the collection of similar DNA sequences for the set of species of interest from the huge set of DNA sequence data that is stored in databases like GSDB and GenBank. In its simplest form, both the set of taxa and the set of genes are completely specified, and the task is simply to determine occurrences (i.e., sequences) of the genes in the taxa. Matching sequences are determined by comparing the given genes with the sequences belonging to each taxon, and applying a set of filtering criteria. This result is typically constructed by iterating (manually or using ad-hoc scripts) the application of *similarity* search programs—e.g., the *Basic Local Alignment Search Tool (BLAST)*—

using the provided genes as input and (manually) filtering the output with respect to the taxa of interest. During each iteration one of the genes is used to detect reasonable matches against a sequence database. The resulting matches have to be filtered to extract only matches relative to the taxa of interest and to remove false matches.

The successive step is to find the most likely evolutionary tree for a given set of species from the given DNA sequence data and a model of evolution [28]. The current methodology to solve this problem is to: *(i)* align the input sequences—sequence alignment can be performed using a standard tool for multiple sequence alignment, such as CLUSTAL W—and *(ii)* use the aligned sequences and the given model(s) to generate and rank possible phylogenetic trees for the sequences, using tools such as PHYLIP and PAUP. Users are responsible for proper pipelined execution of all the components (including data format translation, if needed). Furthermore, most tree building software uses a very limited set of evolutionary models, and existing software considers only limited parameter optimizations for parametric models [28].

3.1 Preliminary DSL

In this section we provide a brief overview of the ΦLOG language; for a more complete description the interested reader is referred to [20].

Overall Program Structure: ΦLOG programs consist of modules. Each module contains a collection of global declarations and a collection of procedures. In turn, each procedure contains a sequence of declarations and a sequence of instructions. The declaration part of each procedure is used *(i)* to describe the data items used by the procedure, *(ii)* to allow user selection of the computational components to be used during execution; and *(iii)* to provide parameters affecting the behavior of the different components. Data items used in the program must be declared. Declarations are used to explicitly describe data items, by providing a name (`<item name>`), a description of the nature of the values that are going to be stored in it (`<item type>`) and eventual properties of the item. E.g., `gene1 : Gene ( gi | 557882 )` declares an entity called `gene1`, of type `Gene`, and identifies the initial value for this object—the gene which has GI number 557882 in the GenBank database. Declarations are also used to identify computational components to be used during the execution—which allows the user to customize some of the operations performed. E.g., the declaration

```
similar: operation (BLASTX -- alignment=ungapped
         database=drosophila matrix=BLOSUM45 )
```

allows the user to explicitly configure the behavior of the language operation `similar`—by associating this operation with the BLAST similarity search program.

Data Types: The design of the collection of data types of ΦLOG has been driven by observing the commonalities present between various languages for biological data description proposed in the literature (e.g., BSML and NEXUS [14]). ΦLOG provides two classes of data types that can be used to create data items. The first class includes generic (non-domain specific) data types, while the second class provides a number of *Domain Specific Data Types*, which are relevant for the specific domain. E.g.,

- `Sequence, Gene, and Taxon:` These data types are used to describe molecular sequence data (e.g., DNA, RNA), Genes, and Taxons (e.g., species along with their traits). The sequence data type allows users to describe the sequence in differ-

ent ways, e.g., by providing its description in the standard formats. Using the FASTA format one could describe a sequence as follows:

```
g : Sequence (protein)
g is FOSB_HUMAN P53539 homo sapiens (HUMAN).
      MFQAFPGDYDSGSRCSSSSPSAESQYLSSVDSFGPPT...
```

Transformation between the different formats is transparent. Each object of type `Gene` and `Taxon` is characterized by a number of (optional and mandatory) attributes, including name, accession number, GI number, and sequence data [14]. The actual set of attributes depends on the detail of the description provided by the user and/or on the existing attributes present in the gene database used. For example:

```
g1 : Gene is (gi | 557882)
se is sequence(g1)
```

assigns to the item `g1` the gene having GI number `gi|557882` and extracts its sequence data, which is stored in the item `se`. The `Taxon` data type is polymorphic in the sense that items of type `Taxon` can be used to represent higher-order taxa as well, which are interpreted in the language as (hierarchically organized) sets of taxa.

- `Model`: A model data type is used to describe models of evolution, used to perform inference of phylogenies [28]. In the simple case, a model is a matrix containing the individual evolutionary rates:

  `t : Model is ({A` $\rightarrow$ `(0.1)C, A` $\rightarrow$ `(0.04)T,` $\cdots$ `}).`

  ΦLOG allows symbolic description of models [20] and access to standard models [28], e.g., `t1 is K80(kappa).`

ΦLOG also provides a number of polymorphic data types which are used to aggregate in different ways collection of entities. ΦLOG allows operations on `Trees` (described using Newick format) and `Sets`. In particular, various externally accessible sources of information are mapped in the DSL as sets of elements. For example, external databases, e.g., GenBank, can be accessed in the language using traditional set operations, e.g.,

```
{name(x) | x: Gene, x in GenBank}
```

In ΦLOG a `Map` represents a function which maps elements of a domain into elements of another domain. The mapping can be specified either as an enumeration of pairs or using an intensional definition—i.e., the mapping is described via properties. E.g.,

```
match ( tax : Taxon, x : Gene ) : map
    ( S is { y : Sequence | y in genes(tax),
           {    score(similar(x,y)) > threshold }
     if S is empty then match is undefined
     else match is (any y in S) )
```

defines a map `match` which associates with each taxon (from a domain of taxa) the sequence of a gene which is sufficiently similar to the input gene `x`. The `similar` operation is expected to return a data item measuring the similarity of the two sequences; in this example we expect an attribute called `score` to be present. A map is a function—in the example we have a function which maps a taxon and a gene to a sequence. Maps can be used in various ways—in the previous example, if `Ta` is a set of taxa, then `match(Ta,seq)` is a set of sequences (constructed according to the map definition), while an expression such as `select X (seq1 is match(X,seq2))` searches for a taxon `X` containing the sequence `seq1` which is similar to sequence `seq2`. Other poly-

morphic types include the ability to describe probability distributions and homologies [20].

Control Structures: The language provides two levels of control constructs. At the higher level, control of the execution is expressed using declarative constructs, such as function applications and quantifications. Existential quantification (`select`) is used to express search tasks, while universal quantification (`forall`) can be used to express iterations and global properties. In this respect, the structure of the language closely resembles the structure of many functional languages. For example, the code

```
select a in Trees
   forall b in Trees
    likelihood(a,model) ≥ likelihood(b,model)
```

selects a tree `a` from the collection of trees `Trees` that has the maximum likelihood.

At the lower level, the language provides control constructs which are closer to those in traditional programming languages, such as `if-then-else` conditional statements and `repeat` iterative constructs. These are mostly used by users who want to customize the basic operations of the language (as described in the following example). The previous example can be rewritten as (`best, t` are of type `Tree`)

```
best in Trees
repeat (t in Trees)
  if (likelihood(t,model)>likelihood(best,model))
     then best is t
```

The first statement selects as initial value for `best` an arbitrary element of `Trees`, and the loop performs an iteration for each element in `Trees`. The language provides also the capability of asserting constraints on the computation. These constraints are assertions that have to be satisfied at any point during the computation. Constraints are asserted using the `constraint` statement. E.g., if we want the computation to generate a tree `t` whose root has only two children, with branch lengths 10 and 20, then we can use the statement

```
constraint : t is (X : label1 , Y : label2)Z and
             label1 is 10 and label2 is 20
```

The execution model adopted by the language provides the user with both batch and interactive execution. Under batch mode, programs are compiled and completely executed. Under interactive mode, programs are executed incrementally under user supervision. The user is allowed to select breakpoints in the execution, run the program statement by statement, and modify *both the data and the program* on the fly. For example, the user is allowed to introduce new constraints *during* the execution, thus modifying on the fly the behavior of the computation. Another feature that ΦLOG provides is the ability to make data items *persistent*, allowing users to create with no extra effort databases containing the results of the computations, and to share partial results.

3.2 Some Examples Coded in ΦLOG

Let us start by looking at how we can express the problem of, given a collection of taxa and a collection of genes, determining in which taxa those genes occur. Both the set of taxa and the set of genes can be either explicitly provided by the user or the result of some computation. Regarding the selection of taxa, we need to determine a set of taxa of interest, eventually involving higher-order taxa. This set could be the result of some computation, e.g., to select 40 taxa out of a given higher-order taxon:

```
Tax is {y : Taxon | y in murinae} and |Tax|=40
```
or, given a collection of higher-order taxa (`InputSet`), select a taxon from each of them: `Tax is union ( x in InputSet , t is (any in x) ).`
Regarding the selection of the genes of interest, this can be either a user defined collection of genes or it could be itself the result of some computation. E.g., to select all genes from a given set of taxa (`T`) which contain a certain name:
```
Gen is {x : Gene | taxa in T, x in genes(taxa),
                   name(x) contains "Adh" }
```
Once a set of taxa (`Tax`) and a set of genes (`Gen`) have been identified, then we would like to determine occurrences of the identified genes in the taxa of interest—this can be accomplished using the `match` map defined earlier. The selection of the components of the map requires a filtering of the result of the similarity search. E.g., if we are interested in defining the map in order to produce the sequence with the longest aligned region, then we simply replace the `any` statement with
```
w in S and forall z in S
  length_alignment(similar(y,w))≥length_alignment(similar(y,z))
```
As part of the definition of the `Sequence` data type, the language provides an operation, `align`, which provides the sequence alignment capabilities. The `align` operation accepts a single argument which represents the set of sequences to be aligned. The result of the operation is a set of sequences, containing the original sequences properly expanded to represent the desired alignment. The behavior of `align` can be customized by the user similarly to what is described in the case of `similar`. E.g., the declaration:
```
align : operation (CLUSTAL W -- model=PAM)
```
asserts that the `align` operation should be performed by accessing the CLUSTAL W software with the appropriate parameters. We envision generalizing the behavior of `align` to produce as result not just a set of aligned sequences but a more general object—a data item of type `Homology`. The next step requires the definition of the model which is going to be used to describe evolutionary rates—i.e., a data item of the type `Model`. At the highest level, we can assume the presence of a `build_tree` operation which directly interfaces to dedicated tools for inference of phylogenies:
```
  build_tree : Operation ( DNAML -- )
  t : Tree is build_tree(Seqs,model1)
```
This reflects the current standard approach, based on the development of a tree using a single model across the entire sequence and the entire tree [28]. The operation `build_tree` can be redesigned by the user whenever a different behavior is required. A simple declarative way of achieving this is as follows: given a list `list`
```
constraint : forall [X,Y] in list
  (likelihood(X,model)≥likelihood(Y,model))
list is [ t : Tree | t in Tree(Seqs) ]
```
which expresses the fact that `list` contains all the trees over the sequences in the set `Seqs`, sorted according to the likelihood of each tree under the model `model`. The best tree is the first in the list. A finer degree of control can be obtained by switching to the imperative constructs of the language and writing explicit code for the search. E.g., the following ΦLOG code picks the best of the first 1000 trees generated:
```
t is initialTree(Seqs)
best is t
```

```
  repeat (i from 1 to 1000)
    t is nextTree(t,Seqs,model)
    if likelihood(t,model)>likelihood(best,model)
      then best is t
```

By providing different definitions of the operations `initialTree` and `nextTree` it is possible to customize the search for the desired tree. The same mechanisms can be used to select a set of trees instead of just one.

Given a set of trees `Trees` computed from a set of sequences and given a model `model`, we can construct a relative likelihood for the given set of trees:

```
prob : Probability(Trees)
total is summation(x in Trees,likelihood(x,model))
     forall x in dom(prob)
        prob(x) is likelihood(x,model)/total
```

This model can be easily extended to accommodate a distribution of models instead of an individual model. This allows us to use expected likelihood for the evaluation and selection of the trees. Assuming a finite collection of models `Models` and an associated probability distribution `prob`, we can replace the `likelihood` function by the `fit` function defined below:

```
fit (t : Tree) : map
  (fit is sum(m in Models,prob(m)*likelihood(t,m)))
```

4 Compilation of ΦLOG Programs

The goal of the ΦLOG compiler is to translate ΦLOG programs—manually developed by a biologist or developed via high-level graphical interfaces [20]—into partial plans. The components of the partial plan are high-level actions, extracted from an ontology of bioinformatic operations; each high-level operation will be successively concretized into one or more invocations of bioinformatic services (by the execution agent).

4.1 Bioinformatic Services

Data sources and software tools employed to accomplish phylogenetic inference tasks are uniformly viewed by ΦLOG as *bioinformatic services*. Each service provides a uniform interface to the data repository or software tool along with a description of the functionalities of the service (e.g., inputs, outputs, capabilities). Service descriptions are represented using a standard notation for service description in the semantic web (specifically DAML-S [5]). Service providers register their services with the *services broker*. The task of the broker is to maintain a directory of active services (including the location of the service and its description) and to provide matchmaking services between service descriptions and service requests.

The creation and management of service descriptions require the presence of a very refined *ontology*, describing all entities involved in service executions. Ontologies provide an objective specification of domain information, representing a community-wide consensus on entities and relations characterizing knowledge within a domain. The broker employs the ontology to instantiate high-level requests incoming from configuration agents into actual service requests. Considerable work has been done in the development of formal ontologies for biological concepts and entities [26, 1]. Our project will

build on these efforts, taking advantage of the integrated description of biological entities provided in these existing proposals. Nevertheless, this project aims at covering aspects that most of these ontologies currently do not provide:

(i) Description of an actual hierarchy of *bioinformatic operations*, their relationships, and their links to biological entities; each operation is described in terms of input and output types as well as its effect.

(ii) Description of an actual hierarchy for *bioinformatic types* and *bioinformatic data representation formats*, their relationships, and the links to biological entities and to the bioinformatic applications.

Thus, our objective is to concretize biological ontologies by introducing an ontology level describing the data formats and transformations that are commonly employed in phylogenetic inference, and linking them to the existing biology ontologies. This additional level (i.e., an ontology for bioinformatic tools) is fundamental to effectively accomplish the instantiation of a partial plan into a concrete plan—e.g., automatically selecting appropriate data format conversion services for interoperation.

Standard semantic web services description languages—i.e., DAML-S—have been employed for the development of these ontologies.

Service descriptions as well as the bioinformatic ontologies are maintained and managed by a service broker. In this project, the service broker is developed using the *Open Agent Architecture (OAA)* [16]. This will allow us to apply previously developed techniques in web services composition and GOLOG programs's execution and monitoring in [17] in our application. The advantages of this approach have been discussed in [18]. Currently, we have developed a minimal set of service descriptions that will allow us to develop simple ΦLOG programs that will be used as testbeds for the development of other components of the systems. We use OAA in this initial phase as it provides us the basic features that we need for the development of other components. In the later phases of the project, we will evaluate alternative architectures with similar capabilities (e.g. InfoSleuth [7]) or multi-agent architectures (e.g. RETSINA [27] or MINERVA [12]).

4.2 ΦLOG Compiler

The derivation of the ΦLOG compiler has been obtained using a semantic-based framework called *Horn-Logic Denotations (HLD)* [9]. In this framework, the syntax and semantics of the DSL is expressed using a form of denotational semantics and encoded using an expressive and tractable subset of first-order logic (Horn clauses). Following traditional denotational semantic specifications, in HLD a DSL $\mathcal{L}$ is described by three components: *(i)* syntax specification—realized by encoding a context free grammars as Horn clauses (using the Declarative Clause Grammars commonly used in logic programming); *(ii)* interpretation domains—described as logic theories; *(iii)* valuation functions—mappings from syntax structures to interpretation domains (encoded as first-order relations). In our specific application, the interpretation domains are composed of formulae in an action theory [21], describing properties and relationships between bioinformatic services. The action theories are encoded using situation calculus [21]—see also Section 5—and they are extracted by the compiler from the services ontologies described in the previous section. The semantic specification also allows for rapid and

correct implementations of the DSL. This is possible thanks to the use of an encoding in formal logic and the employment of logic-based inference systems—i.e., the specifications are *executable*, automatically yielding an interpreter for the DSL. Moreover, the Second Futamura Projection [10] proves that compiled code can be obtained by *partially evaluating* an interpreter w.r.t. a source program. Thus, our DSL interpreter can be partially evaluated (using a partial evaluator for logic programming [22]) w.r.t. a program expressed in the DSL to obtain compiled code [9]. In our context, the final outcome of the process is the automatic transformation of specifications written in ΦLOG to programs written in GOLOG.

The semantic specification can also be extended to support verification and debugging: the denotational specifications provide explicit representation of the state [23, 9], and manipulation of such information can be expressed as an alternative semantics of the DSL—i.e., as an *abstract semantics* of the DSL [9]. This verification process is also possible thanks to the existence of *sound and complete* inference systems for meaningful fragments of Horn clause logic [19]. Given that the interpreter, compiler, and verifiers are obtained directly from the DSL specification, the process of developing and maintaining the development infrastructure for the DSL is very rapid. Furthermore, syntax and semantic specifications are expressed in a *uniform* notation, backed by effective inference models.

5 Execution Agent

5.1 ΦLOG's Program Execution as Planning and Plan Execution Monitoring

A ΦLOG program specifies the general steps that need to be performed in a phylogenetic inference application. Some steps can be achieved using built-in operations—this is the case for the primitive operations associated to the various data types provided by ΦLOG—e.g., compose two sets using a union operation or selecting the members of a set that satisfy a certain condition. Other steps might involve the use of a bioinformatic service, e.g., CLUSTAL W for sequence alignment, BLAST for similarity search, PAUP for phylogeny construction, etc. In several cases, intermediate steps, not specified by the compiler, are required; this may occur whenever the ΦLOG program does not explicitly lay out all the high-level steps required, or whenever additional steps are required to accomplish interoperability between the bioinformatic services. For example, the sequence alignment formats provided by CLUSTAL W cannot be used directly as inputs to the phylogenetic tree inference tool PAUP (which expects a Nexus file as input). As such, a ΦLOG program could be viewed as a skeleton of an application rather than its detailed step-by-step execution. Under this view, *execution of ΦLOG programs could be viewed as an instance of the planning and plan execution monitoring problem*. This will allow us to apply the techniques which have been developed in that area to implement the ΦLOG engine. We employ the approach introduced in [17] in developing the ΦLOG system. In this approach, each bioinformatic service is viewed as an *action* and a phylogenetic inference application as a GOLOG program [13].

We will now review the basics of GOLOG and the agent architecture that will be used in the development of the ΦLOG system. Since GOLOG is built on top of the situation calculus (e.g.,[21]), we begin with a short review of situation calculus.

Situation Calculus: The basic components of the situation calculus language, following the notation of [21], include a special constant S_0, denoting the initial situation, a

binary function symbol do where $do(a, s)$ denotes the successor situation to s resulting from executing the action a, fluent relations of the form $f(s)$, denoting the fact that the fluent f is true in the situation s, and a special predicate $Poss(a, s)$ denoting the fact that the action a is executable in the situation s.

A dynamic domain can be represented by a theory containing: (i) axioms describing the initial situation S_0; (ii) action precondition axioms (one for each action a, characterizing $Poss(a, s)$); (iii) successor state axioms (one for each fluent F, stating under what condition $F(\boldsymbol{x}, do(a, s))$ holds, as a function of what holds in s); (iv) unique name axioms for the primitive actions; and some foundational, domain independent axioms.

GOLOG: The constructs of GOLOG [13] are:

α	*primitive action*	$\phi?$	*wait for a condition*
$(\sigma_1; \sigma_2)$	*sequence*	$(\sigma_1 \mid \sigma_2)$	*choice between actions*
$\pi x.\sigma$	*choice of arguments*	σ^*	*nondeterministic iteration*
if ϕ **then** σ_1 **else** σ_2	*synchronized conditional*	**while** ϕ **do** σ	*synchronized loop*
proc $\beta(\boldsymbol{x})\sigma$	*procedure definition*		

The semantics of GOLOG is described by a formula $Do(\delta, s, s')$, where δ is a program, and s and s' are situations. Intuitively, $Do(\delta, s, s')$ holds whenever the situation s' is a terminating situation of an execution of δ starting from the situation s. For example, if $\delta = a; b \mid c; f?$, and $f(do(b, do(a, s)))$ holds, then the GOLOG interpreter will determine that $a; b$ is a successful execution of δ in the situation s.

The language GOLOG has been extended with various concurrency constructs, leading to the language *ConGolog* [6]. The precise semantic definitions of GOLOG and ConGolog can be found in [13, 6]. Various extensions and implementations of GOLOG and ConGolog can be found at `http://www.cs.toronto.edu/~cogrobo` web site. An *answer set programming* interpreter for GOLOG has been implemented [24].

Bioinformatic Services as Actions and ΦLOG's Programs as GOLOG Programs: A bioinformatic service is a web service—i.e., a computational service accessible via the Web. Adopting the view of considering web services as actions for Web service composition application [17], we view each bioinformatic service as an action in situation calculus. Roughly speaking, an action description of a web service consists of the service name, its invocation's description, and its input and output parameters with their respective formats. Let $\mathcal{D}$ be the set of actions representing the bioinformatic services. Under this framework, the problem of combining bioinformatic services to develop a phylogenetic inference application is a planning problem in $\mathcal{D}$. Since $\mathcal{D}$ contains all the external operations that a user of ΦLOG can use, each ΦLOG program P can be compiled into a GOLOG program in the $\mathcal{D}$ language by:

- Introducing variables and fluents representing the variables of the ΦLOG program.
- Replacing each instruction `V is operation(`$\boldsymbol{x}$`)` in the ΦLOG program with the action $\tau($`operation`$)(\boldsymbol{x}, V)$, where τ is a mapping that associates ΦLOG operations to invocations of bioinformatic services. The ΦLOG compiler implements the τ operation through the following steps:
  - terms in the ΦLOG program are used to identify the set of *high-level actions* requested by the ΦLOG programmer. The process is accomplished by identifying relevant entries in the hierarchy of bioinformatic operations (see Section 4.1).

  - high-level operations are used to query the services broker and retrieve the description of relevant registered services that implement the required operations.
- Replacing the control constructs such as **if-then, for-all**, etc. in the ΦLOG program with their corresponding constructs in GOLOG.

In turn, the description of the various services retrieved from the services broker (DAML-S descriptions) are converted [17] into action precondition axioms and successor state axioms. This translation will be achieved by the compiler of the ΦLOG system. The GOLOG program obtained through this translation may contain the *order* construct (denoted by $\sigma_1 : \sigma_2$), an extended feature of GOLOG suggested in [17], and used to describe a partial order between parts of the program. This feature is essential in our context; the ΦLOG program determines the ordering between the main steps of the computation (expressed using the order construct), while the configuration agent may need to insert additional intermediate steps to ensure interoperation between services and executability of the plan. E.g., if a ΦLOG program indicates the need to perform a sequence alignment (mapped to the `clustalw` service) followed by a tree construction (mapped to the `paup` service), then the GOLOG program will contain `clustalw : paup`; the agent may transform this into the plan `clustalw; parse_nexus; paup`, by inserting a data format convertion action (`parse_nexus`). Using a GOLOG interpreter we can find different sequences of services which can be executed to achieve the goal of the ΦLOG program, provided that $\mathcal{D}$ is given. Precise details in the translation and execution monitoring under development.

5.2 Configuration Component

The configuration component is in charge of transforming a partial plan (expressed as a GOLOG program with extended features, as described in [17]) into a concrete plan (*instantiation*). This process is a *planning problem* [24], where the goal is to develop a concrete plan which meets the following requirements: *(i)* each high-level action in the partial plan is instantiated into one or more low-level actions in the concrete plan, whose global effect correspond to the effect of the high-level action (i.e., the τ operation mentioned earlier); *(ii)* successive steps in the concrete plan correctly interoperate. This process is intuitively illustrated in Example 1. The configuration component makes use of the high-level actions in the partial plan to query the services broker and obtain lists of concrete bioinformatic services that can satisfy the requested actions. The querying is realized through the use of the bioinformatic tools ontology mentioned earlier.

The configuration component attempts to combine these services into an effective concrete plan—i.e., an executable GOLOG program. The process requires fairly complex planning methodologies, since:

- the agent may need to repeatedly backtrack and choose alternative services and/or add intermediate additional services (e.g., filtering and data format transformation services) to create a coherent concrete plan;
- planning may fail if some of the requested actions do not correspond to any service or services cannot be properly assembled (e.g., lack of proper interoperation between data formats); sensing actions may be employed to repair the failure, e.g., by cooperating with the user in locating the missing service;
- planning requires the management of *resources*—e.g., management of a budget when using bioinformatic services with access charges;

- planning requires *user preferences*—e.g., choice of preferred final data formats.

Most of these features are either readily available in GOLOG or they have been added by the investigators [17, 24, 25] to fit the needs of similar planning domains.

Example 1. Consider the original simple ΦLog program (g is of type $Genes$, s of type $Alignment(dna)$ and t of type $Tree(dna)$):

```
g is { x : Gene | name(x)  contains  ''martensii''}
s is align(g)
t is phylogenetic_tree(s)
```

The high-level plan detected by the compiler:

```
database_search(gene,[(GeneName,''martensii'')], o1) ;
                    sequence_alignment(dna, o1, o2) ;
                     phylogenetic_tree(dna, o2, o3) ;
                                      display_output(o3)
```

The action theory derived from the agent broker will include:

$genebank(''GeneName = martensii'', o1)$ **causes** $genes(o1, G)$ **if** ...
$clustalw(o1, o2)$ **causes** $alignment(dna, o2, A)$ **if** $sequences(dna, o1, S), \ldots$
$dnaml(o2, o3)$ **causes** $tree(dna, o3, T)$ **if** $alignment(dna, o2, A), \ldots$

Additional actions are automatically derived from the type system of the language; for example, `Gene` is seen as a subclass of `DNA_Sequence`, which provides an operation of the type: $gene\_to\_dnasequence(o)$ **causes** $sequence(dna, o)$ **if** $gene(o)$. The operation will be associated to built-in type conversion actions.

Finally, the high level plan will have to be replaced by a GOLOG program, where each high-level action should be replaced by a choice of actions construct (a choice from the set of the services implementing such high-level action), e.g.,

```
genebank(''Gene Name = martensii'',o1) | ··· :
                      clustalw(o1, o2) | ··· :
                          dnaml(o2,o3) | ··· :
                         display_output(o3) ;
                             ? tree(dna,o3).
```

5.3 Execution and Monitoring Agent

Once a concrete plan has been developed, the execution/monitoring component of the agent proceeds with its execution. Executing the concrete plan corresponds to the creation and execution of the proper service invocations corresponding to each low-level action. Each service request involves contacting the appropriate service provider—which can be either local or remote—and supply the provider with the appropriate parameters to execute the desired service. The monitoring element supervises the successful completion of each service request. In case of failure (e.g., a timeout or a loss of connection to the remote provider), the monitor takes appropriate repair actions. Repair may involve either repeating the execution of the service or re-entering the configuration component. The latter case may lead to exploring alternative ways of instantiating the partial plan, to avoid the failing service. The replanning process is developed in such a way to attempt to reuse as much as possible of the part of the concrete plan executed before the failure.

6 Discussion and Conclusions

Technology: The development of this framework employs a combination of novel and existing software technology. DAML-S [5] is used as representation format for the description of services. In the preliminary prototype, the ontologies for bioinformatic services and for bioinformatic data formats have been encoded using logic-based descriptions—specifically, using the OO extensions provided by SICStus Prolog.

The reasoning part of the agent (configuration, execution, and monitoring) is based on *situation calculus*. This is in agreement with the view adopted by the semantic web community—Web Services, encoded in DAML-S, can be viewed as actions [5]. GOLOG is employed as language for expressing the partial plans derived from ΦLOG programs, as well as describing the complete plans to be executed. In this work we make use of a GOLOG interpreter encoded in answer set programming [11]. The advantage of this approach is that it allows us to easily extend GOLOG to encompass the advanced reasoning features required by the problem at hand—i.e., user preferences [25] and planning with domain specific knowledge [24].

Related Work: An extensive literature exists in the field of DSL [29]. Design, implementation, and maintenance of DSLs have been identified as key issues in DSL-based software development and various methodologies have been proposed (e.g., [4, 29]. HLD is the first approach based on logic programming and denotational semantics, and capable of completely specifying a DSL in a uniform executable language.

Various languages have been proposed to deal with the issue of describing biological data for bioinformatic applications. Existing languages tend to be application-specific and limited in scope, they offer limited modeling options, are mostly static, and are poorly interconnected. Various efforts are undergoing to unify different languages, through markup languages (e.g., GEML and BSML) and/or ontologies [26]. Some proposals have recently emerged to address the issues of interoperability, e.g., [3, 2].

The work that comes closest to ΦLOG includes programming environments which allows scientists to *write programs* to perform computational biology tasks. Examples of these include TAMBIS [3], Darwin [8] and Mesquite [15]. They combine a standard language (imperative in Darwin, visual in Mesquite) with a collection of modules to perform computational biological tasks (e.g., sequence alignments). Both Darwin and Mesquite provide only a small number of models that a biologist can use to create bioinformatic applications, and they both rely on a "closed-box" approach. The modules which perform the basic operations have been explicitly developed as part of the language and there is little scope for integration of popular bioinformatic tools. TAMBIS provides a knowledge base for mapping graphically expressed queries to accesses of a set of bioinformatic data sources.

Conclusions and Future Work: In this paper we presented a brief overview of the ΦLOG project, aimed at the development of a domain specific framework for the rapid prototyping of applications in evolutionary biology. The framework is based on a DSL, that allows evolutionary biologists to express complex phylogenetic analysis processes at a very high level of abstraction. The execution model of ΦLOG relies on an agent infrastructure, capable of automatically composing and monitoring the execution of bioinformatic services, to accomplish the goals expressed in the original ΦLOG pro-

gram. The framework is currently under development as a collaboration between researchers in Computer Science, Biology, and Biochemistry at NMSU.

References

1. Bio-Ontologies Consortium. `www.bioontology.org`.
2. Standardizing Biological Data Interchange Through Web Services. `omnigene.sourceforge.net`, 2001.
3. P. Baker et al. Transparent Access to Bioinformatics Information Sources. *Intelligent Systems for Molecular Biology*, 1998.
4. J. Bentley. Programming pearls: Little languages. *Communications ACM*, 29(8), 1986.
5. DAML-S Coalition. DAML-S: Semantic markup for Web services. *Int. Semantic Web Working Symposium*, 2001.
6. G. De Giacomo, Y. Lespérance, and H. Levesque. ConGolog, a concurrent programming language based on the situation calculus. Artificial Intelligence, 121:(1-2), 109-169.
7. J. Fowler, B. Perry, M. Nodine, and B. Bargmeyer. Agent-Based Semantic Interoperability in InfoSleuth. SIGMOD Record 28:1, March, 1999, pp. 60-67.
8. G. Gonnet and M. Hallet. *Darwin 2.0*. ETH-Zurich, 2000.
9. G. Gupta and E. Pontelli. A Horn denotational framework for specification, implementation, and verification of DSLs. In *Logic Programming and Beyond*, Springer Verlag, 2002.
10. N. Jones. Introduction to Partial Evaluation. *ACM Computing Surveys*, 28(3):480–503, 1996.
11. V. Lifschitz. Answer set planning. *ICLP*, MIT Press, 1999.
12. J. A. Leite, J. J. Alferes and L. M. Pereira. MINERVA - A Dynamic Logic Programming Agent Architecture. In *Intelligent Agents VIII*, pages 141-157, Springer-Verlag, 2002.
13. H. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–84, 1997.
14. D. R. Maddison et al. NEXUS: An Extensible File Format for Systematic Information. *Syst. Biol.*, 464(4), 1997.
15. W. Maddison. Mesquite: A Modular System for Evolutionary Analysis. U. Arizona, 2000.
16. D. L. Martin, A. J. Cheyer, and D. B. Moran. The Open Agent Architecture: a Framework for Building Distributed Software Systems. *Applied Artificial Intelligence*, 13:91–128, 1999.
17. S. McIlraith and T.C. Son. Adapting Golog for Composition of Semantic Web Services. In *International Conference on Principles of Knowledge Representation and Reasoning*, 2002.
18. S. McIlraith, T.C. Son, and H. Zeng. Semantic Web Services. IEEE Intelligent Systems. Special Issue on the Semantic Web. March/April, 2001.
19. I. Niemela and P. Simons. An Implementation of the Stable Model Semantics. In *LPNMR*, Springer Verlag, 1997.
20. E. Pontelli et al. Design and Implementation of a Domain Specific Language for Phylogenetic Inference. In *Journal of Bioinformatics and Computational Biology*, 1(2):1–29, 2003.
21. R. Reiter. *Knowledge in Action.* MIT Press, 2001.
22. D. Sahlin. *An Automatic Partial Evaluator for Prolog*. PhD, Uppsala, 1994.
23. D. Schmidt. *Denotational Semantics.* W.C. Brown, 1986.
24. T.C. Son et al. Planning with Different Forms of Domain-Dependent Control Knowledge. Approach. *Int. Conf. on Logic Progr. and Nonmonotonic Reasoning*, Springer Verlag, 2001.
25. T.C. Son and E. Pontelli. Reasoning about actions in prioritized default theory. In *JELIA*, Springer Verlag, 2002.
26. R. Stevens. Bio-Ontology Reference Collection. `cs.man.ac.uk/~stevens/onto-publications.html`.
27. K. Sycara, M. Paolucci, M. van Velsen, and J. Giampapa. The RETSINA MAS Infrastructure. In *Autonomous Agents and MAS*, 7(1-2), 2003 (to appear).
28. D. L. Swofford et al. Phylogenetic inference. *Molecular Systematics*, Sunderland, 1996.
29. A. van Deursen et al. DSLs: an Annotated Bibliography. `www.cwi.nl/~arie`, 2000.

Go! for multi-threaded deliberative agents

K. L. Clark[1] and F. G. Mccabe[2]

[1] Dept. of Computing, Imperial College, London
[2] Fujitsu Labs of America, Sunnuvale, CA

Abstract. `Go!` is a multi-paradigm programming language that is oriented to the needs of programming secure, production quality, agent based applications. It is multi-threaded, strongly typed and higher order (in the functional programming sense). It has relation, function and action procedure definitions. Threads execute action procedures, calling functions and querying relations as need be. Threads in different agents communicate and coordinate using asynchronous messages. Threads within the same agent can also use shared dynamic relations acting as memory stores.
In this paper we introduce the essential features of `Go!` illustrating them by programming a simple multi-agent application comprising hybrid reactive/deliberative agents interacting in a simulated ballroom. The dancer agents negotiate to enter into joint commitments to dance a particular dance (e.g. polka) they both desire. When the dance is announced, they dance together. The agents' reactive and deliberative components are concurrently executing threads which communicate and coordinate using belief, desire and intention memory stores. We believe such a multi-threaded agent architecture represents a powerful and natural style of agent implementation, for which `Go!` is well suited.

1 Introduction

`Go!` is a logic programming descendant of the multi-threaded symbolic programming language `April`[15]. `April` was initially developed as the implementation language for the much higher level MAI^{2}L[10] agent programming of the EU Imagine project. It has more recently been used to implement one of the FIPA compliant agent platforms of the EU AgentCities project[21], and the agent services running on that platform at Imperial College and Fujitsu.

A significant theme in the design of `Go!` is software engineering in the service of high-integrity systems. To bring the benefits of logic programming to applications developers requires fitting the language into current best-practice; and, especially since applications are increasingly operating in the public Internet, security, transparency and integrity are critical to the adoption of logic programming technology.

Although `Go!` has many features in common with `Prolog`, particularly multi-threaded `Prolog`'s such as Qu-Prolog[6], there are significant differences related to transparency of code and security. Features of `Prolog` that mitigate against

transparency, such as the infamous *cut* (`!`) primitive, are absent from `Go!`. Instead, its main uses are supported by higher level programming constructs, such as single solution calls, *iff* rules, and the ability to define 'functional' relations as functions.

In `Prolog`, the same clause syntax is used both for defining relations, with a declarative semantics, and for defining procedures, say that read and write to files, which really only have an operational semantics. In `Go!`, behaviours are described using action rules, which have a different syntax. While `Prolog` is a *meta-order* language, `Go!` is higher-order (in the functional programming sense) and strongly typed, using a modified Hindley/Milner style type inference technique[16].

A key feature of `Go!` is the ability to group a set of definitions into a lexical unit by surrounding them with {} braces. We call such a unit a *theta environment*. Theta environments are `Go!`'s program structuring mechanism. Two key uses of theta environments are *where* expressions, analogous to the *let ... in ...* construct of some functional programming languages in which an expression is evaluated relative to a theta environment, and labeled theories, which are labeled theta environments.

Labeled theories are based on McCabe's *L&O* [14] extension of Prolog. A labeled theory is a theta environment labeled by a term where variables of the label term are global variables of the theory. Instances of the theory are created by given values to these label variables. Labeled theories are analogous to class definitions, and their instances are `Go!`'s objects. Objects can have state, recorded by primitive *cell* and *dynamic relation* objects. New labeled theories can be defined in terms of existing theories using inheritance rules. Labeled theories provide a rich knowledge representation notation akin to that of frame systems.

This paper introduces the key features of `Go!` and illustrates its power and succinctness by developing a simple multi-agent application comprising hybrid reactive/deliberative agents interacting at a simulated ball. Although an artificial example we believe it is representative of many multi-agent applications.

In section 2 we give a brief overview of `Go!` and its facilities for programming task orientated agents. In the limited space available we cannot give a comprehensive description of `Go!`. In particular, space does not allow us to fully illustrate the OO features. For a more complete description see [5].

In section 3 we explore `Go!` in the context of the simulated ballroom. Each dancer agent is programmed using multiple concurrently executing threads that implement different aspects of its behaviour – coordinated by shared `belief`, `desire` and `intention` dynamic relation memory stores. This internal run-time architecture has *implicit* interleaving of the various activities of the agent. This contrasts with the *explicit* interleaving of observation, short deliberation and partial execution of the classic single threaded BDI (*B*eliefs,*D*esires,*I*ntentions) architecture[2].

The `belief`, `desire` and `intention` memory stores are used in a manner similar to Linda tuple stores[3]. For example, memory store updates are atomic,

and a thread can suspend waiting for a belief to be added or deleted. Linda tuple stores have been used for inter-agent coordination [17]. For scalability and other reasons, we prefer to use asynchronous point-to-point messages between agents, as in KQML[8]. However, we strongly advocate concurrency and Linda style shared memory co-ordination for internal agent design.

In section 4 we briefly discuss related work before giving our concluding remarks.

2 Key Features of Go!

`Go!` is a multi-paradigm language with a declarative subset of function and relation definitions and an imperative subset comprising action procedure definitions.

2.1 Function, relation and action rules

Functions are defined using sequences of rewrite rules of the form:

$f(A_1,..,A_k)::Test \texttt{ => } Exp$

where the guard *Test* is omitted if not required.

As in most functional programming languages, the testing of whether a function rule can be used to evaluate a function call uses *matching* not unification. Once a function rule has been selected there is no backtracking to select an alternative rule.

Relation definitions comprise sequences of Prolog-style `:-` clauses ; with some modifications – such as permitting expressions as well as data terms, and no cut. We can also define relations using *iff* rules.

The locus of action in `Go!` is a *thread*; each `Go!` thread executes a procedure. Procedures are defined using non-declarative *action* rules of the form:

$a(A_1,..,A_k)::Test \texttt{ -> } Action_1;...;Action_n$

As with equations, the first action rule that matches some call, and whose test is satisfied, is used; once an action rule has been selected there is no backtracking on the choice of rule.

The permissible actions of an action rule include: message dispatch and receipt, I/O, updating of dynamic relations, the calling of a procedure, and the spawning of any action, or sequence of actions, to create a new action thread.

Threads in a single `Go!` invocation can communicate either by thread-to-thread message communication or by synchronisable access and update of shared data, such as dynamic relations. Threads in different `Go!` invocations can only communicate using messages. To support thread-to-thread communication, each thread has its own buffer of messages it has not yet read, which are ordered in the buffer by time of arrival. To place a message in a thread's buffer the sender has to have the threads unique handle identity.

The message send action:

```
Msg >> To
```

sends the message *Msg* to the thread identified by the handle *To*. Handles are terms of the form `hdl(Id,Group)` where `Id` and `Group` are symbols that together uniquely identify the thread. Typically, threads within the same agent share the same `Group` name, which can be the unique agent's name.

To look for and remove from the message buffer a message matching *Ptn* sent by a thread *From* the receive action:

```
Ptn << From
```

can be used.

To look for any one of several messages, and to act appropriately when one is found, the conditional receive:

```
( Ptn1 << From1 -> Actions1
| ...
| Ptnn << Fromn -> Actionsn
)
```

can be used. When executed, the message buffer of the thread is searched to find the first message that will *fire* one of these alternate message receive rules. The matched message is removed from the message buffer and corresponding actions are executed. Messages that don't match are left in the message buffer for a later message receive to pick up.

Both forms of message receive suspend if no matching message is found, causing the thread to suspend. The thread resumes only when a matching message is received. This is the message receive semantics of `Erlang`[1] and `April`[15].

Communication daemons and a special external communications system module allow threads in different invocations of `Go!` to communicate using the same message send and receive actions as are used between threads of a single invocation, see [5]. This allows an application comprising several modules, developed and tested as one multi-threaded `Go!` invocation, to be converted into a distributed application with minimal re-programming.

2.2 Programming behaviour with action rules

As an example of the use of action rules let us consider programming the top level of an agent with a mission: this is to achieve some fixed goal by the repeated execution of an appropriate action. The two action rule procedure:

```
performMission()::Goal -> {}.
performMission() -> doNextStep ; performMission().
```

captures the essence of this goal directed activity. ({} is the empty action.) This procedure would be executed by one thread within an agent whilst another concurrently executing thread is monitoring its environment, constantly updating the agent's beliefs about the environment; these beliefs being queried

by *Goal*, and by *doNextStep*. `performMission` is a tail recursive procedure and will be executed as an iteration by the `Go!` engine.

Some missions – such as survival – do not have a termination goal but rather one or more continuation actions:

```
survive()::detectDanger(D) -> hideFrom(D);survive().
survive()::detectFood(F) -> eat(F); survive().
survive() -> wanderFor(safeTime()); survive().
```

The order of the rules prioritises avoiding danger. `safeTime` is a function that queries the belief store to determine a 'safe' period to wander, given current knowledge about the environment, before re-checking for danger. Again we assume the belief store is being concurrently manipulated by an environment monitoring thread within the agent. `hideFrom(D)` would typically cause the survival thread to suspend until the monitoring thread deletes those beliefs that made `detectDanger(D)` true.

Invoking queries from actions The declarative part of a `Go!` program can be accessed from action rules in a number of ways:

- Any expression can invoke functions.
- An action rule guard – $(A_1,..,A_k)::Q$ – can extend the argument matching with a query Q.
- If Q is a query, $\{Q\}$, indicating a single solution to Q, can appear as an 'action' in an action rule body.
- We can use a set expression $\{Trm \texttt{ || } Q\}$ to find all solutions to some query. This is `Go!`'s `findall`.
- We can use `Go!`'s *forall* action. $(Q \texttt{ *> } A)$ iterates the action A over all solutions to query Q.
- We can use a conditional action. $(Q \texttt{ ? } A_1 \texttt{ | } A_2)$ executes A_1 if Q succeeds, else A_2.

As an example of the use of `*>`:

```
(is_a_task(Task), I_cant_do(Task), cando(Ag,Task)
  *> ('request',Task) >> Ag)
```

might be used to send a `'request'` message for each current sub-task that the agent cannot itself do to some agent it believes can do the task. `'request'` is a *quoted symbol*, which is a primitive data type of `Go!`.

2.3 Dynamic relations

In `Prolog` we can use `assert` and `retract` to change the definition of a dynamic relation whilst a program is executing. The most frequent use of this feature is to modify a definition comprising a sequence of unconditonal clauses. `Go!` has a system class that can be imported and used to create, update and call dynamic

relations defined using unconditional clauses. In addition, lists of higher-order values, such as relations of the same type, can be stored in cells. This allows us to store, modify and call dynamic relations defined by sequences of rules.

The dynamic relations class definition is imported by using:

```
include "sys:go/dynamic.gof"
```

A dynamic relation is an object with methods: `add`, for adding a fact term to the end of the current sequence of facts in the relation object, `del` for removing the first fact term in a dynamic relation object that unifies with some pattern, `delall` for removing all unifying fact terms, `mem`, for accessing the current fact terms in some dynamic relation component, and finally `ext` for retrieving the current extension as a list of fact terms.

Creating a new dynamic relation A dynamic relation object can be created and initialised using:

```
desire = $dynamic([toDance(jive,2), toDance(waltz,1),
                   ...,barWhen(polka)])
```

`dynamic` takes a list of terms that are the initial extension of the relations. This list could be empty. The above initialisation is equivalent to giving the following the sequence of clauses for a Prolog dynamic relation:

```
desire(toDance(jive,2)).
desire(toDance(waltz,1)).
...
desire(barWhen(polka)).
```

Querying a dynamic relation If we want to query such a dynamic relation we use the `mem` method as in:

```
desire.mem(todance(D,N)),N>2
```

Modifying a dynamic relation To modify a dynamic relation we can use the `add`, and `del` action methods. For example:

```
desire.add((barWhen(quickstep))
```

and:

```
desire.del(toDance(jive,N));desire.add(toDance(jive,N-1))
```

The second is analogous to the following sequence of `Prolog` calls:

```
retract(desire(toDance(jive,N)),NewN is N+1,
assert(toDance(jive,NewN))
```

One difference is that we cannot backtrack on a `del` call to delete further matching facts. This is because it is an action, and all `Go!` actions are deterministic. A `del` call always succeeds, even if there is no matching term. The `delall` method deletes all unifying facts as a single action:

```
desire.delall(barWhen(_))
```

will delete all current `barWhen` desires.

2.4 Multi-threaded applications and data sharing

It is often the case, in a multi-threaded `Go!` application, that we want the different threads to be able to share information. For example, in a multi-threaded agent we often want all the threads to be able to access the beliefs of the agent, and we want to allow some or all these threads to be able to update these beliefs.

We can represent the relations for which we will have changing information as dynamic relations. A *linda* subclass of the dynamic relations class has extra methods to facilitate the sharing of dynamic relations across threads. Instances if this subclass are created using initializations such as:

```
LinRel = $linda([...])
```

For example, it has a `replace` method allowing the deleting and adding of a shared linda relation term to be executed atomically, and it has a `memw` relation method. A call:

```
LinRel.memw(Trm)
```

will suspend if no term unifying with `Trm` is currently contained in `LinRel` until such a term is added by *another thread*.

There is also a dual, `notw` such that:

```
LinRel.notw(Trm)
```

will suspend if a term unifying with `Trm` is currently contained in `LinRel` until all such terms are deleted by *other threads*. It also has a suspending delete method, `delw`.

`memw` and `delw` and the analogues of the Linda[3] `readw` and `inw` methods for manipulating a shared tuple store. There is no analogue of `notw` in Linda.

2.5 Type definitions and type inference

`Go!` is a strongly typed language; using a form of Hindley/Milner's type inference system[16]. For the most part it is not necessary for programers to associate types with variables or other expressions. However, all constructors and *unquoted* symbols are required to be introduced using type definitions. If an identifier is used as a function symbol in an expression it is assumed to refer to an 'evaluable' function unless it has been previously introduced in a type definition.

The pair of type definitions:

```
dance::= polka | jive | waltz | tango | quickstep | samba.
Desire::= toDance(dance,number) | barWhen(dance).
```

introduce two new types – an enumerated type `dance`, which has 6 literal values:

```
polka, jive, waltz, tango, quickstep, samba
```

and a `Desire` type that has two constructor functions `toDance` and `barWhen`.

3 Multi-threaded dancer agents at a ball

In our agents' ball, we have male and female dancer agents that are attempting to dance with each other and a band that 'plays' music for different kinds of dances. The two kinds of dancer agent are required to discover like-minded agents and to negotiate over possible dance engagements. In addition to dancing, dancer agents may have additional goals – such as getting refreshed at the bar. This scenario is a compact use case that demonstrates many of the aspects of building intelligent agents and of coordinating their activities.

Following a BDI model[2][19], each agent has a `belief`, a `desire` and an `intention` relation. The `belief` relation contains beliefs about what other dancers there currently are and what dances they like to do. The `desire` relation contains the goals each dancer would like to achieve, for example, which dances it would like to dance. The `intention` relation holds its current intentions – these normally represent the agent's commitments to perform some particular dance with some partner agent; however, it can also be an intention to go to the bar when a dance is announced.

The dancers use a directory server to discover one another. As each dancer agent 'arrives' at the dance in some random and phased order, it registers with the directory server. The dancers also subscribe in order to be informed about other dancers that are already 'at the dance', and those that will arrive later.

The internal execution architecture of each dancer agent comprises three threads – corresponding to the three key activities of the agent: a directory server interface thread, a negotiations thread and an intention execution thread. The directory server interface interacts with the directory server to publish its own description and to subscribe for the descriptions of other dancer agents. The negotiations thread communicates with other dancer agents in order to agree joint intentions to dance the next dance of a particular kind. The intentions execution thread coordinates the actual dance activities and any 'drinking' activities.

These threads communicate using the shared linda dynamic relations: `belief`, `desire` and `intention`. Note that while all the dancers could be executed in a single invocation of the `Go!` engine, they will *not* have direct access to each others' beliefs, desires and intentions. Furthermore, it is a simple task to distribute the progam across multiple invocations and machines, making each dancer a separate `Go!` process.

3.1 A dancer's intention execution thread

A dancer's intention execution thread handles the execution of intentions when they are triggered by dance announcements. We assume a band agent which sends an announcement message to every currently registered dancer when it starts, and when it later stops playing each dance 'number'.

The procedures for the intention execution threads of the male and female dancers are very similar with respect to how they 'listen' for announcements from the band. They differ in what happens when a dance is starting and there

is an intention to do that dance. We present here only the male case – as the male dancer is expected to take the initiative during the dance.[3]

```
maleIntention..{
 ... -- include statements
 dance::= polka | jive | waltz | tango | quickstep | samba.
 Belief::= hasDesires(symbol,Desire[])
          | bandNotPlaying
          | bandPlaying(dance)
          | ballOver
          | haveDanced(dance,symbol).
 Desire::= toDance(dance,number) | barWhen(dance).
 Intention ::= toDance(dance,symbol) | ...
 bandMessage::= starting(dance) | stopping(dance) | ball_over.

 maleIntention(belief,desire,intention,band) -> loop()..{
  loop() ->
    ( starting(D) << band ->
      belief.replace(bandNotPlaying,bandPlaying(D));
      check_intents(D,belief,desire,intention);
      loop()
   | stopping(D) << band ->
      belief.replace(bandPlaying(D),bandNotPlaying);
      loop()
   | ball_over << band -> belief.add(ballOver)
   ).
  check_intents(D)::intention.mem(toDance(D,FNm)) ->
      intention.del(toDance(D,FNm));
      maleDance(D,FNm)).
  ...
  } -- end of inner environment defining loop etc
}
```

The above is a module that exports the `maleIntention` action procedure. The `..` can be read as *where* and the definitions enclosed inside the {} braces following `..` are a *theta environment*. The procedure calls `loop`, which is itself defined using a *where* with an inner theta environment[4]. `loop` iterates listening for messages; in this case messages from the band. It terminates when it receives a `ball_over` message.

When it receives a `starting(D)` message, and there is an intention to do that dance, the `maleDance` procedure is executed. This is given access to the dancer's beliefs and desires as it may need to modify them. The intended partner

[3] This symmetry is an aspect of the ballroom scenario; one that we would not expect for general agent systems.

[4] The parameters `belief`,..,`band` of the `maleIntention` procedure are in scope throughout the inner environment in which `loop` etc are defined.

should similarly have called its corresponding `femaleDance` procedure and the interaction between the dance threads of the two dancers is the joint dancing activity.

Notice that the `maleIntention` procedure reflects its environment by maintaining an appropriate belief regarding what the band is currently doing and when the ball is over. `replace` is an atomic update action on a linda dynamic relation.

3.2 A dancer's negotiation thread

The procedures executed by the negotiation threads of our dancers are the most complex. They represent the rational and pro-active activity of the agent for they convert desires into intentions using current beliefs. In contrast, the intentions execution and directory interface threads are essentially reactive activities.

A male dancer's negotiation thread must decide which uncommitted desire to try to convert into an intention, and, if this is to do some dance the next time it is announced, which female dancer to invite to do the dance. This may result in negotiation over which dance they will do together, for the female who is invited may have a higher priority desire. Remember that each dancer has a partial model of the other dancer in that it has beliefs that tell it the desires the other dancer registered with the directory server on arrival. But it does not know the priorities, or which have already been fully or partially satisfied.

The overall negotiation procedure is `satisfyDesires`:

```
satisfyDesires()::belief.mem(ballOver) -> {}.
satisfyDesires() ->
  {belief.memw(bandNotPlaying)}; -- wait until band not playing
  (chooseDesire(Des,FNm),\+[5]intention.mem(toDance(D,_),
     still_ok_to_negotiate()) *>
              negotiateOver(Des,FNm));  -- negotiation forall
  {belief.memw(bandPlaying(_))}; -- wait until band playing
  satisfyDesires().
still_ok_to_negotiate():-
  \+ believe.mem(bandPlaying(_)),\+ believe.mem(ballOver).
```

The `satisfyDesires` procedure terminates when there is a belief[6] that the band has finished – a belief that will be added by the intentions execution thread when it receives the message `ball_over`. If not, the first action of `satisfyDesires` is the `memw` call. This is a query action to the `belief` relation that will suspend, if need be, until `bandNotPlaying` is believed. For our dancers we only allow negotiations when the band is not playing. This is not a mandatory aspect of all scenarios – other situations may permit uninterrupted negotiations over desires.

[5] `\+` is `Go!` 's negation as failure operator.

[6] All the procedures for this thread access the linda dynamic relations as global variables since the procedures will be defined in the environment where these relations are introduced.

There is then an attempt to convert into commitments to dance as many unsatisfied dance desires as possible, before the band restarts. This is done by negotiation with a named female whom the male dancer believes shares that dance desire. When that iterative action terminates, the dancer checks that the band has restarted and waits if not. This is to ensure there is only one round of negotiation in each dance interval. The next time the band stops playing, the answers returned by `chooseDesire` will almost certainly be different because the beliefs, desires and intentions of the dancer will have changed. (Other female dancers may have arrived, and the dancer may have executed an intention during the last dance.) Even if one of the answers is the same, a re-negotiation with the same female may now have a different outcome because of changes in her mental state.

```
chooseDesire(Dance(D,N),FNm) :-
   uncmtdFeasibleDesire(Dance(D,N),FNm),
   (desire.mem(Dance(OthrD,OthrN)),OthrD\=D *> OthrN < N).
chooseDesire(Dance(D,N),FNm) :-
   uncmtdFeasibleDesire(Dance(D,N),FNm),
   \+ belief.mem(haveDanced(D,_)).
chooseDesire(Dance(D,N),FNm) :-
   uncmtdFeasibleDesire(Dance(D,N),FNm),
   \+ belief.mem(haveDanced(D,FNm)).
chooseDesire(Dance(D,N),FNm) :-
   uncmtdFeasibleDesire(Dance(D,N),FNm).
uncmtdFeasibleDesire(Dance(D,N),FNm) :-
   uncmtdDesire(Dance(D,N)),
   belief.mem(hasDesires(FNm,FDesires)),
   Dance(D,_) in FDesires.
...
uncmtdDesire(Dance(D,N)):-
   desire.mem(Dance(D,N)), N>0,
   \+ intention.mem(toDance(D,_)),
...
```

The above clauses are tried in order, which reflects priorities. All the clauses return a dance desire only if it is currently uncommitted and the male believes some female desires to do that dance. It is an uncommitted desire if it is still desired to perform the dance at least once, and there is not a current intention to do that dance. (We allow a dancer to enter into at most one joint commitment to do a particular type of dance since this is understood as a commitment to do the dance with the identified partner the *next* time *that* dance is announced.)

The first rule selects a dance if, additionally, it is desired more times than any other dance. The second selects a dance if it has not so far been danced with *any* partner. The third rule selects it if it has not so far been danced with the female who will now be asked. The last, default rule, selects any desired, uncommitted feasible dance. A male with this `chooseDesire` definition only actively tries to

satisfy dance desires, but it could still end up with an intention of going to the bar as a result of negotiation with a female dancer.

Below is a `negotiateOver` procedure for a simple negotiation strategy:

```
ngtMess::= willYouDance(dance) | okDance(dance) |
           sorry | barWhen(dance) | okBar(dance).
negotiateOver(Dance(D,N),FNm) ->
  ngtOverDance(D,N,FNm,hdl('neg',FNm),[]).
ngtOverDance(D,N,FNm,FNgtTh,PrevDs) ->
  willYouDance(D) >> FNgtTh; -- invite female to dance D
  ( okDance(D) << FNgtTh ->  -- female has accepted
      desire.replace(Dance(D,N),Dance(D,N-1));
      intention.add(toDance(D,FNm))
  | sorry << FNgtTh -> {}  -- female has declined
  | willYouDance(D2)::uncmtdDesire(Dance(D2,N2))) << FNgtTh ->
       -- she has counter-proposed to dance D2, which is ok
       intention.add(toDance(D2,FNm));
       desire.replace(Dance(D2,N2),Dance(D2,N2-1));
       okDance(D2) >> FNgtTh
  | willYouDance(D2) << FNgtTh -> -- dance counter prop. not ok
       counterP(FNm,FNgtTh,[D,D2,..PrevDs])
  | barWhen(D2)::uncmtdDesire(BarWhen(D2)) << FNgtTh ->
       intention.add(toBarWhen(D2,FNm));
       desire.del(BarWhen(D2));
       okBar(D2) >> FNgtTh
  | barWhen(D2) << FNgtTh ->      -- bar counter prop. not ok
      counterP(FNm,FNgtTh,[D,D2,..PrevDs])
  ).
counterP(FNm,FNgtTh,PrevDs)::
  (chooseDesire(Dance(D,N),FNm),\+(D in PrevDs))->
     ngtOverDance(D,N,FNm,FNgtTh,PrevDs).
counterP(_,FNgtTh,_) -> -- cannot find a new dance desire
     sorry >> FNgtTh)).
```

The negotiation is with the negotiation thread, `hdl('neg',FNm)`, in the female dancer with name `FNm`.

The negotiation to fulfill a dance desire with a named female starts with the male sending a `willYouDance(D)` message to her negotiation thread. There are four possible responses: an `okDance(D)` accepting the invitation, a `sorry` message declining, or a counter proposal to do another dance, or to go to the bar when some dance is played. A counter proposal is accepted if it is currently an uncommitted desire. Otherwise, the `counterP` procedure is called to suggest an alternative dance. This calls `chooseDesire` to try find another feasible dance `D` for female `FNm`, different from all previous dances already mentioned in this negotiation (the `PrevDs` argument). If this succeeds, the dance negotiation procedure is re-called with `D` as the new dance to propose. If not, a `sorry` message is sent and the negotiation with this female ends.

3.3 The male dancer agent

Below we give the overall structure of the male dancer class definition. It uses modules defining the `maleIntention` and `DSinterface` procedures and it spawns them as separate threads.

```
maleDancer(MyNm,MyDesires,DS,band){
  .. -- include statements
  belief=$linda([]).
  desire=$linda([]).
  intention=$linda([]).
  init() ->
    (Des on MyDesires *> desire.add(Des));
    spawn DSinterface(MyNm,male,belief,MyDesires,DS);
    spawn maleIntention(belief,desire,intention,band)
                          as hdl('exec',MyNm);
    spawn satisfyDesires() as hdl('neg',MyNm);
    waitfor(hdl('exec',MyNm)).
  .. -- defs of satisfyDesires etc
}
```

The init procedure of this module is the one called to activate an instance `$maleDancer(MyNm,MyDesires,DS,band)` of the class. An instance is specified by four parameters: the unique symbol name of the dancer agent, such as `'bill l. smith'`, a list of its desires expressed as `Desire` terms, and the handles of the directory server and band agent of the ball it is to attend. Each instance will have its own three linda dynamic relations encoding the dynamic state of the instance.

The `init` procedure adds each desire of `MyDesires` parameter to the dancer's `desire` linda relation. It then `spawns` the directory server interface, the intention execution and the negotiation threads for the dancer. The latter are assigned standard handle identities based on the agents symbol name. The `init` procedure then waits for the intention execution thread to terminate (when the ball is over). Termination of `init` terminates the other two spawned threads.

The negotiation thread executes concurrently with the other two threads. The directory interface thread will be adding beliefs about other agents to the shared `belief` relation as it receives `inform` messages from the directory server, and the execute intentions thread will be concurrently accessing and updating all three shared relations.

The female dancer is similar to the male dancer; we assume that the female never takes the initiative. The female negotiation thread must wait for an initial proposal from a male but thereafter it can make counter proposals. It might immediately counter propose a different dance or to go to the bar, depending on its current desires and commitments. It can handle negotiations with male proposers one at a time, or have simultaneous negotiations by spawning auxiliary negotiation threads. This requires another dynamic relation to keep track of

the latest proposal of each negotiation so that they do not result in conflicting commitments.

4 Related Work

4.1 Logic Programming Languages

Qu-Prolog[6], BinProlog[20], CIAO Prolog [4], SICStus-MT Prolog[7] are all multi-threaded Prolog systems. The closest to `Go!` is Qu-Prolog. Threads in Qu-Prolog communicate using messages or via the dynamic data base. As in `Go!`, threads can suspend waiting for another thread to update some dynamic relation. However, Qu-Prolog has no higher order features or type checking support, and all threads in the same Qu-Prolog invocation share the same global dynamic data base. In `Go!`, using modules, dynamic relations can be restricted to a specified set of threads.

SICStus-MT[7] Prolog threads also each have a single message buffer, which they call a port, and threads can scan the buffer looking for a message of a certain form. But this buffered communication only applies to communication between threads in the same Prolog invocation.

In BinProlog[20], threads in the same invocation communicate through the use of Linda tuple spaces[3] acting as shared information managers. BinProlog also supports the migration of threads, with the state of execution remembered and moved with the thread[7]. The CIAO Prolog system [4] uses just the global dynamic Prolog database for communicating between thread's in the same process. Through front end compilers, the system also supports functional syntax and modules.

Mercury[22] is a pure logic programming language with polymorphic types and modes. The modes are used to aid efficient compilation. It is not multi-threaded.

4.2 Logic and action agent languages

Vip[12], AgentSpeak(L)[18], 3APL[11], Minerva[13] and ConGolog[9] are all proposals for higher level agent programming languages with declarative and action components. We are currently investigating whether the implied architectures of some of these languages can be readily realised in `Go!`. Vip and 3APL have internal agent concurrency.

These languages typically have plan libraries indexed by desire or event descriptors with belief pre-conditions of applicability. Such a plan library can be encoded in `Go!` as a set of `planFor` and `reactTo` action rules of the form:

[7] A `Go!` thread executing a recursive procedure can also be migrated by sending a closure containing a 'continuation' call to this procedure in a message. The recipient then spawns the closure allowing the threads computation to continue in a new location. The original thread can even continue executing, allowing cloning.

```
planFor(Desire)::beliefCond -> Actions
reactTo(Event)::beliefCond -> Actions
```

The actions can include updates of the belief store, or the generation of new desires whose fulfillment will complete the plan. Calls to `planFor` or `reactTo` can be spawned as new threads, allowing concurrent execution of plans.

5 Conclusions

`Go!` is a multi-paradigm programming language – with a strong logic programming aspect – that has been designed to make it easier to build intelligent agents while still meeting strict software engineering best practice. Although many AI practitioners find the restrictions imposed by strong typing and other SE-oriented disciplines to be irksome, we find that we are not significantly hindered. In part this has been because we had the requirements for agent programming in mind in the design of `Go!`.

There are many other important qualities of a production environment that we have not had the space to explore – for example the I/O model, permission and resource constrained execution and the techniques for linking modules together in a safe and scalable fashion. We have also omitted any discussion of how `Go!` applications are distributed and of how `Go!` programs interoperate with standard technologies such as DAML, SOAP and so on. For a more complete description of some of these topics see [5].

The ballroom scenario is an interesting use case for multi-agent programming. Although the agents are quite simple, it encompasses key *behavioural* features of agents: autonomy, adaptability and responsibility. Our implementation features inter-agent communication and co-ordination via messages, multi-threaded agents, intra-agent communication and co-ordination via shared memory stores. We believe these features, which are so easily implemented in `Go!`, are firm foundations on which to explore the development of much more sophisticated deliberative multi-threaded agents.

6 Acknowledgments

The first named author wishes to thank Fujitsu Labs of America for a research contract that supported the collaboration between the authors on the design of `Go!` and the writing of this paper.

References

1. J. Armstrong, R. Virding, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall International, 1993.
2. M. E. Bratman, D. J. Israel, and M. E. Pollack. Plans and resource bounded practical reasoning. *Computational Intelligence*, 4:349–355, 1988.

3. N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.
4. M. Carro and M. Hermenegildo. Concurrency in Prolog using Threads and a Shared Database. In D. D. Schreye, editor, *Proceedings of ICLP99*, pages 320–334. MIT Press, 1999.
5. K. Clark and F. McCabe. Go! – a logic programming language for implementing multi-threaded agents. Technical report, Downloadable from www.doc.ic.ac.uk/~klc, 2003.
6. K. L. Clark, P. J. Robinson, and R. Hagen. Multi-threading and message communication in Qu-Prolog. *Theory and Practice of Logic Programming*, 1(3):283–301, 2001.
7. J. Eskilson and M. Carlsson. Sicstus MT - a multithreaded execution environment for SICStus Prolog. In K. M. Catuscia Palamidessi, Hugh Glaser, editor, *Principles of Declarative Programming*, LNCS 1490, pages 36–53. Springer-Verlag, 1998.
8. T. Finin, R. Fritzson, D. McKay, and R. McEntire. KQML as an agent communication language. In *Proceedings 3rd International Conference on Information and Knowledge Management*, 1994.
9. G. D. Giacomo, Y. Lesperance, and H. Levesque. Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 1–2(121):109–169, 2000.
10. H. Haugeneder and D. Steiner. Co-operative agents: Concepts and applications. In N. R. Jennings and M. J. Wooldridge, editors, *Agent Technology*. Springer-Verlag, 1998.
11. K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J.-J. C. Meyer. Formal semantics for an abstract agent programming language. In Singh, Roa, and Wooldridge, editors, *Intelligent Agents IV*, LNAI. Springer-Verlag, 1997.
12. D. Kinny. VIP:A visual programming language for plan execution systems. In *1st International Joint Conf. Autonomous Agents and Multi-agent Systems*, pages 721–728. ACM Press, 2002.
13. J. A. Leite, J. J. Alferes, and L. M. Pereira. Minerva-A dynamic logic prorgamming agent architecture. In *Intelligent Agents VIII, LNAI 2333*, pages 141–157, 2001.
14. F. McCabe. *L&O: Logic and Objects*. Prentice-Hall International, 1992.
15. F. McCabe and K. Clark. April - Agent PRocess Interaction Language. In N. Jennings and M. Wooldridge, editors, *Intelligent Agents, LNAI, 890*. Springer-Verlag, 1995.
16. R. Milner. A theory of type polymorphism in programming. *Computer and System Sciences*, 17(3):348–375, 1978.
17. A. Omnicini and F. Zambonelli. Coordination for internet application development. *Autonomous Agents and Multi-agent systems*, 2(3):251–269, 1999.
18. A. S. Roa. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Agents Breaking Away*, LNAI 1038. Springer-Verlag, 1996.
19. A. S. Roa and M. P. Georgeff. An abstract architecture for rational agents. In *Proceedings of Knowledge Representation and Reasoning (KR&R92)*, pages 349–349, 1992.
20. P. Tarau and V. Dahl. Mobile Threads through First Order Continuations. In *Proceedings of APPAI-GULP-PRODE'98*, Coruna, Spain, 1998.
21. S. N. Willmott, J. Dale, B. Burg, C. Charlton, and P. O'Brien. Agentcities: A Worldwide Open Agent Network. *Agentlink News*, (8):13–15, November 2001.
22. F. H. Zoltan Somogyi and T. Conway. Mercury: an efficient purely declarative logic programming language. In *Proceedings of the Australian Computer Science Conference*, pages 499–512, 1995.

Operational Semantics for Agents by Iterated Refinement

Federico Bergenti[1], Giovanni Rimassa[1], and Mirko Viroli[2]

[1] AOT Lab - Dipartimento di Ingegneria dell'Informazione
Parco Area delle Scienze 181/A, 43100 Parma, Italy
{bergenti,rimassa}@ce.unipr.it
[2] DEIS, Università degli Studi di Bologna,
via Rasi e Spinelli 176, 47023 Cesena, Italy
mviroli@deis.unibo.it

Abstract. In this paper we evaluate transition systems as a tool for providing a rule-based specification of the operational aspects of autonomous agents. By our technique, different aspects of an agent can be analyzed and designed in a loosely coupled way, enabling the possibility of studying their properties in isolation.
We take as a use case the ParADE framework for building intelligent agents, which leverages a FIPA-like ACL semantics to support semantic interoperability. Our grey-boxing technique is exploited to provide a specification where aspects related to the ACL, the adopted ontology, the agent social role, and the other agent internal details are described separately, in an incremental way.

1 Formalisms, Software Engineering, and Agents

This paper addresses the issue of sound multi-agent design trying to keep a system engineering perspective; that is, both when analyzing a problem and when synthesizing a solution the focus is kept on the whole, large-scale structure of the software artifact that is to be realized.

Striving for a sound design process naturally suggests to rely on some kind of mathematical tools: the precision and coherency afforded by formal and analytical reasoning holds great promises in terms of soundness. But, design is above all a creative process, and care has to be taken so as not to hamper it with too rigid a framework. In particular, formal methods during design activity should act as a tool to nurture an evolving idea, suggesting to the designer viable future choices while checking desirable properties of the current solution.

The formal tool we propose to provide effective assistance to the sound design of multi-agent systems (MAS) applies *labeled transition systems* to the description of interactive behaviors (of software abstractions) [6], as elaborated and promoted in the field of concurrency theory. When specifying a transition system semantics for an agent, a number of rules are given that describe in a quite declarative way the dynamics of its inner machinery, also providing insights on its internal architecture – expressed at a given level of abstraction. We believe that this formalism is particularly well suited because of its calculus-like nature,

which combines the description of a system with the prescription of its possible evolutions; this property buys designers some generativity, while remaining in a well grounded mathematical landscape.

In particular, in this paper we develop a formal framework that exploits basic features of labeled transition systems to address both composability and extensibility. Composability is obtained by dividing an agent model into several transition system specifications, that we call *tiers*; each tier captures the possible evolutions of the agent conceptual subpart dealing with a specific aspect. Beyond providing formal decoupling, tiers also mirror conceptually significant views over an agent behavior, thus easing the adjustable abstraction process. Moreover, starting from a partial specification of an agent behavior, namely a *grey-box model* [18, 15, 17], tiers are used to extend that specification by taking into account a new behavioral aspect, and leading to a new model that is a refinement of the former – according to the standard notion of refinement as "more deterministic implementation" introduced in the context of concurrency [6].

The tiers we use to describe an agent are more than a useful formal technique to us. They are closely related to the different levels of abstractions of an agent description.

The accepted scientific formalization of the idea of level of abstraction is the definition of *system level* [12]. A system level is a set of concepts that provides a means for modeling implementable systems. System levels abstract away from implementation details and are arranged in stack fashion so that higher levels provide concepts that are closer to human intuition and far away from implementation. System levels are structured in terms of the following elements: *(i)* Components, atomic building blocks for building systems at that level; *(ii)* Laws of compositions, laws that rule how components can be assembled into a system; *(iii)* A medium, a set of atomic concepts that the system level processes; and *(iv)* Laws of behavior, laws that determine how the behavior of the system depends on the behavior of each component and on the structure of the system.

In the specific case of agent systems, a relevant application of this idea is the well-known Newell's *knowledge level* [12]. This level describes a single agent as structured in goals, actions, and a body, capable of processing knowledge and pursuing its goals according to the rationality principle. On the other hand, Jennings' proposal of the *social level* [8] moves from agents to MAS trying to raise the level of abstraction of the knowledge level. There, the system is an agent organization, that is, a group of agents with organizational relationships and dependencies on one another, which can interact through channels. Other alternative systems level have been described such as e.g. in [1], taking the benefits of both Newell's and Jennings' trying to provide a concrete and useful compromise.

The various system levels cited above (and all the other, lower abstraction levels that may actually concur to fully describe a real system) provide a conceptual framework that suggests how to meaningfully partition an agent formal model, so that each part expresses properties pertaining to a specific system level. In this paper we adhere to this very methodology, dividing the specifica-

tion of an agent collaborative behavior into different tiers. In particular, due to the increased accuracy of a formal description with respect to a natural language one, these tiers are clearly finer than the usual conceptual system levels, but they can still be associated to a well defined system level.

As a use case for our approach we consider ParADE [2], a complete environment for the development of agents, taking into account social, agent and knowledge level issues, beyond of course interfacing itself with the lower system levels by means of its runtime support.

2 Outline

The remainder of the paper is organized as follows. Section 3 is devoted to describing the basic framework of labeled transition systems [6], which is exploited in this paper to formalize the behavior of agents and of their conceptual subparts.

Section 4 describes our novel approach to agent modeling. This is based on the idea of considering a partial specification of an agent behavior – namely, a grey-box model abstracting away from the agent inner details below a certain abstraction threshold [18] –, which can be refined by the specification of a *completion*. Such a refinement takes into account new aspects of the agent behavior, and leads to a new grey-box model that can be later refined again.

In order to show how our approach can be used to formalize an agent behavior in an incremental way, by separating different aspects of an agent design in different specification tiers, we take as a reference the ParADE framework for building agents [2], whose main characteristics are described in Section 5. ParADE exploits the formal semantics of a FIPA-like Agent Communication Language (ACL) to gain semantic interoperability, and provides *interaction laws* as a design mechanism to specify the agent social role. We chose to use this framework instead of others such as those based on the FIPA standard, the 3APL language [7], or Jack [13], since ParADE incorporates and keeps conceptually separated the many distinctive features that we believe are crucial when modeling MAS.

Section 6 provides a formalization of an agent behavior adhering to the main design choices of ParADE framework and considering five different tiers each dealing with a relevant aspect, namely, interaction, language, ontology, social role, and internal reasoning. Section 7 reports on related and future works, drawing concluding remarks.

3 Transition Systems

The semantic approach we describe in this paper is based on the framework of *(labeled) transition systems* (LTS) [6, 14], which originated in the field of concurrency theory to provide an operational semantics for process algebras [3]. Here, LTS are used in a slightly different, yet common fashion: instead of focusing on the semantics of a language or algebra, we apply them to the description of the interactive behavior of a system, which in our case is an agent (or a subpart of it). This application of LTS is still referred to as an operational

semantics, in that it describes the single-step capabilities of an agent to interact with the environment and to perform internal computations, resembling the idea of operational semantics in the context of programming languages.

A LTS is a triple $\langle X, \longrightarrow, Act \rangle$, where X is the set of states of the system of interest, Act is called the set of *actions*, and $\longrightarrow \subseteq X \times Act \times X$ is the *transition relation*. Let $x, x' \in X$ and $act \in Act$, then $\langle x, act, x' \rangle \in \longrightarrow$ is written $x \xrightarrow{act} x'$ for short, and means that the system may move from state x to state x' by way of action act.

Actions in a LTS can be given various interpretations. In the most abstract setting, they are meant to provide a high-level description of a system evolution, abstracting away from details of the inner system state change. In the sequence of transitions $x_0 \xrightarrow{a_0} x_1 \xrightarrow{a_1} x_2 \xrightarrow{a_2} \ldots \xrightarrow{a_{n-1}} x_n$, the system evolution characterized by states $x_0, x_1, \ldots x_n$ is associated to the actions sequence $a_0, \ldots, a_{n-1}$, which can be though of as an abstract view of that evolution. In a sense, actions can be used to describe what an external, virtual entity is allowed to perceive of the system evolution, generally providing only a partial description. In the case where LTS are exploited to represent the behavior of interactive systems, actions typically represent the single interaction acts of the system.

Several remarkable concepts are promoted by LTS, such as the notions of system *observation* and *refinement* of specifications [11]. Since actions in a LTS can be interpreted as a view of the system transition from a state to another, it is then possible to characterize a whole system evolution in terms of the actions permitted at each step and the actions actually executed. This characterization is made according to a given *observation semantics*, which associates to each evolution a mathematical description called *observation*. For instance, according to observation semantics called *trace semantics* [6], an observation is simply made of an allowed sequence of actions. By nondeterminism, generally more observations are possible for a system – either because of the environment interacting in different ways, or because of different internal behaviors –, so that according to the given observation semantics the software component of interest can be characterized by the set of all its possible observations. The notion of refinement then naturally comes in: a system description is considered an "implementation" of another – i.e., it is more directly executable – if it is more deterministic, that is, if it allows for strictly fewer observations [6]. This notion is particularly crucial in the context of software engineering: when some safety property of interest is verified on a system specification, it continues to hold when the system is substituted with a refinement of it.

4 Grey-box and Refinement

In [17], the rationale behind the grey-box modeling approach for agents is introduced. Its key idea is to represent an agent behavior by focusing on its part dealing with the interactions with the environment, called the *(agent) core*, while abstracting away from the complex inner details, namely, from the *(agent) internal machinery*. Clearly, deciding which aspects should be specified in the core

and which should be abstracted away depends on the abstraction level of interest. In order to deal with agent proactiveness, one of the key notions introduced by this approach is that of *spontaneous move*, which is an event occurring within the internal machinery that may influence the agent interactions, thus affecting the behavior of the core. So, while complexity of agents can be harnessed by choosing the proper abstraction level, the internal machinery behavior can be anyway taken into account by supposing that spontaneous moves can nondeterministically occur.

In [17, 18] a formal framework based on LTS is introduced to show that the grey-box modeling approach is particularly suitable for describing the agent abstraction. Based on this general idea, in this paper we go further developing this formal approach. We not only represent grey-box models as interactive abstractions, but also define a mechanism by which they can be refined by adding the specification of a new behavior, thus lowering the abstraction level and focusing on new details.

Notation In the remainder of the paper, given any set X, this is automatically ranged over by the variable x and its decorations $x', x'', x_0, \ldots$ – and analogously, a set Any is ranged over by any, any', etcetera, and similarly for all sets. The set of multisets over X is denoted by $\overline{X}$ and ranged over by the variable $\overline{x}$ and its decorations; union of multisets $\overline{x}_1$ and $\overline{x}_2$ is denoted by symbol $\overline{x}_1||\overline{x}_2$; $\bullet$ is the empty multiset. Given any set X, symbol $\bot$ is used to denote an exception value in the set $X_\bot$ defined as $X \cup \{\bot\}$, which is ranged over by the variable $x_\bot$ and its decorations.

A Formal Framework for Grey-Box Modeling Formally, an agent core is defined by a mathematical structure $\mathcal{A} = \langle I, O, P, X, E, U, \rightarrow_{\mathcal{A}} \rangle$. I is the set of acts that can be listened by the agent, O is the set of acts that the agent can perform on the environment. Both I and O can be *communicative acts*, involving the exchange of messages between agents, or *physical acts*, involving physical action in the agent environment, such as e.g. listening a stimulus from a sensor or moving a mechanical device by an actuator. P is the set of *place* states (or the set of *places* for short), which are the states of the agent core that can be observed by the agent internal machinery. X is the set of states of the remaining part of the agent core, so that $P \times X$ define the set of core states. E is the set of *events* occurring within the agent core and possibly affecting the internal machinery, while U is the set of *updates* notified by the internal machinery to the agent core, modeling the notion of spontaneous move. Relation $\rightarrow_{\mathcal{A}}$ is a transition relation of the kind $\rightarrow_{\mathcal{A}} \subseteq (P \times X) \times Act_{\mathcal{A}} \times (P \times X)$, defining how the agent core state evolves as actions in the set $Act_{\mathcal{A}}$ occur. These actions can be of five kinds according to the syntax $Act_{\mathcal{A}} ::= \tau \mid ?i \mid !o \mid \rhd e \mid \lhd u$. Orderly, an action can represent the silent, internal computation τ, the agent listening act i, the agent executing act o, the event e occurring, and the update u being notified by the internal machinery.

Refinement of a Grey-Box Model It is common practice of system analysis and design to start considering a system at an high abstraction level, which

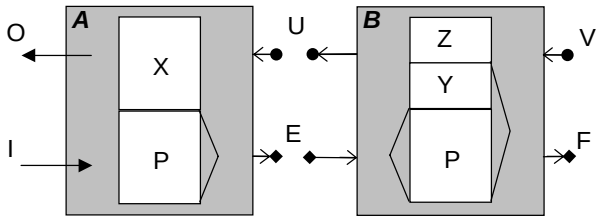

Fig. 1. Core and Completion

is later lowered in order to consider much more details. Then, a key feature of our methodological approach is to refine the specification $\mathcal{A}$ by specifying some of aspects of the internal machinery, that is, by conceptually moving them from the internal machinery to the agent core. This is done by composing the specification $\mathcal{A}$ by a new specification $\mathcal{B}$ called *completion*, so that the resulting system $\mathcal{C} = \mathcal{A} \otimes \mathcal{B}$ has still the structure of an agent core specification likewise $\mathcal{A}$ – hence it is still a grey-box model –, yet providing a number of additional details about the agent behavior.

A pictorial representation of our composition technique is shown in Figure 1. Formally, $\mathcal{B}$ is a structure of the kind $\langle Y, Z, F, V, \rightarrow_{\mathcal{B}} \rangle$. $W = Y \times Z$ is the set of local states added by $\mathcal{B}$ to the core state, Y is the part of it which is observable – along with P – to the new internal machinery, while Z is the local, hidden part of $\mathcal{B}$. $F \subseteq E$ and $V \subseteq U$ are respectively the new events and updates by which $\mathcal{C}$ interacts with its internal machinery. Transition relation $\rightarrow_{\mathcal{B}}$, describing the behavior of the completion, is of the kind $W \times Act_{\mathcal{B}} \times W$, with $Act_{\mathcal{B}}$ being defined by the syntax $Act_{\mathcal{B}} := p : \tau \mid p : e \rhd f_{\perp} \mid p : u_{\perp} \lhd v$. Action $p : \tau$ is the silent action occurring when $\mathcal{A}$ is in place p. The action $p : e \rhd f_{\perp}$ is executed when the place of $\mathcal{A}$ is p: event e occurs that causes f to be propagated towards the internal machinery. The subcase $p : e \rhd \perp$ means that no event f is generated by event e. The action $p : u_{\perp} \lhd v$ means that when the update v is listened from $\mathcal{B}$ while $\mathcal{A}$ is in place p, then u is notified to $\mathcal{A}$ (or nothing if u is $\perp$). Notice that the occurrence of place p in these actions justifies the characterization of the place as the core's subpart observable by the internal machinery.

Given models $\mathcal{A}$ and $\mathcal{B}$, $\mathcal{C} = \mathcal{A} \otimes \mathcal{B}$ is defined as the agent core specification $\langle I, O, P \times Y, X \times Z, F, V, \rightarrow_{\mathcal{C}} \rangle$. The relation $\rightarrow_{\mathcal{C}}$, which is here of the kind $\rightarrow_{\mathcal{C}} \subseteq (P \times X \times W) \times Act_{\mathcal{C}} \times (P \times X \times W)$, is defined by the rules shown in Figure 2. Rules [I] and [O] state that input and output actions of the overall system $\mathcal{C}$ are executed on the agent core $\mathcal{A}$. Rules [AT] and [BT] say that a τ action in either $\mathcal{A}$ and $\mathcal{B}$ is executed in isolation, and makes $\mathcal{C}$ execute a τ action itself. Rules [AE] and [BE] describe the occurrence of an event in $\mathcal{A}$: in the first case no event f is generated by $\mathcal{B}$, in the second case $\mathcal{B}$ produces an event f. Analogously, [AU] and [BU] deal with updates, with [BU] considering the case where an update is propagated from the internal machinery to $\mathcal{A}$ and [AU] where no update is propagated to $\mathcal{A}$.

Following an approach similar to the one reported in [15], which is not reported here for brevity, it is possible to prove that agent core $\mathcal{C}$ has a more

$$\frac{\langle p,x\rangle \xrightarrow{?i}_{\mathcal{A}} \langle p',x'\rangle}{\langle p,x,w\rangle \xrightarrow{?i}_{\mathcal{C}} \langle p',x',w\rangle}\ [\mathrm{I}] \qquad \frac{\langle p,x\rangle \xrightarrow{!o}_{\mathcal{A}} \langle p',x'\rangle}{\langle p,x,w\rangle \xrightarrow{!o}_{\mathcal{C}} \langle p',x',w\rangle}\ [\mathrm{O}]$$

$$\frac{\langle p,x\rangle \xrightarrow{\tau}_{\mathcal{A}} \langle p',x'\rangle}{\langle p,x,w\rangle \xrightarrow{\tau}_{\mathcal{C}} \langle p',x',w\rangle}\ [\mathrm{AT}] \qquad \frac{\langle p,x\rangle \xrightarrow{\triangleright e}_{\mathcal{A}} \langle p',x'\rangle \qquad w \xrightarrow{p':e\triangleright\perp}_{\mathcal{B}} w'}{\langle p,x,w\rangle \xrightarrow{\tau}_{\mathcal{C}} \langle p',x',w'\rangle}\ [\mathrm{AE}]$$

$$\frac{w \xrightarrow{p:\tau}_{\mathcal{B}} w'}{\langle p,x,w\rangle \xrightarrow{\tau}_{\mathcal{C}} \langle p,x,w'\rangle}\ [\mathrm{BT}] \qquad \frac{\langle p,x\rangle \xrightarrow{\triangleright e}_{\mathcal{A}} \langle p',x'\rangle \qquad w \xrightarrow{p':e\triangleright f}_{\mathcal{B}} w'}{\langle p,x,w\rangle \xrightarrow{\triangleright f}_{\mathcal{C}} \langle p',x',w'\rangle}\ [\mathrm{BE}]$$

$$\frac{w \xrightarrow{p:\perp\triangleleft v}_{\mathcal{B}} w'}{\langle p,x,w\rangle \xrightarrow{\triangleleft v}_{\mathcal{C}} \langle p,x,w'\rangle}\ [\mathrm{AU}] \qquad \frac{w \xrightarrow{p:u\triangleleft v}_{\mathcal{B}} w' \qquad \langle p,x\rangle \xrightarrow{\triangleleft u}_{\mathcal{A}} \langle p',x'\rangle}{\langle p,x,w\rangle \xrightarrow{\triangleleft v}_{\mathcal{C}} \langle p',x',w'\rangle}\ [\mathrm{BU}]$$

Fig. 2. Rules for composition of an agent core and its completion

refined behavior than agent core $\mathcal{A}$ in the sense specified e.g. by trace semantics [6]. Hence, our refinement technique can indeed be considered as a way of deepening a specification towards implementation issues.

5 ParADE

ParADE, the *Parma Agent Development Environment* [2], is a development framework that provides the agent developer with high-level abstractions like beliefs and goals. It implements a hybrid agent architecture capable of supporting autonomy and intelligent behaviors by exploiting the semantics of its ACL – resembling the FIPA ACL [5] but with a much lighter semantics. Such an architecture is basically goal-oriented but it also integrates reactive behaviors.

ParADE ACL provides an operational means for agents to exchange representations of beliefs, intentions, and capabilities. The semantics is modeled as the effect that the sender wishes to achieve when sending the message. We stick to a syntax similar to that of FIPA ACL: $\phi \in \Phi$ stands for any predicative formula, a and b for actions, s for the identifier of an agent sending a message, r for the receiver, $B_j\phi$ for "entity j believes ϕ", $I_j\phi$ for "entity j intends ϕ", and $done(a)$ for "action a has just happened". Each message is associated to a feasibility precondition (FP) that must hold in the sender and a rational effect (RE) that the send should intend: for instance $inform(s,r,\phi)$ has the precondition $B_s\phi$ – namely, the sender must believe ϕ – and $request(s,r,a)$ has the rational effect $done(a)$ – the sender should intend a to be executed. As a result, a precise semantics can be assigned to a message by its receiver, for example message $request(s,r,a)$ is associated to the semantics $B_r I_s done(a)$ (the receiver believes that the sender intends a to be executed) – see [2] for more details.

The ParADE ACL provides a means for exchanging complex logic formulae through simple messages as the receiver can assert what the sender is intending. Isolated messages are not sufficient to allow agents to communicate fruitfully. The classic example is the case of an agent requesting another agent to perform an action: in the case of asynchronous messages there is no guarantee that the

receiver would act in response to a message. Moreover, the semantics of a single message might not be sufficient to express application-specific constraints. The semantics of performative *request* does not impose the receiver to communicate to the sender that the requested action has been actually performed. The sender might hang indefinitely while waiting for the receiver to tell it that the action has been performed.

Using an ACL provides an agent with a linguistic environment that attaches semantics to its utterances. By virtue of the speech act theory, utterances become actions, with their attached pre- and post-conditions. Still, an agent is situated within an environment that is not limited to its social milieu; there are actions to perform and events to perceive that are not linguistic. Moreover, even linguistic acts often refer to entities belonging to something else than the language itself. These entities belong to the *domain* of the discourse, i.e. the external environment itself, and are typically described within a *domain model* or *ontology*. This ontology not only shapes the content of exchanged messages, but can also constrain the available actions of an agent associating them with domain-specific pre-conditions. In ParADE, the domain model works as the non-linguistic counterpart of the ACL semantics, enabling uniform processing of both the interactions with other agents and with the external environment.

In order to support socially fruitful communication, the ParADE agent model provides *interaction laws*. These are rules that an agent decides to adopt to govern its interactions with other agents. The interaction laws that an agent decides to follow are part of its capabilities and they are published. As an example, consider the interaction law $I_r done(a) \leftarrow B_r I_s done(a)$ for the agent r, which means that whenever he believes that agent s intends the action a to be executed, then this becomes an intention of r as well. This law should characterize an agent always willing to cooperate with s. In particular, as shown in the example we provide in Section 6.2, if s asks r to execute action a by means of a *request* message, then r will come to believe that the rational effect of the message is intended by the sender, that is $B_r I_s done(a)$. Then, the above interaction rule may be applied that makes $done(a)$ become an intention of r as well. In general, interaction laws are an elegant and flexible way to describe interaction protocols, they can be linked to the possible roles that an agent can play in the MAS, and they may also vary over time.

The ACL semantics, the domain model, and the interaction laws work together to expose an observable model of a ParADE agent that enables semantically meaningful communication with others. However, they don't fully specify the agent behavior; an agent programmer using ParADE is able to define several rules that drive the evolution of the agent mental states, only a subset of which will be published as interaction laws. This means that there is an internal set of rules that completes the agent specification, but is visible to nobody else than the agent designer. Though in most cases the designer wants to abstract away from the agent internal evolution rules, it is still useful to notice that even this relatively low-level part of the agent specification is still described declaratively.

6 Iterated Refinement

In this section we shown an application of our technique for formalizing the behavior of an agent by successive refining an initial description, taking as a reference architecture the ParADE framework. We refer to the term *tier* when describing these different levels, each of which focuses on a different behavioral aspect and is then amenable to a separated description, thus fostering the understanding of its key features. The tiers we analyze not only correspond to the main features of ParADE, but also include most of the main aspects that an agent implementation has to take into account: orderly *(i)* the *interaction tier*, managing interactions with the environment, *(ii)* the *linguistic tier*, dealing with aspects related to the ACL semantics, *(iii)* the *domain tier*, concerning the specific ontology of the application, *(iv)* the *social tier*, where interaction laws define the role and public peculiarities of the agent, and finally *(v)* the *internal tier*, dealing with other aspects such as private laws, planning, and proactiveness. The order of these tiers reflects their impact on the external, observable behavior of the agent, from tackling interaction issues to considering internal reasoning. It worth noting that the actual implementation of a ParADE agent is not actually separated into these tiers, which are instead a modeling tool useful at design-time independently from the actual agent implementation, representing conceptual levels that isolate the different behavioral aspects of agents within MAS.

In order to keep the presentation clear and reasonably compact, locally to a tier we sometime abstract away from some policy or management that is of a too lower abstraction level – either concerning details of the ParADE architecture or of the specific agent peculiarities –, referring to some generic function, relation or set encapsulating the actual behavior.

6.1 Splitting the Agent Specification into Tiers

The Interaction Tier From the point of view of the MAS, the most distinctive aspect of an agent is its ability of interacting with other agents and of performing actions on the environment. So, not surprisingly, the first tier we introduce is the one dealing with the interaction abilities of an agent, namely listening and performing acts. In the following, we denote by Ic and Oc the sets of input and output communicative acts, and by Ip and Op the sets of input and output physical acts, so that $I = Ic \cup Ip$ and $O = Oc \cup Op$.

The task of this tier is to decouple the actual agent interactions with respect to their internal processing, which is a common feature of almost all agent implementations – sometimes mentioned to contrast the notion of agent with respect to that of component (or object). Here, we suppose that the set X of states unobservable by the internal machinery is defined as a multiset of pending input acts waiting to be processed and pending output acts waiting to be executed, namely, $X = \overline{I \cup O}$. The set P of places is defined as the set of agent mental states: a mental state $\Phi_M \in P$ at a given time is a set of formulae $\phi \in \Phi$, representing current beliefs and intentions of the agent. This is explicitly represented

in this tier because, typically, the occurrence of interactions has an immediate representation in the knowledge of an agent.

A composition operator between such formulae is defined so that if $\Phi_M \subseteq \Phi$ is the current mental state and $\Phi_n \subset \Phi$ contains some new beliefs and intentions, then $\Phi_M \circ \Phi_n$ is the new mental state obtained from Φ_M by adding formulae in Φ_n. Clearly, this composition should be defined according to the specific update policy for mental states, which we here abstract away from, analogously e.g. to [16]. Similarly, given a mental state Φ_M, we suppose that if this is of the kind $\Phi' \circ \Phi_n$ then the agent mental state includes all the beliefs and intentions of Φ_n.

An element of the set $P \times X$ is hence a couple $\langle \Phi_M, \overline{i}||\overline{o} \rangle$, here denoted as $\Phi_M||\overline{i}||\overline{o}$ for uniformity of notation. The set of events E coincides to I, representing the input acts listened from outside. The set of updates U is $O \cup \mathcal{P}(\Phi)$, representing either output acts to perform or requests to change the mental state by adding a subset of formulae in Φ (with $\mathcal{P}(\Phi)$ being the powerset of Φ). Transition relation $\rightarrow_{\mathcal{A}}$, describing the behavior of the interaction tier, is the defined by the rules:

$$\Phi_M \xrightarrow{?i}_{\mathcal{A}} i \;||\; \Phi_M \circ \{B_r done(i)\} \qquad \Phi_M||i \xrightarrow{\triangleright i}_{\mathcal{A}} \Phi_M$$

$$\Phi_M||o \xrightarrow{!o}_{\mathcal{A}} \Phi_M \circ \{B_r done(o)\} \qquad \Phi_M \xrightarrow{\triangleleft o}_{\mathcal{A}} \Phi_M||o$$

$$\Phi_M \xrightarrow{\triangleleft \Phi_n}_{\mathcal{A}} \Phi_M \circ \Phi_n$$

The first rule simply states that when input act i is received ($?i$), the mental state is updated so as to reflect the reception ($B_r done(i)$), and then the act i is stored in the state X waiting for being notified as an event by means of the second rule ($\triangleright i$). The third and fourth rule, conversely handle output acts, which are inserted in X by updates ($\triangleleft o$), and are later sent outside by output actions ($!o$), affecting the mental state. Finally, fifth rule handles a mental state update requested by the internal machinery ($\triangleleft \Phi_n$), according to the semantics of composition operator $\circ$.

Different implementations of interactions dispatching could be specified in this tier, including the case where acts are stored into queues representing the agent mailbox instead of being immediately served as in the model above – where a precondition for receiving and sending acts is that no act is still pending.

The Linguistic Tier A fundamental aspect of the ParADE framework is that it exploits an ACL semantics to enjoy true semantic interoperability between agents [1], as currently promoted by the FIPA ACL and KQML approaches. So, as next tier we take into account the constraints imposed on communicative acts by the specific ACL. Then, we define a completion $\mathcal{B}$ specifying the main concepts of the ACL, namely, the syntax of messages and their impact on the agent mental state.

Syntax is defined by introducing sets $I_{ACL} \subseteq Ic$ and $O_{ACL} \subseteq Oc$ of communicative acts allowed by the ACL: when a message ic is received that does not conform to that syntax this is simply ignored. The set of visible states Y is here void, since no new information is to be added to the mental state Φ_M that should

be visible to the internal machinery. Instead, set Z of local, invisible states is used to store scheduled updates $\overline{u}$ to be propagated to the interaction tier, thus $Z = \overline{U}$: in particular, such updates are requests for updating the mental state with formulae Φ_n. On the other hand, events F produced by this tier are of the kind $ic \in I_{ACL}$, while updates V coincides with U.

In order to deal with the effect of a communicative act on the agent mental state, we define $\mathit{eff}^I_M : I_{ACL} \mapsto \mathcal{P}(\Phi)$ as a function associating to a communicative act its intended effect on the mental state of the receiver – formed by beliefs and intentions to be added –, and $\mathit{cnd}^O_M : O_{ACL} \mapsto \mathcal{P}(\Phi)$ as the function associating to a communicative act the condition on the local mental state enabling its sending, expressed as facts the agent has to believe and intend. In the case of ParADE, and similarly to FIPA ACL, we have e.g. $\mathit{eff}^I_M(ic) = \{B_r FP(ic), B_r I_s RE(ic)\}$, that is, when the message is processed the receiver comes to believe that the feasibility preconditions of the act are satisfied, and that the sender intends its rational effects. Analogously, $\mathit{cnd}^O_M(oc) = \{B_s FP(oc), I_s RE(oc)\}$, that is, a message can be sent out only if the sender believes the feasibility preconditions and intends the rational effects of the act. The rules for transition relation $\rightarrow_{\mathcal{B}}$ that deal with communicative acts are as follows:

$$\bullet \xrightarrow{\Phi_M : ic \rhd ic}_{\mathcal{B}} \mathit{eff}^I_M(ic) \qquad \text{if } ic \in I_{ACL}$$

$$\bullet \xrightarrow{\Phi_M : ic \rhd \bot}_{\mathcal{B}} \bullet \qquad \text{if } ic \notin I_{ACL}$$

$$\bullet \xrightarrow{\Phi_M : oc \lhd oc}_{\mathcal{B}} \bullet \qquad \text{if } oc \in O_{ACL} \quad \text{and} \quad \Phi_M = \Phi'_M \circ \mathit{cnd}^O_M(oc)$$

$$\bullet \xrightarrow{\Phi_M : \bot \lhd oc}_{\mathcal{B}} \bullet \qquad \text{if } oc \notin O_{ACL} \quad \text{or} \quad \Phi_M \neq \Phi'_M \circ \mathit{cnd}^O_M(oc)$$

The first rule says that when the communicative act ic arrives that is correct with respect to the ACL ($ic \in I_{ACL}$), then this is redirected as event f (by an action of the kind $p : e \rhd f$), and an update for changing the mental state is correspondingly scheduled; the second rule says that acts that do not belong to the ACL are ignored. Dually, the third rule handles the case where an output act that is correct with respect to the ACL syntax is to be sent out, which is actually performed only if the proper conditions hold on the mental state (ϕ_M includes conditions $\mathit{cnd}^O_M(oc)$). On the other hand, if the act is not correct or these conditions are not satisfied, the fourth rule makes the update be simply ignored and discarded. The remaining rules are as follows:

$$\bullet \xrightarrow{\Phi_M : ip \rhd ip}_{\mathcal{B}} \bullet \qquad \bullet \xrightarrow{\Phi_M : op \lhd op}_{\mathcal{B}} \bullet \qquad \bullet \xrightarrow{\Phi_M : \bot \lhd \Phi_n}_{\mathcal{B}} \Phi_n \;\; u \xrightarrow{\Phi_M : u \lhd \bot}_{\mathcal{B}} \bullet$$

The first and second rule let input physical acts and output physical act to flow across the tier. The third rule reifies requests for changing the mental states within the tier, which by the fourth rule are redirected to the previous tier. Since this reification technique is exploited in the next tiers as well, the latter two rules are are assumed here to be included in the specification of both the ontology, social, and internal tier, even if they are not actually reported for brevity.

The Domain Tier The domain tier is the part of the agent specification that deals with aspects related to the ontology defining the application domain where the agent lives. First of all, the syntax of communicative acts is further constrained with respect to the ACL, since e.g. the actual content of a message is limited to those terms to which the ontology gives an interpretation. We therefore denote by $Ic_{ONT} \subseteq I_{ACL}$ and $Oc_{ONT} \subseteq O_{ACL}$ the communicative acts allowed by the specific ontology. Then, the ontology also defines what are the physical acts that may be listened and executed by the agent, which are denoted by the sets $Ip_{ONT} \subseteq Ip$ and $Op_{ONT} \subseteq Op$. The sets of acts allowed by the ontology are then naturally defined as $I_{ONT} = Ip_{ONT} \cup Ic_{ONT}$ and $O_{ONT} = Op_{ONT} \cup Oc_{ONT}$. Finally, each output physical act is associated to an expected rational effect, by a function $cnd^P_M : O_{ONT} \mapsto \mathcal{P}(\Phi)$, associating output acts to sets of formulae (facts to be intended and believed) that will become satisfied. This function is here supposed to associate to communicative acts a void set, while in the case of a physical act op we have e.g. $cnd^P_M(op) = \{I_s RE(op)\}$. In the case the domain ontology binds physical input acts to predicates in the mental state, it would be sensible to introduce also the management of feasibility preconditions analogously to the linguistic tier, however, this is not considered here for it is subject of current researches.

Similarly to the case of the linguistic tier, set F is a subset of E and $V = U$. Also, sets Y and Z are the same of previous case. The rules for transition relation $\rightarrow_{\mathcal{B}}$ handling input acts are as follows:

$$\bullet \xrightarrow{\Phi_M : i \rhd i}_{\mathcal{B}} \bullet \qquad \text{if } i \in I_{ONT} \qquad\qquad \bullet \xrightarrow{\Phi_M : i \rhd \perp}_{\mathcal{B}} \bullet \qquad \text{if } i \notin I_{ONT}$$

stating that when input act i arrives that is correct with respect to the ontology, then this is redirected as event f; otherwise, by the second rule the act is simply ignored. The case of output acts is handled dually:

$$\bullet \xrightarrow{\Phi_M : o \lhd o}_{\mathcal{B}} \bullet \qquad \text{if } o \in O_{ONT} \quad \text{and} \quad \Phi_M = \Phi'_M \circ cnd^P_M(o)$$
$$\bullet \xrightarrow{\Phi_M : \perp \lhd o}_{\mathcal{B}} \bullet \qquad \text{if } o \notin O_{ONT} \quad \text{or} \quad \Phi_M \neq \Phi'_M \circ cnd^P_M(o)$$

the output act is let flow toward the agent core only if it conforms to the ontology and satisfies the preconditions.

The Social Tier This tier deals with those peculiar aspects of an agent that characterize its collaborative (or social) behavior within the MAS. In particular, interaction laws can be applied that change the mental state under some conditions, e.g. making the agent react to some input by producing some output act. Since interaction laws can be of different kinds, we here just suppose that the set of interaction laws of an agent are modeled by a relation $ilaw \subseteq \mathcal{P}(\Phi) \times \mathcal{P}(\Phi)$, associating to the current mental state the new facts (possibly none) that the agent will believe and intend after applying some enabled law.

In particular, the social tier has void set Y, while set Z – as for previous tier – may contain some pending update, namely, some change on the mental state that

has to be applied. Events F and updates V coincide with E and U, respectively. The rule governing the relation transition of this tier is the following:

$$\bot \xrightarrow{\Phi_M : i \rhd i}_{\mathcal{B}} \Phi_n \qquad \text{if } \mathit{ilaw}(\Phi_M, \Phi_n)$$

saying that whenever a new input is received, interaction laws make a new mental state to be computed which will be propagated to the interaction tier as usual.

The Internal Tier The internal tier is the latter tier of our specification, which includes a number of remaining implementation features of the agent. Most notably, here a further set of laws – which are not however published but forms the hidden, unobservable behavior of the agent – can be applied that may have various purposes. On the one hand, these rules can be used to drive any agent behavior that cannot be ascribed to its social role, but rather to its implementation. To the end of the formalization presented here, we model this behavior by a *private laws* function *plaw* similar to function *ilaw*. On the other hand, the internal tier is also the part of the agent responsible for proactively requesting some action to be performed. To this end, we consider a precondition function $cnd_M^E : o \mapsto \mathcal{P}(\Phi)$ associating to an output act the conditions enabling its execution. In the basic case we have $cnd_M^E(o) = \{I_s done(o), B_s FP(o)\}$, that is, the agent should intend to execute the action and should believe its feasibility preconditions.

Since no further refining is here considered, set of events F and updates U are here void. Moreover, sets Z and Y coincide with previous tier. Other than the two usual rules for dispatching updates as in the previous tier, we have the two rules:

$$\bot \xrightarrow{\Phi_M : i \rhd \bot}_{\mathcal{B}} \Phi_n \qquad \text{if } \mathit{plaw}(\Phi_M, \Phi_n)$$

$$\bot \xrightarrow{\Phi_M \circ cnd_M^E(o) : \tau}_{\mathcal{B}} o$$

While the former makes the mental state be updated by applying a private law, the second rule says that each time a silent action is performed by the agent core and the precondition for executing an action is satisfied, then that action is scheduled by sending the update o.

6.2 An Example

In order to grasp the flavor of this formalization, we here consider an example of simple conversation for an agent, and describe the corresponding sequence of transitions modeling its behavior. In particular, we consider the case that the agent i receives from another agent j a message a of the kind $\mathit{request}(j, i, b)$ where $b = \mathit{inform}(i, j, \phi)$, requesting i to send a message declaring that he believes ϕ. In the case where i actually believes ϕ, and by the interaction law $I_i done(a) \leftarrow B_i I_j done(a)$ – stating that i is always willing to execute the actions that j intends to be executed – we should obtain that i sends the message $\mathit{inform}(i, j, \phi)$. This protocol is a very simplified version of the FIPA-request protocol, and for the

sake of brevity does not take into account aspects such as agreement, refusal, and so on.

The represented portion of the agent state, namely the state of the agent core, is expressed as a tuple with the state of each tier in its elements, orderly. We start by considering the agent i with a mental state of the kind $\Phi_M \circ \{B_i\phi\}$, namely, initially believing ϕ.

(1) $\langle \Phi \circ \{B_i\phi\}, \bullet, \bullet, \bullet, \bullet \rangle \xrightarrow{?a}_{\mathcal{C}}$

(2) $\langle a || \ \Phi \circ \{B_i\phi, B_i done(a)\}, \bullet, \bullet, \bullet, \bullet \rangle \xrightarrow{\tau}_{\mathcal{C}}$

(3) $\langle \Phi \circ \{B_i\phi, B_i done(a)\}, \{B_i I_j done(b)\}, \bullet, \{I_i done(b)\}, \bullet \rangle \xrightarrow{\tau}_{\mathcal{C}}$

(4) $\langle \Phi \circ \{B_i\phi, B_i done(a), B_i I_j done(b)\}, \bullet, \bullet, \{I_i done(b)\}, \bullet \rangle \xrightarrow{\tau}_{\mathcal{C}}$

(5) $\langle \Phi \circ \{B_i\phi, B_i done(a), B_i I_j done(b), I_i done(b)\}, \bullet, \bullet, \bullet, \bullet \rangle \xrightarrow{\tau}_{\mathcal{C}}$

(6) $\langle \Phi \circ \{B_i\phi, B_i done(a), B_i I_j done(b), I_i done(b)\}, \bullet, \bullet, \bullet, b \rangle \xrightarrow{\tau}_{\mathcal{C}}$

(7) $\langle b \ || \ \Phi \circ \{B_i\phi, B_i done(a), B_i I_j done(b), I_i done(b)\}, \bullet, \bullet, \bullet, \bullet \rangle \xrightarrow{!b}_{\mathcal{C}}$
$\langle \Phi \circ \{B_i\phi, B_i done(a), B_i I_j done(b), I_i done(b), B_i done(b)\}, \bullet, \bullet, \bullet, \bullet \rangle$

The first transition models the reception of communicative act a: the interaction tier enqueues the request and adds to its mental state $B_i done(a)$. In the second transition, a is processed by flowing across each tier: in the linguistic tier it makes the agent believing the sender's intentions $B_i I_j done(b)$ (feasibility preconditions are here avoided for simplicity), while in the social tier the interaction law $I_i done(b) \leftarrow B_i I_j done(b)$ is applied leading to the new intention $I_i done(b)$. In the third and fourth rule both these facts are propagated back to the interaction tier affecting the mental state. In the fifth rule, the internal tier recognizes the intention $I_i done(b)$, and by means of function cnd_M^E fires the action b, which the sixth rule moves to the interaction tier. Finally, since the feasibility precondition $B_i\phi$ to b is satisfied the communicative act b is sent outside. Notice that another precondition for sendind would be $I_i B_j \phi$ (the agent i must intend the rational effect of b), which is not considered here for in most agent architectures – such as FIPA – it is generally entailed by $I_i done(b)$. Then, the spurious formulae $\{B_i I_j done(b), I_i done(b)\}$ are meant to be subsequently dropped from the mental state by means of private laws in the internal tier.

7 Conclusions

In this paper we tackle the problem of formalizing the behavior of complex agents by a technique underlying the notion of grey-box modeling. An existing description, focusing on the agent behavior at a given abstraction level, can be refined by adding the specification of a new aspect. This formal approach is here described and put to test on the ParADE framework, describing a number of relevant agent aspects such as the ACL and its semantics, ontology, and social role, providing a means by which different system levels and views of a MAS can be represented in isolation. A crucial role in this methodology is played by the framework of LTS, which allows us to describe in an operational way the behavior of an agent implementation, and facilitates the task of composing specifications.

Related Works LTS have been applied to agent systems in other works as well. The grey-box modeling approach has been first studied in the context of the so-called observation framework [17, 18], where agents are modeled as sources of information. In [15], a grey-box formal approach similar to the one described in this paper has been applied to represent the semantics of ACLs. In [16], specifications based on operational semantics are evaluated for ACLs, using a process algebra to describe how interaction may affect the mental state. In [10], agent internal aspects such as planning are modeled through the *ψ-calculus*, a language whose operational semantics is meant to capture the allowed internal dynamics of the agent. Another significant formalism from concurrency theory that is being successfully applied to MAS interaction specification are Colored Petri Nets [9], as described e.g. in Ferber's book [4]. Most of these previous works applying LTS and CPN to MAS interaction modeling focused on agent conversations and interaction protocols, while we claim that the approach presented in this paper can be fruitfully applied to the whole design of agent-based systems, enjoying the fact that separating specifications may allow to study their properties separately. A deeper investigation will reveal further relationship with the work in [10], which is closer to our spirit.

Future Works A main advantage of an operational specification is that it should allow properties of interest to be formally proved. In our case, where one such specification is applied at design-time to model an agent collaborative behavior, we may expect them to state the soundness – in the broad meaning of the term – of an agent design, especially as far as its interactions with other agents of the MAS are concerned. Notice that the specification we provided is parametric in a number of mathematical structures (I_{ACL}, O_{ACL}, eff^I_M, cnd^O_M, I_{ONT}, O_{ONT}, cnd^P_M, *ilaw*, *plaw*, cnd^E_M), describing both peculiar aspects of the agent model (e.g. ACL semantics) as well as peculiar aspects of the individual agent (e.g. its private laws).

Some of the properties of interest may be independent of these parameters, mostly concerning the internal structure of the specification. For instance, by-hand proofs should emphasize that any reception of an input act a is reified as a believe $B_i\mathit{done}(a)$, and that requests for mental state updates raised by the internal tier are eventually applied (flowing towards the interaction tier) preserving their order.

Other properties, that instead depend on the parameters of the specification, can be more concerned with the correctness of the resulting agent behavior, that is, concerning its rationality and social attitude. As an example, our specification may provide a suitable framework for proving that under a given ACL semantics (eff^I_M, cnd^O_M, I_{ACL}, O_{ACL}) and a given ontology (cnd^P_M, I_{ONT}, O_{ONT}), a specific set of interact laws (*ilaw*) and private laws (*plaw*) are sufficient for the agent correctly participating in a given conversation protocol, e.g. the simple request conversation showed in Section 6.2.

An expected feature of our framework is that properties concerning only one aspect of the MAS design could be studied considering agent specifications only up to the corresponding tier, by virtue of the refinement notion of transition

systems. For instance, stating that a conversation protocol is consistent with respect to the ACL semantics should require us to consider agents as made of the interaction and linguistic tiers only: as mentioned in Section 4 such a specification refines the actual agent behavior, but consider all the aspects of interest. On the other hand, evaluating the evolutions of conversations depending on the agents social attitude requires to consider the domain and social tiers as well.

In general, deepening all these issues, that concern the applicability of formal verification tools to the validation of MAS design, is the main future work of this research.

References

1. F. Bergenti. A discussion of two major benefits of using agents in software development. In *Engineering Societies in the Agents World (ESAW 2002)*, volume 2577 of *LNAI*, pages 1–12. Springer-Verlag, 2003.
2. F. Bergenti and A. Poggi. A development toolkit to realize autonomous and interoperable agents. In *Conference on Autonomous Agents*, pages 632–639, 2001.
3. J. A. Bergstra, A. Ponse, and S. A. Smolka, editors. *Handbook of Process Algebra*. North-Holland, 2001.
4. J. Ferber. *Multi-Agent Systems: An Introduction to Distributed Artifical Intelligence*. Addison-Wesley, London, 1999.
5. FIPA. FIPA communicative act library specification. `http://www.fipa.org`, 2000. Doc. XC00037H.
6. R. v. Glabbeek. The linear time – branching time spectrum I. The semantics of concrete, sequential processes. In Bergstra et al. [3], chapter 1, pages 3–100.
7. K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J.-J. C. Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4), 1999.
8. N. R. Jennings. On agent-based software engineering. *Artificial Intelligence*, 117:277–296, 2000.
9. K. Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use*. Springer-Verlag, Berlin, 1992.
10. D. Kinny. Vip: a visual programming language for plan execution systems. In *AAMAS 2002*, pages 721–728. ACM Press, 2002.
11. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
12. A. Newell. The knowledge level. *Artificial Intelligence*, 18:87–127, 1982.
13. E. Norling and F. E. Ritter. Embodying the jack agent architecture. In *14th Australian Joint Conference on Artificial Intelligence*, volume 2256 of *LNCS*, pages 368–377. Springer, 2001.
14. G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Department of Computer Science, AArhus University, 1991.
15. G. Rimassa and M. Viroli. An operational framework for the semantics of agent communication languages. In *Engineering Societies in the Agents World (ESAW 2002)*, volume 2577 of *LNAI*, pages 111–125. Springer-Verlag, 2003.
16. R. M. van Eijk, F. S. de Boer, W. van der Hoek, and J.-J. C. Meyer. Operational semantics for agent communication languages. In *Issues in Agent Communication*, volume 1916 of *LNAI*, pages 80–95. Springer, 2000.
17. M. Viroli and A. Omicini. Modelling agents as observable sources. *Journal of Universal Computer Science*, 8, 2002.
18. M. Viroli and A. Omicini. Specifying agent observable behaviour. In *AAMAS 2002*, Bologna, Italy, 15–19 July 2002. ACM.

A Logic-Based Infrastructure for Reconfiguring Applications *

Marco Castaldi Stefania Costantini Stefano Gentile Arianna Tocchio

Università degli Studi di L'Aquila
Dipartimento di Informatica
Via Vetoio, Loc. Coppito, I-67010 L'Aquila - Italy
{castaldi, stefcost, gentile, tocchio}@di.univaq.it

Abstract. This paper proposes the DALI Multiagent System, which is a logic programming environment for developing agent-based applications, as a tool for component-based software management based on coordination. In particular we show the usefulness of the integration between DALI and the agent-based Lira system, which is a Light-weight Infrastructure for Reconfiguring Applications. We argue that using intelligent agents for managing component-based software systems makes it possible to: (i) perform monitoring and supervision upon complex properties of a system, such as for instance performance; (ii) perform global reconfigurations dynamically through the cooperation of intelligent agents.

1 Introduction

After a long predominance of imperative and object oriented languages in the software development process, declarative languages, thanks to new effi cient implementations, have recently regained attention as an attractive programming paradigm for the development of complex applications, in particular related to the Internet, or more generally to distributed application contexts. Declarative methods exhibit important well known advantages: (i) the reduction, also in terms of "lines of code", of the effort required for solving a problem, (ii) the actual reduction of errors introduced in the application, (iii) the fast prototyping of complex applications, that reduces "time to market" and development costs of business applications. In many cases, these applications are better implemented by using agents technology: declarative languages make the implementation of intelligent agents easier and effective. In our opinion, agent-based distributed applications give really a chance to Artifi cial Intelligence to show its usefulness in practical contexts.

In this paper we show how DALI, a new logic-based declarative language for agents and multi-agent systems, supports the development of innovative agent-based applications. We consider the topic of distributed component management in the context of

* We acknowledge the support of MIUR 40% project *Aggregate- and number-reasoning for computing: from decision algorithms to constraint programming with multisets, sets, and maps* and by the *Information Society Technologies programme of the European Commission, Future and Emerging Technologies* under the IST-2001-37004 WASP project.

Large Scale Distributed Component Based Applications (LSDCBA). The role of agents is that of monitoring and reconfiguring the system, in order to dynamically maintain some critical non-functional properties such as performance, high availability or security. The possibility of keeping a complex system under control while running is often a key factor for the success of complex and expensive systems. Several so-called "infrastructures" are being proposed to this purpose. In particular, as an interesting example we consider Lira, an agent-based infrastructure created for dynamic and automatic reconfigurations. As a case study, we have integrated Lira and DALI to create agents able to manage heterogeneous software components.

The DALI language [8] [9] is a Prolog-like logic programming language, equipped with reactive and proactive capabilities. The definition of DALI formalizes in a declarative way different basic patterns for reactivity, proactivity, internal "thinking", and "memory". The language introduces different classes of events: external, internal, present and past. Like Prolog, DALI can be useful and effective for rapid prototyping of light applications.

Lira [5] [11] [7] [6] has been recently defined (and fully implemented as a prototypical version in Java) to perform application reconfiguration in critical domains, such as for instance mobile and wireless systems and networks, risk management, e-government, environment supervision and monitoring. Each application/component of a complex system is managed by an attached agent, and there is a hierarchy of agents performing different tasks. Reconfigurations are dynamic and automatic: they are performed while the application is running and as a reaction to some specified events. However, Lira only allows the agents to execute the orders of a Manager, without any kind of internal reasoning, preventing *a priori* any autonomous decision. Moreover, the hierarchical structure of Lira makes the coordination and cooperation among the agents very difficult to implement, thus reducing the applicability in many real contexts.

The problems that we have found when trying to use Java for implementing agents in the context of a critical system (ensuring security of a bank application) suggested us the idea of using DALI instead of Java. In fact, the features of DALI are suitable for enhancing Lira agents by implementing a form of intelligence in the agents. Thus enhanced, Lira may perform reconfigurations in a more flexible and adaptable fashion: not only under predefined conditions, but also proactively, in order to reach some kind of objective, for instance to ensure some properties of the managed system. To this aim, DALI agents managing different components can communicate and cooperate. The Intelligent agents (IAs), created by integrating DALI and Lira, are able to learn, interact and cooperate in order to: (i) perform monitoring and supervision upon complex properties of a system, such as performance; (ii) perform global reconfigurations through the cooperation of intelligent agents in order to fulfill the required properties.

We argue that Lira/DALI agents are light, easy to write, and independent of any specific agent architecture. We support our argument by presenting as a Case Study a practical experience of use of the enhanced infrastructure. In particular, we show how to implement in Lira+DALI the remote management of web sites in a web hosting provider that runs on a Windows 2000 Server. By using the features provided by Lira/DALI agents, the web sites management becomes easier, and it is possible to perform supervision and automatic reconfiguration in order to optimize bandwidth and

space usage. The Case Study demonstrates how a declarative language like DALI has a significant impact in the developing process of complex applications in the practical and real context of LSDCBA. The agents created by the integration of DALI and Lira constitute in our opinion a significant advance in terms of supported functionalities, readability, modifiability and extensibility.

The paper is organized as follows: we start by describing the features of both DALI and Lira, respectively in Sections 2 and 3. In Section 3.5 we elicit some problems found when using Lira infrastructure. Then, in Section 4 we discuss the motivations of using DALI, an then we introduce the general architecture of the Lira/DALI agents. Section 5 describes the Case Study. Finally, we summarize the results in Section 6.

2 DALI Multiagent System

DALI [8] [9] is an Active Logic Programming language, designed for executable specification of logical agents. DALI allows the programmer to define one or more agents, interacting either among themselves, or with an external environment, or with a user. A DALI agent is a logic program containing special rules and classes of events (represented by special atoms) which guarantee the *reactive* and *proactive* behavior of the agent. The kinds of events are: external, internal, present, past.

Proactivity makes an agent able to initiate a behavior according to its own internal reasoning, and not only as a reaction to some external event. Reactivity determines actions that the agent will perform when some kind of event happens. Actions can be messages to other agents and/or interaction with the environment.

An agent is able to manipulate its knowledge base, to have temporary memory, to perceive an environment and consequently to make actions. Moreover, the system provides a treatment of time: the events are kept or "forgotten" according to suitable conditions.

DALI provides a complete run-time support for development of Multiagent Systems. A DALI Multiagent System is composed by communicating environments, and each environment is composed by one server and more agents. Each agent is defined by a *.pl* file, containing the agent's code written in DALI.

The new approach proposed by DALI is compared to other existing logic programming languages and agent architectures such as ConGolog, 3APL, IMPACT, METATEM, BDI in [9]. However, it is useful to remark that DALI is a logic programming language for defining agents and multi-agent systems, and does not commit to any agent architecture. Differently from other significant approaches like, e.g., DESIRE [10], DALI agents do not have pre-defined submodules. Thus, different possible functionalities (problem-solving, cooperation, negotiation, etc.) and their interactions are specific to the particular application. DALI is in fact an "agent-oriented" general-purpose language that provides, as discussed below, a number of primitive mechanisms for supporting this paradigm, all of them within a precise logical semantics.

The declarative semantics of DALI is an *evolutionary semantics*, where the meaning of a given DALI program P is defined in terms of a modified program P_s, where reactive and proactive rules are reinterpreted in terms of standard Horn Clauses. The agent

reception of an event is formalized as a program transformation step. The evolutionary semantics consists of a sequence of logic programs, resulting from these subsequent transformations, together with the sequence of the Least Herbrand Model of these programs. Therefore, this makes it possible to reason about the "state"of an agent, without introducing explicitly such a notion, and to reason about the conclusions reached and the actions performed at a certain stage. Procedurally, the interpreter simulates the program transformation steps, and applies an extended resolution which is correct with respect to the Least Herbrand Model of the program at each stage.

DALI is fully implemented in Sicstus Prolog [14]. The implementation, together with a set of examples, is available at the URL http://gentile.dm.univaq.it/dali/dali.htm.

2.1 Events Classes

In DALI, events are represented as special atoms, called *events atoms*. The corresponding predicates are indicated by a particular prefix.

- **External Events.** When something happens in the "external world" in which the agent is situated, and the agent can perceive it, this is an *external event*. If an agent receives an external event, it can decide to react to it. In order to define rules that specify the reaction, the external event is syntactically indicated by the prefix *eve*. For instance, $eve(alarm\_clock\_rings)$ represents an external event to which the agent is able to respond. When the event happens, the corresponding atom becomes true and, if in the DALI logical program that defines the agent there is a rule with this atom in the head, then the reaction defined in the body of the rule is triggered. The external events are recorded, in the arrival order, in a list called **EV** and are consumed whenever the correspondent reactive rule is activated (i.e., upon reaction the event is removed from **EV**).
  In the implementation, events are time-stamped, and the order in which they are "consumed" corresponds to the arrival order. The time-stamp can be useful for introducing into the language some (limited) possibility of reasoning about time. The head of a reactive rule can contain several events: in order to trigger reaction, they must all happen within an amount of time that can be set by a directive.
  Attached to each external event there is also the indication of the agent that has originated the event For events like $rainsE$ there will be the default indication $environment$. Then, an event atom can be more precisely seen as a triple:

  $$Sender : Event\_Atom : Timestamp$$

  The $Sender$ and $Timestamp$ fields can be omitted whenever not needed.
- **Internal Events.** The internal events define a kind of "individuality" of a DALI agent, making it independent of the environment, of the user and of the other agents, and allowing it to manipulate and revise its knowledge. An internal event is indicated by the prefix *evi*. For instance, $evi(food\_is\_finished)$ is a conclusion that the prefix evi interprets as an internal event, to which the agent may react, for instance by going to buy food. Internal events are attempted with some frequency (customizable by means of directives in an initialization file). Whenever one of them becomes true, it is inserted in a set **IV**. Similarly to external events, internal

events are extracted from this set to trigger reaction. In more detail, the mechanism is the following: if goal G has been indicated to the interpreter as an internal event by means of a suitable directive, from time to time the agent attempts the goal (at the given frequency). If the goal succeeds, it is interpreted as an event, thus determining the corresponding reaction. I.e., internal events are events that do not come from the environment. Rather, they are goals defined in some other part of the program.

There is a default frequency for attempting goals corresponding to internal events, that can be customized by the user when the agent is activated. Also, priorities among different internal events that could be attempted at the same time can be specified. At present, this frequency cannot be dynamically changed by the agent itself, but a future direction is that of providing this possibility, so as the agent will be able to adapt to changing situations.

- **Present Events.** When an agent perceives an event from the "external world", it doesn't necessarily react immediately: it has the possibility of reasoning about the event, before (or instead of) triggering reaction. Reasoning also allows a *proactive* behavior. In this situation, the event is called *present event* and is indicated by the prefix *en*. For instance, $en(alarm\_clock\_ring)$, represents a present event to which the agent has not reacted yet.
- **Past Events.** Past events represent the agent's "memory", that makes it capable to perform its future activities while having experience of previous events, and of its own previous conclusions. A past event is indicated by the prefix *evp*. For instance, $evp(alarm\_clock\_ring)$ is an event to which the agent has reacted and which remains in the agent's memory. Memory of course is not unlimited, neither conceptually nor practically: it is possible to set, for each event, for how long it has to be kept in memory. The agent has the possibility to keep events in memory either forever or for some time or until something happens, based on directives. In fact, an agent cannot keep track of *every* event and action for an unlimited period of time. Moreover, sometimes subsequent events/actions can make former ones no more valid.

  In the implementation, past events are kept for a certain default amount of time, that can be modified by the user through a suitable directive. The user can also express a condition of the form:

  $keep$ *evp(A)* $until$ *HH:MM.*

  The past event will be removed at the specified time. Alternatively, one can specify the terminating condition. As soon as the condition is fulfilled (i.e. the corresponding goal is proved) the event is removed.

  $keep$ *evp(A)* $until$ *Cond.*

  In particular cases, an event should never be dropped from the knowledge base, like in the example below:

  $keep$ *evp(born(daniele)) : 27/Aug/1993* $forever.$

Implicitly, if a second version of the same past event arrives, with a more recent timestamp, the "older" event is overridden, unless this violates a directive.

2.2 Actions: Reactivity in DALI

Actions are the agent's way of affecting its environment, possibly in reaction to an external or internal event. In DALI, actions can have or not *preconditions*: in the former case, the actions are defined by *actions rules*, in the latter they are just action atoms. In **actions rules**, success of preconditions determine the execution of the action: only when all preconditions are verified, then the corresponding action is performed. These preconditions are indicated by the prefix *cd*. In the case of **action atoms**, the actions always succeed. An action is indicated by prefix *a*. An example:

eve(saturday) :- *a(go_to_the_supermarket).*
fridge_full :- *evp(go_to_the_supermarket).*
evi(fridge_full) :- *a(prepare_a_snack).*
eve(child(I), we_are_hungry)) :- *assert(children_are_hungry).*
cd (prepare_a_snack) :- *children_are_hungry..*

When the external event *eve(saturday)* occurs, the agent reacts by performing the action $go\_to\_the\_supermarket$. Since the reaction is recorded as a past event (indicated by $evp(go\_to\_the\_supermarket)$), the recollection triggers the proactive rule and allows the internal event *evi(fridge_full)*. The action $a(prepare\_a\_snack)$ is executed if the precondition $cd(children\_are\_hungry)$ is true. This is conditioned by the external event $eve(we\_are\_hungry)$ coming from agent $child(I)$. As soon as it is observed, DALI executes the subgoal in the body of the rule, that consists in a predefined predicate (namely **assert**) that records the event.

Similarly to events, actions are recorded as *past actions*, with prefix *pa*. The following example illustrates how to exploit past actions. In particular, the action of opening (resp. closing) a door can be performed only if the door is closed (resp. open). The window is closed if the agent remembers to have closed it previously. The window is open if the agent remembers to have opened it previously.

a(open_the_door) :- *door_is_closed.*
door_is_closed :- *pa(close_the_door).*
a(close_the_door) :- *door_is_open.*
door_is_open :- *pa(open_the_door).*

External events and actions are used also for expressing communication acts. An external event can be a message from another agent, and, symmetrically, an action can consist in sending a message. Presently we do not commit to any particular agent communication language, that we consider as a customizable choice that can be changed according to the application domain.

3 Lira

In the context of Large Scale Distributed Systems we usually deal with: (i) thousands of components that are part of one or more Applications; (ii) single Applications that

are part of bigger systems, distributed over a wide area network. A basic objective of remote control is that of making the Java managed system flexible, highly modifiable at run time and stable with respect to many different faults. To these aims, remote (re)configuration should be dynamic: i.e., should be performed while a system is running, possibly as an automatic reaction when some event happens.

Lira (Light-weight Infrastructure for Reconfiguring Applications) [5] [11] [7] [6] is a system that performs remote control and dynamic reconfigurations [13] [3] over single components or applications. It uses and extends the approach of Network Management [12] architectures and protocols, where an agent controls directly the managed device and a Manager orders the reconfigurations. The decision maker could be an Administration Workbench with a graphical interface, or, in a more interesting case, a program that has the necessary knowledge to decide, when a specified precondition is verified, what kind of reconfigurations must be performed.

With component reconfiguration we mean any allowed change in the component's parameters (*component re-parametrization*): the addressed components are usually black-boxes, so Lira is able to dynamically change the values of the provided parameters. An Application reconfiguration [2] can be: (i) any change of the Application in terms of number and location of components; (ii) any kind of architectural modification [16].

Lira has been designed *light-weight* [5], given that components can be very small in size and might be run on limited-resource devices such as mobile phones or PDA. It provides a general interface for interacting with components and applications: this interface is realized using a specified architecture and a very simple protocol that allows one *to set and get variable values* and *to call functions*.

Lira has been created to provide the minimal amount of functionalities necessary to perform components reconfiguration and deployment. There are many others approaches of components reconfiguration, based on heavy weight infrastructures that manage also application dependencies and consistence. A complete description of the existing infrastructures with respect to the Lira approach is provided in [5].

The Lira architecture specifies three main actors: the **Reconfiguration Agent**, which performs the reconfiguration; the **MIB**, which is a list of variables and functions that an agent exports in order to reconfigure the component; the **Management Protocol**, that allows agents to communicate.

There are different kinds of agent, depending of their functionalities: the **Component Agent** is associated to the reconfigurable component; the **Host Agent** manages installation and activation of components and agents on the deployment host; the **Application Agent** is a higher-level agent able to monitor and reconfigure a set of components or a subsystem (for details see [6]); finally the **Manager** is the particular agent providing the interface with the decision maker, having the role to order reconfigurations to other agents.

In the next subsections we will describe the Lira features relevant for the proposed integration.

3.1 Component Agent

The Component Agent (CompAgent) is the most important part of the Lira infrastructure: it directly controls and manages the component. To keep the system general, Lira does not specify how the component is attached to the agent, but it only assumes that the agent is able to act on the component. The CompAgent is composed by a generic part (called *Protocol Manager*) which manages the agent communication, and by a local part (called *Local Agent*) which is the actual interface between the agent and the component. This interface is component-specific and it implements the following functions for the component's life-cycle management:

- `void start(compParams)`: starts the component.
- `void stop()`: stops the component.
- `void suspend()`: suspends the component.
- `void resume()`: resumes the component.
- `void shutdown()`: stops the component and kills the agent.

Moreover, each CompAgent exports the variables:

- `STATUS`: maintains the current status of the component. It can assume one of the following values: **starting, started, stopping, stopped, suspending, suspended, resuming**.
- `NOTIFYTO`: contains the address of the agent that has to be notified when a specified event happens.

All the variables exported for the specific component must be declared in the MIB (Section 3.3).

A very important property of a Lira-based reconfiguration system is the *composability* of the agents: they may be composed in a hierarchical way [15], thus creating a higher level agent which performs reconfigurations at application level, by using variables and functions exported by lower level agents. The Application Agent is a Manager for the agents in the controlled components, but it is a reconfigurations actuator for the global (if present) Manager.

3.2 Manager

The Manager orders reconfigurations on the controlled components through the associated CompAgents. The top-level manager of the hierarchy constitutes the Lira interface with the Decision Maker. It exports the `NOTIFYTO` variable, like every other agent. The Manager is allowed to send Lira messages, but may also receive SET, GET, CALL messages: it means that different Managers can communicate with each other.

3.3 MIB

This description represents the agreement among agents that allows them to communicate in a consistent way.

The description provides the list of variables and functions exported by the agent. In particular, the MIB contains the variables and functions always exported by the agent, such as the STATUS or the start() ones, as well as variables and functions specific for the managed components, that are component dependent.

Finally, the MIB specifies constraints to bind declared variables and performed actions to obtain the specified behavior [7].

3.4 Management Protocol

The management protocol has been designed to be as simple as possible, in order to keep the system *light*. Based on TCP/IP, it specifies seven messages, of which six are synchronous, namely:

- SET(*variable_name, variable_value*) / ACK(*message_text*)
- GET(*variable_name*) / REPLY(*variable_name, variable_value*)
- CALL(*function_name, parameters_list*) / RETURN(*return_value*)

and one is asynchronous, namely:

- NOTIFY(*variable_name, variable_value, agent_name*)

3.5 Some problems with using Lira

The current Lira version specifies a very clean, powerful and effective architecture. The Java prototype works well in the test examples proposed in [4] [5] [11] [6]. Nevertheless, there are still problems to solve, related to both the specification and the implementation.

From the implementation point of view, an object-oriented language such as Java allows one to easily create every kind of Lira agent by inheritance from specified classes, thus encouraging agent's reuse. Also, it provides a direct interface with the managed component. However, it is not so immediate to implement in Java mechanisms to provide agents with some kind of intelligence, such as "internal thinking" or "memory". This is demonstrated by the fact that Java-based frameworks for agent development like JADE [1] have built-in reactive capabilities, but do not directly provide proactivity.

Moreover, the hierarchical structure of Lira inherited by the network management architecture model is useful and powerful but very strict. In fact, a hierarchical management is effective for rigidly structured domains, while it makes agents implementation very hard when coordination and cooperation is needed. In this way, the applicability of Lira is reduced.

4 The integration

In this research, we have tried to overcome Lira problems by implementing a part of Lira agents using DALI.

There are several motivations to propose DALI as a formalism for the implementation of intelligent reconfiguration Lira agents. Firstly, DALI's proactive capabilities allow the agents to timely supervise component -and application- behavior, by using *Internal events*. Secondly, by using *External events* the agents are able to communicate with each other so that they can synchronize and adapt their behavior in a changing environment (e.g., in case of applications oriented to mobile devices). Thirdly, by using *Past Events* the agents have a *memory* and can perform actions automatically whenever a well-known situation occurs. Therefore, the resulting infrastructure is flexible and allows run-time event-driven reconfiguration.

A good reason to keep a Java part of Lira agents is that DALI infrastructure cannot act directly on the component to perform reconfiguration. In fact, DALI does not provide high level mechanisms to interact with the components, while Lira is specified with that purpose.

The agents created by integrating DALI and Lira, that we have called Intelligent Agents (IA) for dynamic reconfiguration, have an intelligent part provided by DALI and a managing part provided by Lira. In other words, we can say that DALI constitutes the "mind", and Lira the "hand" for performing reconfigurations.

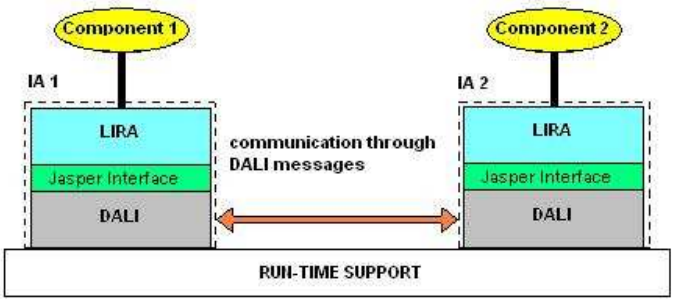

Fig. 1. The architecture of the Intelligent agents

The general architecture of the IA is shown in Figure 1. The interface between DALI and Lira is provided by a SICTUS Prolog library called *Jasper*, which allows one to call the specified Java method inside Prolog code.

Lira loses the TCP message management, but it still provides the access to the exported variables and functions through the following methods:

$$\begin{array}{ll} ACKmsg & msgSET(varName, varValue) \\ REPLYmsg & msgGET(varName) \\ RETURNmsg & msgCALL(funcName, parList) \end{array}$$

$$void \qquad msgNOTIFY(varName, varValue)$$

The communication among IAs is implemented by using DALI messages, and is

managed by its run time support. The Java methods are called by the DALI environment through the Jasper interface whenever the corresponding DALI message is received.

In DALI, the reception of a Lira message is implemented by using an *external event*. When the event is received, the agent performs a specific action which hides the Jasper predicate. For example, the reception of the Lira message *CALL("STOP", "")* is implemented as:

$$eve(CALL(" STOP", void) : -a(daliCALL(stop, void))$$

where *daliCALL* is a macro which hides all the steps (objects creation, method call etc) necessary to actually invoke the Java method.

In the sample DALI code that we will present in the next subsections, all the operations for getting and setting values or more generally for affecting the supervised component are implemented in a similar way.

The reactive capabilities provided by DALI make the IA's able to dynamically perform reconfigurations either upon certain conditions, or upon occurrence of significant events. In some cases reconfigurations can be decided and performed locally (on the controlled component), whenever the managing agent has sufficient knowledge, otherwise they can be decided by means of a process of cooperation and negotiation among the different agents.

Also, the Lira/DALI IA's can manage and exchange meta-information about system configuration and functionality. In perspective, they may have knowledge and competence to detect critical situations, and to activate dynamic security processes in order to ensure system consistency also in presence of faults or attacks.

Finally, by using DALI primitives the agents are able to learn from past situations, for example to repeat the same kind of reconfiguration upon the same conditions, or to retry the same kind of negotiation.

5 The Case Study

The case study proposed here is remote management of web sites in a web hosting provider that runs on a Windows 2000 Server. This particular environment manages single web sites as independent components, allowing the administrator to start and stop web sites independently from the actual web server that hosts them.

The features of the example are the following: we have a general server W that manages the web sites W_i through the IAs IA_i. Each agent can communicate with the other agents and with the Manager. In particular, for each web site we are interested to supervise the disk space and the bandwidth.

In order to show the flexibility of these new IA's, we propose (a sketch of) the implementation of two different policies of reconfiguration, aimed at optimizing space and bandwidth usage.

The space is managed by using a hierarchical model, where a Manager maintains the global knowledge about the space usage, and eventually orders the reconfigurations to the agents.

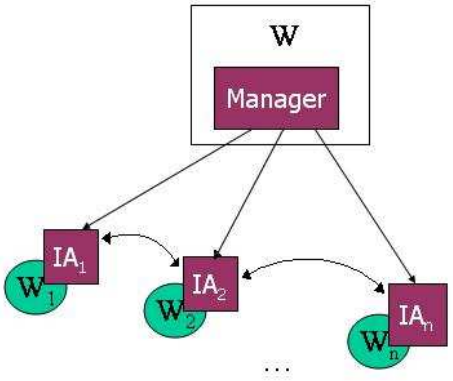

Fig. 2. The hosting web provider

A high quality of service for each web site is guaranteed through a dynamic distribution of the available bandwidth. We employ to this purpose a cooperative model, where an agent that needs more bandwidth asks other agents for obtaining the possibility to increase its own usage.

The details of these policies are described in the Sections 5.1 and 5.2.

In order to act on the component (web site) and perform the reconfigurations, the IA exports the following variables and functions. USED_SPACE, that contains the used space; MAX_SPACE, i.e., the max space allowed; USED_BAND, i.e., the band used; MAX_BANDWIDTH, i.e., the max bandwidth allowed; STATUS, which is the state of the web site; NOTIFYTO, i.e., the agent that must be notified. ERASE(fileType, space) erases the specified files, thus freeing some space on the disk. COMPRESS(files, space) compresses the specified files thus making available a larger space quota on the disk.

5.1 Space Management

DALI definition of the Manager agent

Implementation of Site Maintenance
The manager starts the maintenance of a web site managed by an IA whenever a certain timeout has expired. The exact mechanism in DALI is that the predicate $activate\_maintenance$ is automatically attempted at a predefined (customizable) frequency, and succeeds as soon as the timeout is expired. Since this predicate is an internal event, its success triggers the proactive clause with head $evi(activate\_maintenance(IA))$, thus executing the body, and sends messages to IA to stop the web site, and perform the maintenance. At the end, the site is restarted.

$activate\_maintenance(IA) : -timeout\_expired(IA).$
$evi(activate\_maintenance(IA)) : -a(message(IA, CALL("STOP", void))),$
$\qquad\qquad a(message(IA, perform\_maintenance(IA))).$
$eve(maintenance\_terminated(IA)) : -a(message(IA, CALL("START", void))).$

Remote Reconfiguration
If an IA that manages a web site asks for more disk space, the manager assigns more space to this web site if available. Only as *extrema ratio* the Manager eliminates old web sites with expired life time.

$eve(space\_not\_recovered(IA)) : -once(find\_space(IA)).$
$find\_space(IA) : -a(message(IA, SET(MAX\_SPACE, New\_space))).$
$cd(message(IA, SET(MAX\_SPACE, New\_space))) : -space\_available(New\_space).$
$find\_space(IA) : -once(check\_accounts(IA)).$
$check\_accounts(IA) : -a(erase\_expired\_web\_site).$
$cd(erase\_expired\_web\_site) : - \ldots$

DALI Intelligent agent definition for the Web Sites

Site Maintenance
The following piece of code defines how IA becomes aware of the orders by the manager of stopping/starting the site, and of performing maintenance. Notice that IA knows that maintenance is finished as soon as $perform\_maintenance$ becomes a past event (prefix *evp*), i.e., as soon as action $a(perform\_maintenance)$ has been done. If so, $end\_maintenance$ becomes true, and, since it is an internal event, it triggers a reaction that sends a message to the manager to signal that maintenance is over.

$eve(CALL("STOP", void)) : -a(daliCALL(stop, void)).$
$eve(CALL("START", void)) : -a(daliCALL(start, void)).$
$eve(perform\_maintenance) : -a(perform\_maintenance).$
$end\_maintenance(IA) : -evp(perform\_maintenance).$
$evi(end\_maintenance(IA)) : -a(message(M, maintenance\_over(IA)).$

Managing lack of space
As an example of adaptive behavior, the following piece of code included in the definition of a local agent specifies that if the used space of the managed web site is close to MAX_SPACE, then IA tries to find more space. First, the agent tries to recovery space locally, by either erasing or compressing files. If this is impossible, then it asks the manager. These local attempts of reconfigurations can be done only if they have not been performed recently, i.e., only if the corresponding past events (prefix *evp*) are not present (notice that the past events expire after a pre-set, customizable amount of time). Otherwise, the manager is informed by sending the message $space\_not\_recovered$.

$more\_space\_needed : -a(daliGET(MAX\_SPACE)),$
$\quad a(daliGET(USED\_SPACE)),$
$\quad MAX\_SPACE - USED\_SPACE \leq threshold.$
$evi(more\_space\_needed) : -recovery\_space(IA).$
$recovery\_space(IA) : -a(erase\_useless\_files(IA)).$
$cd(erase\_useless\_files) : -not(evp(erase\_useless\_files(IA))).$
$recovery\_space(IA) : -a(compress\_files(IA)).$
$cd(compress\_files) : -not(evp(compress\_files(IA))).$
$recovery\_space(IA) : -a(message(M, space\_not\_recovered(IA))).$

Updating space limit
If asked, the manager can give three different answers, corresponding to the following external events: (i) send an enlarged value new_space of MAX_SPACE; (ii) order to erase all files; (iii) stop the web site.

$eve(SET(MAX\_SPACE, new\_space)) : -a(daliSET(MAX\_SPACE, new\_space)).$
$eve(CALL(ERASE, all\_files)) : -a(daliCALL(ERASE, all\_files)).$
$eve(CALL(KILL, void)) : -a(daliCALL(KILL, void)).$

5.2 Bandwidth management

In order to exhibit a good performance to the end user, the Intelligent Agents cooperate for a dynamic band distribution according to the component needs. In particular, when an IA detects that the available band is less than the bandwidth needed, the internal event $seek\_band$ triggers a reaction: the agent checks which agents are present in the system and creates a list. Then it takes the first one and sends a request for a part of the band. If the agent receives the external event that indicates that more band is available, it sets the Lira variable MAX_BANDWIDTH, while the giving agent reduces its max bandwidth by taking off the given value. If the bandwidth is still insufficient, the agent keeps asking for band to the other agents are present in the system.

$seek\_band : -band\_insufficient.$
$band\_insufficient : - \ldots$
$evi(seek\_band) : -findallagents(Askable\_agents\_list), askfb(Askable\_agents\_list).$
$askfb(Askable\_agents\_list) : -member(IA1, Askable\_agents\_list),$
$\quad a(message(IA1, ask\_for\_band(IA, IA1))).$
$\ldots$

Symmetrically, if IA is asked for some band, it checks if it is actually in the condition to give it. When the external event ask_for_band arrives, the agent checks its bandwidth. If it has some unused band (the USED_BAND is minor of the MAX_BANDWIDTH) it keeps 80% of its band, and offers the remaining amount Bt to the other agent. Otherwise, the agent sends the message $impossible\_to\_transfer\_band$.

$eve(ask\_for\_band(IA, IA1)) : -check\_available\_band(Bt).$
$check\_available\_band(Bt) : - \ldots$
$evi(check\_available\_band(Bt)) : -Bt \neq 0, a(message(IA1, offer\_band(Bt, IA)).$
$evi(check\_available\_band(0)) : -a(message(IA1, impossible\_to\_transfer\_band)$
$\ldots$

6 Concluding Remarks

In this paper we have proposed our practical experience of using the logic programming language DALI for enriching the functionalities of Lira, an infrastructure for managing and reconfiguring Large Scale Component Based Applications.

The advantage of Lira is that of being lightweight, although able to perform both component-level reconfigurations and scalable application-level reconfigurations. The key design choice of Lira has been that of providing a minimal basic set of functionalities, while assuming that advanced capabilities are implemented in the agents, according to the application at hand. This has allowed us to gracefully integrate Lira with DALI, by replacing Java agents with DALI agents. To the best of our knowledge, this is the first running prototype of a logic-based infrastructure.

We have argued that DALI brings practical advantages under several respects. (i) The task of developing agents with memory and reasoning capabilities becomes easier, and the resulting agent programs are easier to understand, extend and modify. (ii) Reactivity, proactivity and learning capabilities of logical agents make the system more powerful through the intelligent cooperation among logical agents that can supervise and solve critical situations. (iii) Intelligent agents with reasoning abilities can cooperatively perform many tasks and reach overall objectives, also by means on suitable forms of delegation and learning. The coordination and cooperation among agents that are difficult to implement with Lira because of its hierarchical architecture can be easily realized by using LIRE/DALI intelligent agents. This makes the resulting infrastructure powerful and effective, especially in real-time contexts.

Both DALI and Lira are fully implemented, and the Intelligent Agents have been successfully experimented. Future applications are being specified, in challenging contexts such as system security in critical applications.

References

1. F. Bellifemine, A. Poggi and G. Rimassa. "JADE A FIPA-compliant agent framework". *Proceedings of PAAM'99*, held in London, April 1999, pp.97-108. Project URL: http://sharon.cselt.it/projects/jade/
2. L. Bellissard, N. de Palma and M. Riveill. "Dynamic Reconfiguration of agent-Based Applications". *Proceedings of European Research Seminar on Advances in Distributed systems, April 1999.*

3. *C. Bidan, V. Issarny, T. Saridakis and A. Zarras. "A Dynamic Reconfiguration Service for CORBA". In* Proc. of the 4th International Conference on Configurable Distributed Systems*, May 1998, Annapolis, Maryland, USA, pp. 35–42.*
4. *M. Castaldi. "Lira: a practitioner approach".* Technical Report*, University of L'Aquila, July 2002.*
5. *M. Castaldi, A. Carzaniga, P. Inverardi and A. L. Wolf. "A Light-weight Infrastructure for Reconfiguring Applications".* Proceedings of 11th Software Configuration Management Workshop*, Portland, USA, May 2003.*
6. *M. Castaldi, G. De Angelis and P. Inverardi. "A Reconfiguration Language for Remote Analysis and Application Adaptation".* Proceedings of ICSE Workshop on Remote Analysis and Measurement of Software Systems*, Portland, USA, May 2003.*
7. *M. Castaldi and N. D. Ryan. "Supporting Component-based Development by Enriching the Traditional API".* Proceedings of 4th European GCSE Young Researchers Workshop 2002*, in conjunction with NoDE, to be held in Erfurt, Germany, 7-10 October 2002.*
8. *S. Costantini. "Towards active logic programming". In A. Brogi and P. Hill, editors,* Proc. of 2nd International Workshop on component-based Software Development in Computational Logic (COCL'99)*, PLI'99, Paris, France, September 1999. http://www.di.unipi.it/ brogi/ResearchActivity/COCL99/ proceedings/index.html.*
9. *S. Costantini and A. Tocchio. "A logic programming language for multi-agent systems".* Proceedings of JELIA02, 8th European Conference on Logics in Artificial Intelligence, held in Cosenza, Italy, September 23-26, 2002. LNCS 2424, Springer-Verlag.
10. C. M. Jonker, R. A. Lam and J. Treur. "A Reusable Multi-Agent Architecture for Active Intelligent Websites". *Journal of Applied Intelligence*, vol. 15, 2001, pp. 7-24.
11. S. Porcarelli, M. Castaldi, F. Di Giandomenico, P. Inverardi and A. Bondavalli. "An Approach to Manage Reconfiguration in Fault-Tolerant Distributed Systems". *Proceedings of ICSE Workshop on Software Architectures for Dependable Systems*, Portland, USA, May 2003.
12. M. T. Rose. "The Simple Book: An Introduction to Networking Management". Prentice Hall, April 1996.
13. S.K. Shrivastava and S.M. Wheater. "Architectural Support for Dynamic Reconfiguration of Large Scale Distributed Applications". *Technical Report 645, pp. 1-14, Department of Computing Science, University of Newcastle upon Tyne, 1998.*
14. *SICStus home page.* http://www.sics.se/sicstus/.
15. *M. Wermelinger. "A Hierarchical Architecture Model for Dynamic Reconfiguration". In* Proc. of the 2nd Intl. Workshop on Software Engineering for Parallel and Distributed Systems*, IEEE Computer Society Press, 1997, pp. 243–254.*
16. *M. Wermelinger, A.Lopes and J.Fiadeiro. "A Graph Based Architectural (Re)configuration Language". In* Proc. of ESEC/FSE'01*, ACM Press, 2001.*

Logic-Based Electronic Institutions

Wamberto W. Vasconcelos

Department of Computing Science, University of Aberdeen
Aberdeen AB24 3UE, United Kingdom
wvasconcelos@acm.org

Abstract. We propose a logic-based rendition of electronic institutions – these are means to specify open agent organisations. We employ a simple notation based on first-order logic and set theory to represent an expressive class of electronic institutions. We also provide a formal semantics to our constructs and present a distributed implementation of a platform to enact electronic institutions specified in our formalism.

1 Introduction

In this paper we propose a logical formalism that allows the representation of a useful class of protocols involving many agents. This formalism combines first-order logic and set theory to allow the specification of interactions among agents, whether an auction, a more sophisticated negotiation or an argumentation framework. We introduce and exploit the logic-based formalism within the context of electronic institutions: these are means to modularly describe open agent organisations [6]. As well as providing a flexible syntax for interactions, we also formalise their semantics via the construction of models.

Current efforts at standardising agent communication languages like KIF and KQML [12] and FIPA-ACL [7] do not cater for dialogues: they do not offer means to represent relationships among messages. Work on dialogues (*e.g.* [11], [15] and [21]), on the other hand, prescribe the actual format, meaning and ultimate goal of the interactions. Our effort aims at providing engineers with a notation for specifying interactions among the components of a Multi-Agent System (MAS, for short), but which allows *relationships* to be forged among the interactions. A typical interaction we are able to express in our formalism is "all seller agents advertise their goods; after this, all buyer agents send their offers for the goods to the respective seller agent". In this interaction, it is essential that the buyer agents send offers to the appropriate seller agents, that is, each seller agent should receive an appropriate offer to the good(s) it advertised.

This paper is structured as follows. In Section 2 we describe the syntax and semantics of our proposed logic-based formalism to describe protocols. In Section 3 we introduce a definition of electronic institutions using our logic-based notation for protocols giving their formal meaning; in that section we illustrate our approach with a practical example and we describe how we implemented a platform to enact electronic institutions expressed in our formalism. Finally, in Section 4 we draw conclusions and give directions for future work.

2 A Set-Based Logic $\mathcal{L}$ for Protocols

In this section, we describe a set-based first-order logic $\mathcal{L}$ with which we can define protocols. Our proposed logic provides us with a compact notation to formally describe relationships among messages in a protocol. Intuitively, these constructs define (pre- and post-) conditions that should hold as agents follow a protocol.

We aim at a broad class of protocols in which many-to-many interactions (and, in particular, one-to-one, one-to-many and many-to-one) can be formally expressed. The protocols are *global* in the sense that they describe any and all interactions that may take place in the MAS. One example of the kind of interactions we want to be able to express is "an agent x sends a message to another agent y offering an item k for sale; agent y replies to x's message making an offer n to buy k" and so on. We define $\mathcal{L}$ as below:

Definition 1. *$\mathcal{L}$ consists of formulae Qtf ($Atfs \Rightarrow SetCtrs$) where Qtf is the quantification, Atfs is a conjunction of atomic formulae and SetCtrs is a conjunction of set constraints.*

Qtf provides our constructs with universal and existential quantification over (finite) sets; *Atfs* expresses atomic formulae that must hold true and *SetCtrs* represents set constraints that (are made to) hold true. We define the classes of constructs *Qtf*, *Atfs* and *SetCtrs* in the sequel. We refer to a well-formed formulae of $\mathcal{L}$ generically as *Fml*.

We shall adopt some notational conventions in our formulae. Sets will be represented by words starting with capital letters and in this typefont, as in, for example "S", "Set" and "Buyers". Variables will be denoted by words starting with capital letters in *this typefont*, as in, for example, "X", "*Var*" and "*Buyer*". We shall represent constants by words starting with non-capital letters in *this font*; some examples are "a" and "*item*". We shall assume the existence of a recursively enumerable set *Vars* of variables and a recursively enumerable set *Consts* of constants.

In order to define the class *Atfs* of atomic formulae conjunctions, we first put forth the concept of *terms*:

Definition 2. *All elements from Vars and Consts are in Terms. If $t_1, \ldots, t_n$ are in Terms, then $f(t_1, \ldots, t_n)$ is also in Terms, f being a function symbol.*

The class *Terms* is thus defined recursively, based on variables and constants and their combination with functional symbols. An example of a term using our conventions is $enter(buyer)$. We can now define the class *Atfs*:

Definition 3. *If $t_1, \ldots, t_n$ are Terms, then $p(t_1, \ldots, t_n)$ is an atomic formula (or, simply, an atf), where p is any predicate symbol. A special atomic formula is defined via the "=" symbol, as $t_1 = t_2$. The class Atfs consists of all atfs; furthermore, for any Atf_1 and Atf_2 in Atfs, $Atf_1 \wedge Atf_2$ is also in Atfs.*

This is another recursive definition: the basic components are the simple atomic formulae built with terms. These components (and their combinations) can be put together as conjuncts.

We now define the class of *set constraints*. These are restrictions on set operations such as union, intersection, Cartesian product and set difference [8]:

Definition 4. *Set constraints are conjunctions of set operations, defined by the following grammar:*

$$
\begin{aligned}
\mathit{SetCtrs} &\rightarrow \mathit{SetCtrs} \wedge \mathit{SetCtrs} \mid (\mathit{SetCtrs}) \mid \mathit{MTest} \mid \mathit{SetProp} \\
\mathit{MTest} &\rightarrow \mathit{Term} \in \mathit{SetOp} \mid \mathit{Term} \notin \mathit{SetOp} \\
\mathit{SetProp} &\rightarrow \mathit{card}(\mathit{SetOp})\ \mathit{Op}\ \mathbb{N} \mid \mathit{card}(\mathit{SetOp})\ \mathit{Op}\ \mathit{card}(\mathit{SetOp}) \mid \mathit{SetOp} = \mathit{SetOp} \\
\mathit{Op} &\rightarrow = \mid > \mid \geq \mid < \mid \leq \\
\mathit{SetOp} &\rightarrow \mathit{SetOp} \cup \mathit{SetOp} \mid \mathit{SetOp} \cap \mathit{SetOp} \mid \mathit{SetOp} - \mathit{SetOp} \\
&\quad \mid \mathit{SetOp} \times \mathit{SetOp} \mid (\mathit{SetOp}) \mid \mathsf{Set} \mid \emptyset
\end{aligned}
$$

MTest is a *membership test*, that is, a test whether an element belongs or not to the result of a set operation *SetOp* (in particular, to a specific set). *SetProp* represents the *set properties*, that is, restrictions on set operations as regards to their size (*card*) or their contents. $\mathbb{N}$ is the set of natural numbers. *Op* stands for the allowed operators of the set properties. *SetOp* stands for the *set operations*, that is, expressions whose final result is a set. An example of a set constraint is $B \in \mathsf{Buyers} \wedge \mathit{card}(\mathsf{Buyers}) \geq 0 \wedge \mathit{card}(\mathsf{Buyers}) \leq 10$. We may, alternatively, employ $|\mathsf{Set}|$ to refer to the cardinality of a set, that is, $|\mathsf{Set}| = \mathit{card}(\mathsf{Set})$. Additionally, in order to simplify our set expressions and improve their presentation, we can use $0 \leq |\mathsf{Buyers}| \leq 10$ instead of the previous expression.

Finally, we define the quantifications *Qtf*:

Definition 5. *The quantification Qtf is defined as:*

$$
\begin{aligned}
\mathit{Qtf} &\rightarrow \mathit{Qtf}'\ \mathit{Qtf} \mid \mathit{Qtf}' \\
\mathit{Qtf}' &\rightarrow Q\ \mathit{Var} \in \mathit{SetOp} \mid Q\ \mathit{Var} \in \mathit{SetOp}, \mathit{Var} = \mathit{Term} \\
Q &\rightarrow \forall \mid \exists \mid \exists!
\end{aligned}
$$

Where $\mathit{Term} \in \mathit{Terms}$ *and* $\mathit{Var} \in \mathit{Vars}$.

We pose an important additional restriction on our quantifications: either *Var* or subterms of *Term* must occur in $(\mathit{Atfs} \Rightarrow \mathit{SetCtrs})$.

Using the typographic conventions presented above, we can now build correct formulae; an example is $\exists B \in \mathsf{Ags}\,(m(B, \mathit{adm}, \mathit{enter}(\mathit{buyer})) \Rightarrow (B \in \mathsf{Bs} \wedge 1 \leq |\mathsf{Bs}| \leq 10))$. To simplify our formulae, we shall also write quantifications of the form $\mathit{Qtf}\ \mathit{Var} \in \mathit{SetOp}, \mathit{Var} = \mathit{Term}$ simply as $\mathit{Qtf}\ \mathit{Term} \in \mathit{SetOp}$. For instance, $\forall X \in \mathsf{Set}, X = f(a, Z)$ will be written as $\forall f(a, Z) \in \mathsf{Set}$.

2.1 The Semantics of $\mathcal{L}$

In this section we show how *Fml* is mapped to truth values $\top$ (true) or $\bot$ (false). For that, we first define the *interpretation* of our formulae:

Definition 6. *An interpretation* $\Im$ *for Fml is the pair* $\Im = (\sigma, \Omega)$ *where* σ *is a possibly empty set of ground atomic formulae (i.e. atfs without variables) and* Ω *is a set of sets.*

Intuitively our interpretations provide in σ what is required to determine the truth value of $\mathit{Qtf}(\mathit{Atfs})$ and in Ω what is needed in order to assign a truth value to $\mathit{Qtf}(\mathit{SetCtrs})$.

We did not include in our definition of interpretation above the notion of *universe of discourse* (also called *domain*) nor the usual mapping between constants and elements of this universe, neither the mapping between function and predicate symbols of the formula and functions and relations in the universe of discourse [4, 13]. This is because we are only interested in the relationships between *Atfs* and *SetCtrs* and how we can automatically obtain an interpretation for a given formula. However, we can define the union of all sets in Ω as our domain. It is worth mentioning that the use of a set of sets to represent Ω does not cause undesirable paradoxes: since we do not allow the formulae in $\mathcal{L}$ to make references to Ω, but only to sets in Ω, this will not happen.

The semantic mapping $\mathbf{k} : \mathit{Fml} \times \Im \mapsto \{\top, \bot\}$ is:

1. $\mathbf{k}(\forall\, Terms \in SetOp\ Fml, \Im) = \top$ iff $\mathbf{k}(Fml|_e^{Terms}, \Im) = \top$ for all $e \in \mathbf{k}'(SetOp, \Im)$
   $\mathbf{k}(\exists\, Terms \in SetOp\ Fml, \Im) = \top$ iff $\mathbf{k}(Fml|_e^{Terms}, \Im) = \top$ for some $e \in \mathbf{k}'(SetOp, \Im)$
   $\mathbf{k}(\exists!\, Terms \in SetOp\ Fml, \Im) = \top$ iff $\mathbf{k}(Fml|_e^{Terms}, \Im) = \top$ for a single $e \in \mathbf{k}'(SetOp, \Im)$
2. $\mathbf{k}((Atfs \Rightarrow SetCtrs), \Im) = \bot$ iff $\mathbf{k}(Atfs, \Im) = \top$ and $\mathbf{k}(SetCtrs, \Im) = \bot$
3. $\mathbf{k}(Atfs_1 \wedge Atfs_2, \Im) = \top$ iff $\mathbf{k}(Atfs_1, \Im) = \mathbf{k}(Atfs_2, \Im) = \top$
   $\mathbf{k}(Atf, \Im) = \top$ iff $Atf \in \sigma, \Im = (\sigma, \Omega)$
4. $\mathbf{k}(SetCtrs_1 \wedge SetCtrs_2, \Im) = \top$ iff $\mathbf{k}(SetCtrs_1, \Im) = \mathbf{k}(SetCtrs_2, \Im) = \top$
5. $\mathbf{k}(Terms \in SetOp, \Im) = \top$ iff $Terms \in \mathbf{k}'(SetOp, \Im)$;
   $\mathbf{k}(Terms \notin SetOp, \Im) = \top$ iff $Terms \notin \mathbf{k}'(SetOp, \Im)$
   $\mathbf{k}(|SetOp|\ Op\ \mathbb{N}, \Im) = \top$ iff $|\mathbf{k}'(SetOp, \Im)|\ Op\ \mathbb{N}$ holds.
   $\mathbf{k}(|SetOp_1|\ Op\ |SetOp_2|, \Im) = \top$ iff $|\mathbf{k}'(SetOp_1, \Im)|\ Op\ |\mathbf{k}'(SetOp_2, \Im)|$ holds.
   $\mathbf{k}(SetOp_1 = SetOp_2, \Im) = \top$ iff $\mathbf{k}'(SetOp_1, \Im) = \mathbf{k}'(SetOp_2, \Im)$

In item 1 we address the three quantifiers over *Fml* formulae, where $Fml|_e^{Terms}$ is the result of replacing every occurrence of *Terms* by e in *Fml*. Item 2 describes the usual meaning of the right implication. Item 3 formalises the meaning of conjunctions *Atfs* and the basic case for individual atomic formulae – these are only considered true if they belong to the associated set σ of the interpretation $\Im$. Item 4 formalises the meaning of the conjunct and disjunct operations over set constraints *SetCtrs* and the basic membership test to the result of a set operation *SetOp*. Item 5 describes the truth-value of the distinct set properties *SetProp*. These definitions describe only one case of the mapping: since ours is a total mapping, the situations which are not described represent a mapping with the remaining value $\top$ or $\bot$.

The auxiliary mapping $\mathbf{k}' : SetOp \times \Im \mapsto \mathsf{Set}$ in $\Omega, \Im = (\sigma, \Omega)$, referred to above and which gives meaning to the set operations is thus defined:

1. $\mathbf{k}'(SetOp_1 \cup SetOp_2, \Im) = \{e \mid e \in \mathbf{k}'(SetOp_1, \Im)\ \text{or}\ e \in \mathbf{k}'(SetOp_2, \Im)\}$
2. $\mathbf{k}'(SetOp_1 \cap SetOp_2, \Im) = \{e \mid e \in \mathbf{k}'(SetOp_1, \Im)\ \text{and}\ e \in \mathbf{k}'(SetOp_2, \Im)\}$
3. $\mathbf{k}'(SetOp_1 - SetOp_2, \Im) = \{e \mid e \in \mathbf{k}'(SetOp_1, \Im)\ \text{and}\ e \notin \mathbf{k}'(SetOp_2, \Im)\}$
4. $\mathbf{k}'(SetOp_1 \times SetOp_2, \Im) = \{(e_1, e_2) \mid e_1 \in \mathbf{k}'(SetOp_1, \Im)\ \text{and}\ e_2 \in \mathbf{k}'(SetOp_2, \Im)\}$
5. $\mathbf{k}'((SetOp), \Im) = (\mathbf{k}'(SetOp, \Im))$.
6. $\mathbf{k}'(\mathsf{Set}, \Im) = \{e \mid e \in \mathsf{Set}\ \text{in}\ \Omega, \Im = (\sigma, \Omega)\}$, $\mathbf{k}'(\emptyset, \Im) = \emptyset$.

The 4 set operations are respectively given their usual definitions [8]. The meaning of a particular set Set is its actual contents, as given by Ω in $\Im$. Lastly, the meaning of an empty set $\emptyset$ in a set operation is, of course, the empty set.

We are interested in *models* for our formulae, that is, interpretations that map *Fml* to the truth value $\top$ (true). We are only interested in those interpretations in which *both* sides of the "$\Rightarrow$" in the *Fml*'s hold true. Formally:

Definition 7. *An interpretation $\Im = (\sigma, \Omega)$ is a model for a formula $Fml = Qtf\,(Atfs \Rightarrow SetCtrs)$, denoted by $\mathbf{m}(Fml, \Im)$ iff σ and Ω are the smallest possible sets such that $\mathbf{k}(Qtf\ Atfs, \Im) = \mathbf{k}(Qtf\ SetCtrs, \Im) = \top$.*

The scenarios arising when the left-hand side of the *Fml* is false do not interest us: we want this formalisation to restrict the meanings of our constructs only to those desirable (correct) ones. The study of the anomalies and implications caused by not respecting the restrictions of a protocol albeit important is not in the scope of this work.

We now define the extension of an interpretation, necessary to build models for more than one formula *Fml*:

Definition 8. *$\Im' = (\sigma', \Omega')$ is an extension of $\Im = (\sigma, \Omega)$ which accommodates Fml, denoted by $\mathbf{ext}(\Im, Fml) = \Im'$, iff $\mathbf{m}(Fml, \Im''), \Im'' = (\sigma'', \Omega'')$ and $\sigma' = \sigma \cup \sigma'', \Omega' = \Omega \cup \Omega''$.*

3 Logic-Based Electronic Institutions

In the same way that social institutions, such as a constitution of a country or the rules of a club, are somehow forged (say, in print or by common knowledge), the laws that should govern the interactions among heterogeneous agents can be defined by means of electronic institutions (e-institutions, for short) [5, 6, 14, 16]. E-institutions are non-deterministic finite-state machines describing possible interactions among agents. The interactions are only by means of message exchanges, that is, messages that are sent and received by agents. E-institutions define communication protocols among agents with a view to achieving global and individual goals.

Although different formulations of e-institutions can be found in the literature [5, 6, 14, 16, 19], they all demand additional informal explanations concerning the precise meaning of its constructs. In an e-institution the interactions among agents are described as finite-state machines with messages labelling the edges between two states. A simple example is graphically depicted in Fig. 1 where two agents x and y engage in a simple two-step conversation

Fig. 1: Protocol as a Finite-State Machine

– to save space, we have represented messages as atomic formulae of the form $m(Sender, Addressee, Conts)$, meaning that *Sender* is sending to *Addressee* message *Conts*; alternative formats such as FIPA-ACL [7] could be used instead. Agent x informs agent y that it wants to sell item k and y replies with an offer n. In the example above we employed variables x, y, k and n but it is not clear what their actual meaning is: is x the same in both edges? is it just *one* agent x or can many agents follow the transition? It is not clear from the notation only what the meaning of the label is. Surely, informal explanations could solve any ambiguity, but by tacitly assuming the meaning of constructs (*i.e.* "hardwiring" the meaning to the syntax), then variations cannot be offered. For instance, if we assume that the variables in Fig. 1 are universally quantified, then it is not possible to express the existential quantification and vice-versa. Similar expressiveness losses occur when other assumptions are made.

We have incorporated our proposed logic $\mathcal{L}$ to the definition of e-institutions. In this combination, constructs of $\mathcal{L}$ label edges of finite-state machines. This allows for precisely defined and expressive edges thus extending the class of e-institutions one can represent. Furthermore, by embedding $\mathcal{L}$ within e-institutions, we can exploit the model-theoretic issues in an operational framework.

3.1 Scenes

Scenes are the basic components of an e-institution, describing interactions among agents:

Definition 9. *A scene is* $\mathbf{S} = \langle R, W, w_0, W_f, WA, WE, f^{Guard}, Edges, f^{Label} \rangle$ *where*

- $R = \{r_1, \ldots, r_n\}$ *is the set of roles;*
- $W = \{w_0, \ldots, w_m\}$ *is a finite, non-empty set of states;*
- $w_0 \in W$ *is the initial state;*
- $W_f \subseteq W$ *is the non-empty set of final states;*
- WA *is a set of sets* $WA = \{ WA_r \subseteq W \mid r \in R\}$ *where each* WA_r, $r \in R$, *is the set of access states for role* r*;*

- *WE is a set of sets* $WE = \{WE_r \subseteq W \mid r \in R\}$ *where each* WE_r, $r \in R$, *is the set of exit states for role* r*;*
- $f^{Guard} : WA_r \mapsto Fml$ *and* $f^{Guard} : WE_r \mapsto Fml$ *associates with each access state* WA_r *and exit state* WE_r *of role* r *a formula* Fml*.*
- $Edges \subseteq W \times W$ *is a set of directed edges;*
- $f^{Label} : Edges \mapsto Fml$ *associates each element of* $Edges$ *with a formula* Fml*.*

This definition is a variation of that found in [19]. We have added to access and exit states, via function f^{Guard}, explicit restrictions formulated as formulae of $\mathcal{L}$. The labelling function f^{Label} is defined similarly, but mapping *Edges* to our formulae *Fml*.

3.2 Transitions

The scenes, as formalised above, are where the communication among agents actually take place. However, individual scenes can be part of a more complex context in which specific sequences of scenes have to be followed. For example, in some kinds of electronic markets, a scene where agents meet other agents to choose their partners to trade is followed by a scene where the negotiations actually take place. We define *transitions* as a means to connect and relate scenes:

Definition 10. *A transition is* $\mathbf{T} = \langle CI, w_a, Fml, w_e, CO \rangle$ *where*

- $CI \subseteq \bigcup_{i=1}^{n}(WE_i \times w_a)$*, is the set of connections into the transition,* $WE_i, 1 \leq i \leq n$ *being the sets of exit states for all roles from all scenes;*
- w_a *is the access state of the transition;*
- w_e *is the exit state of the transition;*
- Fml*, a formula of* $\mathcal{L}$*, labels the pair* $(w_a, w_e) \mapsto Fml$*;*
- $CO \subseteq \bigcup_{j=1}^{m}(w_e \times WA_j)$*, is the set of connections out of the transition,* $WA_j, 1 \leq j \leq m$ *being the sets of access states for all roles onto all scenes.*

A transition has only two states w_a, its access state, and w_e, its exit state, and a set of connections *CI* relating the exit states of scenes to w_a and a set of connections *CO* relating w_e to the access states of scenes. The conditions under which agents are allowed to move from w_a to w_e are specified by a formula *Fml* of our set-based logic, introduced above.

Transitions can be seen as simplified scenes where agents' movements can be grouped together and synchronised out of a scene and into another one. The roles of agents may change, as they go through a transition. An important feature of transitions lies in the kinds of formula *Fml* we are allowed to use. Contrary to scenes, where there can only be references to constructs within the scene, within a transition we can make references to constructs of any scene that connects to the transition. This difference is formally represented by the semantics of e-institutions below.

3.3 $\mathcal{L}$-Based E-Institutions

Our e-institutions are collections of scenes and transitions:

Definition 11. *An e-institution is* $\mathbf{E} = \langle Scenes, \mathbf{S}_0, \mathbf{S}_f, Trans \rangle$ *where*

- $Scenes = \{\mathbf{S}_0, \ldots, \mathbf{S}_n\}$ *is a finite and non-empty set of scenes;*
- $\mathbf{S}_0 \in Scenes$ *is the root scene;*
- $\mathbf{S}_f \in Scenes$ *is the output scene;*
- $Trans = \{\mathbf{T}_0, \ldots, \mathbf{T}_m\}$ *is a finite and non-empty set of transitions;*

We shall impose the restriction that the transitions of an e-institution can only connect scenes from the set *Scenes*, that is, for all $\mathbf{T} \in \mathit{Trans}$, $\mathit{CI} \subseteq \bigcup_{i=0}^{n}(\mathit{WE}_i \times w_a), i \neq f$ (the exit states of the output scene can not be connected to a transition) and $\mathit{CO} \subseteq \bigcup_{j=1}^{n}(w_e \times \mathit{WA}_j)$ (the access state of the root scene cannot be connected to a transition).

For the sake of simplicity, we have not included in our definition above the *normative rules* [5] which capture the obligations agents get bound to as they exchange messages. We are aware that this makes our definition above closer to the notion of *performative structure* [5] rather than an e-institution.

3.4 Models for $\mathcal{L}$-Based E-Institutions

In this section we introduce models for scenes, transitions and e-institutions using the definitions above.

A model for a scene is built using the formulae that label edges connecting the initial state to a final state. The formulae guarding access and exit states are also taken into account: they are used to extend the model of the previous formulae and this extension is further employed with the formula connecting the state onwards. Since there might be more than one final state and more than one possible way of going from the initial state to a final state, models for scenes are not unique. More formally:

Definition 12. *An interpretation $\Im$ is a model for a scene $\mathbf{S} = \langle R, W, w_0, W_f, \mathit{WA}, \mathit{WE}, f^{\mathit{Guard}}, \mathit{Edges}, f^{\mathit{Label}}\rangle$, given an initial interpretation $\Im_0$, denoted by $\mathbf{m}(\mathbf{S}, \Im)$, iff $\Im = \Im_n$, where:*

- *$f^{\mathit{Label}}(w_{i-1}, w_i) = \mathit{Fml}_i, 1 \leq i \leq n, w_n \in W_f$, are the formulae labelling edges which connect the initial state w_0 to a final state w_n.*
- *for $w_i \in \mathit{WA}_r$ or $w_i \in \mathit{WE}_r$ for some role r, that is, w_i is an access or exit state, then $f^{\mathit{Guard}}(w_i) = \mathit{Fml}_{[\mathit{WA},i]}$ or $f^{\mathit{Guard}}(w_i) = \mathit{Fml}_{[\mathit{WE},i]}$, respectively.*
- *for $1 \leq i \leq n$, then $\Im_i = \begin{cases} \mathbf{ext}(\mathbf{ext}(\Im_{i-1}, \mathit{Fml}_{[\mathit{WA},i]}), \mathit{Fml}_i), & \text{if } w_i \in \mathit{WA}_r \\ \mathbf{ext}(\mathbf{ext}(\Im_{i-1}, \mathit{Fml}_{[\mathit{WE},i]}), \mathit{Fml}_i), & \text{if } w_i \in \mathit{WE}_r \\ \mathbf{ext}(\Im_{i-1}, \mathit{Fml}_i), & \text{otherwise} \end{cases}$*

One should notice that the existential quantification allows for the *choice* of components for the sets in Ω and hence more potential for different models. In order to obtain a model for a scene, an initial model $\Im_0$, possibly empty, must be provided.

The model of a transition extends the models of scenes connecting to it:

Definition 13. *An interpretation $\Im$ is a model for a transition $\mathbf{T} = \langle \mathit{CI}, w_a, \mathit{Fml}, w_e, \mathit{CO}\rangle$, denoted by $\mathbf{m}(\mathbf{T}, \Im)$, iff*

- *$\mathbf{S}_1, \ldots, \mathbf{S}_n$ are all the scenes that connect with CI, i.e. the set WE_i of exit states of each scene $\mathbf{S}_i, 1 \leq i \leq n$, has at least one element $\mathit{WE}_{i,r} \times w_a$ in CI, and*
- *$\mathbf{m}(\mathbf{S}_i, \Im_i), \Im_i = (\sigma_i, \Omega_i), \Im' = (\bigcup_{i=1}^{n} \sigma_i, \bigcup_{i=1}^{n} \Omega_i), 1 \leq i \leq n$, and $\mathbf{ext}(\Im', \mathit{Fml}) = \Im$*

The model of a transition is an extension of the union of the models of all its connecting scenes to accommodate *Fml*. Finally, we define the meaning of e-institutions:

Definition 14. *An interpretation $\Im$ is a model for an e-institution $\mathbf{E} = \langle \mathit{Scenes}, \mathbf{S}_0, \mathbf{S}_f, \mathit{Trans}\rangle$, denoted by $\mathbf{m}(\mathbf{E}, \Im)$, iff*

- *$\mathit{Scenes} = \{\mathbf{S}_0, \ldots, \mathbf{S}_n\}, \mathbf{m}(\mathbf{S}_i, \Im), 0 \leq i \leq n$; and*
- *$\mathit{Trans} = \{\mathbf{T}_0, \ldots, \mathbf{T}_m\}, \mathbf{m}(\mathbf{T}_j, \Im), 0 \leq j \leq m$.*

3.5 Building Models for $\mathcal{L}$-Based E-Institutions

We can build a model $\Im$ for a formula *Fml* if we are given an initial value for the sets in Ω. We need only those sets that are referred to in the quantification of *Fml*: with this information we can define the atomic formulae that make the left-hand side of "$\Rightarrow$" true. If the conditions on the left-hand side of *Fml* are fulfilled then we proceed to *make* the conditions on the right-hand side true, by means of the appropriate creation of other sets.

Building a model $\Im$ is a computationally expensive task, involving combinatorial efforts to find the atomic formulae that ought to be in σ and the contents of the sets in Ω. If, however, the formulae *Fml* of a scene have a simple property, *viz.* the quantification of each formula Fml_i only refers to sets that appear on preceding formulae $Fml_j, j < i$, then we can build an interpretation gradually, taking into account each formula at a time. This property can be syntactically checked: we can ensure that all sets appearing in Fml_i quantification appears on the right-hand side of a Fml_j which leads on to Fml_i in a scene. Only if all scenes and transitions of an e-institution fulfill this property is that we can automatically build a model for it in feasible time.

If this property holds in our e-institutions, then we can build for any formula Fml_i a model $\Im_i$ that uses the $\Im_{i-1}$ of the preceding formula (assuming an ordering among the edges of a path). The models of a scene are then built gradually, each formula at a time, via $\mathbf{ext}(\Im_{i-1}, Fml_i) = \Im_i$. The quantifiers in *Fml* assign values to variables in its body, following the semantic mapping $\mathbf{k}$ shown previously. The existential quantifiers $\exists$ and $\exists!$ introduce non-determinism: in the case of $\exists$ a subset of the elements of the quantified set has to be chosen; in the case of $\exists!$ a single element has to be chosen. Additional constraints on the choice to be made can be expressed as part of *Fml*.

Given an initial interpretation $\Im = (\emptyset, \Omega)$ in which Ω is possibly empty or may contain any initial values of sets, so that we can start building the models of the ensuing formulae. Given $\Im_{i-1}$ and Fml_i we can automatically compute the value $\mathbf{ext}(\Im_{i-1}, Fml_i) = \Im_i$. Since the quantifiers of Fml_i only refer to sets of the right-hand side of preceding Fml_j, then $\Im_{i-1}$ should have the actual contents of these sets. We exhaustively generate values for the quantified variables – this is only possible because all the sets are finite – and hence we can assemble the atomic formulae for a possible σ_i of $\Im_i$. With this σ and Ω_{i-1} we then assemble Ω_i, an extension of Ω_{i-1} which satisfies the set constraints of Fml_i.

3.6 Example: A Simple Agoric Market

We illustrate the definitions above with an example comprising a complete virtual agoric marketplace. We provide in Fig. 2 a graphic rendition of an e-institution for our market – the same e-institution is, of course, amenable to different

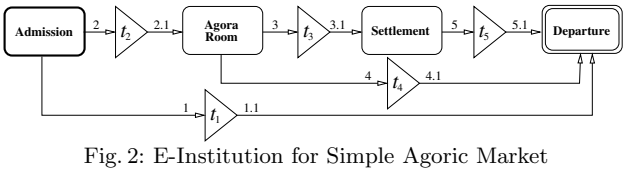

Fig. 2: E-Institution for Simple Agoric Market

visual renditions. Scenes are represented as boxes with rounded edges; the root

scene **Admission** has a thicker box and the output scene **Departure** has a double box. Transitions are represented as triangles. The arcs in our diagram connect exit states of a scene with the access state of a transition and the exit state of a transition with an access state of a scene. Agents have to be initially admitted in the e-institution (**Admission** scene) where their details are recorded; agents then may proceed to trade their goods in the **Agora Room** scene, after which they may (if they have bought or sold goods) have to settle any debts in the **Settlement** scene. Finally, agents leave the institution, via the **Departure** scene.

We now focus on a specific scene in the e-institution above. In Fig. 3 we "zoom in" on the **Agora Room** scene, in which agents willing to acquire goods interact

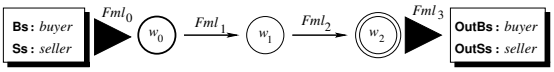

Fig. 3: Diagram for **Agora Room** Scene

with agents intending to sell such goods. This agora scene has been simplified – no auctions or negotiations are contemplated. The sellers announce the goods they want to sell and collect the replies from buyers (all buyers must reply). The simplicity of this scene is deliberate, so as to allow us to fully represent and discuss it. A more friendly visual rendition of the formal definition is employed in the figure and is explained below.

The states $W = \{w_0, w_1, w_2\}$ are displayed in circles and $Edges = \{(w_0, w_1), (w_1, w_2)\}$ are shown as arrows: if $(w_i, w_j) \in Edges$, then $w_i \longrightarrow w_j$. The initial state w_0 is shown enclosed in a thicker circle; the final state $W_f = \{w_2\}$ is enclosed in a double circle. We define the set of roles as $R = \{seller, buyer\}$. An access state $w \in WA$ is marked with a "▶" pointing towards the state with a box containing the role(s) of the agents that may enter the scene at that point and a set name. Exit states are also marked with a "▶" but pointing away from the state; they are also shown with a box containing the roles of the agents that may leave the scene at that point and a set name. We have defined the formulae $Fml_i, 0 \leq i \leq 3$, as:

$$Fml_0\colon \exists B, S \in \mathsf{Ags}\left(\begin{pmatrix} m(B, adm, enter(buyer)) \wedge \\ m(S, adm, enter(seller)) \end{pmatrix} \Rightarrow \begin{pmatrix} B \in \mathsf{Bs} \ \wedge \ 1 \leq |\mathsf{Bs}| \leq 10 \wedge \\ S \in \mathsf{Ss} \ \wedge \ 1 \leq |\mathsf{Ss}| \leq 10 \end{pmatrix}\right)$$
$$Fml_1\colon \forall S \in \mathsf{Ss}\, \forall B \in \mathsf{Bs}\, \exists I \in \mathsf{Is}\ (m(S, B, \mathit{offer}(I, P)) \Rightarrow \langle S, B, I, P\rangle \in \mathsf{Ofs})$$
$$Fml_2\colon \forall \langle S, B, I, P\rangle \in \mathsf{Ofs} \exists! A \in \mathsf{As} (m(B, S, \mathit{reply}(I, P, A)) \Rightarrow \langle B, S, I, P, A\rangle \in \mathsf{Rs})$$
$$Fml_3\colon \forall B \in \mathsf{Bs}\, \forall S \in \mathsf{Ss}\ \left(\begin{pmatrix} m(B, adm, leave) \wedge \\ m(S, adm, leave) \end{pmatrix} \Rightarrow \begin{pmatrix} B \in \mathsf{OutBs} \ \wedge \ S \in \mathsf{OutSs} \wedge \\ \mathsf{OutBs} = \mathsf{Bs} \ \wedge \ \mathsf{OutSs} = \mathsf{Ss} \end{pmatrix}\right)$$

The left-hand side of the Fml_i are atomic formulae which must hold in σ_i and the right-hand side are set constraints that must hold in Ω_i. The atomic formula stand for messages exchanged among the agents as they move along the edges of the scene. The above definitions give rise to the following semantics:

$$\mathbf{ext}\left(\left(\emptyset, \begin{Bmatrix} \mathsf{Ags}, \\ \mathsf{As}, \\ \mathsf{Is} \end{Bmatrix}\right), Fml_0\right) = \Im_0 = \left(\begin{Bmatrix} m(ag_1, adm, enter(seller)), \\ m(ag_2, adm, enter(buyer)), \\ m(ag_3, adm, enter(buyer)) \end{Bmatrix}, \begin{Bmatrix} \mathsf{Ags}, \mathsf{As}, \mathsf{Is}, \\ \mathsf{Bs}, \mathsf{Ss} \end{Bmatrix}\right)$$

We assume that $\mathsf{Ags} = \{ag_1, \ldots, ag_4\}$, $\mathsf{As} = \{ok, not\_ok\}$ and $\mathsf{Is} = \{car, boat, plane\}$, and we obtain $\mathsf{Bs} = \{ag_2, ag_3\}$ and $\mathsf{Ss} = \{ag_1\}$;

$$\mathbf{ext}(\Im_0, Fml_1) = \Im_1 = \left(\sigma_0 \cup \begin{Bmatrix} m(ag_1, ag_2, \mathit{offer}(car, 4)), \\ m(ag_1, ag_3, \mathit{offer}(boat, 3)) \end{Bmatrix}, \begin{Bmatrix} \mathsf{Ags}, \mathsf{As}, \mathsf{Is}, \\ \mathsf{Bs}, \mathsf{Ss}, \mathsf{Ofs} \end{Bmatrix}\right)$$

where $\mathsf{Ofs} = \{\langle ag_1, ag_2, car, 4\rangle, \langle ag_1, ag_3, boat, 3\rangle\}$;

$$\mathbf{ext}(\Im_1, Fml_2) = \Im_2 = \left(\sigma_1 \cup \left\{\begin{array}{l} m(ag_2, ag_1, reply(car, 4, ok)), \\ m(ag_3, ag_1, reply(boat, 3, not\_ok))\end{array}\right\}, \left\{\begin{array}{l}\mathsf{Ags}, \mathsf{As}, \mathsf{Is}, \\ \mathsf{Bs}, \mathsf{Ss}, \mathsf{Ofs}, \\ \mathsf{Rs}\end{array}\right\}\right)$$

where $\mathsf{Rs} = \{\langle ag_2, ag_1, car, 4, ok\rangle, \langle ag_3, ag_1, boat, 3, not\_ok\rangle\}$;

$$\mathbf{ext}(\Im_2, Fml_3) = \Im_3 = \left(\sigma_3 \cup \left\{\begin{array}{l} m(ag_1, adm, leave), \\ m(ag_2, adm, leave), \\ m(ag_3, adm, leave)\end{array}\right\}, \left\{\begin{array}{l}\mathsf{Ags}, \mathsf{As}, \mathsf{Is}, \mathsf{Bs}, \mathsf{Ss}, \mathsf{Ofs}, \\ \mathsf{Rs}, \mathsf{OutBs}, \mathsf{OutSs}\end{array}\right\}\right)$$

where $\mathsf{OutBs} = \{ag_2, ag_3\}$ and $\mathsf{OutSs} = \{ag_1\}$.

Intuitively, the σ_i provide "snapshots" of those messages that were sent up to a particular state: the record of the messages sent characterises the state of the scene. Each state is associated with a σ_i and Ω_i. We explicitly list the messages that should be sent at each state of the scene, indicating that the complete set σ_i is an extension of σ_{i-1}. The contents of the sets in Ω_i shown above represent information relevant for defining future steps in the global protocol. This information is gathered as the protocol is followed and it defines the subsequent steps.

The semantics of transitions is defined similarly. However, the sets over which the formulae Fml in $\mathbf{T}_i$ are quantified are built by merging the sets of all those scenes that are connected to the transition, as formally stated in definition 13 above.

3.7 Design Rationale of $\mathcal{L}$

The class of protocols we aim at require the unambiguous reference to details of previous interactions so as to determine the ensuing message exchanges among the participating agents. In the example above, the model is gradually built taking any such restrictions into account: the quantification of the formulae labelling each edge ensures that restrictions be taken into account. For instance, Fml_2 restricts the interaction and only permits buyer agents send messages; these messages must be a reply to their respective offers. This is only possible because the semantics of $\mathcal{L}$ allows the reference to sets built to store any relevant information as edges are followed. This information is then employed via the quantifiers to restrict ensuing steps of the protocol. We are thus able to capture dynamic aspects of the protocol in a generic and abstract fashion.

The logic $\mathcal{L}$, a restricted form of first-order logic, has been engineered for our purposes of labelling connections of a finite-state machine. The set quantifications are just a notational variant of first-order quantification. It is easy to see that, for any arbitrary formula α, if $\forall X \in \mathsf{Set}.\ \alpha$ holds then $\forall X.X \in \mathsf{Set} \wedge \alpha$ also holds. The same is true for the other quantifiers $\exists$ and $\exists!$. The set constraints are just first-order predicates whose intended meaning has been "hardwired" to the underlying semantics.

There are connections between $\mathcal{L}$ and many-sorted logics [4]. The sets employed in our quantifications can be viewed as explicit sorts. However, the set constraints do not have a counterpart in many-sorted logics since sets are not part of the allowed syntax. Set-based logics are not more powerful than standard first-order logic [4]. However, we have decided to employ a set-based logic to provide for a more disciplined design with a cleaner representation. Clearly, all the sets of an $\mathcal{L}$ formula can be put together as one single set (*i.e.* the union

of all sets) but if we needed to differentiate among elements (say, agents that are of different roles) then we should provide extra means. Another advantage of set-based logics stems from the potential reduction on the search space for a model: if our universe of discourse is organised in sets, our search procedure can concentrate only on the sets concerned with the formulae, thus avoiding having to unnecessarily examine large numbers of spurious elements.

3.8 Representing and Checking $\mathcal{L}$-Based E-Institutions

$\mathcal{L}$-based e-institutions can be readily represented in many different ways. We show in Fig. 4 a Prolog [1] representation for the **Agora Room** scene graphi-

```
roles(market,agora,[buyer,seller]). states(market,agora,[w0,w1,w2,w3]).
initial_state(market,agora,w0).     final_states(market,agora,[w3]).
access_states(market,agora,[buyer:[w0],seller:[w0,w2]]).
exit_states(market,agora,[buyer:[w3],seller:[w1,w3]]).
edges(market,agora,[(w0,w1),(w1,w2),(w2,w3)])
guard(market,agora,w0,[exists(B,agents),exists(S,agents)],
                      [m(B,adm,enter(buyer)),m(S,adm,enter(seller))],
                      [in(B,buyers),1 =< card(buyers) =< 10,
                       in(S,sellers),1 =< card(buyers) =< 10]).
label(market,agora,w0,w1,[forall(S,sellers),forall(B,buyers),exists(I,items)],
                         [m(S:seller,B:buyer,offer(I))],
                         [in([S,B,I],offers)]).
...
```

Fig. 4: Representation of **Agora Room** Scene

cally depicted in Fig. 3 above. Each component of the formal definition has its corresponding representation. Since many e-institutions and scenes may co-exist, the components are parameterised by the e-institution and scene names (first and second parameters, respectively). The f^{Guard} component is represented as a `guard`/6 term; to save space, we only show the first of them. Component f^{Label} is represented as `label`/7 – we only show the first of them to save space. Both `guard`/6 and `label`/7 incorporate the same representation for $\mathcal{L}$ formulae, in their last three arguments: a list for the quantifications *Qtf*, a list for the conjunction *Atfs* and a list for the set constraints *SetCtrs*. The actual coding of the logical constructs into a Prolog format is done in a simple fashion: "$\forall x \in \mathsf{Set}$" is coded as `forall(X,set)`, "$\exists x \in \mathsf{Set}$" is encoded as `exists(X,set)`, "$x \in \mathsf{Set}$" (set operation) is encoded as `in(X,set)` and so on.

The terms standing for the messages sent in the `label`s of our representation have been augmented with information on the *role* of the agents which sent them (and the roles of the agents the messages are aimed at). In our example above, the roles `seller` and `buyer` were added, respectively, to the first and second arguments of the message, that is, `m(S:seller,B:buyer,offer(I))`. We shall use this information when we automatically synthesise agents to enact our e-institutions, as explained below. This information can be inferred from the scene specification, by propagating the roles adopted by the agents which `enter`ed the scene in access states. We have adopted the standard messages `enter(`*Role*`)` and `leave` in our scenes to convey, respectively, that the agent wants to enter the scene and incorporate *Role* and that the agent wants to leave the scene.

The representation above renders itself to straightforward automatic checks for well-formedness. For instance, we can check whether all `label`/7 terms are indeed defined with elements of `states`/3, whether all `label`/6 are defined either for `access_states`/3 or `exit_states`/3, if all `access_states`/3 and `exit_states`/3 have their `guard`/6 definition, whether all pairs in `edges`/3 have a corresponding `label`/7, and so on. However, the representation is also amenable for

checking important graph-related properties using standard algorithms [3]. It is useful to check, for instance, if from the state specified in `initial_state`/3 we can reach all other `states`/3, whether there are `states`/3 from which it is not possible to reach an `exit_state` (absence of *sinks*), and so on.

The use of logics for labels in our e-institutions also allows us to explore logic-theoretic issues. Given a scene, we might want to know if the protocol it describes is feasible, that is, if it is possible for a number of agents to successfully enact it. This question amounts to finding out whether there is at least one path connecting the initial (access) state to a final (exit) state, such that the conjunction of the formulae labelling its edges is satisfiable, that is, the conjunction has at least one model. Since the quantified sets in our formulae are finite then the satisfiability test for conjunctions of $\mathcal{L}$ formulae has the same complexity of propositional logic: each atomic formula with variables can be seen as a conjunction of atomic formulae in which the variables are replaced with the actual values over which they are quantified; atomic formulae without variables amount to propositions (because they can be directly mapped to $\top$ or $\bot$). There are algorithms to carry out the satisfiability test for propositional logic which will always terminate [4, 13]. Model-checking techniques (*e.g.*, [9] and [10]) come in handy here, helping engineers to cope with the exponential complexity of this problem.

Transitions are represented in a similar fashion. We show in Fig. 5 how we represented transition $\mathbf{T}_1$ of our agoric market e-institution of Fig. 2. Transition

```
access_state(market,t1,w0). exit_state(market,t1,w1).
connections_into(market,t1,[(admission,client:[w2])]).
connections_outof(market,t1,[(agora,buyer:[w0]),(agora,seller:[w0])]).
label(market,t1,w0,w1,[exists(C,registered_clients)],
                      [m(C,adm,move(agora))],
                      [in(C,agora_agents)]).
```

Fig. 5: Representation of Transition $\mathbf{T}_1$

$\mathbf{T}_1$ guarantees that only those agents that successfully registered in the **Admission** scene (their identification being included in the set `registered_clients`) and that showed their interest in joining the **Agora Room** scene (by sending the message `m(C,adm,move(agora))`) will be able to move through it. We employed the same `label`/7 construct as in the scene representation, but here it stores the *Fml* labelling the edge connecting `w0` and `w1` in the transition.

The above representation for transitions is also amenable for automatic checks. We can automatically verify, for instance, that the scenes, states and roles referred to in `connections_into`/3 and `connections_ outof`/3 are properly defined. A desirable property in transitions is that the connecting scenes have at least one model – this property, as explained above, can be automatically checked. E-institutions are collections of scenes and transitions in the format above, plus the extra components of the tuple comprising its formal definition.

3.9 Enacting $\mathcal{L}$-Based E-Institutions

We have incorporated the concepts above into a distributed enactment platform. This platform, implemented in SICStus Prolog [17], uses the semantics of our constructs to perform a simulation of an e-institution. The platform relies on a number of administrative agents, implemented as independent processes, to overlook the enactment, building models and interacting with the agents par-

taking the enactment via a blackboard architecture, using SICStus Linda tuple space [2, 17].

The platform starts up for each scene an administrative agent *admScene*. An initial model is available for all scenes, $\Im = (\emptyset, \Omega)$ where Ω (possibly empty) contains the values of any sets that need to be initially defined. Some of such sets are, for instance, the identity of those agents that may join the e-institution, the possible values for items and their prices, and so on. Agent *admScene* follows the edges of a scene, starting from w_0 and, using $\Im$, creates the set σ_0 of atomic formulae. Set σ_0 is assembled by evaluating the quantification of $\mathcal{L}_0$ over the sets in Ω.

An enactment of an e-institution begins with the enactment of the root scene and terminates when all agents leave the output scene. Engineers may specify whether a scene can have many instances enacted simultaneously, depending on the number and order of agents willing to enter it. We did not include this feature in our formal presentation because in logic-theoretic terms instances of a scene can be safely seen as different scenes: they are enacted independently from each other, although they all conform to the same specification.

Our platform takes into account the agents that will partake it. These are called the *performing agents* and are automatically synthesised from the description of the e-institution, as described in [19]. A performing agent sends a message by checking if the corresponding σ set contains the message it wants to send; if the message is available then the agent "sends" it by marking it as sent. This mark is for the benefit of the *admScene* agent: the *admScene* agent creates templates for *all* messages that can be sent, but not all of them may in fact be sent. The messages that have been marked as sent are those that were actually sent by the performing agents.

Similarly, a performing agent receives a messages by marking it as received. However, it can only receive a message that has been previously marked as sent by another agent. Both the sending and receiving agents use the format of the messages to ensure they conform to the format specified in the edge they are following. To ensure that an agent does not try to receive a message that has not yet been marked as sent but that may still be sent by some agent, the *admScene* agent synchronises the agents in the scene: it first lets the sending agents change state by moving along the corresponding edge, marking their messages as sent. When all sending agents have moved, then the *admScene* agent lets the receiving agents receive their messages and move to the following state of the scene.

The synchronisation among the agents of a scene is achieved via a simple semaphore represented as a term in the tuple space. The performing agents trying to send a message must wait until this semaphore has a specific value. Likewise, the agents that will receive messages are locked until the semaphore allows them to move. The performing agents inform to the *admScene* agent, via the tuple space, the state of the scene they are currently at. With this information the *admScene* agent is able to "herd" agents from one state to another, as it creates messages templates, lets the sending agents mark them as sent and then lets the receiving agents mark them as received (also retrieving their contents). Those agents that do not send nor receive can move between states without having to wait for the semaphore. All agents though synchronise at every state of the

scene, that is, there is a moment in the enactment when all agents are at state w_i, then after sending and receiving (or just moving) they are all at state w_{i+1}.

Transitions are enacted in a similar fashion. The platform assigns an agent *admTrans* to look after each transition. Transitions, however, differ from scenes in two ways. Firstly, we do not allow instances of transitions. This is strictly a methodological restriction, rather than a technical one: we want transitions to work as "meeting points" for agents moving between scenes and instances of transitions could prevent this. Secondly, transitions are *permanent*, that is, their enactment never comes to an end. Scenes (or their instances), once enacted (*i.e.* all the agents have left it at an exit state), cease to exist, that is, the *admScene* agent looking after it stops.

When a scene comes to an end, the *admScene* agent records in the tuple space the model it built as a result of the scene's enactment. The atomic formulae are only important during the enactment since they actively define the interpretations being built. However, only the sets in the Ω part of the interpretation is left as a record of the enactment. This is useful for following the dynamics of the e-institution, and it is also essential for the transitions. The *admTrans* agents looking after transitions use the sets left behind by the *admScene* agents to build their models.

A model can be explicitly represented and used to guide the distributed enactment of a $\mathcal{L}$-based e-institution. The model representation should be shared by all administrative agents which would use it instead of building its own (sub-)model. Variations of an enactment can still be explored by using *partially defined* models, in which variables are allowed as part of the atfs in σ_i. For instance, σ_1 of our previous agora room scene example, could be defined as $\sigma_1 = \{m(Ag_1, Ag_2, \mathit{offer}(I_1, P_1)), m(Ag_3, Ag_4, \mathit{offer}(I_2, P_2))\} \cup \sigma_0$ that is, the actual values of the agents' identification and items/price are not relevant, but there should be *exactly* two such messages. Restrictions can be imposed or relaxed by adequately using variables or specific values.

4 Conclusions and Future Work

In this paper we have presented a formalism to represent global protocols, that is, all possible interactions among components of a multi-agent system, from a global perspective. The proposed formalism is $\mathcal{L}$ a set-based restricted kind of first-order logic that allows engineers to describe a protocol and to forge relationships among messages of one-to-one, one-to-many and many-to-many interactions.

We have put this formalism to work by embedding it within the definition of electronic institutions [5], giving rise to $\mathcal{L}$-based electronic institutions. Existing formulations of electronic institutions, *e.g.* [5, 6, 14, 19], resort to informal explanations when defining the meaning of their constructs. Our rendition, on the other hand, has its syntax and semantics formally defined using $\mathcal{L}$. We have also presented an implementation of a platform to enact e-institutions represented in our formalism. Our proposal has been exploited for rapid prototyping of large Multi-Agent Systems [20].

Our platform is a proof-of-concept prototype, engineered with two principles in mind: a minimum number of messages should be exchanged and a maximum

distribution and asynchrony among processes should be achieved. Its distributed implementation allows its scale-up: more machines can be used to host its agents. Starting from an e-institution description in our formalism, represented as a sequence of Prolog constructs, the platform starts up a number of administrative agents to overlook the scenes and transitions. The same e-institution formulation is employed to synthesise the agents that will perform in the e-institution, following our approach described in [19]. The specification of the e-institution is used to guide the synthesis of the performing agents and also to control the execution of the administrative agents.

The e-institutions are represented as Prolog terms, in a declarative fashion. We have noticed that this representation is amenable for many different sorts of manipulation. We have used it, for instance, to synthesise agents [18, 19] – these are guaranteed to conform to the e-institution they were synthesised from – and also to guide the execution of general-purpose administrative agents. However, the declarative representation also allows for desirable properties to be checked before we run the e-institution, as explained before.

Our implementation does not take into account message loss or delays. We also assume that there are no malignant agents intercepting messages and impersonating other agents. Our platform can be seen as an idealised correct version of a multi-agent system to be built, whereby the performing agents stands for "proxies" of foreign heterogeneous agents, guaranteed to follow an e-institution. The practical security issues that actual heterogeneous agents are prone to are not transferred on to the e-institution platform. We are working on how agents synthesised from the e-institution specification [19] could be presented to foreign agents and customised as their proxy agents.

References

1. K. R. Apt. *From Logic Programming to Prolog*. Prentice-Hall, U.K., 1997.
2. N. Carriero and D. Gelernter. Linda in Context. *Comm. of the ACM*, 32(4):444–458, Apr. 1989.
3. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, USA, 1990.
4. H. B. Enderton. *A Mathematical Introduction to Logic*. Harcourt/Academic Press, Mass., USA, 2nd edition, 2001.
5. M. Esteva, J. Padget, and C. Sierra. Formalizing a Language for Institutions and Norms. volume 2333 of *LNAI*. Springer-Verlag, 2001.
6. M. Esteva, J.-A. Rodríguez-Aguilar, C. Sierra, P. Garcia, and J. L. Arcos. On the Formal Specification of Electronic Institutions. In F. Dignum and C. Sierra, editors, *Agent Mediated E-Commerce*, volume 1991 of *LNAI*. Springer-Verlag, 2001.
7. FIPA. The Foundation for Physical Agents. `http://www.fipa.org`, 2002.
8. P. R. Halmos. *Naive Set Theory*. Van Nostrand, Princeton, New Jersey, 1960.
9. G. J. Holzmann. The SPIN Model Checker. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.
10. G. J. Holzmann, E. Najm, and A. Serhrouchni. SPIN Model Checking: an Introduction. *Int. Journal of Software Tools for Technology Transfer*, 2(4):321–327, Mar. 2000.
11. J. Hulstijn. *Dialogue Models for Inquiry and Transaction*. PhD thesis, University of Twente, 2000.
12. Y. Labrou, T. Finin, and Y. Peng. Agent Communication Languages: the Current Landscape. *IEEE Intelligent Systems*, 14(2):45–52, 1999.

13. Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill Kogakusha, Ltd., Tokio, Japan, 1974.
14. J. A. Rodríguez Aguilar, F. J. Martín, P. Noriega, P. Garcia, and C. Sierra. *Towards a Formal Specification of Complex Social Structures in Multi-Agent Systems*, pages 284–300. Number 1624 in LNAI. Springer-Verlag, Berlin, 1997.
15. P. McBurney, R. van Eijk, S. Parsons, and L. Amgoud. A Dialogue-Game Protocol for Agent Purchase Negotiations. *Journal of Autonomous Agents and Multi-Agent Systems*, 2002. In Press.
16. J. A. Rodriguez. *On the Design and Construction of Agent-mediated Electronic Institutions*. PhD thesis, Institut d'Investigació en Intel.ligència Artificial (IIIA), Consejo Superior de Investigaciones Científicas (CSIC), Spain, 2001.
17. SICS. SICStus Prolog User's Manual. Swedish Institute of Computer Science, available at `http://www.sics.se/isl/sicstus2.html#Manuals`, Feb. 2000.
18. W. W. Vasconcelos, D. Robertson, C. Sierra, M. Esteva, J. Sabater, and M. Wooldridge. Rapid Prototyping of Large Multi-Agent Systems through Logic Programming. Submitted for publication, 2003. Document available from authors upon request.
19. W. W. Vasconcelos, J. Sabater, C. Sierra, and J. Querol. Skeleton-based Agent Development for Electronic Institutions. In *Proc. 1st Int'l Joint Conf. on Autonomous Agents & Multi-Agent Systems (AAMAS 2002)*, Bologna, Italy, 2002. ACM, U.S.A.
20. W. W. Vasconcelos, C. Sierra, and M. Esteva. An Approach to Rapid Prototyping of Large Multi-Agent Systems. In *Proc. 17th IEEE Int'l Conf. on Automated Software Engineering (ASE 2002)*, Edinburgh, UK, 2002. IEEE Computer Society, U.S.A.
21. T. Wagner, B. Benyo, V. Lesser, and P. Xuan. Investigating Interactions between Agent Conversations and Agent Control Components. In F. Dignum and M. Greaves, editors, *Issues in Agent Communication*, pages 314–330. Springer-Verlag: Heidelberg, Germany, 2000.

Modeling interactions using *social integrity constraints*: a resource sharing case study*

Marco Alberti[1], Marco Gavanelli[1], Evelina Lamma[1],
Paola Mello[2], and Paolo Torroni[2]

[1] Dipartimento di Ingegneria, Università degli Studi di Ferrara
Via Saragat, 1 - 44100 Ferrara (Italy)
{malberti|mgavanelli|elamma}@ing.unife.it
[2] DEIS, Università degli Studi di Bologna
Viale Risorgimento, 2 - 40136 Bologna (Italy)
{ptorroni|pmello}@deis.unibo.it

Abstract. *Computees* are abstractions of the entities that populate global and open computing environments. The *societies* that they populate give an institutional meaning to their interactions and define the allowed interaction protocols. *Social integrity constraints* represent a powerful though simple formalism to express such protocols. Using social integrity constraints, it is possible to give a formal definition of concepts such as violation, fulfillment, and social expectation. This allows for the automatic verification of the social behaviour of computees. The aim of this paper is to show by a concrete example how the theoretical framework can be used in practical situations where computees can operate. The example that we choose is a resource exchange scenario.

1 Introduction

Global Computing [12] is a European Union initiative which aims at obtaining technologies to harness the flexibility and power of rapidly evolving interacting systems composed of autonomous computational entities, where activity is not centrally controlled, where the computational entities can be mobile, the configuration may vary over time, and the systems operate with incomplete information about the environment.

The problems that such an ambitious objective poses can be tackled from different perspectives: along with aspects such as efficiency and scalability, there are also other ones like formal soundness, possibility to prove properties and to reason on the behaviour of the system, predictability, verifiability, semantics. Declarative methods and formalisms can be used to face the problem from this second perspective. Logic programming in particular is a suitable tool to model the reasoning capabilities of autonomous entities, to give semantics to their interaction, and to throw a bridge between formal specification and operational model.

* This work is partially funded by the Information Society Technologies programme of the European Commission under the IST-2001-32530 SOCS project.

Computees [25] are abstractions of the entities that populate global and open computing environments. Computees have a declarative representation of knowledge, capabilities, resources, objectives and rules of behaviour. Each computee typically has only a partial, incomplete and possibly inaccurate view of the society and of the environment and of the other computees, and it might have inadequate resources or capabilities to achieve its objectives. Computees are characterized by exhibiting reasoning abilities, based on a declarative representation of knowledge and on computational logic-based functionalities (e.g. abduction, learning, planning, and so forth). These entities can form complex organizations, which we call *societies of computees*.

Using a computational logic based approach it is possible to specify the set of allowed interaction patterns among computees in the society. In general, such patterns are expressed as protocols, which we model by means of *social integrity constraints*.

Using social integrity constraints, it is possible to give a formal definition of concepts such as violation, fulfillment, and social expectation. The use of social integrity constraints is twofold. In fact, through them the society can automatically verify the compliance of its members to the protocols, and can actively suggest to its members what are possible conforming behaviours, thus guiding them in their social life.

The idea is to exploit abduction for checking the compliance of the computation at a social level. Abduction captures relevant events (or hypotheses about future events, that we call *expectations*), and a suitably extended abductive proof procedure can be used for integrity constraint checking. Expectations are therefore mapped into abducible predicates, and the social infrastructure is based on an extended abductive framework where socially relevant events are dynamically taken into consideration, as they happen.

In this work, we explain our ideas with a focus on protocols: we provide an intuitive understanding of the semantics of social integrity constraints, and we show how they can be used to verify protocol compliance. We adopt as a running example a resource sharing problem, inspired by the work done by Sadri & al. [23], where the authors propose a multi-agent solution to the problem, based on a logic programming framework. In such a framework, agents initiate negotiation dialogues to share resources along time, and can follow one among several protocols, depending on how much information they wish to disclose in order to achieve some missing resources.

Drawing inspiration from [23], in our example the society of computees defines as interaction protocols those for resource sharing. As the interactions proceed, we show the evolution of social expectations and their possible fulfillment, or the raising of a violation if some expectations are disregarded.

The paper is structured as follows. In Section 2 we introduce the idea of social integrity constraints. In Section 3 we explain the resource exchange scenario. In Section 4 we give an example of social integrity constraints and we show how social expectations are generated as the computees interact among each other. In Section 5 we show how the formalism that we propose can be used to verify

the conformance of computees to social interaction protocols. In Section 6 we relate our work with other proposals of literature and with our past and current work within the SOCS project. Conclusions follow.

2 Social integrity constraints

We envisage a model of society, tolerant to partial information, which continues to operate despite the incompleteness of the available knowledge. In such a model, the society is time by time aware of social events that dynamically happen in the social environment. Moreover, the society can reason upon the happened events and the protocols that must be followed by the computees, and therefore define what are the *expected social events*. These are events which are not yet available (to the society), but which are expected if we want computees to exhibit a *proper* behaviour, i.e., conforming to protocols.

Such expectations can be used by the society to behave pro-actively: suitable social policies could make them public, in order to try and influence the behaviour of the computees towards an ideal behaviour.

Indeed, the set of expectations of the society are adjusted when it acquires new knowledge from the environment on social events that was not available while planning such expectations. In this perspective, the society should be able to deal with unexpected social events from the environment, which violate the previous expectations.

This can be the case in an open environment where "regimentation" (see [7]) cannot be assumed. In an open society, where computees are autonomous, unexpected events can bring to a state of *violation*, from which it could be necessary to recover by taking appropriate measures (e.g., sanctions), in order to bring the society back to a consistent state.

The knowledge in a society is composed of three parts: organizational knowledge, environmental (including an events record), and Social Integrity Constraints to express the allowed interaction protocols. In this document, we will focus more on this last part of the society knowledge, which expresses what is expected to happen or not to happen, given some event record. For example, a social integrity constraint could state that the manager of a resource should give an answer to whomever has made a request for that resource.

Protocols are specified by means of Social Integrity Constraints (IC_S). IC_S relate socially significant happened events and expected events. Intuitively, IC_S are forward implications used to produce *expectations* about the behavior of computees. They are used to check if a computee inside the society behaves in a permissible way with respect to its "social" behavior.

IC_S are forward implications

$$\chi \rightarrow \phi$$

which contain in χ a conjunction of social events or expectations, and in ϕ a disjunction of conjunctions of expectations. Expectations can be of two kinds: positive (**E**) and negative (**NE**), and their variables can be constrained.

Happened events are denoted by **H**. Intuitively, an **H** atom represents a socially significant event that happened in the society, i.e., social events are mapped into **H** predicates. Events that happen, such as dialogue moves (social events), are part of the environmental knowledge of the society.

Being the focus of this paper on the motivation of the use of social integrity constraints in a declarative agent programming setting, we will not give here more detail about IC_S. In a companion paper [3] we define the full syntax of social integrity constraints, the scope of variables, quantification, and we give some results about the conditions for a proper behaviour of the framework. Also, we give a formal semantic characterization of concepts such as coherence and consistency of sets of expectations and their fulfillment. Instead, in [3] we do not discuss how the framework can be used to prove properties of interactions. The aim of this paper is to show by a concrete example how the theoretical framework can be used in practical situations where computees can operate. We will therefore give below a flavour of the operational behaviour of the framework.

In our approach, computees autonomously perform some form of reasoning, and the society infrastructure is devoted to ensure that in performing their tasks they do not violate the established rules and protocols.

H events, **E**/**NE** expectations, and society knowledge and protocols can be smoothly recovered into an abductive framework, so to exploit well-assessed proof-theoretic techniques in order to check the compliance of the overall computation with respect to the expected social behavior.

Here, we adopt an approach where abduction is used to record expectations. The dynamic knowledge available at social level grows up during the computees' own reasoning, through knowledge acquisition.

We represent the knowledge available at the social level as an Abductive Logic Program (ALP) [8], since we want to deal with incomplete knowledge. In particular, in order to model interactions, the incompleteness of their knowledge includes ignorance about communicative acts that still have to be made. The idea of modelling communicative acts by abduction is derived from [16], where the abducibles are produced within an agent cycle, and represent actions in the external world.

In our proposal, social events are recorded as **H** events. A second class of events is represented by raised expectations (about the happening, **E**, and not happening, **NE**, of events: we will call **E** and **NE** respectively positive and negative expectations), represented as abducible atoms in our case. Finally, we have negated expectations ($\neg$**E** and $\neg$**NE**), also represented as abducible atoms, in accordance with the usual way abduction can be used to deal with negation [8]. The set **EXP** can be seen as a set of hypotheses (possibly, as it will be exemplified in Section 4, a set of disjunctions of atomic hypotheses [21,11]).

At the society level, knowledge can be represented as an abductive logic program, i.e., the triple: $\langle KB, \mathcal{E}, IC \rangle$ where:

- KB is the knowledge base of the society. It contains the organizational and environmental knowledge, including happened events (we denote by **HAP** the event record: $KB \supseteq$ **HAP**);

– $\mathcal{E}$ is a set of *abducible predicates*, standing for positive and negative expectations, and their negation;
– IC is the set of social integrity constraints, IC_S.

The idea is to exploit abduction for checking the compliance of the computation at a social level. Abduction captures relevant events (or hypotheses about future events), and a suitably extended abductive proof procedure can be used for integrity constraint checking. **EXP** is a coherent and consistent set of abducibles if and only if

$$KB \cup \mathbf{EXP} \cup IC_S \not\vdash \bot \tag{1}$$

and conformance to rules and protocols is guaranteed by:

$$\mathbf{HAP} \cup \{\mathbf{E}(p) \rightarrow \mathbf{H}(p)\} \cup \{\mathbf{NE}(p) \rightarrow \neg\mathbf{H}(p)\} \cup \mathbf{EXP} \not\vdash \bot \tag{2}$$

In this last condition, expectations are put into relationship with the events, which gives a notion of fulfillments. If (2) is not verified, then a *violation* occurs.

In [3] we provide a declarative semantics. A suitable proof procedure still has to be defined in order to efficiently deal with such a semantics for this framework. In particular, we envisage an incremental fulfillment check, in order to detect violations as soon as possible.

3 Negotiation for resource sharing

In this section we briefly recall the resource exchange scenario defined in [23]. Let us consider a system where computees have *goals* to achieve, and in order to achieve them they use plans. Plans are partially ordered sequences of activities. Activities have a duration and a time window in which they have been scheduled by the computee. In order to execute the activities, computees may need some resources during the scheduled time window.

An activity that requires a resource r is said to be *infeasible* if r is not available to the computee that intends to execute it. Similarly, infeasible is also a *plan* that contains an activity which is infeasible, and so is the *intention* of a computee, containing such plan.[1] The resources that computees *need* in order to perform an action in a plan but that they do not possess are called *missing* resources.

In fact, what we mean when we say *resource* is only an abstract entity, identified by its *name*, which possibly symbolizes a physical resource, such as a bike or a scooter. We do not explicitly model the actual delivery of physical resources either.

The *resource exchange problem* is the problem of answering to the following *question*: Does there exist a time τ during the negotiation process when the resource distribution is such that each computee has the resources it requires for

[1] In [23], *plans* are modeled as part of the computee *intentions*.

time periods that would allow it to perform the activities in its intention, within their specified time windows?

In [23], the authors propose a framework for resource exchange, where the computees can interact following different protocols. Without loss of generality, resources are assumed to be non consumable. The protocols are ordered into stages, each characterised by an increased chance of a mutually agreeable deal but at the price of disclosing more and more information. In the sequence of stages, the computees may agree to move on to the next stage if the previous stage fails to produce a deal amongst them.

In particular, the authors define two different stages of negotiation, each characterized by the degree of flexibility of the computees and the amount of information disclosed and used by them:

Stage 1: Request/flexible schedule
Stage 2: Blind deal

The first stage implements a two-step request/accept/refuse protocol. The policy that a computee may adopt upon receival of a request is to try and give the resource in question. This may require a change in its own activity schedule. The second stage implements a more elaborate protocol, where the computee who answers to a request can either accept or refuse the request, as in Stage 1, but can also propose a series of deals.

In this paper, we choose the multi-stage negotiation architecture not only because it is based on abductive logic programming, which makes it easy to define the negotiation protocols into social integrity constraints, but also because it presents different protocols, which suggests interesting observations about society engineering and the need to have a formal definition of interaction protocols, and an operational framework to reason upon.

The policies adopted by the negotiating peers at Stage 1 and Stage 2 implement a very collaborative behaviour. The set of problems that can be solved at the various stages can be formally defined (see [23] for details). But in an open society, where computees are free to choose their own policy, the only thing that we can look at, from a social verification perspective, are the exchanged messages, and the interaction protocols that they follow.

In the next section, we will define the protocols that are followed when negotiating at Stage 1 and at Stage 2, and show how expectations evolve as the negotiation proceeds.

4 Social expectations for a resource sharing scenario

The protocol followed at Stage 1 is shown in Figure 1 for two generic computees X and Y: once X makes a *request* to Y for R, Y can either *accept* or *refuse* the request. Note that there is no mention of the policies adopted by the computees (for instance, a computee could well refuse all requests and not be collaborative at all, whatever the request and its actual resource allocation: it would still be compiant to the protocol).

$$(IC_{S1})\ \ \mathbf{H}(tell(X, Y, \mathbf{request}(\mathbf{give}(R, (Ts, Te))), D, T)) \\ \rightarrow \mathbf{E}(tell(Y, X, \mathbf{accept}(\mathbf{request}(\mathbf{give}(R, (Ts, Te)))), D, T')) : T < T' \\ \vee \mathbf{E}(tell(Y, X, \mathbf{refuse}(\mathbf{request}(\mathbf{give}(R, (Ts, Te)))), D, T')) : T < T'$$

Fig. 1. *Stage 1* protocol

H and **E** enclose *tell* atoms, which represent communicative acts, and whose parameters are: sender, receiver, subject, dialogue identifier and time of the communicative act.

Social integrity constraints can encode extra knowledge about this protocol: in particular, in Figure 1 there is nothing saying how a protocol starts/ends. In Figure 2, we see how by IC_S we can express the fact that some moves are "final", in the sense that they mean to put an end to a conversation, while some others are "initial", i.e., they are never preceded by other moves in the same conversation. We refer to the language $\mathcal{L}_{1-TW}$ defined in [23], where *request*(...) is the only allowed initial move, and *accept*(*request*(...)), *refuse*(*request*(...)) and *refuse*(*promise*(...)) are final moves. We see here another use of integrity constraints, different from that in Figure 1: IC_S are used in fact to generate *negative* expectations, which tell what should not happen, given a certain course of events.

$$(IC_{S2})\ \ \mathbf{H}(tell(\_, \_, \mathbf{request}(Req), D, T)) \\ \rightarrow \mathbf{NE}(tell(\_, \_, \_, D, T')) : T' < T$$

$$(IC_{S3})\ \ \mathbf{H}(tell(\_, \_, \mathbf{accept}(Req), D, T)) \\ \rightarrow \mathbf{NE}(tell(\_, \_, \_, D, T')) : T' > T$$

$$(IC_{S4})\ \ \mathbf{H}(tell(\_, \_, \mathbf{refuse}(Req), D, T)) \\ \rightarrow \mathbf{NE}(tell(\_, \_, \_, D, T')) : T' > T$$

Fig. 2. Negative expectations following from the semantics of the negotiation language

In Figure 2, *Req* can be any kind of request or proposed deal (*promise*).

In Figure 3 we define the protocol for Stage 2 negotiation. By IC_{S5}, after a *request* the possible moves are those of Stage 1, plus *promise*. By IC_{S6}, after *promise* one expects either an *accept* or a *change*(*promise*(...)). By IC_{S7}, after a *change*(*promise*(...)) one expects either another *promise* or a *refuse* of the original request, which terminates the dialogue. Finally, by IC_{S8}, one does not expect the same *promise* to be made twice.

$$
\begin{array}{ll}
(IC_{S5}) & \mathbf{H}(\mathit{tell}(X,Y,\mathbf{request}(\mathbf{give}(R,(Ts,Te))),D,T)) \\
& \quad \rightarrow \mathbf{E}(\mathit{tell}(Y,X,\mathbf{accept}(\mathbf{request}(\mathbf{give}(R,(Ts,Te)))),D,T')) : T < T' \\
& \quad \vee \mathbf{E}(\mathit{tell}(Y,X,\mathbf{refuse}(\mathbf{request}(\mathbf{give}(R,(Ts,Te)))),D,T')) : T < T' \\
& \quad \vee \mathbf{E}(\mathit{tell}(Y,X,\mathbf{promise}(R,(Ts',Te'),(Ts,Te)),D,T')) : T < T' \\
\\
(IC_{S6}) & \mathbf{H}(\mathit{tell}(X,Y,\mathbf{promise}(R,(Ts,Te)),D,T)) \\
& \quad \rightarrow \mathbf{E}(\mathit{tell}(Y,X,\mathbf{change}(\mathbf{promise}(R,(Ts',Ts'),(Ts,Te)),D,T'))) : \\
& T < T' \\
& \quad \vee \mathbf{E}(\mathit{tell}(Y,X,\mathbf{accept}(\mathbf{promise}(R,(Ts,Te))),D,T')) : T < T' \\
\\
(IC_{S7}) & \mathbf{H}(\mathit{tell}(X,Y,\mathbf{change}(\mathbf{promise}(R,(Ts',Te'),(Ts,Te))),D,T)) \\
& \quad \rightarrow \mathbf{E}(\mathit{tell}(Y,X,\mathbf{promise}(R,(Ts",Te"),(Ts,Te)),D,T')) : T < T' \\
& \quad \vee \mathbf{E}(\mathit{tell}(Y,X,\mathbf{refuse}(\mathbf{request}(\mathbf{give}(R,(Ts,Te)))),D,T')) : T < T' \\
\\
(IC_{S8}) & \mathbf{H}(\mathit{tell}(X,Y,\mathbf{promise}(R,(Ts',Te'),(Ts,Te)),D,T)) \\
& \quad \rightarrow \mathbf{NE}(\mathit{tell}(X,Y,\mathbf{promise}(R,(Ts',Te'),(Ts,Te)),D,T')) : T < T'
\end{array}
$$

Fig. 3. *Stage 2* protocol

If we look at both protocols, for Stage 1 and Stage 2, we understand that $IC_{S1} \Rightarrow IC_{S5}$, in the sense that IC_{S5} is more general than IC_{S1}. In a more elaborate solution, we could explicitly add a "stage identifier" in the communication acts, and keep a *request* made at Stage 1 different from a *request* made at Stage 2. Here, for the sake of brevity, we will simply override Stage 2 with Stage 1. The social integrity constraints that we consider, in particular, are only those of Figure 2 and 3.

From this example it is possible to see a first achievement of the use of social integrity constraints to formally define protocols: we are able to formally reason on the protocols and adopt a modular approach to society engineering. For instance, it would be interesting to know that if we put Stage 1 constraints and Stage 2 constraints together as they are (without the stage identifier), then the resulting protocol is not the union of the two stages, but a more restrictive protocol than that. The study of tools to automatically prove some relationships among integrity constraints or protocols is subject for current investigation.

We would now like to show how social expectations are created and evolve as computees exchange requests and reply to each other. For the sake of the example, we consider a system composed of three computees: *david*, *yves*, and *thomas*. The resource that they share is a *scooter*. *david* is normally entitled to have the scooter in the afternoon, while *yves* in the morning.

Let us assume that at time 1 *david* makes a request to *yves*, because he needs the scooter from 10 to 11 in the morning.[2] This request is recorded by the society and put into **HAP**:

$$\mathbf{H}(tell(david, yves, request(give(scooter(10, 11))), d, 1))$$

By looking at the history, the society modifies its set of expectations. Such a set could be initially empty. After the event at time 1, due to IC_{S5}, the society grows with expectations. Since IC_{S5} has a disjunction as a consequence of an **H** atom, the expectations will also be disjunctions of atoms. In general, we will have an expression **EXP** which could be put in the form of a disjunction of conjunction of expectations.

At time 2, we have

$$\begin{gathered}\mathbf{EXP}_1 = \{ \\ (\ (\ \mathbf{E}(tell(yves, david, accept(request(give(scooter, (10, 11)))), d, T) : T > 1)\ ) \\ \vee(\ \mathbf{E}(tell(yves, david, refuse(request(give(scooter, (10, 11)))), d, T) : T > 1)\ ) \\ \vee(\ \mathbf{E}(tell(yves, david, promise(scooter, (10, 11), (Ts, Te)), d, T) : T > 1)\ )\ ) \\ \wedge(\ \mathbf{NE}(tell(\_, \_, \_, d, T') : T' < 1)\ ) \\ \}\end{gathered}$$

Let us assume that at time 3 *yves* proposes a deal to *david*:

$$\mathbf{H}(tell(yves, david, promise(scooter, (10, 11), (20, 23)), d, 3))$$

EXP changes. It becomes:

$$\begin{gathered}\mathbf{EXP}_2 = \{ \\ (\ (\mathbf{E}(tell(yves, david, accept(request(give(scooter, (10, 11)))), d, T) : T > 1)) \\ \vee(\mathbf{E}(tell(yves, david, refuse(request(give(scooter, (10, 11)))), d, T) : T > 1)) \\ \vee(\mathbf{E}(tell(yves, david, promise(scooter, (10, 11), (Ts, Te)), d, T) : T > 1))\ ) \\ \wedge\ (\ (\mathbf{E}(tell(david, yves, accept(promise(scooter, (10, 11), (20, 23))), d, T) : T > 3)) \\ \vee(\mathbf{E}(tell(david, yves, change(promise(scooter, (10, 11), (20, 23))), d, T) : T > 3))\ ) \\ \wedge\ \mathbf{NE}(tell(yves, david, promise(scooter, (10, 11), (20, 23)), d, T') : T' > 3) \\ \}\end{gathered}$$

The disjunction **EXP** will keep evolving along with the social events. In [3] the authors give a declarative semantics to expectations and social integrity constraints. They define a notion of fulfillment of expectations. For instance, in our example, we can see that an expectation which is present in $\mathbf{EXP}_1$ at time 2 is then fulfilled by *yves*' message to *david* at time 3. Similarly, we have a violation if at a later time something happens which is expected not to happen, e.g., *yves* repeats the same promise for a second time.

The resource exchange scenario allows us to exemplify some advantages of our formalism. Social integrity constraints gave use the ability to:

[2] We make the simplifying assumption that the time of communication acts is centrally assigned, e.g. by the "social infrastructure", and that all computees are able to cope with this. We can then consider the time of the society as a transaction time.

– express allowed communication patterns inside a society (e.g., the protocols for Stage 1 and Stage 2). This is done independently of the computees' internal policies and implementation;
– use the protocol definitions and the history of socially relevant events to generate at run-time the possible combinations of future events that represent a "proper" behaviour of the computees (e.g., $\mathbf{EXP}_1$ and $\mathbf{EXP}_2$);
– formally reason on the composition of protocols, and adopt a modular approach to society engineering (e.g. the composition of Stage 1 and Stage 2 defines a new, more restrictive protocol);
– verify at run-time the correct behaviour of some computees, with respect to the protocols, without having access to their internals (e.g., if *yves* repeats the same promise for a second time we enter a violation state).

As future extensions, we intend to investigate the issue of protocol composition, and the notion of violation, particularly about how to recover from a state of violation, and possibly — given such a state — how to identify one or more "culprits" and generate appropriate sanctions. Having all these elements in a unified declarative framework would be an important achievement.

5 Social integrity constraints for verification

In [13,20], F. Guerin and J. Pitt propose a classification of properties that are relevant for e-commerce systems, in particular with respect to properties of protocols and interactions. In this setting, they propose a formal framework for verification of properties of "low level computing theories required to implement a mechanism for agents" in an *open* environment, where by open the author mean that the internals of agents are not public.

Verification is classified into three types, depending on the information available and whether the verification is done at design time or at run time:

Type 1: verify that an agent will always comply;
Type 2: verify compliance by observation;
Type 3: verify protocol properties.

As for *Type 1* verification, the authors propose using a model checking algorithm for agents implemented by a finite state program. As for *Type 2* verification, the authors refer to work done by Singh [24], where "agents can be tested for compliance on the basis of their communications", and suggest policing the society as a way to enforce a correct behaviour of its inhabitants. As for verification of *Type 3*, the authors show how it is possible to prove properties of protocols by using only the ACL specification. They construct a fair transition system representing all possible observable sequences of states and prove by hand that the desired properties hold over all computations of the multi-agent system. This type of verification is demonstrated by an auction example.

The formal framework that we propose for modelling interactions in an open society of computees lends itself very well to all these kinds of verification. In

fact, both (public) protocols and (internal) policies are expressed in the same formalism, which makes it possible to relate social aspects with individual aspects in a static verification. We believe that all the above three types of verification can be *automatically* done in a computational logic setting.

Verification that a computee will always comply cannot be done by externally monitoring its behaviour. As Hume says [15], we are in a natural state of ignorance with regard to the powers and influence of all objects when we consider them a priori (IV.ii.32). For this kind of verification, we need to have access to the computees' internals, or to its specifications. We would be advantaged in this task if we could express the computee program and policies by means of an abductive logic programming based formalism, as it is shown in [22]: in that case, specification and implementation coincide.

The proof that a given computee c will always comply with a set $\mathcal{IC}_S$ of constraints representing a protocol, based on the agents' specifications, could be the following. We show that for all the constraints in $\mathcal{IC}_S$, if in the head of a constraint there is a social event which is expected from c, then, the history of events that from a social viewpoint leads to such expectation, leads by that computee's viewpoint to producing an event that fulfills it (or prevents the computee from producing any event that violates it, in the case of negative expectations).

In the scooter example, if *yves*'s specifications are such that the constraint:

$$\mathbf{H}(tell(C, yves, request(give(scooter, (10, 11))), d, T))$$
$$\rightarrow \mathbf{H}(tell(yves, C, accept(request(give(scooter, (10, 11)))), d, T') : T' > T)$$

is never violated, then *yves* is compliant with the protocol, because the event that raises the expectation

$$\mathbf{E}(tell(yves, C, accept(request(give(scooter, (10, 11)))), d, T))$$

in the society also makes *yves* generate an event which fulfills that expectation.

The study of a mechanism to automatically obtain such a proof (or its failure) is subject for current investigation.

For verification of Type 2 we need to be able to observe the computees' social actions, i.e., the communicative acts that they exchange. As in [20], we can assume that this can be achieved by policing the society. In particular, "police" computees will "snoop" the communicative acts exchanged by the society members and check at run time if they comply with the protocols specifications (social integrity constraints).

This kind of run-time verification is theoretically already built in our framework and we do not need to provide additional formal tools to achieve it. In [1] Alberti & al. show an implementation of social integrity constraints based on the CHR language [10].

Verification of Type 3 is about protocol properties. In order to prove them we do not need to access the computees' internals, nor to know anything about the communication acts of the system, because it is a verification which is statically done at design time. As we already mentioned in Section 4, we are working on

the design of a logic-based formalism to automatically detect inconsistencies of combinations of protocols. In general, one of the main motivations behind our formal approach is to be able to prove all properties that can be formally defined, e.g. by means of an invariant or an implication, as they are defined in [20].

6 Discussion

Computees are abstractions of the entities that populate global and open computing environments. The intuition behind a computee is very similar to that behind an agent. The reason why we adopt a different name is because we want to refer to a particular class of agents, which can rely on a declarative representation of knowledge, and on reasoning methods grounded on computational logic. In this way, a declarative representation of the society knowledge and of social categories such as expectations, as we defined them in this work, can be smoothly integrated with the knowledge of its inhabitants, and similar techniques can be used by them to reason upon either knowledge base or upon a combination of them. The main motivation of our approach is in its declarative nature, which has the potential to aiding a user's understanding of a system's specification, and in its solid formal basis, which puts together, under the same formalism, specification, implementation, and verification for multi-agent systems.

In [9], Esteva & al. give a formal specification of agents societies, focussing on the social level of electronic institutions. In that model, each agents in a society plays one or more *roles*, and must conform to the pattern of behavior attached to its roles. The *dialogic framework* of a society defines the common ground (ontology, communication language, knowledge representation) that allows heterogeneous agents to communicate. Interactions between agents take place in group meetings called *scenes*, each regulated by a communication protocol; connections between scenes (for instance, an agent may have to choose between different scenes, or its participation in a scene may be causally dependent on its participation in another) are captured by the *performative structure*. *Normative rules* specify how agents' actions affect their subsequent possible behavior, by raising obligations on them.

In our framework, agents are only required to perform their communicative acts by a given computee communication language; they are not required to share an ontology (which, strictly speaking, need not even be specified). Possible interactions are not organized in separate (although inter-connected) scenes; agents' interactions are supposed to take place in one shared interaction space. For this reason, we do not distinguish, as in [9], between intra-scene (possible interaction paths *inside* inside a scene) and inter-scene (possible paths of an agent *through* scenes) normative rules. Without this distinction, normative rules are strictly related to our social integrity constraints, in that both constrain agents' future behavior as a consequence of their past actions. We would like to stress that Esteva & al., based on the analysis presented in [6] by Dellarocas and Klein, start from the same requisites as we, considering as major issues heterogeneity, trust and accountability, exception handling and societal changes,

and provide in their work a basis for a verified design of electronic marketplaces, by giving a very detailed formal specification of all the aspects of their electronic institutions, and graphical tools to make such a specification easy to understand. However, the focus of their work does not seem to be on a direct and automatic specification-verified implementation relationship, as in ours.

Our framework does not specifically cater for roles (very little we said about he structure of the knowledge related to the society). However, our aim is to propose a declarative framework where roles can indeed be expressed, but do not represent a first class entity in the definition of agent interactions in general. For instance, in a semi-open society [5] roles could be assigned to incoming computees by "custom officer" computees, or they could be statically defined in the society knowledge base, or else dynamically acquired along the time. It is not difficult to imagine social integrity constraints that define the protocols for role management.

In [27], Vasconcelos presents a very neat formalization based on first order logics and set theory to represent an expressive class of electronic institutions. As in our work, the use of variables and quantification over finite sets allow to express significant protocols patterns. The definition of allowed interaction patterns is based on the concept of scene, similarly to what is done in [9]. How expressive is the formalism proposed in [27] with respect to ours, and what are the differences in terms of verification, are issues that we would like to address in the future.

In [19] Moses and Tennenholtz focus on the problem of helping interacting agents achieve their individual goals, by guaranteeing not only mere achievability of the goals, but also computational feasibility of planning courses of actions to realize them. Since an agent's behavior can affect the achievability of other agents' goals, the agents' possible actions are restricted by means of *social laws*, in order to prevent agents from performing actions that would be detrimental to other agents.

In our work, we aim more at verifying sound social interaction, rather than at guaranteeing properties of individual computees. Moreover, computing appropriate social laws requires a (correct) modelling of individual agents in terms of states, state-actions relations and plans, thus assuming reliable knowledge about agents' internals, which we do not require.

Gaia is a methodology in software engineering developed by Wooldridge & al. for agent-oriented analysis and design [28]. The developer of a MAS has to define *roles* that agents will then embody. Each role is defined by its *responsibilities*, *permissions*, *activities* and *protocols*. In particular, responsibilities explain how an agent embodying the corresponding role should behave in terms of *liveness* (good things that should happen) and *safety* (bad things that should not happen) expressions.[3] These expressions can be seen as abstractions of our expectations, or, in a sense, our Social Integrity Constraints could be seen as a further refinement and a formalization of the concepts of responsibilities. More-

[3] This distinction is due to Lamport [17].

over, we use Social Integrity Constraints also to model *protocols*, i.e., interaction with other agents/roles.

Considerable work has been done about the formal definition and verification of properties of multi-agent systems, but to the best of our knowledge there is not yet a reference paper with a formal classification and definition of properties that are considered interesting in a general setting. In Section 5, we referred to the work by Guerin and Pitt because it neatly pins down the main requisites and characteristics of a multi-agent system verification process (run-time verification vs. static-verification, and visibility of agents's internals). Although it is specially oriented to electronic commerce applications, we believe that its main ideas can be extended to a more general case of agent interactions.

Among other work that proposes list of properties of agent systems, more or less formally defined and related to particular domains, we cite work done by Mazouzi & al. [18], Yolum & Singh [29], Hewitt [14], Artikis & al. [4], and Davidsson [5].

We conclude this section by putting this work in relationship with our past and current activity within the SOCS project. In [26] a layered architecture for societies of computees has been proposed, where at the bottom level a platform is used to implement the system and give support to computees' communication, and a communication language layer defines syntax and semantics of communicative acts, while society and protocols are in a higher layer. The purpose the higher layers is to determine the set of allowed interaction patterns among computees in the society. A social semantics for communicative acts has been presented in [2], along with a discussion about the advantages and motivation of a social semantics of communication with respect to other approaches, and in [1] an implementation of a restricted class of Social Integrity Constraints is proposed, based on the CHR language [10]. [3] defines the full syntax of social integrity constraints, the scope of variables, quantification, and gives some results about the conditions for a proper behaviour of the framework, along with a formal semantic characterization of concepts such as coherence and consistency of sets of expectations and their fulfillment.

With respect to the work that we have been doing and that we have briefly reviewed above, this is the first paper which aims at showing the practical use of our theoretical framework, by means of a simple though realistic case study. The protocols that we used for our example are taken from the literature, and could be used to solve resource sharing problems among agents. We also contributed in showing how the framework can be the basis for the automatic proof of properties of interactions and protocols.

7 Conclusion

In this work, we have shown how a logic programming based framework is a suitable tool to give semantics to the interactions of autonomous entities populating a global computing environment. We illustrated the framework by means of a resource sharing example. The main idea is that of social integrity constraints,

and of a correspondence with abductive frameworks of logic programming to provide a semantic to such constraints.

This paper wants to give a motivation to a formal approach by showing a concrete example of the expected operation of the framework.

The use of social integrity constraints could be twofold. In fact, through them the society can automatically verify the compliance of its members to the protocols, and ideally it could actively suggest to its members what are possible conforming behaviours, thus guiding them in their social life. The paper does not cover the last issue, which is subject for future work. In both aspects, our work represents a novel contribution to the research in the area.

References

1. M. Alberti, A. Ciampolini, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. Logic Based Semantics for an Agent Communication Language. In *Proceedings of the International Workshop on Formal Approaches to Multi-Agent Systems (FAMAS)*, Warsaw, Poland, April 12 2003. To appear.
2. M. Alberti, A. Ciampolini, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. A social ACL semantics by deontic constraints. In V. Marik, J. Müller, and M. Pechoucek, editors, *Proceedings of the 3rd International/Central and Eastern European Conference on Multi-Agent Systems (CEEMAS)*, 2003.
3. M. Alberti, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. An Abductive Computational Model for Open Societies. 2003. Under review.
4. A. Artikis, J. Pitt, and M. Sergot. Animated specifications of computational societies. In C. Castelfranchi and W. Lewis Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2002), Part III, Bologna, Italy*, pages 1053–1061. ACM, 2002.
5. P. Davidsson. Categories of artificial societies. In A. Omicini, P. Petta, and R. Tolksdorf, editors, *Engineering Societies in the Agents World II*, volume 2203 of *LNAI*, pages 1–9. Springer-Verlag, December 2001. 2nd International Workshop (ESAW'01), Prague, Czech Republic, 7 July 2001, Revised Papers.
6. C. Dellarocas and M. Klein. Civil agent societies: Tools for inventing open agent-mediated electronic marketplaces. In *Agent Mediated Electronic Commerce (IJCAI Workshop)*, pages 24–39, 1999.
7. T. Eiter, V.S. Subrahmanian, and G. Pick. Heterogeneous active agents, I: Semantics. *Artificial Intelligence*, 108(1-2):179–255, March 1999.
8. K. Eshgi and R. A. Kowalski. Abduction compared with negation by failure. In G. Levi and M. Martelli, editors, *Proceedings of the 6th International Conference on Logic Programming*, pages 234–255. MIT Press, 1989.
9. M. Esteva, J. A. Rodriguez-Aguilar, C. Sierra, P. Garcia, and J. L. Arcos. On the formal specification of electronic institutions. In F. Dignum and C. Sierra, editors, *Agent-mediated Electronic Commerce (The European AgentLink Perspective)*, number 1991 in LNAI, pages 126–147. Springer Verlag, 2001.
10. T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1-3):95–138, October 1998.
11. T. H. Fung and R. A. Kowalski. The IFF proof procedure for abductive logic programming. *Journal of Logic Programming*, 33(2):151–165, November 1997.
12. Global Computing: Co-operation of Autonomous and Mobile Entities in Dynamic Environments. `http://www.cordis.lu/ist/fetgc.htm`.

13. F. Guerin and J. Pitt. Proving properties of open agent systems. In C. Castelfranchi and W. Lewis Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2002), Part II, Bologna, Italy*, pages 557–558. ACM, 2002.
14. C. Hewitt. Open information systems semantics for distributed artificial intelligence. *Artificial Intelligence*, 47(1-3):79–106, 1991.
15. D. Hume. *An Enquiry Concerning Human Understanding*. 1748.
16. R. A. Kowalski and F. Sadri. From logic programming to multi-agent systems. *Annals of Mathematics and Artificial Intelligence*, 1999.
17. L. Lamport. What Good Is Temporal Logic? In R. E. A. Mason, editor, *Information Processing*, volume 83, pages 657–668. Elsevier Sciene Publishers, 1983.
18. H. Mazouzi, A. El Fallah Seghrouchni, and S. Haddad. Open protocol design fo complex interactions in multi-agent systems. In C. Castelfranchi and W. Lewis Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2002), Part I, Bologna, Italy*, pages 402–409. ACM, 2002.
19. Y. Moses and M. Tennenholtz. Artificial social systems. *Computers and AI*, 14(6):533–562, 1995.
20. J. Pitt and F. Guerin. Guaranteeing properties for e-commerce systems. Technical Report TRS020015, Department of Electrical and Electronic Engineering, Imperial College, London, UK, 2002.
21. D. L. Poole. A logical framework for default reasoning. *Artificial Intelligence*, 36(1):27–47, 1988.
22. F. Sadri, F. Toni, and P. Torroni. An abductive logic programming architecture for negotiating agents. In S. Greco and N. Leone, editors, *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA)*, volume 2424 of *LNCS*, pages 419–431. Springer Verlag, September 2002.
23. F. Sadri, F. Toni, and P. Torroni. Minimally intrusive negotiating agents for resource sharing. In G. Gottlob, editor, *Proceedings of the 18th International Joint Conference on Artificial Intelligence*. AAAI Press, August 2003. To appear.
24. M. P. Singh. A social semantics for agent communication languages. In F. Dignum and M. Greaves, editors, *Issues in Agent Communication*, pages 31–45. Springer-Verlag, Heidelberg, Germany, 2000.
25. SOCS: Societies Of ComputeeS (SOCS): a computational logic model for the description, analysis and verification of global and open societies of heterogeneous computees. `http://lia.deis.unibo.it/Research/SOCS/`.
26. P. Torroni, P. Mello, N. Maudet, M. Alberti, A. Ciampolini, E. Lamma, F. Sadri, and F. Toni. A logic-based approach to modeling interaction among computees (preliminary report). In *UK Multi-Agent Systems (UKMAS) Annual Conference, Liverpool, UK*, December 2002.
27. W. W. Vasconcelos. Logic-based electronic institutions. In this volume, pages 65–80, 2003.
28. M. Wooldridge, N. R. Jennings, and D. Kinny. The gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, September 2000.
29. P. Yolum and M.P. Singh. Flexible protocol specification and execution: applying event calculus planning using commitments. In C. Castelfranchi and W. Lewis Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2002), Part II, Bologna, Italy*, pages 527–534. ACM, 2002.

Linear Logic, Partial Deduction and Cooperative Problem Solving

Peep Küngas[1], Mihhail Matskin[2]

[1] Norwegian University of Science and Technology
Department of Computer and Information Science
peep@idi.ntnu.no
[2] Royal Institute of Technology
Department of Microelectronics and Information Technology
misha@imit.kth.se

Abstract. In this paper we present a model of cooperative problem solving (CPS). Linear Logic (LL) is used for encoding agents' initial states, goals and capabilities. LL theorem proving is applied by each agent to determine whether the particular agent is capable of solving the problem alone. If no individual solution can be constructed, then the agent may start negotiation with other agents in order to find a cooperative solution. Partial deduction in LL is used to derive a possible deal. Finally proofs are generated and plans are extracted from the proofs. The extracted plans determine agents' responsibilities in cooperative solutions.

1 Introduction

It is quite usual in multi-agent systems that an agent needs a help of another agent for performing its tasks and this situation has been considered in several works on cooperative problems solving (CPS). Several attempts were made in order to formalize CPS (see Section 5). Most of them are based on classical or modal logics. In particular, Wooldridge and Jennings [21] provide a formalisation of CPS process where a multi-modal logic is used as a formal specification language. However, since the multi-modal logic lacks a strategy for generating constructive proofs of satisfiability, the formalisation does not lead to direct execution of specifications. Moreover, since modal logics (like classical logic) lack the mechanism for keeping track of resources, it is not possible for agents neither to *count* nor dynamically update the number of instances of the same object belonging to their internal states.

In order to overcome the mentioned shortages of classical and modal logics we use a fragment of Linear Logic [6] (LL) for CPS. LL provides a mechanism for keeping track of resources (in LL one instance of a formula is distinguished from two or more instances of the formulae) and this makes possible more natural representation of dynamical processes and agents' internal states.

The cooperative problem solving has been considered to consist of four steps [21]— recognition of potential for cooperation, team formation, plan formation and plan execution.

An important feature of our methodology is that we do not separate team and plan formation into different processes and that negotiation is already embedded into the reasoning. Although this approach does not preserve the accepted structure of CPS, we think that it may be more natural for representing computational aspects of CPS, where team and plan formation processes interact with eachother.

Basically, we are applying LL theorem proving for generating a constructive proof representing the first 3 steps of CPS: recognition, team and plan formation. Negotiation is reformulated as distributed proof search. Then a solution, summarising the first 3 steps of CPS process, is extracted from a proof and executed.

In CPS models it is often implicitly expected that agents have knowledge about sequences of actions, whose executions lead agents to their goals, while the sequence construction process is not explicitly explained. Our CPS model is more planning-centric. First an agent is trying to find a plan that achieves its goals. Doing this, the agent may discover that either this is not possible or it is more efficient to involve other agents into problem solving process. However, since other agents may be self-interested, they may propose their offers and start a negotiation process. The process lasts until a (shared) plan has been found. The plan determines agents commitments and takes into account requirements determined during the negotiation.

In order to stimulate cooperation, agents should have a common goal [21]. We assume that all agents have a common *meta*-goal: as much agents as possible should become satisfied during run-time. All agents ask for minimum they need and provide maximum they can, during negotiation. This is biased with distributed theorem proving strategies. During negotiation the offers are derived using Partial Deduction (PD) for LL. PD allows determining missing links between proof fragments.

The rest of the paper is organised as follows. In Section 2 we present a general model of distributed problem solving and illustrate it with a working example. Section 3 gives an introduction to LL and PD. Section 4 proceeds with the working example from Section 2 by applying LL theorem proving and PD for CPS. Section 5 reviews related work and Section 6 concludes the article.

2 General CPS model and a working example

For further usage we define an agent $\mathcal{A}_i$ as a triple

$$\mathcal{A}_i = (\Gamma_i, S_i, G_i),$$

where:

- Γ_i represents a set of agent's capabilities in form $Delete \mapsto Add$. $Delete$ and Add are sets of formulae, which are respectively deleted from and added to the agent's current state S_i, if the particular capability is applied. The capability can be applied only if $Delete \subseteq S_i$.

– S_i is a set of literals representing the current state of the agent.
– G_i is a set of literals representing the goal state of the agent.

By agent's capabilities we mean actions that an agent can perform. While performing a capability $X \mapsto Y$, an agent consumes resources denoted by X and generates resources referred with Y. The current state of an agent reflects agent's perception of the current world together with agent's internal state. The word "state" instead of "beliefs" is used herein to emphasise that our approach in planning-centric. Moreover, it expresses explicitly that our approach is not related much to BDI theory. We write S_i' and G_i' to denote modification of S_i and G_i respectively. Similarly we write S_i'' and G_i'' to denote modifications of S_i' and G_i', and so forth. While representing states and goals of agents we write A^n to specify that there are n instances of objects A in a particular state or a goal.

While planning, agents may discover that a plan, whose execution would lead them from state S_i to goal G_i, could not be found. Then they determine subproblems (missing links), which cannot be solved by themselves. A missing link with respect to S_i and G_i is a pair (S_i', G_i'), which is achieved by applying agent's capabilities to its initial state S_i in forward chaining and to its goal G_i in backward chaining manner. Thus $S_i \longmapsto_* S_i'$ and $G_i' \longmapsto_* G_i$, where $\longmapsto$ represents application of a capability from Γ_i and $\longmapsto_*$ denotes that 0 or more capabilities are applied in sequence.

For illustrating detection of missing links let us consider the case where $S_1 = \{A\}$, $G_1 = \{D\}$ and $\Gamma_1 = \{A \mapsto B, C \mapsto D\}$. Then possible missing links are $(\{B\},\{C\})$, $(\{B\},\{D\})$, $(\{A\},\{C\})$ and $(\{A\},\{D\})$. If, for example, agent $\mathcal{A}_1$ decides that the most relevant missing link is $(\{B\},\{C\})$, it sends a message with $S_1' = \{B\}$, $G_1' = \{C\}$ to another agent $\mathcal{A}_2$. Additionally $\mathcal{A}_1$ may send a list of its capabilities Γ_1 (or a fragment of Γ_1) that might help $\mathcal{A}_2$ in reasoning about possible offers.

Our general CPS model is depicted in Figure 1. Communication between agents goes via a Communication Adapter (CA), whose purpose is:

– to provide translations between agents' communication languages and
– to change the viewpoint of proposals

Usage of Communication Adapter will be explained in details in Section 3.5. Our general CPS model can be described as follows:

1. Agent $\mathcal{A}_1$ tries to generate a plan from elements of Γ_1, such that execution of the plan would lead $\mathcal{A}_1$ from state S_1 to G_1. However, $\mathcal{A}_1$ fails to construct such a solution.
2. Agent $\mathcal{A}_1$ identifies possible missing links for achieving a complete solution. Every missing link derived from S_1 and G_1 is presented with a pair (S_1', G_1').
3. Agent $\mathcal{A}_1$ delivers missing links to other agents and asks them to solve the missing links. If no answers are received, the problem is not solvable.
4. Agent $\mathcal{A}_2$ agrees to help $\mathcal{A}_1$.
5. If $\mathcal{A}_2$ finds a complete solution for link (S_1', G_1'), then S_1'' and G_1'' will be just copies of S_1' and G_1' respectively and P is a complete solution. Otherwise,

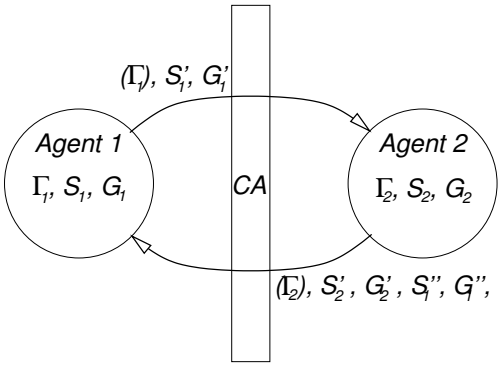

Fig. 1. General CPS model.

S_1'' and G_1'' are new missing links based on S_1' and G_1' and P is a partial solution to the initial problem.

6. $\mathcal{A}_2$ delivers (S_1'', G_1'') and P back to $\mathcal{A}_1$.
7. Additionally, $\mathcal{A}_2$ may require that $\mathcal{A}_1$ helps to solve a missing link (G_2', S_2') in return of a solution P and a (new) missing link (S_1'', G_1''). The link (G_2', S_2') is achieved through application of CA to link (S_2', G_2').
8. Agent $\mathcal{A}_2$ may send the new missing link (S_1'', G_1'') further to other agents as well for solving. In that case $\mathcal{A}_2$ acts as a mediator agent.

Negotiation proceeds until the set of possible missing links will be exhausted or a solution satisfying both agents is found. As an extreme case all agents may be explicitly aware of each other capabilities, beliefs and goals a priori. In this case less messages are sent during negotiation, however, agents' privacy, in terms of hiding their internal state, goal and capabilities, is less supported. In general, it is also possible that an agent is involved into negotiation with several agents simultaneously and at each step selects the best available offer. While implementing the presented model we apply LL theorem proving for planning and PD for determining subtasks, which have to be solved by other agents. The iterative CPS process is depicted in Figure 2.

Taking into account the presented general model, our working example is described as follows. Let us assume that two students, John and Peter, are looking for ways to relax after long days of studying and a final successful examination. John has a CD and he wants to listen music ($G_{John} = \{Music\}$). Unfortunately, his CD player is broken and this makes his goal non-realistic. John has also decided to visit a library to return books and this gives him possibility to return also books of other students when this may be useful for him. For covering all his expenses, related to relaxing, John has 10 USD. Considering that he has a broken CD player and a CD his initial state is as follows $S_{John} = \{Dollar^{10}, CD, BrokenCDPlayer\}$ and his capabilities are $\Gamma_{John} = \{Books \mapsto BooksReturned, CDPlayer\&CD \mapsto Music\}$.

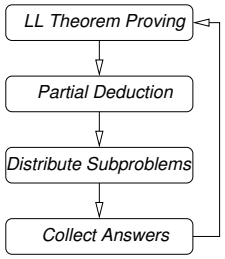

Fig. 2. Iterative CPS process.

Peter is skilled in electronics and can repair the CD player. He has decided to spend his day in a park with his girlfriend. However, he has to return books to the library. Since he has to take a taxi to reach the library, he has to spend 10 USD to cover his transportation expenses. This does not match well with his goals, since he has only 15 USD while he needs 25 USD for food, drinks and attractions in the park. Therefore he lacks 20 USD to achieve his goals. Peter's initial state, goal and capabilities are described as follows: $S_{Peter} = \{Dollar^{15}, Books\}$, $G_{Peter} = \{BooksReturned, Beer\}$ and $\Gamma_{Peter} = \{Dollar^{10} \& Books \mapsto BooksReturned, BrokenCDPlayer \mapsto CDPlayer, Dollar^{25} \mapsto Beer\}$.

Taking into account Γ_{John}, G_{John} and S_{John}, John constructs new state and goal and asks Peter whether he can repair his CD player: $S'_{John} = \{BrokenCDPlayer\}$, $G'_{John} = \{CDPlayer\}$. Peter decides to take advantage of the situation. He agrees to repair the CD player and asks 20 USD for performing this task: $S'_{Peter} = \{\}$, $G'_{Peter} = \{Dollar^{20}\}$, $G''_{John} = G'_{John}$ and $S''_{John} = S'_{John}$.

However, John has only 10 USD. Therefore he discloses additional information about his capabilities to Peter $\Gamma''_{John} = \{Books \mapsto BooksReturned\}$. Peter discovers that John has decided to visit a university library and agrees to decrease a fee for repairing the CD player by 10 USD, if John delivers his books to the library. John agrees and negotiation is successfully finished. During the negotiation agents' plans have also been determined.

3 Formalisation of CPS process

3.1 Linear logic

LL is a refinement of classical logic introduced by J.-Y. Girard to provide means for keeping track of "resources". In LL two assumptions of a propositional constant A are distinguished from a single assumption of A. This does not apply in classical logic, since there the truth value of a fact does not depend on the number of copies of the fact. Indeed, LL is not about truth, it is about computation.

Although LL is not the first attempt to develop resource-oriented logics, it is by now the most investigated one. Since its introduction LL has enjoyed increasing attention both from proof theorists and computer scientists. Therefore,

because of its maturity, LL is useful as a declarative language and an inference kernel.

In following we are considering intuitionistic multiplicative additive fragment of LL (IMALL or MAILL) consisting of multiplicative conjunction ($\otimes$), additive disjunction ($\oplus$), additive conjunction ($\&$) and linear implication ($\multimap$). In terms of resource acquisition the logical expression $A \otimes B \vdash C \otimes D$ means that resources C and D are obtainable only if both A and B are obtainable. After the sequent has been applied, A and B are consumed and C and D are generated.

The expression $A \vdash B \oplus C$ in contrary means that, if we have resource A, we can obtain either B or C, but we do not know which one of those. The expression $A \& B \vdash C$ on the other hand means that while having resources A and B we can choose, which one of them to trade for C. Therefore it is said that $\oplus$ and $\&$ are representing *external* and *internal* choice.

To illustrate preceding let us consider the following LL sequent from [14]—$(D \otimes D \otimes D \otimes D \otimes D) \vdash (H \otimes C \otimes (O \& S) \otimes !F \otimes (P \oplus I))$, which encodes a fixed price menu in a fast-food restaurant: for 5 dollars (D) you can get an hamburger (H), a coke (C), either onion soup O or salad S depending, which one *you* select, all the french fries (F) you can eat plus a pie (P) or an ice cream (I) depending on availability (restaurant owner selects for you). The formula $!F$ here means that we can use or generate a resource F as much as we want—the amount of the resource is unbounded.

To increase the expressiveness of formulae, we are using in the following sometime abbreviation $a^n = \underbrace{a \otimes \ldots \otimes a}_{n}$, for $n \geq 0$.

Lincoln [15] summarises complexity results for several fragments of LL. Propositional MALL is indicated to be PSPACE-complete, whilst first-order MALL is at most NEXPTIME-hard. If we would discard additives $\oplus$ and $\&$ from MALL, we would get multiplicative LL (MLL). Both, propositional and first-order MLL, are NP-complete. It is also identified that these complexity results do not change, if intuitionistic fragments of LL are considered. These results hint that for practical computations either MLL or propositional MALL (or their intuitionistic variants MILL and MAILL (IMALL), respectively) could be used. The complete set of IMALL inference rules is given in Appendix A.

3.2 Partial deduction and LL

Partial deduction (PD) (or partial evaluation of logic programs first introduced in [11]) is known as one of optimisation techniques in logic programming. Given a logic program, partial deduction derives a more specific program while preserving the meaning of the original program. Since the program is more specialised, it is usually more efficient than the original program, if executed. For instance, let A, B, C and D be propositional variables and $A \multimap B$, $B \multimap C$ and $C \multimap D$ computability statements in LL. Then possible partial deductions are $A \multimap C$, $B \multimap D$ and $A \multimap D$. It is easy to notice that the first corresponds to forward chaining (from beliefs to goals), the second to backward chaining (from goals to beliefs) and the third could be either forward or backward chaining.

Lloyd and Shepherdson [16] formalised PD for normal logic programs and showed its correctness with respect to Clark's program completion semantics. Partial deduction in logic programming is often defined as unfolding of program clauses.

Although the original motivation behind PD was to deduce specialised logic programs with respect to a given goal, our motivation for PD is a bit different. We are applying PD for determining subtasks, which cannot be performed by a single agent, but still are possibly closer to a solution than an initial task. This means that given a state S and a goal G of an agent we compute a new state S' and a new goal G'. This information is forwarded to another agent for further inference. Similar approach has been applied by Matskin and Komorowski [17] in automatic software synthesis. One of their motivations was debugging of declarative software specification.

The main problem with PD in LL is that although new derived states and goals are sound with respect to an initial specification, they may not preserve completeness anymore. This is due to resource-consciousness of LL—if a wrong proof branch is followed, initial beliefs may be consumed and thus further search become more limited. Therefore agents have to search, in the worst case, all possible PDs of initial specification to preserve completeness of distributed search mechanism. In [16] completeness and soundness issues of PD are considered for classical logic programs. Issues of complexity, completeness and soundness of PD in LL will be considered within another paper.

The following inference figures are applied as PD steps for back- and forward chaining:

Backward chaining step $\mathcal{R}_b(L_i)$:

$$\dfrac{\dfrac{\dfrac{}{S \vdash A \otimes C}\ Axiom \quad \dfrac{}{\vdash (A \multimap_{L_i} B)}\ Axiom}{S \vdash A \otimes C \otimes (A \multimap_{L_i} B)}\ R\otimes \quad \dfrac{\dfrac{\dfrac{}{C \vdash C}\ Id \quad \dfrac{\dfrac{\dfrac{}{A \vdash A}\ Id \quad \dfrac{}{B \vdash B}\ Id}{A, (A \multimap_{L_i} B) \vdash B}\ L\multimap}{A \otimes (A \multimap_{L_i} B) \vdash B}\ L\otimes}{C, A \otimes (A \multimap_{L_i} B) \vdash B \otimes C}\ R\otimes}{A \otimes C \otimes (A \multimap_{L_i} B) \vdash B \otimes C}\ L\otimes}{S \vdash B \otimes C}\ Cut$$

Forward chaining step $\mathcal{R}_f(L_i)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{B \otimes C \vdash B \otimes C}\ Id \quad \dfrac{}{\vdash (B \multimap_{L_i} A)}\ Axiom}{B \otimes C \vdash B \otimes C \otimes (B \multimap_{L_i} A)}\ R\otimes \quad \dfrac{\dfrac{\dfrac{\dfrac{}{C \vdash C}\ Id \quad \dfrac{\dfrac{\dfrac{}{B \vdash B}\ Id \quad \dfrac{}{A \vdash A}\ Id}{B, (B \multimap_{L_i} A) \vdash A}\ L\multimap}{B \otimes (B \multimap_{L_i} A) \vdash A}\ L\otimes}{C, B \otimes (B \multimap_{L_i} A) \vdash A \otimes C}\ R\otimes}{B \otimes C \otimes (B \multimap_{L_i} A) \vdash A \otimes C}\ L\otimes \quad \dfrac{}{A \otimes C \vdash G}}{B \otimes C \otimes (B \multimap_{L_i} A) \vdash G}\ Cut}{B \otimes C \vdash G}\ Cut$$

L_i in previous inference figures is a labelling for a particular LL axiom representing agent's capability. PD steps $\mathcal{R}_f(L_i)$ and $\mathcal{R}_b(L_i)$, respectively, apply clause L_i to move the initial state towards the goal state or vice versa. A, B and C are multiplicative conjunctions. In $\mathcal{R}_b(L_i)$ inference figure formulae $B \otimes C$

and $A \otimes C$ denote respectively G and G'. Thus the inference figure encodes that, if there is an extralogical axiom $\vdash A \multimap B$, then we can change goal $B \otimes C$ to $A \otimes C$. Analogously, in the inference figure $\mathcal{R}_f(L_i)$ formulae $B \otimes C$ and $A \otimes C$ denote respectively S and S'. And the inference figure encodes that, if there is an extralogical axiom $\vdash B \multimap A$, then we can change initial state $B \otimes C$ to $A \otimes C$.

3.3 Agents in LL

An agent is presented with the following LL sequent:

$$\Gamma; S \vdash G,$$

where Γ is a set of extralogical LL axioms representing agent's capabilities, S is the initial state and G the goal state of an agent. Both S and G are multiplicative conjunctions of literals. Every element of Γ is in form

$$\vdash I \multimap O,$$

whereas I and O are multiplicative conjunctions of formulae, which are respectively consumed and generated when a particular capability is applied. It has to be noted that a capability can be applied only, if conjuncts in I form a subset of conjuncts in S.

3.4 Encoding offers in LL

Harland and Winikoff [8] presented the first ideas about applying LL theorem proving for agent negotiation. The main advantages of LL over classical logic is its resource-consciousness and existence two kinds of nondeterminism. Both internal and external nondeterminism in negotiation rules can be represented. In the case of internal nondeterminism a choice is made by resource provider, whereas in the case of external nondeterminism a choice is made by resource consumer. For instance, formula $Dollar^5 \multimap Beer \oplus Soda$ means that an agent can provide either some *Beer* or *Soda* in return for 5 dollars, but the choice is made by the provider agent. The consumer agent has to be ready to obtain either a beer or a soda. The formula $Dollar \multimap Tobacco \& Lighter$ in contrary means that the consumer may select which resource, *Tobacco* or *Lighter*, s/he gets for a *Dollar*.

In the context of negotiations, operators & and $\oplus$ have symmetrical meanings—what is $A \oplus B$ for one agent, is $A \& B$ to her/his partner. It means that if one agent gives to another an opportunity to choose between A and B, then the former agent has to be ready for providing both choices, A and B. When initial resources owned by agents and expected negotiation results have been specified, LL theorem proving is used for determining the negotiation process.

We augment the ideas of Harland and Winikoff by allowing trading also services (agent capabilities) for resources and vice versa. This is a step further to the world where agents not only exchange resources, but also work for other agents in order to achieve their own goals. We write $A \vdash B \multimap C$ to indicate that an agent can trade resource A for a service $B \multimap C$.

3.5 Communication Adapter

In [18] bridge rules are used for translating formulae from one logic into another, if agents exchange offers. We adopt this idea for Communication Adapter (CA) for two reasons. First, it would allow us to encapsulate agents' internal state and second, while offers are delivered from one agent to another, viewpoint to the offer is changing and internal and external choices are reversed. By viewpoint we mean agent's role, which can be either receiver or sender of an offer.

The *CA* rule is described in the following way. As long as formulae on the left and the right hand side of sequents consist of only $\otimes$ and $\multimap$ operators, the left and the right hand sides of sequents are inversed. However, if formulae contain disjunctions, their types have to be inversed. This has to be done because there are 2 disjunctions in LL—one with internal and another with external choice. Since internal and external choices are context-dependent, they have to be inversed, if changing viewpoints. For instance, sequent $A \otimes (A \multimap B) \vdash C \oplus D$ is translated to $C \& D \vdash A \otimes (A \multimap B)$ through the *CA* rule:

$$\frac{\mathop{\&}_j B_j \vdash \bigoplus_i A_i}{\mathop{\&}_i A_i \vdash \bigoplus_j B_j}\ CA$$

In the *CA* rule A and B consist of multiplicative conjunctions and linear implications. We allow & and $\oplus$ only in the left and the right hand side of a sequent, respectively. Due to LL rules $R\&$ and $L\oplus$ the following conversions are allowed:

$$D \vdash \mathop{\&}_j D_j \Longrightarrow \bigcup_j (D \vdash D_j)$$

$$\bigoplus_j D_j \vdash D \Longrightarrow \bigcup_j (D_j \vdash D)$$

Therefore we do not lose anything in expressive power of LL, when limiting where disjunctions may occur, but proposals are kept declaratively more understandable.

Although our bridge rule is intended for agents reasoning in LL only, additional bridge rules may be constructed for communication with other non-LL agents. However, it should be mentioned that there is no one-to-one translation between most of logics and therefore information loss may occur during translation.

4 Running example

Let us consider encoding of the working example from Section 2 in LL and explain how to represent agents' capabilities (actions they can perform), negotiation arguments, agents' states and goals with LL formulae. In order to keep the

resulting proof simple, we take advantage of propositional MILL (a fragment of MAILL) only.

The initial scenario is described formally as follows:

$$\Gamma_{John} = \begin{array}{l} \vdash_{John} Books \multimap_{returnBooks} BooksReturned \\ \vdash_{John} CDPlayer \otimes CD \multimap_{playMusic} Music \end{array}$$

$$\Gamma_{Peter} = \begin{array}{l} \vdash_{Peter} Dollar^{10} \otimes Books \multimap_{returnBooks} BooksReturned \\ \vdash_{Peter} BrokenCDPlayer \multimap_{repairCDPlayer} CDPlayer \\ \vdash_{Peter} Dollar^{25} \multimap_{buyBeer} Beer \end{array}$$

The sets of extralogical axioms Γ_{John} and Γ_{Peter} represent capabilities of John and Peter, respectively. We write $\vdash_X$ to indicate that a capability is provided by X, $\multimap_Y$ labels a capability with name Y. The internal state of John is described by the following sequent:

$$\Gamma_{John}; Dollar^{10} \otimes CD \otimes BrokenCDPlayer \vdash_{John} Music$$

This means that John has 10 USD, a CD and a broken CD player. His goal is to listen music. Peter's state and goal are described by another sequent:

$$\Gamma_{Peter}; Dollar^{15} \otimes Books \vdash_{Peter} BooksReturned \otimes Beer$$

We write B, BR, BE, CD, P, BP, M and D to denote *Books*, *BooksReturned*, *Beer*, *CD*, *CDPlayer*, *BrokenCDPlayer*, *Music* and *Dollar* respectively. Given John's and Peter's capabilities and internal states, both agents start individually with theorem proving. Initially they fail, since they are unable to reach their goals individually. Then PD in LL is applied to the same set of formulae and new subtasks are derived. These subtasks indicate problems, which could not be solved by agents themselves and need cooperation with other agents. In particular, John has to ask help for solving the following sequent, which is derived by PD:

$$D^{10} \vdash_{John} BP \multimap P$$

The sequent is achieved by applying the backward chaining step of PD in the following way (to allow shorter proof, we write here $\vdash$ instead of $\vdash_{John}$):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{CD \vdash CD}\ Id \quad \dfrac{D^{10} \vdash_{John} BP \multimap P}{D^{10} \otimes BP \vdash P}\ Shift}{CD, D^{10} \otimes BP \vdash CD \otimes P}\ R\otimes}{D^{10} \otimes BP \otimes CD \vdash CD \otimes P}\ L\otimes}{D^{10} \otimes BP \otimes CD \vdash_{John} M}\ \mathcal{R}_b(playMusic)$$

where *Shift* is another inference figure:

$$\dfrac{\dfrac{}{C \vdash A \multimap B}\ Axiom \quad \dfrac{\dfrac{}{A \vdash A}\ Id \quad \dfrac{}{B \vdash B}\ Id}{A, A \multimap B \vdash B}\ L\multimap}{C \otimes A \vdash B}\ Cut$$

The purpose of the inference figure above, is to transform sequents of the form $C \otimes A \vdash B$ to the form $C \vdash A \multimap B$, which is our representation sequent for agents' capabilities. It has to be mentioned that C is possibly an empty formula.

Peter sees the offer as follows:

$$BP \multimap P \vdash_{Peter} D^{10}$$

Changing a viewpoint for offers is done by Communication Adapter (CA, see Section 3.5).

Since Peter can repair the CD player, he agrees partially with the proposal. However, because he needs 20 USD instead of the proposed 10 USD, he increases the number of dollars required. In addition, assuming that John may not have the asked amount of money, Peter provides him with information about tasks (Γ_{Peter}), which may be traded for money:

$$\Gamma_{Peter}; BP \multimap P \vdash_{Peter} D^{20}$$

This is viewed by John as follows:

$$\Gamma_{Peter}; D^{20} \vdash_{John} BP \multimap P$$

By exploring Γ_{Peter} John discovers that both his and John's goals would be reached, if he delivers Peters' books to the library for 10 USD and Peter repairs the CD player for 20 USD. Peter accepts the offer.

Another possible solution could be that Peter repairs the CD player for 10 USD and John delivers his books. However, while the former solution makes it possible to introduce binding contracts between agents, the latter expects that both agents trust each other. Depending on the proof strategy, we can represent different solutions to cooperation problem solving.

The final proposal from John is presented as the following sequent:

$$\Gamma_{Peter}; D^{20} \otimes (D^{10} \otimes B \multimap BR) \vdash_{John} BP \multimap P$$

After applying CA it is perceived by Peter as follows:

$$\Gamma_{Peter}; BP \multimap P \vdash_{Peter} D^{20} \otimes (D^{10} \otimes B \multimap BR)$$

During communication agents can send information about their capabilities (Γ_{Peter} in the current case). This provides their partners with hints about possible counteroffers. The following proof is constructed for John during the above-described series of message exchange (we omit Γ_{John} and Γ_{Peter} in the proof for simplicity):

$$\dfrac{\dfrac{}{BP \vdash BP}\,Id \qquad \dfrac{\dfrac{\dfrac{}{D^{10} \vdash D^{10}}\,Id \qquad \dfrac{returnBooks}{\vdash B \multimap BR}}{D^{10} \vdash D^{10} \otimes (B \multimap BR)}\,R\otimes \qquad \dfrac{\dfrac{\dfrac{}{D^{10} \vdash D^{10}}\,Id \qquad \dfrac{returnBooks_{Peter}}{B \multimap BR \vdash D^{10}}}{D^{10}, (B \multimap BR) \vdash D^{20}}\,R\otimes}{D^{10} \otimes (B \multimap BR) \vdash D^{20}}\,L\otimes}{D^{10} \vdash D^{20}}\,Cut}{D^{10}, BP \vdash D^{20} \otimes BP}\,R\otimes$$

$$\vdots$$

$$\dfrac{\dfrac{}{CD \vdash CD}\,Id \qquad \dfrac{\dfrac{\dfrac{\vdots}{D^{10}, BP \vdash D^{20} \otimes BP}}{D^{10} \otimes BP \vdash D^{20} \otimes BP}\,L\otimes \qquad \dfrac{repairCDPlayer}{D^{20} \otimes BP \vdash P}\,Shift}{D^{10} \otimes BP \vdash P}\,Cut}{\dfrac{D^{10} \otimes BP, CD \vdash P \otimes CD}{D^{10} \otimes BP \otimes CD \vdash P \otimes CD}\,L\otimes}\,R\otimes$$

$$\dfrac{D^{10} \otimes BP \otimes CD \vdash P \otimes CD \qquad \dfrac{playMusic}{P \otimes CD \vdash M}\,Shift}{D^{10} \otimes BP \otimes CD \vdash M}\,Cut$$

John's plan can be extracted from the proof and then executed. In the proof action *returnBooks* occurs twice. While *returnBooks* means that John has to execute it, $returnBooks_{Peter}$ explicitly represents that Peter has to be charged with 10 dollars for performing previous action. A proof for Peter can be constructed in a similar way.

The proof above does not present the whole negotiation process. It rather demonstrates the *result* of negotiation and team and plan formation. The proof also indicates how John and Peter have joined the efforts to achieve their goals. The dependency of plans for John and Peter is shown in Figure 3.

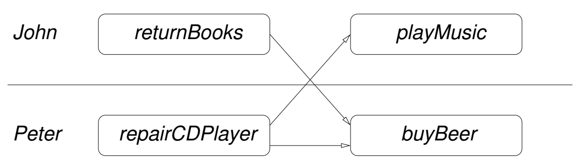

Fig. 3. Interdependent plans.

Arrows in Figure 3 indicate a partial order of actions to be executed (or capabilities applied) and the line separates actions performed by different agents (actions above the line are executed by John and actions below the line are executed by Peter).

5 Related work

As it has been indicated in [10] negotiation is the most fundamental and powerful mechanism for managing inter-agent dependencies at run-time. Negotiation may be required both for self-interested and cooperative agents. It allows to reach a mutually acceptable agreement on some matter by a group of agents.

Kraus et al [12] give a logical description for negotiation via argumentation for BDI agents. They classify arguments as threats and promises, which are identified as most common arguments in human negotiations. In our case only promises are considered, since in order to figure out possible threats to goals of particular agents, agents' beliefs, goals and capabilities should be known in

advance to the persuader. We assume, that our agents do not explicitly communicate about their internal state.

Fisher [4] introduced the idea of distributed theorem proving in classical logic as agent negotiation. In his approach all agents share the common view to the world and if a new clause is inferred, all agents would sense it. Inferred clauses are distributed among agents via broadcasting. Then, considering the received information, agents infer new clauses and broadcast them further again. Although agents have a common knowledge about inferred clauses, they may have different sets of inference rules. Distribution of a collection of rules between agents means that different agents may have different capabilities and make different inferences. The latter implies that different agents contribute to different phases of proof search.

Parsons et al [18] defined negotiation as interleaved formal reasoning and arguing. Arguments and counterarguments are derived using theorem proving while taking into consideration agents' own goals. While Parsons et al [18] perform reasoning in classical logic, it is possible to infer missing clauses needed for achieving a goal. The situation gets more complicated, if several instances of a formulae are available and moreover, the actions performed by agents or resources they spend can be interdependent. Thereby, inference in LL is not so straightforward, since some clauses are "consumed" while inferring others. Due to the aforementioned reasons we apply PD to determine missing parts of a proof. Then a missing part is announced to other possibly interested agents.

Case-based planning has been used for coordinating agent teams in [5]. The planner generates a so called shared mental model of the team plan. Then all agents adapt their plans to the team plan. This work is influenced by the joint intentions [13, 3] and shared plans [7] theory.

In [19] agent coordination is performed through task agents by planning. First problem solving goals are raised and then solutions satisfying these goals are computed. Finally these plans are decomposed and coordinated with appropriate task or other agents for plan execution, monitoring and result collection. Other agents are information and interface agents, for information collection and interfacing with a human user, respectively.

The joint intentions theory [3] determines the means how agents should act to fulfill joint goals, when they should exchange messages, synchronise between themselves, leave the team, etc. It also determines when the joint goal is considered to be achieved or when and how to break up commitment to it, if it should, for instance, turn out that one agent is not able anymore to perform its task(s). Decision making about whether a goal has been achieved, is not achievable or there is no need to achieve it anymore, is based on consensus—every agent can initiate a discussion through which consensus is (presumably) achieved. Then everybody acts as stated by the consensus. However, it is not stated how joint goals are formed through negotiation or other processes.

One of the first formalisations of cooperative problem solving is given by Wooldridge and Jennings [21] (also other approaches presented so far are reviewed there). One of the earliest *implemented* general models of teamwork is

described in [20], which is based of joint intentions theory and partially borrows from shared plans theory.

In [1] LL has been used for prototyping multi-agent systems at conceptual level. Because of fixed semantics of LL, it is possible to verify whether a system functions at conceptual level as intended. Although the prototype LL program is executable, it is still too high level to produce a final agent-based software. Thus another logic programming language is embedded to compose the final software.

Harland and Winikoff [9] address the question how to integrate both proactive and reactive properties of agents into LL programming framework. There forward chaining is used to model the reactive behaviour of an agent and backward chaining for the proactive behaviour respectively. That sort of computation is called there mixed mode computation, because both, forward and backward chaining are allowed.

6 Conclusions

In contrast to other formalisations of CPS, we presented a computational model, where the main emphasis is placed on negotiation and planning. Need for co-operation is sensed, if an agent alone is unable to find a plan for achieving its goals. Agent teams are formed on basis of interdependent plans.

For formalising our general CPS model we chose LL. Planning is implemented on top of LL theorem proving. Initially a planning problem is coded in terms of LL sequents and theorem proving is applied for generating constructive proofs. From generated proofs then plans are extracted. We specified negotiation as distributed LL theorem, whereas PD is used for generating offers. Offers are generally in form—I can grant you X if you provide me with Y, where X and Y can be both resources and capabilities of agents. X can be empty.

We have implemented a planner on top of theorem prover for first-order MILL and performed initial experiments. The planner is available at address `http://www.idi.ntnu.no/~peep/RAPS`.

Acknowledgements

This work is partially supported by the Norwegian Research Foundation in the framework of the Information and Communication Technology (IKT-2010) program—the ADIS project. We would like to thank the anonymous referees for their useful comments.

References

1. M. Bozzano, G. Delzanno, M. Martelli, V. Mascardi, F. Zini. Logic Programming & Multi-Agent Systems: a Synergic Combination for Applications and Semantics. In The Logic Programming Paradigm: a 25-Year Perspective, pp. 5–32, Springer-Verlag, 1999.

2. P. R. Cohen, H. J. Levesque. Intention is Choice with Commitment. Artificial Intelligence, Vol. 42, No. 3, pp. 213–261, 1990.
3. P. R. Cohen, H. J. Levesque. Teamwork. Nous, Vol. 25, No. 4, pp. 487–512, 1991.
4. M. Fisher. Characterising Simple Negotiation as Distributed Agent-Based Theorem-Proving—A Preliminary Report. In Proceedings of the Fourth International Conference on Multi-Agent Systems, Boston, July 2000, IEEE Press.
5. J. A. Giampapa, K. Sycara. Conversational Case-Based Planning for Agent Team Coordination. In D. W. Aha, I. Watson (eds). Case-Based Reasoning Research and Development: Proceedings of the Fourth International Conference on Case-Based Reasoning, ICCBR 2001, July 2001, Lecture Notes in Artificial Intelligence, Vol. 2080, pp. 189–203, Springer-Verlag, 2001.
6. J.-Y. Girard. Linear Logic. Theoretical Computer Science, Vol. 50, pp. 1–102, 1987.
7. B. Grosz, S. Kraus. Collaborative Plans for Complex Group Actions. Artificial Intelligence, Vol. 86, pp. 269–357, 1996.
8. J. Harland, M. Winikoff. Agent Negotiation as Proof Search in Linear Logic. In Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2002), July 15–19, 2002, Bologna, Italy.
9. J. Harland, M. Winikoff. Language Design Issues for Agents based on Linear Logic. In Proceedings of the Workshop on Computational Logic in Multi-Agent Systems (CLIMA'02), August 2002.
10. N. R. Jennings, P. Faratin, A. R. Lomuscio, S. Parsons, C. Sierra, M. Wooldridge. Automated Negotiation: Prospects, Methods and Challenges, International Journal of Group Decision and Negotiation, Vol. 10, No. 2, pp. 199–215, 2001.
11. J. Komorowski. A Specification of An Abstract Prolog Machine and Its Application to Partial Evaluation. PhD thesis, Technical Report LSST 69, Department of Computer and Information Science, Linkoping University, Linkoping, Sweden, 1981.
12. S. Kraus, K. Sycara, A. Evenchik. Reaching Agreements through Argumentation: A Logical Model and Implementation. Artificial Intelligence, Vol. 104, No. 1–2, pp. 1–69, 1998.
13. H. J. Levesque, P. R. Cohen, J. H. T. Nunes. On Acting Together. In Proceedings of the Eighth National Conference on Artificial Intelligence, AAAI-90, pp. 94–99, 1990.
14. P. Lincoln. Linear Logic. ACM SIGACT Notices, Vol. 23, No. 2, pp. 29–37, Spring 1992.
15. P. Lincoln. Deciding Provability of Linear Logic Formulas. In J.-Y. Girard, Y. Lafont, L. Regnier (eds). Advances in Linear Logic, London Mathematical Society Lecture Note Series, Vol. 222, pp. 109–122, 1995.
16. J. W. Lloyd, J. C. Shepherdson. Partial Evaluation in Logic Programming. Journal of Logic Programming, Vol. 11, pp. 217–242, 1991.
17. M. Matskin, J. Komorowski. Partial Structural Synthesis of Programs. Fundamenta Informaticae, Vol. 30, pp.23–41, 1997.
18. S. Parsons, C. Sierra, N. Jennings. Agents that Reason and Negotiate by Arguing. Journal of Logic and Computation, Vol. 8, No. 3, pp. 261–292, 1998.
19. K. Sycara,D. Zeng. Coordination of Multiple Intelligent Software Agents. International Journal of Intelligent and Cooperative Information Systems, Vol. 5, No. 2–3, pp. 181–211, 1996.
20. M. Tambe. Towards Flexible Teamwork. Journal of Artificial Intelligence Research, Vol. 7, pp. 83–124, 1997.
21. M. Wooldridge, N. R. Jennings. The Cooperative Problem-Solving Process. Journal of Logic and Computation, Vol. 9, No. 4, pp. 563–592, 1999.

A IMALL rules

Logical axiom and Cut rule:

$$A \vdash A \ \ (Axiom) \qquad \frac{\Gamma \vdash A, \Delta \quad \Gamma', A \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'} \ (Cut)$$

Rules for the propositional constants:

$$\vdash 1 \qquad \frac{\Gamma \vdash A}{\Gamma, 1 \vdash A}$$

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \otimes B \vdash \Delta} \ (L\otimes) \qquad \frac{\Gamma \vdash A, \Delta \quad \Gamma' \vdash B, \Delta'}{\Gamma, \Gamma' \vdash A \otimes B, \Delta, \Delta'} \ (R\otimes)$$

$$\frac{\Sigma_1 \vdash A \quad B, \Sigma_2 \vdash C}{\Sigma_1, (A \multimap B), \Sigma_2 \vdash C} \ L\multimap \qquad \frac{\Sigma, A \vdash B}{\Sigma \vdash (A \multimap B)} \ R\multimap$$

$$\frac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \oplus B \vdash \Delta} \ (L\oplus) \qquad \frac{\Gamma \vdash A, \Delta}{\Gamma \vdash A \oplus B, \Delta} \ (R\oplus)(a) \qquad \frac{\Gamma \vdash B, \Delta}{\Gamma \vdash A \oplus B, \Delta} \ (R\oplus)(b)$$

$$\frac{\Gamma, A \vdash \Delta}{\Gamma, A \& B \vdash \Delta} \ (L\&)(a) \qquad \frac{\Gamma, B \vdash \Delta}{\Gamma, A \& B \vdash \Delta} \ (L\&)(b) \qquad \frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma \vdash A \& B, \Delta} \ (R\&)$$

Rules for quantifiers:

$$\frac{\Gamma, A[a/x] \vdash \Delta}{\Gamma, \forall x A \vdash \Delta} \ L\forall \qquad \frac{\Gamma \vdash \Delta, A[t/x]}{\Gamma \vdash \Delta, \forall x A} \ R\forall$$

$$\frac{\Gamma, A[t/x] \vdash \Delta}{\Gamma, \exists x A \vdash \Delta} \ L\exists \qquad \frac{\Gamma \vdash A[a/x], \Delta}{\Gamma \vdash \exists x A, \Delta} \ R\exists$$

where t is not free in Γ and Δ.

A Proposal for Reasoning in Agents: Restricted Entailment

Lee Flax

Macquarie University, Sydney NSW 2109, Australia
flax@ics.mq.edu.au

Abstract. Johnson-Laird proposes a semantic theory of human reasoning taking into account finite human capacities. We cast this into logical formalism and define a notion of restricted semantic entailment. Corresponding to any set of logical structures, R, there is a restricted entailment with parameter R. The family of restricted entailments, generated as R varies over sets of structures, is shown to be a complete lattice and to approximate ordinary entailment in the sense of domain theory. A given restricted entailment, $\models_R$ say, can be modelled in a modal language with an operator $\downarrow_R$. The modal language is sound and complete and there is a correspondence result: $X \models_R \varphi$ iff $\downarrow_R X \Vdash \downarrow_R \varphi$, where X is a set of first-order sentences and φ is first-order. This forms the basis for the proposal that $\models_R$ be identified with agent reasoning and that $\downarrow_R$ encapsulate an agent. The existence of the lattice structure mentioned above means that several agents can be integrated into a super-agent or else distilled into a sub-agent by taking joins or meets.

1 Introduction

When we address the question of what could constitute the basis of rationality in an agent, we quickly arrive at a difficulty. If we seek an algorithmic basis founded on first-order logic, then we come up against the problem of undecidability [3, page 159]: there is no procedure which can decide for an arbitrary sentence whether it is valid (true in every model) or not. The reason for this is that algorithms are finite in nature and so, in general, cannot deal with non-enumerable processes.

Humans face these same limitations because of their finite capacities; nevertheless we know that humans are capable of reasoning. Researchers such as Cherniak [2] and cognitive psychologists such as Johnson-Laird and others [6, 7] have developed theories about how agents and humans might reason which take into account the limitation of their finite capacities.

Cherniak in [2] develops the concept of *minimal rationality* and uses this to account for an agent's reasoning capabilities in the face of finite limitations. The following quotes from [2, pages 7 to 9] show that, at least in principle, minimal rationality can be expressed in *linguistic* terms using first-order logic, say. His *ideal general rationality* condition is:

> If A has a particular belief-desire set, A would undertake *all* and only actions that are apparently appropriate.

where

> ... an action is *apparently appropriate* if and only if according to A's beliefs, it would tend to satisfy A's desires.

Now

> ... The most important unsatisfactoriness of the ideal general rationality condition arises from its denial of a fundamental feature of human existence, that human beings are in the *finitary predicament* of having fixed limits on their cognitive capacities and the time available to them.

So the *minimal general rationality* condition is posited:

> If A has a particular belief-desire set, A would undertake some, but not necessarily all, of those actions that are apparently appropriate.

In contrast to this, linguistic, approach Johnson-Laird in [6] claims that human reasoning and deduction is carried out by people forming mental models of the situation being reasoned about and then manipulating the mental models in certain ways. The precise nature of these mental models is not important to our argument so we do not describe them. Johnson-Laird and Byrne say in [7, page 36]:

> The theory is compatible with the way in which logicians formulate a semantics for a calculus ... But, logical accounts depend on assigning an infinite number of models to each proposition, and an infinite set is far too big to fit inside anyone's head ... people construct a minimum of models: they try to work with just a single representative sample from the set of possible models ...

So Johnson-Laird claims that humans reason by building mental models but the model checking is not exhaustive at any stage because of limited human capacities. One can map this situation into the context of an agent doing semantics of first-order logic and say that in checking the validity of a first-order sentence the agent is only able to check the truth of a subset of all models. That is the agent checks a *restricted* set of models.

We take this as our point of departure and approach first-order semantics in a manner which takes seriously this limitation of restricted model checking. An appropriate notion of satisfaction is defined for this approach as well as a notion of *restricted entailment*. A restricted entailment is like an ordinary entailment except that instead of checking all possible models for satisfaction, one restricts one's checking to a subset R, say, of models. The set R is taken to be a parameter of the entailment and the restricted entailment with parameter R is denoted $\models_R$. It has the usual properties one would expect of an entailment relation: reflexivity, cut and monotony. It also has properties that are significant to the claim that

it captures features characteristic of rationality: it is sound, complete and can approximate ordinary entailment in the sense of domain theory.

The idea of restricted entailment provides the basis for the crystallisation of our proposal for the notion of a finite agent. This is explained as follows. A modal operator, called "approximately true" and denoted $\downarrow_R$, is introduced which also has a parameter R, a *finite* set of models. In our proposal the operator $\downarrow_R$ defines an agent; a different agent is defined for each different parameter.

We claim that the language with modal operator "approximately true" models restricted entailment because for finite R, X a set of first-order sentences and φ a first-order sentence, $X \models_R \varphi$ if and only if $\downarrow_R X \Vdash \downarrow_R \varphi$, where $\Vdash$ is the modal forcing relation. The significance of this is that it turns out that restricted entailment is able to be modelled by a modal language which has a semantics and proof theory which are sound and complete.

The restriction parameters are sets and so they are automatically endowed with lattice operations: set union and intersection. These are carried over to the set of restricted entailments; they form a lattice. However, the relationship between lattice operations in the entailment set and in the parameter set is contravariant. As a result the meet of a set of restricted entailments is given by the entailment whose parameter is the union of the individual parameters. The join is slightly more complex, but it does involve taking an intersection of sets induced by the restriction parameters. These results mean that given a set of agents one can form a new, inclusive, canonical agent encompassing the deductive powers of the individual agents, and another having the common part of the deductive powers of the individual agents. In the conclusion we sketch how this also holds for agents when they are represented by the modal operator "approximately true". The paper ends with some remarks about the computability of restricted entailment and also some remarks about the fact that the modal language with a single operator $\downarrow_R$ is sound and complete.

In the next section we define restricted entailment and develop machinery to show that the family of restricted entailments has a lattice structure.

2 Restricted Entailment

In this section the set of *restricted entailments* is defined. The members of this set are generalisations of ordinary entailment. For ordinary entailment, one checks that a set of sentences X entails a set Y by examining every structure that is a model of X and checking that it is also a model of Y. If this is always the case then X entails Y. In contrast, one checks for restricted entailment by specifying a subset of structures, R say, and then checking that each model of X lying in R is also a model of Y. So instead of checking all structures, one *restricts* one's checking to a subset, R, of structures. As we have said, this approach to the approximation of entailment is motivated by Johnsohn-Laird's theory of mental models in cognitive psychology.

The main result of this section is corollary 11: any set of restricted entailments has a greatest lower bound and a least upper bound, or meet and join. The existence of the meet is used in the definition of "domain" in section 3.

We work in a standard first-order language whose *vocabulary* consists of a countable number of constant symbols, a countable number of function symbols of any finite arity and a countable number of relation symbols of any finite arity. A *structure*, $\mathcal{S}$, is a function having a *domain of interpretation*, $\mathsf{dom}(\mathcal{S})$, which is a set. It maps constants to elements in $\mathsf{dom}(\mathcal{S})$, function symbols to functions defined on $\mathsf{dom}(\mathcal{S})$ and relation symbols to relations on $\mathsf{dom}(\mathcal{S})$. The language also has a countable number of *individual variables*; the *connectives* $\neg$, $\vee$, $\wedge$ and $\rightarrow$; and the *quantifiers* $\forall$ and $\exists$. The *terms* of the language are defined in the usual way, as are the *formulas*. Given any formula, an individual variable is *free* in that formula if the variable is not in the scope of any quantifier in the formula. A *sentence* is a formula with no free variables. Meaning is given to sentences by defining, in the usual way, the *satisfaction relation* between structures and sentences. If the structure $\mathcal{S}$ satisfies the sentence φ, it is written thus: $\mathcal{S} \models \varphi$.

In order to avoid difficulties with set-theoretical foundations we work entirely in a universe of sets [1]; all collections of objects are sets and there are no classes which are not sets.

We suppose a language is given and remains fixed. The set of all structures defined on the vocabulary is denoted STRUC. The set of all subsets of STRUC is denoted PSTRUC. The relation of *elementary equivalence* between stuctures is defined as follows: two structures $\mathcal{S}$ and $\mathcal{S}'$ are elementarily equivalent, denoted $\mathcal{S} \equiv \mathcal{S}'$, if and only if they satisfy the same sentences.

The following fundamental *restricted* notions are now defined: set of models and entailment. The definitions are made by analogy with the unrestricted ones, which can be obtained from the following definition by omitting the subscript R.

Definition 1 *Let X and Y be sets of sentences and let $R \subseteq \mathsf{STRUC}$.*

1. *The* set of models of X restricted to R, *denoted* $\mathsf{mod}_R(X)$, *is* $\mathsf{mod}_R(X) = \{\mathcal{S} \in R : \mathcal{S} \models X\}$.
2. X entails Y with restriction R *iff* $\mathsf{mod}_R X \subseteq \mathsf{mod}_R Y$*; this is written as* $X \models_R Y$.

Note that $\models_R$ generalises ordinary entailment, $\models$, because $\models_{\mathsf{STRUC}}$ equals $\models$.

It is well-known that entailment satisfies three properties called *reflexivity*, *cut* and *monotony*. A straightforward argument shows that restricted entailment also satisfies appropriate versions of these properties. As they are not central to our argument we do not give them here.

In what follows, propositions 2 and 4 are needed for the important propositions 9 and 10 on the bounds of families of restricted entailments. Proposition 2 says that if a set of structures is enlarged then the associated restricted entailment is reduced, while proposition 4 provides a converse to proposition 2 under a *fullness* condition on the restriction of the entailment.

Proposition 2 *Let $I \subseteq J \subseteq \mathsf{STRUC}$, then $\models_J \subseteq \models_I$.*

Proof. Let X and Y be sets of sentences and suppose that $X \models_J Y$; that is $\mathsf{mod}_J(X) \subseteq \mathsf{mod}_J(Y)$. We must show $\mathsf{mod}_I(X) \subseteq \mathsf{mod}_I(Y)$.

$$\begin{aligned}
\mathsf{mod}_I(X) &= I \cap \mathsf{mod}(X) \\
&= (I \cap J) \cap \mathsf{mod}(X) \text{ since } I \subseteq J \\
&= I \cap (J \cap \mathsf{mod}(X) \\
&= I \cap \mathsf{mod}_J(X) \\
&\subseteq I \cap \mathsf{mod}_J(Y) \qquad \text{since } \mathsf{mod}_J(X) \subseteq \mathsf{mod}_J(Y) \\
&= I \cap (J \cap \mathsf{mod}(Y)) \\
&= (I \cap J) \cap \mathsf{mod}(Y) \\
&= I \cap \mathsf{mod}(Y) \qquad \text{since } I \subseteq J \\
&= \mathsf{mod}_I(Y)
\end{aligned}$$

■

There is a converse to proposition 2 provided J is *full*. This is defined next.

Definition 3 *Let $I \subseteq \mathsf{STRUC}$, then I is* full *if and only if given elements $\mathcal{S}$ and $\mathcal{S}'$ of STRUC, if $\mathcal{S} \in I$ and $\mathcal{S}$ is elementarily equivalent to $\mathcal{S}'$ then $\mathcal{S}'$ is in I.*

Proposition 4 *Let I and J be subsets of STRUC and suppose J is full. If $\models_J \subseteq \models_I$, then $I \subseteq J$.*

Proof. We suppose $I \not\subseteq J$ and prove that $\models_J \not\subseteq \models_I$. There are two cases to consider: either J is empty or not empty.

CASE $J = \emptyset$. Let $\mathcal{S} \in I$ and let φ be a sentence with $\mathcal{S} \models \varphi$. Set $X = \{\varphi\}$ and $Y = \{\neg\varphi\}$, then $\mathcal{S} \models X$ and $\mathcal{S} \not\models Y$. So $\mathcal{S} \in \mathsf{mod}_I(X)$ but $\mathcal{S} \notin \mathsf{mod}_I(Y)$. Since $J = \emptyset$, $X \models_J Y$ but as we have seen $X \not\models_I Y$.

CASE $J \neq \emptyset$. Suppose $\mathcal{S} \in I$ and $\mathcal{S} \notin J$, then $\mathcal{S}$ is not elementarily equivalent to any $\mathcal{S}' \in J$. This is so because if $\mathcal{S}$ is elementarily equivalent to some $\mathcal{S}' \in J$ then since J is full $\mathcal{S} \in J$, which is contrary to our supposition. So for each $\mathcal{S}' \in J$ there is a sentence $\varphi_{\mathcal{S}'}$, say, with $\mathcal{S} \models \varphi_{\mathcal{S}'}$ and $\mathcal{S}' \not\models \varphi_{\mathcal{S}'}$. Set $X = \{\varphi_{\mathcal{S}'} : \mathcal{S}' \in J\}$, then $\mathcal{S} \in \mathsf{mod}_I(X)$ but $\mathsf{mod}_J(X) = \emptyset$. Also pick $\mathcal{S}'' \in J$ and set $Y = \{\neg\varphi_{\mathcal{S}''}\}$, then $\mathcal{S} \notin \mathsf{mod}_I(Y)$.

Now $\mathsf{mod}_J(X) = \emptyset \subseteq \mathsf{mod}_J(Y)$, so $X \models_J Y$. But $\mathcal{S} \in \mathsf{mod}_I(X)$ and $\mathcal{S} \notin \mathsf{mod}_I(Y)$ so $X \not\models_I Y$. ■

Next we define the *greatest lower bound* and *least upper bound* of a set of restricted entailments and then show how to calculate them. We recall that a partial order is a reflexive, transitive, antisymmetric relation.

Definition 5

1. $\mathsf{ENT} = \{\models_I : I \subseteq \mathsf{STRUC}\}$.
2. $(\mathsf{ENT}, \subseteq)$ *is the partially ordered set where the elements of* ENT *are partially ordered by set inclusion.*

3. *Let* $E \subseteq \mathsf{ENT}$. *An element* $\models_H$ *of* ENT *is a* lower bound *of* E *if* $\models_H \subseteq \models_I$ *for each* $\models_I \in E$.
4. *An element* $\models_G$ *of* ENT *is the* greatest lower bound *of* $E \subseteq \mathsf{ENT}$ *if* $\models_G$ *is a lower bound of* E *and it is a superset of, or equal to any other lower bound of* E. *The greatest lower bound of* E, *if it exists, is denoted* $\bigwedge E$.
5. *Let* $E \subseteq \mathsf{ENT}$. *An element* $\models_H$ *of* ENT *is an* upper bound *of* E *if* $\models_I \subseteq \models_H$ *for each* $\models_I \in E$.
6. *An element* $\models_L$ *of* ENT *is the* least upper bound *of* $E \subseteq \mathsf{ENT}$ *if* $\models_L$ *is an upper bound of* E *and it is a subset of, or equal to any other upper bound of* E. *The least upper bound of* E, *if it exists, is denoted* $\bigvee E$.

We want to show that the meet of $\models_I$ and $\models_J$ is $\models_{I \cup J}$. We do this with the help of an operator that takes any subset I of STRUC and turns it into a full subset containing I denoted $[I]$. The operator is called the *full expansion* operator.

We recall that PSTRUC stands for the set of all subsets of STRUC. If $E \subseteq \mathsf{PSTRUC}$ then each element of E is a subset of STRUC; $\bigcup E$ means the union of all members of E.

Definition 6

1. *Let* $I \subseteq \mathsf{STRUC}$. *The* full expansion *of* I, *denoted* $[I]$, *is* $\{\mathcal{S} \in \mathsf{STRUC} : \exists \mathcal{S}' \in I \ \& \ \mathcal{S} \equiv \mathcal{S}'\}$.
2. *Let* $E \subseteq \mathsf{PSTRUC}$, *then* $[E] = \{[I] : I \in E\}$.

Proposition 7

1. *The full expansion of a set of structures is full.*
2. *The full expansion of a set of structures is unique.*
3. *The full expansion operator satisfies inclusion, idempotence and monotony:*
   (a) $I \subseteq [I]$.
   (b) $[[I]] = [I]$.
   (c) *If* $I \subseteq J$, *then* $[I] \subseteq [J]$.

Proof. Straightforward. ■

Proposition 8 *Let* $I \subseteq \mathsf{STRUC}$, *then* $\models_I = \models_{[I]}$.

Proof. Full expansion satisfies inclusion so $I \subseteq [I]$ and by proposition 2 $\models_{[I]} \subseteq \models_I$.

On the other hand suppose $X \models_I Y$ and let $\mathcal{S}' \in [I] \cap \mathsf{mod}(X)$. We must show that $\mathcal{S}' \in [I] \cap \mathsf{mod}(Y)$. There is $\mathcal{S} \in I$ such that $\mathcal{S}' \equiv \mathcal{S}$. Also $\mathcal{S} \in \mathsf{mod}(X)$ because, as is easily seen, $\mathsf{mod}(X)$ is full and $\mathcal{S}' \in \mathsf{mod}(X)$. So $\mathcal{S} \in \mathsf{mod}_I(X)$ and therefore by our supposition $\mathcal{S} \in \mathsf{mod}_I(Y)$. By the inclusion property of full expansion $I \subseteq [I]$, therefore $\mathcal{S} \in I \cap \mathsf{mod}(Y) \subseteq [I] \cap \mathsf{mod}(Y)$. But since, as is easily seen, $[I] \cap \mathsf{mod}(Y)$ is full, we have that $\mathcal{S}' \in [I] \cap \mathsf{mod}(Y)$. ■

The following results allow one to calculate meets and joins. The meet of a family of restricted entailments is generated by taking the union of the restriction sets. The join, however, is generated by taking the intersection of the full expansions of the restriction sets.

Proposition 9 *Let $E \subseteq \mathsf{PSTRUC}$. The following are true.*

1. $\models_{\cup E} \subseteq \models_I$, *for each* $I \in E$.
2. *Suppose there is* $G \subseteq \mathsf{STRUC}$ *with* $\models_{\cup E} \subseteq \models_G \subseteq \models_I$ *for each* $I \in E$, *then* $\models_G = \models_{\cup E}$.

Proof. 1. This follows from proposition 2 because $I \subseteq \bigcup E$ for each $I \in E$.
2. We have supposed that
$\models_{\cup E} \subseteq \models_G \subseteq \models_I$ so
$\models_{[\cup E]} \subseteq \models_{[G]} \subseteq \models_I$ (by 8) and
$I \subseteq [G] \subseteq [\cup E]$ (by 4) so
$\cup E \subseteq [G] \subseteq [\cup E]$ and
$[\cup E] \subseteq [G] \subseteq [\cup E]$ (by 7 monotony and idempotence) giving
$[G] = [\cup E]$ and so
$\models_G = \models_{\cup E}$ (using 8)

■

The next proposition depends on the evaluation of the expression $\cap[E]$ where $E \subseteq \mathsf{PSTRUC}$. It can be shown that if a Bernays-von Neumann-Gödel set theory (such as in Kelley [8]) is adopted $[E]$ can be empty even when E is not (see [5, Appendix]). If this is the case then $\cap[E]$ is not a set, it is the class of all sets. We prefer to work in an environment without such "explosions". To achieve this we could either dispense with defining the least upper bound altogether or use an approach to set theory yielding more tractable results. As mentioned earlier, we have decided to work within a universe of sets (see [1, 9]) where the results of all constructions are sets. In a universe of sets $[E]$ is not empty if E is not and also $\cap[E]$ is a set.

Proposition 10 *Let $E \subseteq \mathsf{PSTRUC}$. The following are true.*

1. $\models_I \subseteq \models_{\cap[E]}$, *for each* $I \in E$.
2. *Suppose there is* $L \subseteq \mathsf{STRUC}$ *with* $\models_I \subseteq \models_L \subseteq \models_{\cap[E]}$ *for each* $I \in E$, *then* $\models_L = \models_{\cap[E]}$.

Proof. 1. For each $I \in E$ we have $\cap[E] \subseteq [I]$, so by 2 $\models_{[I]} \subseteq \models_{\cap[E]}$ and by 8 $\models_{[I]} = \models_I$.
2. We have supposed that
$\models_I \subseteq \models_L \subseteq \models_{\cap[E]}$ so
$\models_{[I]} \subseteq \models_{[L]} \subseteq \models_{\cap[E]}$ (by 8) and
$\cap[E] \subseteq [L] \subseteq [I]$ (by 4) so
$\cap[E] \subseteq [L] \subseteq \cap[E]$ giving
$[L] = \cap[E]$ and so
$\models_L = \models_{\cap[E]}$ (using 8)

■

We have shown that arbitrary infinite (and hence finite) meets and joins exist in PSTRUC.

Corollary 11 *Let $E \subseteq \mathsf{PSTRUC}$, then*

1. $\bigwedge_{I \in E} \models_I = \models_{\cup E}$.
2. $\bigvee_{I \in E} \models_I = \models_{\cap [E]}$.

The next section shows that the restricted entailments approximate ordinary entailment in the sense of domain theory.

3 Restricted Entailments as a Domain

Here we show that an arbitrary restricted entailment of ENT can be approximated by *compact* ones. These are of the form $\models_I$ where $I \subseteq \mathsf{STRUC}$ has only a *finite* number of structures which are not elementarily equivalent to each other. That is, I has finitely many equivalence classes under the relation of elementary equivalence. As a byproduct of this approach it will be shown how ordinary entailment, $\models$, can be approximated in this way.

It turns out that ENT is a domain in which each element is approximated by the compact elements which are set theoretically *greater than* it. So when regarding ENT as a domain the domain partial order will be taken to be $\supseteq$. The partially ordered set $(\mathsf{ENT}, \subseteq)$ has an infinitary meet operation which is taken to be the domain least upper bound operator with respect to the partial order $\supseteq$. The least element for the domain is taken to be $\models_\emptyset$. The definitions for a domain and its constituents come from [10].

Definition 12 (Complete Partial Order) *Let $D = (D, \sqsubseteq, \bot)$ be a partially ordered set with least element $\bot$.*

1. **(Directed Set)** *Let $A \subseteq D$, then A is* directed *if whenever w and x are members of A there is $y \in A$ satisfying $w \sqsubseteq y$ and $x \sqsubseteq y$.*
2. **(Complete Partial Order)** *D is called a* complete partial order (CPO) *if whenever $A \subseteq D$ and A is directed then the least upper bound of A, denoted $\bigsqcup A$, exists in D.*

Proposition 9 and corollary 11 justify the use of $\bigwedge$ as $\bigsqcup$ in the definition below.

Definition 13 *The set $\mathsf{ENT} = \{\models_I : I \subseteq \mathsf{STRUC}\}$ can be regarded as a CPO $\mathsf{ENT} = (\mathsf{ENT}, \sqsubseteq, \bot)$ by taking*

1. *$\sqsubseteq$ to be $\supseteq$,*
2. *$\bot$ to be $\models_\emptyset$,*
3. *$\bigsqcup$ to be $\bigwedge$.*

The compact elements in a CPO play a major role in approximation of members of the CPO so compactness is defined next.

Definition 14 (Compact element) *Let D be a CPO.*

1. *An element $d \in D$ is* compact *if and only if the following condition is satisfied:*
   For all $A \subseteq D$ if A is directed and $d \sqsubseteq \bigsqcup A$, then there is $x \in A$ with $d \sqsubseteq x$.
2. *The set of compact members of D is denoted* $\mathsf{comp}(D)$.

In order to characterise the compact elements in proposition 20 some results (lemma 15 to proposition 19) are needed on the cardinality of the set of equivalence classes of expansions, and related matters. Let I be a subset of STRUC, the family of equivalence classes of I under elementary equivalence is denoted $I/_{\equiv}$. If $\mathcal{S} \in I$ the equivalence class of $\mathcal{S}$ in $I/_{\equiv}$ is denoted $\|\mathcal{S}\|_I$.

Lemma 15 *Suppose $I \subseteq [H] \subseteq \mathsf{STRUC}$.*

1. *There is a one-to-one function $f : I/_{\equiv} \to [H]/_{\equiv}$.*
2. *If $I = H$, then f is onto.*

Proof. Straightforward. ■

For any set X let $\mathsf{card}(X)$ denote the cardinality of X.

Proposition 16 *Let $H \subseteq \mathsf{STRUC}$.*

1. *Both $[H]/_{\equiv}$ and $H/_{\equiv}$ are sets satisfying*
   (a) $\mathsf{card}([H]/_{\equiv}) = \mathsf{card}(H/_{\equiv})$.
   (b) $\mathsf{card}(H/_{\equiv}) \leq \mathsf{card}(H)$.
2. *If $I \subseteq [H]$, then* $\mathsf{card}(I/_{\equiv}) \leq \mathsf{card}(H)$.

Proof. Straightforward, uses lemma 15. ■

Definition 17 (Reduction) *Let $I \subseteq \mathsf{STRUC}$.*

1. *$J \subseteq \mathsf{STRUC}$ is a* reduction *of I, denoted $J \preceq I$, if and only if*
   (a) $J \subseteq I$.
   (b) For each element $x \in I/_{\equiv}$, there is an element $\mathcal{S} \in J$ satisfying $\mathcal{S} \in x$.
   (c) No two elements of J are elementarily equivalent.
2. *I is said to be* reduced *if and only if $I \preceq I$.*

The proof of the following is straightforward.

Proposition 18 *Let J and I be subsets of* STRUC.

1. *If I is reduced, then $J \preceq I$ iff $J = I$.*
2. *I has a reduction.*

Proposition 19 *Suppose $J \preceq I$, then the following are true.*

1. *J is reduced.*
2. *$[J] = [I]$.*
3. *If $I/_{\equiv}$ is finite, then so is J.*

Proof. Straightforward. Part 2 uses monotony of expansion (see 7). Part 3 uses 15 and 16. ■

The next result characterises compact restricted entailments.

Proposition 20 *The restricted entailment $\models_I$ is compact if and only if $I/_{\equiv}$ is finite.*

Proof. Suppose $\models_I$ is compact. Let $E = \{F : F \subseteq I \ \&\ F \text{ is finite}\}$ and consider $T = \{\models_F : F \in E\}$. T is directed and $\bigwedge T = \models_{\cup E} \subseteq \models_I$ because $I \subseteq \bigcup E$. Since $\models_I$ is compact there is $\models_H \in T$ with $\models_H \subseteq \models_I$. By definition of T, H is finite. By propositions 4 and 8 $I \subseteq [H]$ and so $I/_{\equiv}$ is finite by 16. This proves one half of the proposition.

To prove the other half suppose $I/_{\equiv}$ is finite. We must show that $\models_I$ is compact. Let $T = \{\models_R : R \in E\}$ be directed where $E \subseteq \mathsf{PSTRUC}$ is arbitrary. Suppose $\bigwedge T \subseteq \models_I$. We must show there is a member of T that is a subset of $\models_I$. Now $\models_{[\cup E]} = \models_{\cup E} = \bigwedge T \subseteq \models_I$ so $I \subseteq [\bigcup E]$ by 4.

Let $J \preceq I$, then by proposition 19 J is finite. Also $J \subseteq I$ so $J \subseteq [\bigcup E]$. It follows that for each $\mathcal{S} \in J$ there is a member of E, denote it $R_{\mathcal{S}}$, such that $\mathcal{S} \in [R_{\mathcal{S}}]$. But J is finite and T is directed so there is $\models_G \in T$ with $\models_G \subseteq \models_{R_{\mathcal{S}}}$ for each $\mathcal{S} \in J$. Because $\models_{[G]} = \models_G$ we have $R_{\mathcal{S}} \subseteq [G]$ by 4 and idempotence, and by monotony of expansion and idempotence $[R_{\mathcal{S}}] \subseteq [G]$. So we have $J \subseteq \bigcup\{[R_{\mathcal{S}}] : \mathcal{S} \in J\} \subseteq [G]$ and it follows that $\models_G = \models_{[G]} \subseteq \models_J = \models_{[J]}$. Since $\models_{[J]} = \models_{[I]}$ by 19, we have that $\models_G \subseteq \models_{[I]} = \models_I$. ■

A CPO $(D, \sqsubseteq, \bot)$ is *algebraic* if any element of D is equal to the the least upper bound of the compact elements below it.

Definition 21 (Algebraic CPO) *Let $D = (D, \sqsubseteq, \bot)$ be a CPO. For $x \in D$, denote $\{c \in \mathsf{comp}(D) : c \sqsubseteq x\}$ by $\mathsf{approx}(x)$. D is* algebraic *if and only if the following condition holds.*

If $x \in D$, then $\mathsf{approx}(x)$ is directed and $x = \bigsqcup \mathsf{approx}(x)$.

The next lemma is used to show that ENT is algebraic.

Lemma 22 *Let I and J be subsets of STRUC. If both $I/_{\equiv}$ and $J/_{\equiv}$ are finite, then so is $(I \cup J)/_{\equiv}$.*

Proof. Straightforward. ■

Proposition 23 *The CPO $\mathsf{ENT} = (\mathsf{ENT}, \sqsubseteq, \bot)$ is algebraic.*

Proof. From definition 13 $\sqsubseteq$ is $\supseteq$ so the set $\mathsf{approx}(\models_R) = \{\models_I : \models_R \subseteq \models_I \ \&\ I/_{\equiv} \text{ is finite}\}$. It is directed because if $\models_I$ and $\models_J$ are elements of $\mathsf{approx}(\models_R)$, then $\models_R \subseteq \models_I \wedge \models_J = \models_{I \cup J}$, which is an element of $\mathsf{approx}(\models_R)$ since by lemma 22 $(I \cup J)/_{\equiv}$ is finite if both $I/_{\equiv}$ and $J/_{\equiv}$ are.

Finally we must show that $\bigwedge \mathsf{approx}(\models_R) = \models_R$. Any finite set $F \subseteq \mathsf{STRUC}$ has finite $F/_{\equiv}$. Now consider $E = \{F : F \subseteq R \ \&\ F \text{ is finite}\}$. For each $F \in E$, $\models_F \in \mathsf{approx}(\models_R)$ and $\bigcup E = R$. So $\models_R \subseteq \bigwedge \mathsf{approx}(\models_R) \subseteq \bigwedge\{\models_F : F \in E \ \&\ F \text{ is finite}\} = \models_{\bigcup E} = \models_R$. ■

A *domain* is an algebraic CPO satisfying a certain consistency condition.

Definition 24 (Consistent Set) *Let $D = (D, \sqsubseteq)$ be partially ordered and let $A \subseteq D$, then A is* consistent *if and only if A has an upper bound in D.*

Definition 25 (Domain) *Let $D = (D, \sqsubseteq, \bot)$ be a CPO, then D is a* domain *if and only if the following two conditions are satisfied.*

1. *D is an algebraic CPO.*
2. *If d and e are compact members of D and $\{d, e\}$ is consistent, then $d \vee e$ exists and lies in D.*

We have seen in proposition 23 that $\mathsf{ENT} = (\mathsf{ENT}, \sqsubseteq, \bot)$ is algebraic. By 9 any pair of elements of ENT has a least upper bound given by meet. The following proposition summarises the situation.

Proposition 26 *The CPO $\mathsf{ENT} = (\mathsf{ENT}, \sqsubseteq, \bot)$ defined in* 13 *is a domain in which*

1. *Ordinary entailment $\models = \models_{\mathsf{STRUC}} \in \mathsf{ENT}$.*
2. *The compact members of ENT are the restricted entailments $\models_I$, where $I/_{\equiv}$ is finite.*
3. *For $\models_R \in \mathsf{ENT}$, $\mathsf{approx}(\models_R) = \{\models_I : \models_R \subseteq \models_I \ \ \& \ I/_{\equiv} \text{ is finite}\}$.*

In fact, the last sentence in the proof of proposition 23 shows that any restricted entailment is the meet of a subset of the compact ones greater than it, namely the ones with *finite restrictions*.

In the next section we show how restricted entailment can be modelled in a modal language, and discuss its semantics and proof theory.

4 The Modality "Approximately True"

In this section we sketch the properties of a first-order language augmented with a single modal operator, "approximately true" with parameter R, which is denoted $\downarrow_R$. The parameter R is a finite set of structures. With R finite, $\downarrow_R$ is taken to embody a finite agent. As was mentioned earlier this is justified by the "correspondence" result for first-order sentences: proposition 30. Also the modal language has a proof theory and a semantics which are sound and complete. So by using the correspondence result to translate a restricted entailment into the modal language, one can use the proof theoretic and semantic facilities of a language which is sound and complete.

The semantics and proof theory of $\downarrow_R$ are summarised as well as their soundness and completeness. The discussion is brief and no proofs are given. Full detail can be found in [5, chapters 7 to 10]. In the final section some comments are made about the logic of interaction of several different agents when they are modelled by "approximately true" with different parameters.

Modal formulas are defined in the following way so as to allow the proof of completeness to go through smoothly. The definition is in two stages.

Definition 27 (Modal Formula)

1. *Let $R \subseteq \mathsf{STRUC}$. The approximation sentences are defined inductively as follows.*
   (a) *Any first-order sentence is an approximation sentence.*
   (b) *If φ and ψ are approximation sentences, then so are $\downarrow_R \varphi$, $\neg\varphi$ and $\varphi \circ \psi$, where $\circ$ is a boolean connective. Note that sometimes the subscript, R, in $\downarrow_R$ will be dropped if there is no danger of confusion.*
2. *The modal formulas are defined inductively as follows.*
   (a) *Any approximation sentence is a modal formula, and any atomic first-order formula is a modal formula.*
   (b) *If φ and ψ are modal formulas then so are $\neg\varphi$ and $\varphi \circ \psi$, where $\circ$ is a boolean connective, as well as $(\forall x)\varphi$ and $(\exists x)\varphi$.*

Modal Semantics

The modal counterpart of *satisfaction* is *forcing*, denoted $\Vdash$. Modal *models* force modal formulas. The semantics is unusual in that it uses a single possible world. Models will now be defined.

Definition 28 (Modal Model) *A model is a four-tuple $\mathcal{M} = (\mathcal{G}, \mathcal{R}, \mathcal{D}, \mathcal{S})$ where:*

1. *$\mathcal{G}$ is a set with one element, G, called a world.*
2. *$\mathcal{R}$ is a binary relation defined on $\mathcal{G}$ called the accessibility relation. There are only two possibilities for $\mathcal{R}$ in our system: it is either empty or it equals (G, G).*
3. *$\mathcal{D}$ is a set called the domain of the model.*
4. *$\mathcal{S}$ is a first-order structure with domain $\mathsf{dom}(\mathcal{S}) = \mathcal{D}$.*

The following is an abbreviated definition of modal forcing.

Definition 29 (Forcing) *Let $\mathcal{M} = (\mathcal{G}, \mathcal{R}, \mathcal{D}, \mathcal{S})$ be a model and $\mathcal{I} = (\mathcal{S}, u)$ be a first-order interpretation, where u is an assignment of variables. The forcing relation, $\Vdash$, is defined recursively as follows. Let φ and ψ be modal formulas.*

1. *If φ is an atomic formula, then $\mathcal{M} \Vdash_u \varphi$ iff $\mathcal{I} \vDash \varphi$.*
2. *Rules for forcing of $\neg\varphi$, $\varphi \circ \psi$ (where $\circ$ is a boolean connective) and quantified formulas are analogous to first-order satisfaction rules.*
3. *$\mathcal{M} \Vdash \downarrow_R \varphi$ if and only if the following holds: if $G\mathcal{R}G$ and $\mathcal{S} \in R$ then $\mathcal{M} \Vdash \varphi$.*

If X is a set of modal sentences, then we write $\mathcal{M} \Vdash X$ if and only if $\mathcal{M} \Vdash \varphi$ for each $\varphi \in X$. Also, $\downarrow X = \{\downarrow \varphi : \varphi \in X\}$. Modal entailment is defined as follows: $X \Vdash \varphi$ if and only if for every model $\mathcal{M}$, $\mathcal{M} \Vdash X$ implies $\mathcal{M} \Vdash \varphi$.

The following "correspondence" result connects restricted entailment and forcing of "approximately true". It provides the theoretical basis for regarding forcing of "approximately true" as encapsulating the reasoning of a finite agent, and hence for regarding "approximately true" as the embodiment of a finite agent.

Proposition 30 *Let φ be a first-order sentence and X a set of first-order sentences, then $X \vDash_R \varphi$ iff $\downarrow_R X \Vdash \downarrow_R \varphi$.*

Proof Theory, Soundness and Completeness

Our proof theory is based on tableaux and we use Fitting and Mendelsohn's approach [4]. Tableau rules for first-order logic are augmented to include ones for sentences issuing from "approximately true", $\downarrow$. Our method uses signed sentences. A signed sentence is one which begins with either the character T or F. A tableau is a tree having sentences as nodes, with one root node and branches which end in leaf nodes.

Generic tableau branch extension rules are given in figure 1 and then the working of a tableau proof is outlined. The rules show what can be appended to a leaf of a branch if a certain sentence (referred to as the parent) lies on the branch.

Some explanatory comments follow for the rules involving $\downarrow$; its parameter, R, has been omitted.

First, an infinitely long *separator sentence*, σ, can be set up with the property that every member of R satisfies σ and every structure not elementarily equivalent to a member of R does not satisfy σ (see [5, page 14]). The sentence σ is used as a kind of "token" in tableau rules and it is also used in the proof of completeness; σ is of the form $\bigvee_{i=1}^{m} C_i$, where m is an integer and $C_i = \bigwedge_{n \in N_i} \sigma_{n,i}$ for N_i some subset (possibly all) of the integers. Each of the $\sigma_{n,i}$ is a first-order sentence. Next, the comments for the rules involving $\downarrow$ follow.

1. The parent is $\mathsf{T} \downarrow\varphi$; the node $\mathsf{T}\varphi$ is appended to the leaf if $\mathsf{T}\sigma$ already appears on the branch, otherwise nothing is appended.
2. The parent is $\mathsf{F} \downarrow\varphi$; then two nodes are appended in sequence: $\mathsf{F}\varphi$ and $\mathsf{T}\sigma$.
3. The parent is $\mathsf{T}\sigma$; then m nodes are appended in parallel: $\mathsf{T}C_1, \ldots, \mathsf{T}C_m$.
4. The parent is $\mathsf{T}C_i$; then nodes $\mathsf{T}\sigma_{n,i}$ are appended in sequence, for each $n \in N_i$. We recall that each of the $\sigma_{n,i}$ is a first order sentence.

A tableau proof for the sentence ψ consists of using tableau rules to build a tree with root node $\mathsf{F}\psi$ and then checking that every branch of the tree has a contradiction, that is, a pair of expressions in sequence which are identical except that one begins with the character T and the other with F. If a branch has a contradiction it is said to be closed; if every branch has a contradiction then the tableau is said to be closed. A closed tableau is taken to be a proof.

Taken together, the tableau proof method and modal semantics can be shown to be sound and complete (see [5]). Soundness means that if a modal sentence is provable then it is valid, that is, forced by every model. Completeness is the converse of soundness: if a sentence is forced by every model then it is provable. It is not hard to extend the argument to show that a sentence ψ can be proved from premises X (a set of sentences) if and only if X forces ψ. Denoting the provability relation by $\vdash$, this means that $X \vdash \psi$ if and only if $X \Vdash \psi$. In particular, this holds for the "correspondence" of proposition 30: $\downarrow_R X \Vdash \downarrow_R \varphi$ iff $\downarrow_R X \vdash \downarrow_R \varphi$. So if we have a restricted entailment $X \models_R \varphi$, we can be sure $\downarrow_R \varphi$ is modally provable from $\downarrow_R X$.

Some concluding remarks and directions for future work follow.

$$\frac{\mathsf{T}\varphi}{\text{any atomic } \varphi} \qquad \frac{\mathsf{F}\varphi}{\text{any atomic } \varphi} \qquad \frac{\mathsf{T}\neg\varphi}{\mathsf{F}\varphi} \; \frac{\mathsf{F}\neg\varphi}{\mathsf{T}\varphi}$$

$$\frac{\mathsf{T}\varphi \vee \psi}{\mathsf{T}\varphi \mid \mathsf{T}\psi} \; \frac{\mathsf{F}\varphi \vee \psi}{\begin{array}{c}\mathsf{F}\varphi \\ \mathsf{F}\psi\end{array}} \qquad \frac{\mathsf{T}\varphi \wedge \psi}{\begin{array}{c}\mathsf{T}\varphi \\ \mathsf{T}\psi\end{array}} \; \frac{\mathsf{F}\varphi \wedge \psi}{\mathsf{F}\varphi \mid \mathsf{F}\psi}$$

$$\frac{\mathsf{T}\varphi \to \psi}{\mathsf{F}\varphi \mid \mathsf{T}\psi} \; \frac{\mathsf{F}\varphi \to \psi}{\begin{array}{c}\mathsf{T}\varphi \\ \mathsf{F}\psi\end{array}}$$

$$\frac{\mathsf{T}\exists x\varphi(x)}{\mathsf{T}\varphi(c)} \qquad \frac{\mathsf{F}\exists x\varphi(x)}{\mathsf{F}\varphi(t)} \qquad \frac{\mathsf{T}\forall x\varphi(x)}{\mathsf{T}\varphi(t)} \qquad \frac{\mathsf{F}\forall x\varphi(x)}{\mathsf{F}\varphi(c)}$$

some new c | any closed term t | any closed term t | some new c

$$\frac{\mathsf{T}\downarrow\varphi}{\mathsf{T}\varphi} \qquad \frac{\mathsf{F}\downarrow\varphi}{\mathsf{F}\varphi}$$

If $\mathsf{T}\sigma$ is on branch $\mathsf{T}\sigma$

$$\frac{\mathsf{T}\sigma}{\mathsf{T}C_1 \mid \cdots \mid \mathsf{T}C_m} \qquad \frac{\mathsf{T}C_i}{\begin{array}{c}\mathsf{T}\sigma_{1,i} \\ \vdots \\ \mathsf{T}\sigma_{n,i} \\ \vdots\end{array}}$$

Fig. 1. Tableaux Branch Extension Rules

5 Discussion and Future Work

In the previous sections we have described a basis for a reasoning mechanism, restricted entailment, which can be made finitary in character by a suitable choice of parameter. We have shown that it approximates ordinary entailment in the sense of domain theory and that it can be modelled in a modal language with a semantics and proof theory that are sound and complete. We have proposed that a restricted entailment be taken to encapsulate the deductive process of an agent. This is important because it is an approach that takes seriously the finite limitations of a reasoning agent.

We need to bear in mind that even humans cannot store infinite objects in their heads, or brains. To be able to do so would require an addressing system in the brain capable of making infinitely many links to the infinitely many tokens representing the elements of the infinite object. But there are only finitely many particles in the brain to provide the raw materials. (This, however, does not rule out the possibility of a human carrying in their brain a name for an infinite object and being able to reason about that object.) In any case even if one quibbles with this argument about humans, it is still true that an artificial reasoning agent cannot carry an infinite object, such as an infinite list, in its memory. So it seems worthwhile to make a start, as we have done, in analysing what can be accomplished in a finitary way.

This brings us to another point. If we take the logical essence of rationality to be encompassed by (ordinary) entailment, it is still not clear which of its properties would be generally accepted as being definitive of rationality. When considering restricted entailment as a candidate for finitary rational agency we suggest that perhaps the selection of defining properties can be deferred because restricted entailment already enjoys several important properties of entailment such as reflexivity, cut and monotony, and it can be modelled modally by a sound and complete system.

We now touch on the topics of computability and a calculus for multi-agent rationality. In [5, chapter 14] it is shown that a restricted entailment, $\models_R$, is decidable provided that R is finite; the structures in R have have finite domains and are computable; assignments of variables are computable; the interpretations of function and relation symbols under structures are computable and only finite sets of sentences are considered. This means that the deductive process is decidable for those agents meeting these requirements. Also, because a finite union of finite sets is finite, the integration of finitely many agents into a canonical super-agent produces one with a reasoning process that is decidable (under the provisos mentioned above). Consideration of the computability of the meet of a finite number of agents requires further work because the meet is induced by the intersection of the full expansions of parameter sets, which may not be finite. (But note that the resulting parameter has only a finite number of equivalence classes under elementary equivalence of structures.)

The semantics for $\downarrow_R$ presented in definition 29 can be easily extended to deal with several operators with different parameters co-existing in the same modal

language. The first steps in a calculus can even be taken. Semantic arguments can be used to show:

1. $\mathcal{M} \Vdash \downarrow_{I \cup J} \varphi$ iff $\mathcal{M} \Vdash (\downarrow_I \varphi \vee \downarrow_J \varphi)$.
2. $\mathcal{M} \Vdash \downarrow_{I \cap J} \varphi$ iff $\mathcal{M} \Vdash (\downarrow_I \varphi \wedge \downarrow_J \varphi)$.

Also it would not be difficult to include other modal operators such as "box" and "diamond" in the language. Further work needs to be done to examine the properties of this language.

Our formalisation of agents here is semantically based. Wooldridge in [11, page 296] comments on the advantages and difficulties of semantic approaches to agent system verification. Further steps in the use of the approach proposed here would be to integrate it into an agent architecture and to see how the specification and verification of agent systems could be handled in the resulting system.

I would like to thank the reviewers of this paper for their many helpful comments.

References

1. Peter J. Cameron. *Sets, Logic and Categories*. Springer, 1999.
2. Christopher Cherniak. *Minimal Rationality*. MIT Press, 1986.
3. H-D Ebbinghaus, J Flum, and W Thomas. *Mathematical Logic*. Springer, 1984.
4. Melvin Fitting and Richard L. Mendelsohn. *First-order Modal Logic*. Kluwer Academic Publishers, 1998.
5. Cyril Lee Flax. *Algebraic Aspects of Entailment: Approximation, Belief Revision and Computability*. PhD thesis, Macquarie University, Sydney, NSW 2109, Australia, 2002. Available from http://www.comp.mq.edu.au/~flax/.
6. P. N. Johnson-Laird. *Mental Models*. Cambridge University Press, 1983.
7. P. N. Johnson-Laird and Ruth M. J. Byrne. *Deduction*. Lawrence Erlbaum Associates, Hove, 1991.
8. John L. Kelley. *General Topology*. D. Van Nostrand Company, 1955.
9. Yiannis N. Moschovakis. *Notes on Set Theory*. Springer, 1994.
10. Viggo Stoltenberg-Hansen, Ingrid Linström, and Edward R. Griffor. *Mathematical Theory of Domains*. Number 22 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1994.
11. Michael Wooldridge. *An Introduction to Multiagent Systems*. John Wiley, 2002.

Extending the Operational Semantics of a BDI Agent-Oriented Programming Language for Introducing Speech-Act Based Communication

Álvaro F. Moreira[1], Renata Vieira[2], and Rafael H. Bordini[3]

[1] Departamento de Informática Teórica
Instituto de Informática
Universidade Federal do Rio Grande do Sul
Porto Alegre RS, 91501-970, Brazil
afmoreira@inf.ufrgs.br

[2] Centro de Ciências Exatas
Universidade do Vale do Rio dos Sinos
CP 275, CEP 93022-000, São Leopoldo, RS, Brazil
renata@exatas.unisinos.br

[3] Department of Computer Science,
University of Liverpool,
Liverpool L69 7ZF, U.K.
R.Bordini@csc.liv.ac.uk

Abstract. Work on agent communication languages has since long striven to achieve adequate speech act semantics; partly, the problem is that references to an agent's architecture (in particular a BDI-like architecture) would be required in giving such semantics more rigorously. On the other hand, BDI agent-oriented programming languages have had their semantics formalised for an abstract versions only, neglecting "practical" aspects such as communication primitives; this means that, at least in what concerns communication, implementations of BDI programming languages have been *ad hoc*. This paper tackles, however preliminarily, both these problems by giving semantics to speech-act based messages received by an AgentSpeak(L) agent. AgentSpeak(L) is a BDI, agent-oriented, logic programming language that has received a great deal of attention in recent years. The work in this paper builds upon a structural operational semantics to AgentSpeak(L) that we have given in previous work. The contribution of this paper is two-fold: we here extend our earlier work on providing a solid theoretical background on which to base existing implementations of AgentSpeak(L) interpreters, as well as we shed light on a more computationally grounded approach to giving semantics for (the core) illocutionary forces used in speech-act based agent communication languages.

1 Introduction

The AgentSpeak(L) programming language was introduced by Rao in [15]. The language was quite influential in the definition of other agent-oriented programming languages. After a period of apparent disinterest, work related to AgentSpeak(L) has been done in many fronts recently, from developing interpreters [3] and applications to formal semantics and model-checking [7,4]. AgentSpeak(L) is particularly interesting, in comparison to other agent-oriented languages, in that it retains the most important aspects of the BDI-based reactive planning systems on which it was based, it has a working interpreter, and at the same time its formal semantics and relation to BDI logics [16] is being thoroughly studied [14,5,6].

In our previous work on giving operational semantics to AgentSpeak(L) [14], we considered only the main constructs of the language, as originally defined by Rao. However, Rao's definition considered an abstract programming language; for AgentSpeak(L) to be useful in practice, various extensions to it have been proposed [3]. Still, the interpreter described in that paper does not yet support communication, which is essential when it comes to engineering *multi-agent systems*. Speech act theory (see Section 3) is particularly adequate as a foundation for communication among intentional agents. Through communication, an agent can share its internal state (beliefs, desires, intentions) with other agents, as well as it can influence other agents' states. Speech-act based communication for AgentSpeak(L) agents has already been used, in a very simple way (so as to allow model checking), in [4].

This paper deals exactly with this aspect of AgentSpeak(L), by extending its operational semantics to account for the main illocutionary forces related to communicating AgentSpeak(L) agents. Our semantics tells exactly how to implement the processing of messages received by an AgentSpeak(L) agent (how its computational representation of Beliefs-Desires-Intentions are changed when a message is received). Note that in implementations of the BDI architecture, the concept of *plan* is used to simplify aspects of deliberation and knowing what course of action to take in order to achieve desired states of the world. Therefore, an AgentSpeak(L) agent *sends* a message whenever a communicative action appears in the body of an intended plan that is being executed. The important issue is then how to interpret a message that has been *received*. This is precisely the aspect of agent communication that we consider in this paper.

In extending the operational semantics of AgentSpeak(L) to account for inter-agent communication, we also touch upon another long-standing problem in the area of multi-agent systems, namely the semantics of speech acts. As Singh pointed out [19], semantics of agent communication languages such as KQML and FIPA have incurred in the mistake (for those concerned with general interoperability and legacy systems) of emphasising *mental agency*. The problem is that if the semantics makes reference to an agent believing (or intending a state satisfying) a certain proposition, there is no way to ensure that any software using that communication language complies with the expected underlying semantics of belief (or intention, or mental attitudes in general). This problem is

avoided here as the AgentSpeak(L) agents for which such speech-act based communication language is being used, are indeed agents for which a precise notion of Belief-Desire-Intention has been given [5,6]. Another way of putting this, is that we give a more "computationally grounded" [21] semantics of speech-act based agent communication.

Previous attempts to give semantics to agent communication languages, e.g. [12], were based on the "pre-conditions – action – post-conditions" approach, referring to agent mental states in modal languages based on Cohen and Levesque's work on intention [10]. Our semantics for communication, besides being more instrumental in implementations (as it serves as the specification for an interpreter), can also be used in the proof of communication properties [23]. More recently, the work reported in [11,20] also provides an operational semantics for an agent communication language. However, again they do not consider the effects of communication in terms of BDI agents. To the best of our knowledge, our work is the first to give operational semantics incorporating the core illocutionary forces in a BDI programming language.

The paper is organised as follows. Section 2 gives a very brief (informal) overview of AgentSpeak(L), and Section 3 provides the general background on speech-act based agent communication. We then present formally the syntax and semantics of AgentSpeak(L) in Section 4; this is the syntax and semantics of AgentSpeak(L) first presented in [14], and reproduced here so that the paper is self-contained. Section 5 provides the main contribution in this paper, i.e., the semantics of speech-act based communication for AgentSpeak(L) agents. Brief conclusions and plans for future work are given in the last section.

2 An Overview of AgentSpeak(L)

The AgentSpeak(L) programming language was introduced in [15]. It is a natural extension of logic programming for the BDI agent architecture, and provides an elegant abstract framework for programming BDI agents. The BDI architecture is, in turn, the predominant approach to the implementation of "intelligent" or "rational" agents [22]. An AgentSpeak(L) agent is created by the specification of a set of base beliefs and a set of plans. A *belief atom* is simply a first-order predicate in the usual notation, and belief atoms or their negations are termed *belief literals*. An *initial set of beliefs* is just a collection of ground belief atoms.

AgentSpeak(L) distinguishes two types of goals: *achievement goals* and *test goals*. Achievement and test goals are predicates (as for beliefs) prefixed with operators '`!`' and '`?`' respectively. Achievement goals state that the agent wants to achieve a state of the world where the associated predicate is true. (In practice, these initiate the execution of *subplans*.) A *test goal* returns a unification for the associated predicate with one of the agent's beliefs; they fail otherwise. A *triggering event* defines which events may initiate the execution of a plan. An *event* can be internal, when a subgoal needs to be achieved, or external, when generated from belief updates as a result of perceiving the environment. There

are two types of triggering events: those related to the *addition* ('+') and *deletion* ('-') of mental attitudes (beliefs or goals).

Plans refer to the *basic actions* that an agent is able to perform on its environment. Such actions are also defined as first-order predicates, but with special predicate symbols (called action symbols) used to distinguish them from other predicates. A plan is formed by a *triggering event* (denoting the purpose for that plan), followed by a conjunction of belief literals representing a *context*. The context must be a logical consequence of that agent's current beliefs for the plan to be *applicable*. The remainder of the plan is a sequence of basic actions or (sub)goals that the agent has to achieve (or test) when the plan, if applicable, is chosen for execution.

```
+concert(A,V) : likes(A)
   ← !book_tickets(A,V).

+!book_tickets(A,V) : ¬busy(phone)
   ← call(V);
      ...;
      !choose_seats(A,V).
```

Fig. 1. Examples of AgentSpeak(L) Plans

Figure 1 shows some examples of AgentSpeak(L) plans. They tell us that, when a concert is announced for artist `A` at venue `V` (so that, from perception of the environment, a belief `concert(A,V)` is *added*), then if this agent in fact likes artist `A`, then it will have the new goal of booking tickets for that concert. The second plan tells us that whenever this agent adopts the goal of booking tickets for `A`'s performance at `V`, if it is the case that the telephone is not busy, then it can execute a plan consisting of performing the basic action `call(V)` (assuming that making a phone call is an atomic action that the agent can perform) followed by a certain protocol for booking tickets (indicated by '...'), which in this case ends with the execution of a plan for choosing the seats for such performance at that particular venue.

3 Background on Speech-Act Based Agent Communication Languages

As BDI theory is based on the philosophical literature on practical reasoning [8], agent communication in multi-agent systems is inspired by philosophical studies on the speech act theory, in particular the work of Austin [1] and Searle [17].

Speech act theory is based on the conception of language as action, and different types of speech actions are explained according to their illocutionary force, which represents the speaker's intention for a certain semantic content.

In natural language, one has an illocutionary force associated to a utterance (or locutionary act) such as "the door is open" and another to a utterance "open the door". The former intends belief revision, whereas the latter intends a change in the plans of the hearer. As seen in the utterances above, in natural language the illocutionary force is implicit. When the theory is adapted to agent communication the illocutionary force is made explicit, to facilitate the computational processing of the communication act.

Other pragmatical factors related to communication such as social roles and conventions have been discussed in the literature [13,2,18]. Illocutionary forces may require the existence of certain relationships between speaker and hearer for them to be felicitous. A *command*, for instance, requires a subordination relation between the individuals involved in the communication, whereas such subordination is not required in a *request*. To some extent, this has been made explicit in the semantics of communicating AgentSpeak(L) agents that we give in Section 5. We require that an agent specification provides "trust" and "power" relations to other agents, as the semantics of illocutionary forces in messages being processed by that agent depends on them. This is a simple mechanism, which should incorporate more elaborate conceptions of trust and power from the literature on the subject (see, e.g., [9]).

The "Agent Communication Language" developed in the context of the "Knowledge Sharing Effort" project, attempt to define a practical agent communication language that included high level (speech-act based) communication as proposed in the distributed artificial intelligence literature. The Knowledge Query and Manipulation Language (KQML) is a language that adds intentional context to communication messages [12]. KQML "performatives" refer to illocutionary forces and they make explicit the agent intentions with a message being sent.

In this paper, as a first attempt to give semantics to communication aspects of BDI programming languages, we focus on four illocutionary forces (which are particularly relevant for AgentSpeak(L) agents), named exactly as the associated KQML performatives; we also consider two new illocutionary forces, which we introduce in Section 5. The four KQML performatives which we consider in the semantics given in this paper are listed below, where S denotes the agent that sends the message, and R denotes the agent that receives the message.

`tell:` S informs R that the sentence in the message (i.e., the message content) is true of S — that is, the sentence is in the knowledge base of S (or S believes that the content of the message is true, for BDI-like agents);
`untell:` the message content is *not* in the knowledge base of S;
`achieve:` S requests that R try to achieve a state of the world where the message content is true;
`unachieve:` S wants to revert the effect of an `achieve` previously sent.

A complete semantic representation for speech acts needs to consider action and planning theories, and to conceive communication as a special kind of action which has direct effect over the mental states of agents. Most formalisms used for

the representation of actions use the "pre-conditions – action – post-conditions" approach. For the description of mental states, most of the work in the area is based on the Cohen and Levesque's work on intention [10].

In [12], an initial attempt to introduce semantic principles to KQML was made. The communication process is defined according to the influence that messages have on the agents' beliefs and intentions. The semantics is given through the description of agent states before and after sending or receiving a message, in terms of pre- and post-conditions. Pre-conditions describe the required state for an agent to send a message and for the receiver to accept and process it. Post-conditions describe the interlocutors' states after an utterance or after receiving and processing a message.

Agent states are described through metal attitudes whose arguments may be propositions or state of the world descriptions. Mental attitudes are, e.g., belief (*bel*), knowledge (*know*), desire (*want*), and intention (*intend*). In the style introduced in [12], the semantics for $tell(S, R, X)$ (S tells R that S believes that X is true), for instance, is given as:

- Pre-conditions on the states of S and R:
  - $Pre(S)$: $bel(S, X) \wedge know(S, want(R, know(R, bel(S, X))))$
  - $Pre(R)$: $intend(R, know(R, bel(S, X)))$
- Post-conditions on S and R:
  - $Pos(S)$: $know(S, know(R, bel(S, X)))$
  - $Pos(R)$: $know(R, bel(S, X))$
- Action completion:
  - $know(R, bel(S, X))$

Post-condition and *completion* hold unless a *sorry* or *error* are returned (i.e., when R was unable to process the *tell* message).

As we mentioned in the introduction, the problem with the usual approach to giving semantics to an agent communication language is that it makes reference to an agent's mental attitudes, and there is no way to ensure that any software in general, using that communication language, complies with the expected underlying semantics of the mental attitudes. This is true of the semantic approaches to both KQML and FIPA (see [19] for a discussion). As an example, consider a legacy software wrapped in an agent that uses KQML or FIPA to interoperate with other agents. One could not prove communication properties of the system, as there is not such a thing as a precise definition of when that legacy system believes or intends a formula X, as appears in the semantics of the communication language. In our approach, it is possible to prove such properties given that in [5,6] we gave a precise definition of what it means for an AgentSpeak(L) agent to believe, desire, or intend a certain formula.

4 Syntax and Semantics of AgentSpeak(L)

This section presents the syntax and semantics of AgentSpeak(L), as originally in [14] and then used in [5,6]. The semantics is given in the style of Plotkin's structural operational semantics, a standard notation for semantics of programming languages.

4.1 Abstract Syntax

An AgentSpeak(L) agent specification ag is given by the following grammar:

$$
\begin{array}{lcll}
ag & ::= & bs \quad ps & \\
bs & ::= & b_1 \ldots b_n & (n \geq 0) \\
at & ::= & \mathtt{P}(t_1, \ldots, t_n) & (n \geq 0) \\
ps & ::= & p_1 \ldots p_n & (n \geq 1) \\
p & ::= & te : ct \leftarrow h & \\
te & ::= & +at \mid -at \mid +g \mid -g & \\
ct & ::= & at \mid \neg at \mid ct \wedge ct \mid \mathsf{T} & \\
h & ::= & a \mid g \mid u \mid h; h & \\
a & ::= & \mathtt{A}(t_1, \ldots, t_n) & (n \geq 0) \\
g & ::= & !at \mid ?at & \\
u & ::= & +at \mid -at &
\end{array}
$$

In AgentSpeak(L), an agent is simply specified by a set bs of beliefs (the agent's initial belief base) and a set ps of plans (the agent's plan library). The atomic formulæ at of the language are predicates where $\mathtt{P}$ is a predicate symbol and $t_1, \ldots, t_n$ are standard terms of first order logic. We call a *belief* an atomic formula at with no variables and we use b as a metavariable for beliefs. The set of initial beliefs of an AgentSpeak(L) program is a sequence of beliefs bs.

A plan in AgentSpeak(L) is given by p above, where te is the *triggering event*, ct is the plan's context, and h is sequence of actions, goals, or belief updates; $te : ct$ is referred as the *head* of the plan, and h is its *body*. Then the set of plans of an agent is given by ps as a list of plans. Each plan has in its head a formula ct that specifies the conditions under which the plan can be executed. The formula ct must be a logical consequence of the agent's beliefs if the plan is to be considered applicable.

A triggering event te can then be the addition or the deletion of a belief from an agent's belief base ($+at$ and $-at$, respectively), or the addition or the deletion of a goal ($+g$ and $-g$, respectively). A sequence h of actions, goals, and belief updates defines the body of a plan. We assume the agent has at its disposal a set of *actions* and we use a as a metavariable ranging over them. They are given as normal predicates except that an action symbol $\mathtt{A}$ is used instead of a predicate symbol. Goals g can be either *achievement goals* ($!at$) or *test goals* ($?at$). Finally, $+at$ and $-at$ (in the body of a plan) represent operations for updating (u) the belief base by, respectively, adding and removing at.

4.2 Semantics

An agent and its circumstance form a configuration of the transition system giving operational semantics to AgentSpeak(L). The transition relation:

$$\langle ag, C \rangle \longrightarrow \langle ag', C' \rangle$$

is defined by the semantic rules given in the next section.

An agent's circumstance C is a tuple $\langle I, E, A, R, Ap, \iota, \rho, \varepsilon \rangle$ where:

– I is a set of *intentions* $\{i, i', \ldots\}$. Each intention i is a stack of partially instantiated plans.
– E is a set of *events* $\{(te, i), (te', i'), \ldots\}$. Each event is a pair (te, i), where te is a triggering event and the plan on top of intention i is the one that generated te.
When the belief revision function, which is not part of the AgentSpeak(L) interpreter but rather of the general architecture of the agent, updates the belief base, the associated events are included in this set.
– A is a set of *actions* to be performed in the environment.
An action expression included in this set tells other architecture components to actually perform the respective action on the environment, thus changing it.
– R is a set of *relevant plans*. In Definition 1 below we state precisely how the set of relevant plans is obtained.
– Ap is a set of *applicable plans*. The way this set is obtained is given in Definition 2 below.
– Each circumstance C also has three components called ι, ε, and ρ. They keep record of a particular intention, event and applicable plan (respectively) being considered along the execution of an agent.

Auxiliary Functions

We define some auxiliary syntactic functions to be used in the semantics. If p is a plan of the form $te : ct \leftarrow h$, we define $\mathsf{TrEv}(p) = te$ and $\mathsf{Ctxt}(p) = ct$, which retrieve the triggering event and the context of the plan, respectively. We use these to define the auxiliary functions below, which will be needed in the semantic rules.

A plan is considered relevant in relation to a triggering event if it has been written to deal with that event. In practice, that is verified by trying to unify the triggering event part of the plan with the triggering event that has been selected from E for treatment. In the definition bellow, we write mgu for the procedure that computes the most general unifying substitution of two triggering events.

Definition 1. *Given the plans ps of an agent and a triggering event te, the set $\mathsf{RelPlans}(ps, te)$ of relevant plans is given as follows:*

$$\mathsf{RelPlans}(ps, te) = \{p\theta \mid p \in ps \ \wedge \ \theta = \mathsf{mgu}(te, \mathsf{TrEv}(p))\}.$$

A plan is applicable if it is both relevant and its context is a logical consequence of the agent's beliefs.

Definition 2. *Given a set of relevant plans R and the beliefs bs of an agent, the set of applicable plans $\mathsf{AppPlans}(bs, R)$ is defined as follows:*

$$\mathsf{AppPlans}(bs, R) = \{p\theta \mid p \in R \ \wedge \ \theta \text{ is s.t. } bs \models \mathsf{Ctxt}(p)\theta\}.$$

An agent can also perform a test goal. The evaluation of a test goal $?at$ consists in testing if the formula at is a logical consequence of the agent's beliefs. One of the effects of this test is the production of a set of substitutions:

Definition 3. *Given the beliefs bs of an agent and a formula at, the set of substitutions* $\mathsf{Test}(bs, at)$ *produced by testing at against bs is defined as follows:*

$$\mathsf{Test}(bs, at) = \{\theta \mid bs \models at\theta\}.$$

Notation

In order to keep the semantic rules neat, we adopt the following notations:

- If C is an AgentSpeak(L) agent circumstance, we write C_E to make reference to the component E of C. Similarly for all the other components of C.
- We write $C_\iota = \_$ (the underline symbol) to indicate that there is no intention being considered in the agent's execution. Similarly for C_ρ and C_ϵ.
- We use $i, i', \ldots$ to denote intentions, and we write $i[p]$ to denote an intention that has plan p on its top, i being the remaining plans in that intention.

We use the following notation for AgentSpeak(L) selection functions: S_E for the event selection function, S_{Ap} for the applicable plan selection function, and S_I for the intention selection function.

The semantic rules are given in Appendix A. They appeared in [14,5], but we include them here so that the paper is self-contained.

5 Semantics of Communicating AgentSpeak(L) Agents

In order to endow AgentSpeak(L) agents with the capability of processing communication messages, we first change the syntax of atomic propositions so that we can annotate, for each belief, what is its source. This annotation mechanism provides a very neat notation for making explicit the sources of an agent's belief. It has advantages in terms of expressive power and readability, besides allowing the use of such explicit information in an agent's reasoning (i.e., in selecting plans for achieving goals).

By using this information source annotation mechanism, we also clear up some practical problems in the implementation of AgentSpeak(L) interpreters concerning internal beliefs (the ones added during the execution of a plan). In the interpreter reported in [3], we temporarily dealt with the problem by creating a separate belief base where the internal beliefs are included or removed; this extra belief base, together with the current list of percepts, is then used in the belief revision process.

The following new grammar rule is used instead of the one given in Section 4, so that we can annotate each *atomic proposition* with its source: either a term identifying which was the agent in the society (id) that previously sent the information in a message, `self` to denote internal beliefs, or `percept` to indicate that the belief was acquired through perception of the environment.

$$at \quad ::= \quad \mathtt{P}(t_1, \ldots, t_n)[an_1, \ldots, an_m]$$

where $n \geq 0$, $m > 0$, and $an_i \in \{\mathtt{percept}, \mathtt{self}, id\}$, $0 \leq i \leq m$.

Note that with this new language construct, it is possible to make sure, in a plan context, what was the source of a belief before using that plan as an intended means. All of these details that are consequence of the new syntax that we introduce here are in fact hidden in the mgu function used in the auxiliary functions of the semantics in Section 4, as well as in the logical consequence relation that is also referred to in the semantics. We intend to make such details more clear in future work.

Next, we need to change the definition of an agent's circumstance, as we now need a set M which represents an agent's mail box. As usual in practice (and used, e.g., in [4]), we assume that the implementation of the AgentSpeak(L) interpreter provides, as part of the overall agent architecture, a mechanism for receiving and sending messages asynchronously; messages are stored in an mail box and one of them is processed by the agent at the beginning of a reasoning cycle. The format of those messages is $\langle \mathit{Ilf}, id, \mathit{content}\rangle$, where $\mathit{Ilf} \in \{\mathit{Tell}, \mathit{Untell}, \mathit{Achieve}, \mathit{Unachieve}, \mathit{TellHow}, \mathit{UntellHow}\}$ is the illocutionary force associated with the message, id identifies the agent that sent the message (as mentioned above), and *content* is the message content, which can be either an atomic proposition (at) or a plan (p).

An agent's circumstance C is now defined as a tuple $\langle I, E, M, A, R, Ap, \iota, \rho, \varepsilon\rangle$ where M is the set of messages that the agent has received and has not processed yet, and everything else is as in Section 4. For processing messages, a new selection function is necessary, which operates in the same way as the three selection functions described in the previous section as well. The new selection function is called S_M, and selects one particular message from M.

Further, in processing messages we now need two more "given" functions, in the same way that the selection functions are assumed as given in an agent's specification. $\mathsf{Trust}(id)$ returns true if id identifies a trusted information source. It is used to decide whether *Tell* messages will be processed at all. Even though we annotate the information source when a belief is acquired from communication, the agent may ignore messages arriving from untrusted agents. $\mathsf{Power}(id)$, is true if the agent has a subordination relation towards agent id. In that case, messages of type *Achieve* should be respected. The idea of having user-defined "trust" and "power" functions has already been used in practice in [4].

We now extend the semantics to cope with the processing of speech-act based messages received by an AgentSpeak(L) agent. The new semantic rules are as follows.

Receiving a Tell Message

$$\textbf{TellRec}\ \frac{S_M(C_M) = \langle \mathit{Tell}, id, at\rangle \quad \mathsf{Trust}(id)}{\langle ag, C\rangle \longrightarrow \langle ag', C'\rangle}$$

$$\begin{aligned} \textit{where: } C'_M &= C_M - \{\langle \mathit{Tell}, id, at\rangle\} \\ bs' &\models \begin{cases} at[\mathit{sources} \cup \{id\}], & \text{if } bs \models at[\mathit{sources}] \\ at[id], & \text{otherwise} \end{cases} \\ C'_E &= C_E \cup \{\langle +at[id], \top\rangle\} \end{aligned}$$

The content of the message is added to the belief base in case it was not there previously. If the information is already there, the sender of the message is included in the set of sources giving accreditation to that belief.

Receiving an Untell Message

$$\textbf{UnTellRec}\ \frac{S_M(C_M) = \langle Untell, id, at\rangle \quad \mathsf{Trust}(id)}{\langle ag, C\rangle \longrightarrow \langle ag', C'\rangle}$$

$$\begin{array}{rlll} \textit{where: } C'_M & = C_M - \{\langle Untell, id, at\rangle\} & \\ & bs' \not\models at[id], & \text{if } bs \models at[id] \\ & bs' \models at[sources - \{id\}], & \text{if } bs \models at[sources] \\ C'_E & = C_E \cup \{\langle -at[id], \top\rangle\} & \end{array}$$

The sender of the message is removed from the set of sources giving accreditation to the belief. If the sender was the only source for that information, the belief is removed from the belief base.

Receiving an Achieve Message

$$\textbf{AchieveRec}\ \frac{S_M(C_M) = \langle Achieve, id, at\rangle \quad \mathsf{Power}(id)}{\langle ag, C\rangle \longrightarrow \langle ag, C'\rangle}$$

$$\begin{array}{rl} \textit{where: } C'_M & = C_M - \{\langle Achieve, id, at\rangle\} \\ C'_E & = C_E \cup \{\langle +!at, \top\rangle\} \end{array}$$

If the sender has power over the AgentSpeak(L) agent, the agent will try to execute a plan whose triggering event is $+!at$; that is, it will try to achieve the goal associated with the propositional content of the message. All that needs to be done is to include an external event in the set of events (recall that external events have the triggering event associated with the true intention $\top$).

Note that, interestingly, now it is possible to have a new focus of attention (each of the stacks of plans in the set of intentions I) being started by an addition (or deletion, see below) of an achievement goal. Originally, only a change of belief from perception of the environment started a new focus of attention; the plan chosen for that event could, in turn, have achievement goals in its body, thus pushing new plans onto the stack.

Receiving an Unachieve Message

$$\textbf{UnAchieveRec}\ \frac{S_M(C_M) = \langle Unachieve, id, at\rangle \quad \mathsf{Power}(id)}{\langle ag, C\rangle \longrightarrow \langle ag, C'\rangle}$$

$$\begin{array}{rl} \textit{where: } C'_M & = C_M - \{\langle Unachieve, id, at\rangle\} \\ C'_E & = C_E \cup \{\langle -!at, \top\rangle\} \end{array}$$

Similarly to the previous rule, except that now a deletion (rather than addition) of achievement goal is included in the set of events. If the agent has a

plan with such triggering event, that plan should handle all aspects of dropping an intention. However, doing so in practice may require the alteration of the set of intentions, thus requiring special mechanisms which are not available in AgentSpeak(L) as yet (neither formally, nor in implemented AgentSpeak(L) interpreters, to the best of our knowledge).

Receiving a Tell-How Message

$$\textbf{TellHowRec}\ \frac{S_M(C_M) = \langle \mathit{TellHow}, id, p\rangle \quad \mathsf{Trust}(id)}{\langle ag, C\rangle \longrightarrow \langle ag', C'\rangle}$$

$$\begin{aligned} \textit{where: } C'_M &= C_M - \{\langle \mathit{TellHow}, id, p\rangle\} \\ ps' &= ps \cup \{p\} \end{aligned}$$

The concept of plans in reactive planning systems such as those defined by AgentSpeak(L) agents is associated with Singh's notion of know-how [18]. Accordingly, we use the *TellHow* performative when an external source wants to inform an AgentSpeak(L) agent of a plan it uses for handling certain types of events (given by the plan's triggering event). If the source is trusted, the plan (which is in the message content) is simply added to the agent's plan library.

Receiving an Untell-How Message

$$\textbf{UntellHowRec}\ \frac{S_M(C_M) = \langle \mathit{UntellHow}, id, p\rangle \quad \mathsf{Trust}(id)}{\langle ag, C\rangle \longrightarrow \langle ag', C'\rangle}$$

$$\begin{aligned} \textit{where: } C'_M &= C_M - \{\langle \mathit{UntellHow}, id, p\rangle\} \\ ps' &= ps - \{p\} \end{aligned}$$

Similarly to the rule above. An external source may find that a plan is no longer valid, or efficient, for handling the events it was supposed to handle. It may then want to inform an AgentSpeak(L) agent of that fact. Thus, when receiving and *UntellHow*, the agent drops the associated plan (in the message content) from its plan library.

6 Conclusion

We have given formal semantics to the processing of speech-acted based messages received by an AgentSpeak(L) agent. The operational semantics we have used in previous work proved quite handy: in this extension, all we had to do (apart from minor changes in the syntax of the language and in the configuration of the transition system) was to provide new semantic rules, one for each illocutionary force used in the communication language. In giving semantics to communicating AgentSpeak(L) agents, we have provided the means for precise implementation of AgentSpeak(L) interpreters with such functionality, as well as given a more computationally grounded semantics of speech-act based agent communication.

If on one hand Singh's [19] proposal for a social-agency based semantics may be the best way towards giving semantics to general purpose agent communication languages such as FIPA or KQML, on the other hand, within the context of a BDI agent programming language, a mental agency approach to semantics of communication can be used without any of the drawbacks pointed out by Singh.

Future work should consider other performatives (in particular *AskIf*, *AskAll*, *Reply*, and for plans *AskHow* and *ReplyHow*), as well as giving a better formal treatment of information sources (that are annotated to atomic propositions) — unification and logical consequence of annotated atomic propositions was not formalised in this paper. Further communication aspects such as ontological agreement among AgentSpeak(L) agents, and reasoning about information sources (e.g., in executing test goals or choosing plans based on those annotations) should also be considered in future work. We also expect that this work will be used when speech-act based communication is implemented in existing AgentSpeak(L) interpreters.

References

1. J. L. Austin. *How to Do Things with Words.* Oxford University Press, London, 1962.
2. T. T. Ballmer and W. Brennenstuhl. *Speech Act Classification: A Study in the Lexical Analysis of English Speech Activity Verbs.* Springer-Verlag, Berlin, 1981.
3. R. H. Bordini, A. L. C. Bazzan, R. O. Jannone, D. M. Basso, R. M. Vicari, and V. R. Lesser. AgentSpeak(XL): Efficient intention selection in BDI agents via decision-theoretic task scheduling. In C. Castelfranchi and W. L. Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2002), 15–19 July, Bologna, Italy*, pages 1294–1302, New York, NY, 2002. ACM Press.
4. R. H. Bordini, M. Fisher, C. Pardavila, and M. Wooldridge. Model checking AgentSpeak. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2003), Melbourne, Australia, 14–18 July*, 2003. To appear.
5. R. H. Bordini and Á. F. Moreira. Proving the asymmetry thesis principles for a BDI agent-oriented programming language. In J. Dix, J. A. Leite, and K. Satoh, editors, *Proceedings of the Third International Workshop on Computational Logic in Multi-Agent Systems (CLIMA-02), 1st August, Copenhagen, Denmark*, Electronic Notes in Theoretical Computer Science 70(5). Elsevier, 2002. URL: <http://www.elsevier.nl/locate/entcs/volume70.html>. CLIMA-02 was held as part of FLoC-02. This paper was originally published in Datalogiske Skrifter number 93, Roskilde University, Denmark, pages 94–108.
6. R. H. Bordini and Á. F. Moreira. Proving BDI properties of agent-oriented programming languages: The asymmetry thesis principles in AgentSpeak(L). *Annals of Mathematics and Artificial Intelligence*, 2003. Accepted for publication in a Special Issue on Computational Logic and Multi-Agency.
7. R. H. Bordini, W. Visser, M. Fisher, C. Pardavila, and M. Wooldridge. Model checking multi-agent programs with CASP. In *Proceedgins of the Fifteenth Conference on Computer-Aided Verification (CAV-2003), Boulder, CO, 8–12 July*, 2003. Tool description. To appear.

8. M. E. Bratman. *Intentions, Plans and Practical Reason*. Harvard University Press, Cambridge, MA, 1987.
9. C. Castelfranchi and R. Falcone. Principles of trust for MAS: Cognitive anatomy, social importance, and quantification. In Y. Demazeau, editor, *Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS'98), Agents' World, 4–7 July, Paris*, pages 72–79, Washington, 1998. IEEE Computer Society Press.
10. P. R. Cohen and H. J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42(3):213–261, 1990.
11. F. S. de Boer, R. M. van Eijk, W. Van Der Hoek, and J.-J. C. Meyer. Failure semantics for the exchange of information in multi-agent systems. In C. Palamidessi, editor, *Eleventh International Conference on Concurrency Theory (CONCUR 2000), University Park, PA, 22–25 August*, number 1877 in LNCS, pages 214–228. Springer-Verlag, 2000.
12. Y. Labrou and T. Finin. A semantics approach for KQML—a general purpose communication language for software agents. In *Proceedings of the Third International Conference on Information and Knowledge Management (CIKM'94)*. ACM Press, Nov. 1994. URL: `http://www.cs.umbc.edu/kqml/papers/`.
13. S. C. Levinson. The essential inadequacies of speech act models of dialogue. In H. Parret, M. Sbisa, and J. Verschuren, editors, *Possibilities and limitations of pragmatics: Proceedings of the Conference on Pragmatics at Urbino, July, 1979*, pages 473–492. Benjamins, Amsterdam, 1981.
14. Á. F. Moreira and R. H. Bordini. An operational semantics for a BDI agent-oriented programming language. In *Proceedings of the Workshop on Logics for Agent-Based Systems (LABS-02), held in conjunction with the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR2002), April 22–25, Toulouse, France*, pages 45–59, 2002.
15. A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. Van de Velde and J. Perram, editors, *Proceedings of the Seventh Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96), 22–25 January, Eindhoven, The Netherlands*, number 1038 in LNAI, pages 42–55, London, 1996. Springer-Verlag.
16. A. S. Rao and M. P. Georgeff. Decision procedures for BDI logics. *Journal of Logic and Computation*, 8(3):293–343, 1998.
17. J. R. Searle. *Speech Acts: An Essay in the Philosophy of Language*. Cambridge University Press, Cambridge, 1969.
18. M. P. Singh. *Multiagent Systems—A Theoretic Framework for Intentions, Know-How, and Communications*. Number 799 in LNAI. Springer-Verlag, Berlin, 1994.
19. M. P. Singh. Agent communication languages: Rethinking the principles. *IEEE Computer*, 31(12):40–47, December 1998.
20. R. M. van Eijk, F. S. de Boer, W. Van Der Hoek, and J.-J. C. Meyer. A verification framework for agent communication. *Autonomous Agents and Multi-Agent Systems*, 6(2):185–219, 2003.
21. M. Wooldridge. Computationally grounded theories of agency. In E. Durfee, editor, *Proceedings of the Fourth International Conference on Multi-Agent Systems (ICMAS-2000), 10–12 July, Boston*, pages 13–20, Los Alamitos, CA, 2000. IEEE Computer Society. Paper for an Invited Talk.
22. M. Wooldridge. *Reasoning about Rational Agents*. The MIT Press, Cambridge, MA, 2000.
23. M. Wooldridge. Semantic issues in the verification of agent communication languages. *Autonomous Agents and Multi-Agent Systems*, 3(1):9–31, 2000.

A Semantic Rules

Event Selection: The rule below assumes the existence of a selection function S_E that selects events from a set of events E. The selected event is removed from E and it is assigned to the ϵ component of the circumstance.

$$\textbf{SelEv}\ \frac{S_E(C_E) = \langle te, i\rangle}{\langle ag, C\rangle \longrightarrow \langle ag, C'\rangle} \quad \sharp\ C_\varepsilon = \_\ ,\quad C_{Ap} = C_R = \{\}$$
$$\textit{where: } C'_E = C_E - \langle te, i\rangle$$
$$C'_\varepsilon = \langle te, i\rangle$$

Relevant Plans: The rule $\textbf{Rel}_\textbf{1}$ initialises the R component with the set of relevant plans. If no plan is relevant, the event is discarded from ϵ by $\textbf{Rel}_\textbf{2}$.

$$\textbf{Rel}_\textbf{1}\ \frac{\mathsf{RelPlans}(ps, te) \neq \{\}}{\langle ag, C\rangle \longrightarrow \langle ag, C'\rangle}$$
$$\sharp\ C_\varepsilon = \langle te, i\rangle\ \ C_{Ap}, C_R = \{\}$$
$$\textit{where: } C'_R = \mathsf{RelPlans}(ps, te)$$

$$\textbf{Rel}_\textbf{2}\ \frac{\mathsf{RelPlans}(ps, te) = \{\}}{\langle ag, C\rangle \longrightarrow \langle ag, C'\rangle}$$
$$\sharp\ C_\varepsilon = \langle te, i\rangle\ \ C_{Ap}, C_R = \{\}$$
$$\textit{where: } C'_\varepsilon = \_$$

Applicable Plans: The rule $\textbf{Appl}_\textbf{1}$ initialises the Ap component with the set of applicable plans. If no plan is applicable, the event is discarded from ϵ by $\textbf{Appl}_\textbf{2}$. In either case the relevant plans are also discarded.

$$\textbf{Appl}_\textbf{1}\ \frac{\mathsf{AppPlans}(bs, C_R) \neq \{\}}{\langle ag, C\rangle \longrightarrow \langle ag, C'\rangle}$$
$$\sharp\ C_\varepsilon \neq \_\ , C_{Ap} = \{\}, C_R \neq \{\}$$
$$\textit{where: } C'_R = \{\}$$
$$C'_{Ap} = \mathsf{AppPlans}(bs, C_R)$$

$$\textbf{Appl}_\textbf{2}\ \frac{\mathsf{AppPlans}(bs, C_R) = \{\}}{\langle ag, C\rangle \longrightarrow \langle ag, C'\rangle}$$
$$\sharp\ C_\varepsilon \neq \_\ , C_{Ap} = \{\}, C_R \neq \{\}$$
$$\textit{where: } C'_R = \{\}$$
$$C'_\varepsilon = \_$$
$$C'_E = C_E \cup \langle te, i\rangle$$

Selection of Applicable Plan: This rule assumes the existence of a selection function S_{Ap} that selects a plan from a set of applicable plans Ap. The plan selected is then assigned to the ρ component of the circumstance and the set of applicable plans is discarded.

$$\textbf{SelAppl}\ \frac{S_{Ap}(C_{Ap}) = p}{\langle ag, C\rangle \longrightarrow \langle ag, C'\rangle} \quad \sharp\ C_\varepsilon \neq \_\ , C_{Ap} \neq \{\}$$
$$\textit{where: } C'_\rho = p$$
$$C'_{Ap} = \{\}$$

Preparing the Set of Intentions: Events can be classified as external or internal (depending one whether they were generated from the agent's perception, or whether they were generated by the previous execution of other plans, respectively). Rule **ExtEv** says that if the event ϵ is external (which is indicated by T in the intention

associated to ϵ) a new intention is created and its single plan is the plan p annotated in the ρ component. If the event is internal, rule **IntEv** says that the plan in ρ should be put on top of the intention associated with the event. Either way, both the event and the plan can be discarded from the ϵ and ι components, respectively.

$$\textbf{ExtEv}\ \frac{}{\langle ag, C\rangle \longrightarrow \langle ag, C'\rangle} \quad \sharp\ C_\varepsilon = \langle te, \mathsf{T}\rangle,\ \ C_\rho = p \quad \textit{where: } C'_I = C_I \cup \{\ [p]\ \},\ C'_\varepsilon = C'_\rho = \_$$

$$\textbf{IntEv}\ \frac{}{\langle ag, C\rangle \longrightarrow \langle ag, C'\rangle} \quad \sharp\ C_\varepsilon = \langle te, i\rangle,\ \ C_\rho = p \quad \textit{where: } C'_I = C_I \cup \{\ i[p]\ \},\ C'_\varepsilon = C'_\rho = \_$$

Note that, in rule **IntEv**, the whole intention i that generated the internal event needs to be inserted back in C_I, with p on its top. This is related to suspended intentions, in rule **Achieve**.

Intention Selection: This rule uses a function that selects an intention (i.e., a stack of plans) for processing.

$$\textbf{IntSel}\ \frac{S_I(C_I) = i}{\langle ag, C\rangle \longrightarrow \langle ag, C'\rangle} \quad \sharp\ C_\iota = \_ \quad \textit{where: } C'_\iota = i$$

Executing the Body of Plans: This group of rules expresses the effects of executing the body of plans. The plan being executed is always the one on the top of the intention that has been previously selected. Observe that all the rules in this group discard the intention ι. After that, another intention can be eventually selected.

– *Basic Actions:* the action a on the body of the plan is added to the set of actions A. The action is removed from the body of the plan and the intention is updated to reflect this removal.

$$\textbf{Action}\ \frac{}{\langle ag, C\rangle \longrightarrow \langle ag, C'\rangle} \quad \sharp\ C_\iota = i[head \leftarrow a\,;h] \quad \textit{where: } C'_\iota = \_,\ C'_A = C_A \cup \{a\},\ C'_I = (C_I - \{C_\iota\}) \cup \{i[head \leftarrow h]\}$$

– *Achievement Goals:* this rule registers a new internal event in the set of events E. This event can then be eventually selected (see rule **SelEv**).

$$\textbf{Achieve}\ \frac{}{\langle ag, C\rangle \longrightarrow \langle ag, C'\rangle} \quad \sharp\ C_\iota = i[head \leftarrow\ !at\,;h] \quad \textit{where: } C'_\iota = \_,\ C'_E = C_E \cup \{\langle +!at, C_\iota\rangle\},\ C'_I = C_I - \{C_\iota\}$$

Note how the intention that generated the internal event is removed from the set of intentions C_I. This denotes the idea of *suspended intentions* (see [5] for details).

– *Test Goals:* these rules are used when a test goal $?at$ should be executed. Both rules try to produce a set of substitutions that can make at a logical consequence of the agent's beliefs. The rule **Test$_1$** says basically that nothing is done if no substitution is found, and the rule **Test$_2$** says that one of the substitutions is applied to the plan.

$$\textbf{Test}_1\ \frac{\mathsf{Test}(bs, at) = \{\}}{\langle ag, C\rangle \longrightarrow \langle ag, C'\rangle}$$
$$\sharp\ C_\iota = i[head \leftarrow ?at; h]$$
$$\textit{where: } C'_\iota = \_$$
$$C'_I = (C_I - \{C_\iota\}) \cup \{i[head \leftarrow h]\}$$

$$\textbf{Test}_2\ \frac{\mathsf{Test}(bs, at) \neq \{\}}{\langle ag, C\rangle \longrightarrow \langle ag, C'\rangle}$$
$$\sharp\ C_\iota = i[head \leftarrow ?at; h]$$
$$\textit{where: } C'_\iota = \_$$
$$C'_I = (C_I - \{C_\iota\}) \cup \{i[(head \leftarrow h)\theta]\}$$
$$\theta \in \mathsf{Test}(bs, at)$$

– *Updating Beliefs:* rule **AddBel** simply adds a new event to the set of events E. The formula $+b$ is removed from the body of the plan and the set of intentions is updated properly. Rule **DelBel** works similarly. In both rules, the set of beliefs of the agent should be modified in a way that either the predicate b follows from the new set of beliefs (rule **AddBel**) or it does not (rule **DelBel**).

$$\textbf{AddBel}\ \frac{}{\langle ag, C\rangle \longrightarrow \langle ag', C'\rangle}$$
$$\sharp\ C_\iota = i[head \leftarrow +b; h]$$
$$\textit{where: } C'_\iota = \_$$
$$bs' \models b$$
$$C'_E = C_E \cup \{\langle +b, C_\iota\rangle\}$$
$$C'_I = (C_I - \{C_\iota\}) \cup \{i[head \leftarrow h]\}$$

$$\textbf{DelBel}\ \frac{}{\langle ag, C\rangle \longrightarrow \langle ag', C'\rangle}$$
$$\sharp\ C_\iota = i[head \leftarrow -b; h]$$
$$\textit{where: } C'_\iota = \_$$
$$bs' \not\models b$$
$$C'_E = C_E \cup \{\langle -b, C_\iota\rangle\}$$
$$C'_I = (C_I - \{C_\iota\}) \cup \{i[head \leftarrow h]\}$$

Removing intentions: The two rules below can be seen as "clearing house" rules. The rule **ClearInt$_1$** simply removes an intention from the set of intentions of an agent when there is nothing left (goal or action) in that intention. The rule **ClearInt$_2$** removes from the intention what is left from the plan that had been put on the top of the intention on behalf of the achievement goal $!at$ (which is also removed as it has been accomplished).

$$\textbf{ClrInt}_1\ \frac{}{\langle ag, C\rangle \longrightarrow \langle ag, C'\rangle}$$
$$\sharp\ C_\iota = [head \leftarrow]$$
$$\textit{where: } C'_\iota = \_$$
$$C'_I = C_I - \{C_\iota\}$$

$$\textbf{ClrInt}_2\ \frac{}{\langle ag, C\rangle \longrightarrow \langle ag, C'\rangle}$$
$$\sharp\ C_\iota = i'[head' \leftarrow !at; h'][head \leftarrow]$$
$$\textit{where: } C'_\iota = \_$$
$$C'_I = (C_I - \{C_\iota\}) \cup \{i'[head' \leftarrow h']\}$$

Coo-BDI: Extending the BDI Model with Cooperativity

Davide Ancona and Viviana Mascardi

DISI - Università di Genova
Via Dodecaneso 35, 16146, Genova, Italy
{davide,mascardi}@disi.unige.it

Abstract. We extend the BDI architecture with the notion of cooperativity. Agents can cooperate by exchanging and sharing plans in a quite flexible way. As a main result Coo-BDI promotes adaptivity and sharing of resources; as a by product, it provides a better support for dealing with the situation when agents do not possess their own procedural knowledge for processing a given event.

1 Introduction

Intelligent agents [20, 9] are a powerful abstraction for conceptualizing and modeling complex systems in a clear and intuitive way. For this reason many declarative agent models have been proposed in the last years. One of the most successful is the Belief, Desire and Intention (BDI) one [17], whose wide appreciation is witnessed by the development of a BDI logic [18], the definition of BDI-based languages (AgentTalk[1], 3APL [6], AgentSpeak(L) [16]) and the creation of BDI-based development tools such as JACK [4], PRS [12] and dMARS [7].

Despite this large consensus, however, many BDI concepts are not interpreted in a uniform way. To make an example, let us consider the notion of "goals", namely, consistent sets of desires. For many authors, they should be explicitly represented in order to reason about their properties and ensure their consistence. This explicit representation is not available in current BDI systems and languages such as dMARS, AgentSpeak(L) and JACK where goals, if represented at all, have only a transient representation as a type of event. A similar situation takes place with events (or subgoals, since they are considered the same entity in implemented systems) which cannot be dealt with because of the lack of plans suitable for managing them. In the original specification of the BDI architecture and in many high level BDI languages this case is not considered at all, and implemented systems solve it in very different ways: in most of them, the event or subgoal is simply dropped; in other ones the definition of a default "catch-all" plan is required. In this paper we propose an extension to the basic BDI model, for

1. representing events, goals, and the relationships between them in a conceptually clear way, and

[1] http://goanna.cs.rmit.edu.au/~winikoff/agenttalk

2. allowing agents to cooperate by exchanging plans to cope with the case no local applicable plans are available for managing an event (or goal, or subgoal).

The idea behind the first extension is to keep events and goals (that we will call *desires* for uniformity with the Belief, Desires, Intentions acronym) separate. The agent has an event queue containing events generated by the environment or by other agents. Managing an event means removing it from the event queue and generating the set of desires associated with the event. For example, the event "*reception of an e-mail asking to join friends for a drink at the Madeleine Café at 7 p.m.*" may generate the desires "*cancel all the appointments for this evening*" and "*reach the Madeleine Café at 7 p.m.*". Desires are maintained as explicit information in an ad hoc structure.

Let us go on with the previous example, and let us suppose that the agent desires to reach the Madeleine Café at 7 p.m., but it has no procedural knowledge (namely, no plans) for achieving this desire. In the existing BDI languages and systems, the agent would give up or would apply a default plan. In the extension we propose, the agent asks to other agents if they have a good plan for reaching the Madeleine Café. Some of the agents contacted for advice could cooperate by suggesting plans for achieving this desire, allowing the requiring agent to reach its friends.

Our extension allows to implement agents that dynamically change their behavior by cooperating with other agents. This feature turns out to be useful in many application fields such as:

Personal Digital Assistants (PDAs). The physical resources of a PDA are usually limited and for the PDA technology to be really effective, dynamic linking of code is a fundamental aspect. If the traditional BDI approach is adopted, modeling dynamic linking is not easy: BDI agents are expected neither to possess the capability of looking for external plans nor to dynamically extend their plan library with the plans retrieved from other agents. Our extension can cope with both issues and allows an agent to discard the external plans after their usage: this may prove useful when the space resources of the PDA have strict bounds.

Self-repairing agents. By self-repairing agent we mean an agent situated in a dynamically changing software environment and able to identify the portions of its code that should be updated to ensure its correct functioning in the evolving environment. When the agent finds out pieces of code that have become obsolete, the agent replaces them with new code, provided by a server agent, without needing to stop its activity and the activity of the whole system. Coo-BDI allows to dynamically replace local plans with external ones and thus can be used for modeling such a self-repairing agent.

Digital butlers. A "digital butler" is an agent that assists the user in some task such as managing her/his agenda, filtering incoming mail, retrieving interesting information from the web. A typical feature of a digital butler is its ability to dynamically adapt its behavior to the user needs. This ability is achieved by cooperating both with more experienced digital butlers and with the assisted user. In our setting we may think that, through a user friendly interface, the

user may train the agent by showing the right sequence of actions to perform in particular situations. The agent may treat this sequence as an externally retrieved plan which enriches its plan library, as well as plans retrieved from peer agents.
More in general, our extension is suitable for modeling all kinds of "learning agents" which learn from the interaction with other agents.

The structure of the paper is the following: in Section 2 we recall the main BDI concepts and survey the common approaches to the "no applicable plans" problem; in Section 3 we introduce Coo-BDI, our BDI model extended with *cooperativity* among agents for sharing plans, and then provide its full specification. Section 4 concludes.

2 The BDI Model

The BDI model is characterized by the following concepts:

– *Beliefs*: the agent's knowledge about the world.

– *Desires*: objectives to be accomplished; are similar to goals, but they do not need to be consistent.

– *Intentions*: plans currently under execution.

– *Plans*: "recipes" representing the procedural knowledge of the agent. They are usually characterized by a trigger which fires the adoption of the plan, a precondition that the current state must satisfy for the plan to be applicable, a body of actions to perform, an invariant condition that must hold during all the plan execution, a set of actions to be executed if the plan execution terminates successfully and a set of actions to be executed in case of plan failure. Plans are static: they do not change during the agent's life cycle.

All BDI systems also include an *event queue* where external events (perceived from the environment) and internal subgoals (generated by the agent itself while trying to achieve a main goal) are stored.

The typical BDI execution cycle is characterized by the following steps:

1. observe the world and the agent's internal state, and update the *event queue* consequently;
2. generate possible new plans whose trigger event matches an event in the event queue ("relevant" plans) and whose precondition is satisfied ("applicable" plans);
3. select one plan from this set of matching plans for execution;
4. push the selected plan onto an existing or new *intention stack*, according to whether or not the event is a (sub)goal;
5. select an intention stack, take the topmost plan and execute the next step of this current plan: if the step is an action, perform it, otherwise, if it is a subgoal, post it on the event queue.

This description is not detailed enough to answer to the question: *what should the agent do if the set of applicable plans is empty?* This issue is usually not considered by the BDI high level specifications. In [17], Rao and Georgeff write that after the option generator reads the event queue and returns a list of options

(namely, a set of applicable plans), the deliberator selects a subset of the options to be adopted and adds these to the intention structure. We can guess that if no options (plans) can be adopted the interpreter simply jumps to the successive step but this is not explicitly established by the specification. In [7], d'Inverno, Kinny, Luck and Wooldridge provide a more detailed specification than the one discussed in [17], but nevertheless, the case that no applicable plans are found is not considered. In the specification of AgentSpeak(L) [16], the $\mathcal{S}_{\mathcal{O}}$ function selects an option or an applicable plan from a set of applicable plans O_ϵ. Also in this case, if O_ϵ is empty, it is not clear what the interpreter does.

When we move from specification to implementation, the problem above cannot be ignored any more. There are different implementation solutions BDI systems adopt. When the implemented BDI systems AgentTalk, JACK, dMARS select an event from the event queue for which there are no applicable plans, they simply ignore the event and delete it from the event queue. The implementation of 3APL (`http://www.cs.uu.nl/3apl/`) adopts a default plan if no other plans can be applied to deal with the selected event. In the implementation of AgentSpeak(L) developed by R. Bordini et al. [1] (`http://www.inf.ufrgs.bf/~massoc`) there are two options of what to do with events which are relevant but not applicable: the user can either ask the interpreter to discard these events or to keep them in the set of events. In the latter case, if eventually a plan becomes applicable, then the agent can do something.

This short overview shows that both at the specification and implementation level the "no applicable plans" problem is solved in a rather ad hoc way. Our proposal, motivated by the applications sketched in the introduction, focuses on the agents ability of cooperating.

3 A Full Specification of Coo-BDI

3.1 Overview

Coo-BDI (Cooperative BDI) is based on the dMARS specification [7] and extends the traditional BDI architecture described in Section 2 in many respects. The first Coo-BDI extension is that external events and main desires are kept separate. In Coo-BDI there are two structures which maintain events and desires: the "event queue" which only contains external events, and the "desire set", which only contains *achieve* desires generated by events. We distinguish between *main desires*, which are kept in the desire set, and *subaltern desires*, which are generated while trying to achieve a main desire and remain implicit in the intention stack structure. When a main desires fails backtracking is applied and a "fresh" (not already attempted) plan instance is selected for it. The main desire is removed by the set if there are no more fresh plan instances for it. Subaltern desires are not backtracked, both to keep the Coo-BDI interpreter and data structures simpler and to maintain the same strategy implemented by dMARS.

The main extension of Coo-BDI, however, involves the introduction of *co-operations* among agents to retrieve external plans for achieving desires (both

main and subaltern), the introduction of *default* plans, the extension of *specific* (non-default) plans with *access specifiers*, the extension of *intentions* to take into account the external plan instances retrieval mechanism and the modification of the Coo-BDI *engine* to cope with all these issues.

Cooperation strategy. The cooperation strategy (or, simply, the cooperation) of an agent A includes the set of agents with which is expected to cooperate, the plan retrieval policy and the plan acquisition policy. The cooperation strategy may evolve during time allowing the maximum flexibility and autonomy of the agents.

Plans. Coo-BDI specific plans share with "classical" ones the trigger, the precondition, the body, the invariant and the two sets of success and failure actions. Besides these components, they also have an access specifier which determines the set of agents the plan can be shared with. It may assume three values: *private* (the plan cannot be shared), *public* (the plan can be shared with any agent) and *only(TrustedAgents)* (the plan can be shared only with the agents contained in the *TrustedAgents* set). Default plans are introduced to ensure that for each desire, at least one plan instance exists for its management. Default plans are private, otherwise they would be continuously exchanged between agents and this makes little sense.

Intentions. Coo-BDI intentions are in a one-to-one relation with main desires: each intention is created when a new main desire enters the desires set, and it is deleted when the main desire fails or is achieved. Intentions are characterized by "standard" components plus components introduced to manage the external plan retrieval mechanism. External plans are retrieved, according to the retrieval policy, both for main and for subaltern desires.

Coo-BDI engine. The engine of Coo-BDI departs from the classical one to take into account both desires generation and cooperations. It is characterized by three macro-steps:

1. process the event queue;
2. process suspended intentions;
3. process active intentions.

Before describing these three steps, we need to explain the mechanism for retrieving relevant plans, since such mechanism is essential for understanding how steps 1 and 3 work. 1) The intention is suspended. 2) The local relevant plans for the desire are generated and associated to the intention. 3) According to the cooperation, the set S of the agents expected to cooperate is defined. 4) A plan request for the desire is created and it is sent to all the agents in S.

Events in the event queue may be either *cooperation* or *ordinary* events. Cooperation events include requests of relevant plan instances for a desire and answers to a plan request. Ordinary events include at least messages reception and notification of updates performed on the agent's beliefs set. When an agent

receives a request of relevant plan instances for a desire, it looks for all the local relevant plan instances for the desire which can be shared with the requesting agent (the ones whose access specifier is *public* or includes the requesting agent) and sends them to it. On the other hand, when an agent receives an answer to a plan request for a desire, it checks if the answer is still valid and if so it updates the intention associated with the desire to include the just obtained plan instances and to remember that answer. Finally, if the event is an ordinary one, the set of corresponding desires is generated and added to the desire set. For each new desire an empty intention is created and the mechanism for retrieving relevant plans is started.

The management of suspended intentions consists in looking for all suspended intentions which can be resumed. When an intention is resumed the set of applicable plan instances is generated from the set of relevant plan instances except for the already failed instances, one applicable plan is selected and the corresponding plan instance is created. If the applicable plan instances set is empty, the desire fails and is deleted from the desire set, and the intention is destroyed. Otherwise, the selected plan instance is pushed onto the intention stack. If the selected plan is an externally retrieved one, it may be used and discarded, or added to the plan library or used to replace plans with a unifying trigger according to the acquisition policy.

Finally, the management of active intentions is similar to the one discussed in [7] apart from the management of a desire achievement which starts the mechanism for retrieving relevant plans in case the desire is not a logical consequence of the agent's current beliefs.

3.2 Structural specification of Coo-BDI

The following subsections describe in detail the structure of the components of the Coo-BDI architecture.

Beliefs, desires, queries and actions. Beliefs are ground atoms (namely, symbols of predicates applied to terms, without variables). To keep the implementation of a Coo-BDI working interpreter as simple as possible, we avoid including negative atoms among beliefs.

Desires have the form *achieve(Atom)*, where *Atom* is a positive atom.

Queries are denoted by *query(SituationFormula)* where a situation formula is either an atom, or the constant *true* or *false*, or conjunctions and disjunctions of situation formulas.

Actions may be internal or external. Internal actions are updates of the data base of an agent's beliefs: *add(Atom)* and *remove(Atom)*. External actions include at least the ability of sending messages to agents: *send(AgentId, Message)*.

Plans. Plans can be of two different forms: either *specific* or *default*. A specific plan is defined by the following components: an access specifier *AccessSpecifier*, a trigger *Trigger*, a precondition *Precondition*, a body *Body*, an invariant *Invariant*,

and two sets of internal actions *SuccessActions* and *FailureActions*. The access specifier may assume the following values:

- *private*, meaning that the plan cannot be provided;
- *public*, meaning that the plan can be provided to any agent;
- *only(TrustedAgents)* where *TrustedAgents* is a set of agent identifiers specifying the only trusted agents which can receive instances of the plan.

The trigger of a specific plan is the desire the plan is designed to achieve.

Like queries, preconditions and invariants are situation formulas.

Plan bodies are non empty trees where nodes are execution states and edges are labelled by either desires, or queries, or (both external and internal) actions. Desires in plan bodies are called *subaltern*.

Finally, success and failure actions are sequences of internal actions.

A default plan is just a specific plan with no access specifier, no trigger and no precondition.

Plan instances. A plan instance is a pair *(Plan,Substitution)* formed by a plan and a substitution, that is a map from variables to terms (we refer to [19] for the notion of substitution and most general unifier).

A plan instance *(Plan,Substitution)* is called *relevant* w. r. t. a desire if either *Plan* is specific and *Substitution* is a most general unifier for that desire and the trigger of *Plan*, or *Plan* is a default plan and *Substitution* is the empty substitution.

A plan instance *(Plan,Substitution)* is called *applicable* if either *Plan* is specific and the formula obtained by applying *Substitution* to the precondition of *Plan* is a logical consequence of the beliefs of the agent's data base, or *Plan* is a default plan and *Substitution* is the empty substitution.

The execution of a plan instance is defined by its plan instance *Instance* together with the computed substitution *Substitution*, the current state *CurrentState* of the body of the plan of *Instance*, the set *NextStates* of the remaining siblings of *CurrentState* which have not been executed yet.

Intentions and cooperation requests. An intention is composed by a stack *Stack* of executions of plan instances, a unique identifier *IntentionId*, a status *Status*, a set of plan instances *RelevantInstances*, a set of agent identifiers *WaitingOnAgents*, and a set of failed plan instances *FailedInstances*; furthermore a relation *IntentionRequest* associate with any intention identity its current (if any) request for cooperation, and with any request the intention it originates from, if the request is still valid.

The stack of executions of plan instances is similar in spirit to the execution stack of logic programs. The intention identifier is necessary to relate an intention to the desire it originated from; such a desire is called *main*.

The status may be either *suspended* or *active*. An intention is suspended if the execution of the plan instance on top of the stack needs to achieve a desire *Desire* for which no plan instance has been selected yet; in this case

RelevantInstances contains the relevant plan instances which have been already collected for *Desire*, *WaitingOnAgents* contains all the identifiers of those agents which are still expected to cooperate for achieving *Desire*, and *IntentionRequest* associates the intention identifier with the current cooperation request for *Desire*.

A cooperation request is specified by a unique identifier *RequestId*, the identifier *AgentId* of the requesting agent, and the desire *Desire* to achieve.

Events. There are two kinds of events: *cooperation* and *ordinary* events. A cooperation event is either *requested(Request)* with the meaning "agent identified by *Request.AgentId* is requesting relevant plan instances for *Request.Desire*", or *provided(AgentId,Request,Instances)* with the meaning "agent identified by *AgentId* has cooperated in response of the request *Request* by providing a set *Instances* of relevant plan instances for *Request.Desire*". In both cases *Desire* has the form *achieve(Atom)*, where *Atom* may contain variables.

Ordinary events include at least the following ones:

- *received(AgentId,Message)* with the meaning "message *Message* has been received from the agent identified by *AgentId*";
- *added(Belief)* with the meaning "the new belief *Belief* has been added to the agent's data base";
- *removed(Belief)* with the meaning "the belief *Belief* has been removed from the agent's data base".

Events perceived from the agent are stored in a priority queue.

Agents and cooperations. An agent is defined by the following data:

- a unique identifier *TheAgentId*;
- a priority queue *TheEventQueue* containing the events perceived by the agent;
- a set of desires *TheDesires* containing all main desires to be achieved;
- a set of specific plans *TheSpecificPlans*;
- a non empty set of default plans *TheDefaultPlans*;
- a set of intentions *TheIntentions*;
- a set of beliefs *TheBeliefs*;
- a cooperation *TheCooperation* specifying a set of trusted agent identifiers *TrustedAgents*, a plan retrieval policy *Retrieval* and a plan acquisition policy *Acquisition*. A plan retrieval policy ranges over the two values *always* and *noLocal*, whereas a plan acquisition ranges over the three values *discard*, *add* and *replace*;
- a relation *DesireIntention* between desires and intention identifiers which must be one-to-one between *TheDesires* and the set of intention identifiers $IntentionIdentifiers = \{Id \ s.t. \ \exists \ Intention \in TheIntentions \wedge Intention.IntentionId = Id\}$;

- a relation *IntentionRequest* between intention identifiers and requests s.t. the identifier of any suspended intention in *TheIntentions* is associated with at most one request, and any request is associated with at most one intention identifier identifying a suspended intention in *TheIntentions*;
- a constant predicate *CanResume* taking a cooperation request, a set of agent identifiers, and a set of plan instances;
- a constant total function *GetDesires* taking an ordinary event and returning a possibly empty set of desires;
- a constant partial function *SelectInstance* taking a possibly empty set of plan instances and returning a member of such a set (the function is undefined if such a set is empty). The function must meet the following requirement: if *SelectInstance(Instances)* returns an instance of a default plan, then *Instances* does not contain instances of specific plans;
- a constant total function *SelectIntention* taking a non empty set of active intentions and returning a member of such a set;
- a constant total function *SelectState* taking a non empty set of states and returning a member of such a state.

3.3 Primitive Operations

The primitive operations used in the specification are listed below. For each data structure, *Empty* is the boolean primitive checking whether its argument is empty.

- Primitives on the event queue: *Empty*, *Get* and *Put*.
- Primitives on sets and relations: $\emptyset$, $\cup$, $\setminus$, $\in$.
- Primitives on intention stacks: *EmptyStack* (constant empty stack), *Empty*, *Push*, *Pop*.
- primitives on bodies (trees): *Root*, *Children Leaf*, *Label*.
- primitives on internal actions sequences: *Empty*, *Head*, *Tail*.
- Primitives for collaborative plan exchange and for sociality:
  - *RequestOp*: takes an agent identifier *AgentId* and a request *Request* and puts *requested(Request)* into the event queue of *AgentId*;
  - *ProvideOp*: takes an agent identifier *ProviderId*, a request *Request* and a set of instances *Instances* and puts *provided(ProviderId,Request,Instances)* into the event queue of *Request.AgentId*;
  - *SendOp*: takes two agent identifiers, *SenderId* and *ReceiverId*, and a message *Message*, and puts *received(SenderId,Message)* into the event queue of *ReceiverId*. Currently we do not commit to any agent communication language: *Message* may be any kind of logical expression eventually containing free variables implementing information passing.
- Logical primitives:
  - *Ground*: takes an atom as its argument and returns *true* if it does not contain variables, *false* otherwise.
  - *Apply*: takes a substitution θ and a term t and returns $t\theta$, namely the term obtained from t by replacing each variable x by $\theta(x)$.

- *Compose*: takes two substitutions σ and θ and returns the substitution $\sigma\theta$ s.t. for every variable x, $(\sigma\theta)(x) = (x\sigma)\theta$.
- *Mgu*: takes two expressions and returns their most general unifier.
- $\models$: takes a couple (S, A) such that S is a set of atoms and A is an atom and returns *true* if A is a logical consequence of S, *false* otherwise.

3.4 Behavioral Specification

The behavioral specification of Coo-BDI is defined using an informal functional language based on Extended ML [10] including boolean connectors and existential and universal quantifiers. Formalizing the given specification using standard notations is part of our future work.

Due to space constraints we will omit the specification of those auxiliary functions whose behavior can be expressed in natural language. We will also skip the details of processing active intentions since, w.r.t. dMARS, this step has been only marginally modified by the introduction of cooperations. The interested reader can find the complete specification of Coo-BDI in the technical report downloadable from `ftp://ftp.disi.unige.it/pub/person/AnconaD/BCDI.ps.gz`.

Processing Events. If the event queue is empty, the successive engine step is performed. Otherwise, an event *Event* is taken from the event queue.

if not Empty(TheEventQueue) then
  let Event=Get(TheEventQueue)

Three situations may occur:
1) If *Event* is of kind *requested(Request)*, then *GetRelInstances(Request)* retrieves all the instances of specific plans relevant for *Request.Desire* and whose specifier is either *public* or *only(TrustedAgents)* and *TrustedAgents* includes *Request.AgentId*. The retrieved *Instances* set is posted into the *Request.AgentId*'s event queue by calling *ProvideOp(TheAgentId,Request,Instances)*.

if $\exists$ *Request s.t. Event=requested(Request) then*
  let Instances=GetRelInstances(Request)
  ProvideOp(TheAgentId,Request,Instances)

$\forall$ *Plan,Sustitution,Request*
  (Plan,Substitution)$\in$*GetRelInstances(Request)*$\Leftrightarrow$
    Plan$\in$*TheSpecificPlans*$\wedge$
    (Plan.AccessSpecifier=public$\vee$
      (*Plan.AccessSpecifier=only(AgentIds)*
      $\wedge$*Request.AgentId*$\in$*AgentIds))*$\wedge$
    Substitution=Mgu(Plan.Trigger,Request.Desire)

2) *Event* is of kind *provided(AgentId,Request,Instances)*. If there exists *Intention* s.t. *(Intention.IntentionId,Request)*$\in$*IntentionRequest* (the retrieval of relevant plan instances for *Request.Desire* is still ongoing) *Intention* is updated by adding the retrieved *Instances* to the *RelevantInstances* field and by removing the identifier of the answering agent (*AgentId*) from the *WaitingOn* field. The updated intention replaces *Intention* in *TheIntentions*.

else if $\exists$ *AgentId, Request, Instances s.t.*
Event=provided(AgentId,Request,Instances) then
    if $\exists$ *Intention s.t. (Intention.IntentionId,Request)*$\in$*IntentionRequest then*
        let UpdatedIntention=Intention
        UpdatedIntention.RelevantInstances:=
            Intention.RelevantInstances$\cup$*Instances*
        UpdatedIntention.WaitingOnAgents:=
            Intention.WaitingOnAgents$\setminus\{$*AgentId*$\}$
        TheIntentions:=(TheIntentions$\setminus\{$*Intention*$\}$*)*$\cup\{$*UpdatedIntention*$\}$

3) If *Event* is an ordinary event, the set of corresponding desires is evaluated by means of *getDesires(Event)* and the desire set is updated to contain them. For each generated desire, a new intention stack is created and initialized (*CreateIntention*), the intention set and the *DesireIntention* relation are updated to take into account the newly created intention and the retrieval of relevant plan instances for the intention and the desire is started (*RetrieveRelInstances(Intention,Desire)*.

else let NewDesires=getDesires(Event)
TheDesires:=TheDesires$\cup$*NewDesires*
$\forall$ *Desire*$\in$*NewDesires*
    $\exists$ *Intention s.t. CreateIntention(Intention)*
    let UpdatedIntention=RetrieveRelInstances(Intention,Desire)
    TheIntentions:=TheIntentions$\cup\{$*UpdatedIntention*$\}$
    DesireIntention:=DesireIntention$\cup$*(Desire,UpdatedIntention.IntentionId)*

When a new intention is created, an unused identifier is assigned to it, its *RelevantInstances*, *WaitingOnAgents* and *FailedInstances* fields are set to the empty set, and its status is set to *suspended*.
Updating an intention by retrieving the relevant plan instances for the desire that the intention is currently achieving (either main or subaltern) means setting the intention status to *suspended*, getting the instances of local plans relevant for the desire (*GetLocalRelInstances(Desire)*), setting the *RelevantInstances* field to the returned set and updating the *WaitingOnAgents* fields according to the plan retrieval policy. If the policy is *always* or it is *noLocal* and no local relevant instances of *specific* plans are found (at least one instance of one default plan is always found by definition of default plan) then the *WaitingOnAgents* field is set to the trusted agents set, a plan request for the desire is issued to each trusted agent (*RequestOp(AgentId,Request)*) and the *IntentionRequest* relation is updated. Otherwise, *WaitingOnAgents* is set to the empty set.

$\forall$ *UpdatedIntention, Intention, Desire, Request*
    UpdatedIntention=RetrieveRelInstances(Intention,Desire) $\Leftrightarrow$
        UpdatedIntention.IntentionId=Intention.IntentionId $\wedge$
        UpdatedIntention.Stack=Intention.Stack $\wedge$
        UpdatedIntention.FailedInstances=Intention.FailedInstances $\wedge$
        UpdatedIntention.Status=suspended $\wedge$
        UpdatedIntention.RelevantInstances=GetLocalRelInstances(Desire) $\wedge$
        (((TheCooperation.Retrieval=always $\vee$
        SpecificPlans(UpdatedIntention.RelevantInstances)=$\emptyset$*)* $\wedge$
        UpdatedIntention.WaitingOnAgents=TheCooperation.TrustedAgents $\wedge$

Request=CreateRequest(TheAgentId,Desire) $\wedge$
$\forall$ *AgentId*$\in$*TheCooperation.TrustedAgents*
RequestOp(AgentId,Request) $\wedge$
IntentionRequest=
IntentionRequest$\cup$*(UpdatedIntention.IntentionId,Request))* $\vee$
(TheCooperation.Retrieval=noLocal $\wedge$
SpecificPlans(UpdatedIntention.RelevantInstances)$\neq\emptyset$ $\wedge$
UpdatedIntention.WaitingOnAgents=$\emptyset$*))*

$\forall$ *Plan, PlanInstancesSet*
Plan$\in$*SpecificPlans(PlanInstancesSet)*$\Leftrightarrow$
$\exists$ *Substitution s.t. (Plan,Substitution)*$\in$*PlanInstancesSet* $\wedge$
Plan$\in$*TheSpecificPlans*

$\forall$ *Plan,Sustitution,Instances*
(Plan,Substitution)$\in$*GetLocalRelInstances(Desire)*$\Leftrightarrow$
Plan$\in$*TheSpecificPlans* $\wedge$ *Substitution=Mgu(Plan.Trigger,Desire)* $\vee$
Plan$\in$*TheDefaultPlans* $\wedge$ *Substitution=*$\emptyset$

$\forall$ *Request*
Request=CreateRequest(AgentId,Desire) $\Leftrightarrow$
Request.RequestId=Id $\wedge$
$\nexists$ *Another*$\in$*ran(IntentionRequest) s.t. Another.RequestId=Id* $\wedge$
Request.AgentId=AgentId $\wedge$ *Request.Desire=Desire*

Processing Suspended Intentions. During this step the engine checks if there are suspended intentions which can be resumed since the associated *Request* and the *WaitingOnAgents* and *RelevantInstances* fields satisfy the *CanResume* condition.

$\forall$ *Intention*$\in$*TheIntentions s.t. Intention.Status=suspended*
if $\exists$ *Request s.t.*
(Intention.IntentionId,Request)$\in$ *IntentionRequest* $\wedge$
CanResume(Request,Intention.WaitingOnAgents,Intention.RelevantInstances)

If one resumable intention is found, two cases must be considered:
1) The intention stack is empty: the desire for which the relevant plan instances have been collected is a main desire. To implement backtracking on the plan instances which can be used to achieve the main desire, the already failed plan instances (*Intention.FailedInstances*) are not re-attempted. The applicable plan instances are thus evaluated starting from the collected relevant instances *Intention.RelevantInstances* minus the failed instances.
2) The intention stack is not empty: the desire for which the relevant plan instances have been collected is a subaltern desire, and no backtracking is implemented for it. The applicable plan instances are thus evaluated starting from *Intention.RelevantInstances*.

then
if Empty(Intention.Stack) then
let Applicable=GetApplInstances(
Intention.RelevantInstances$\setminus$*Intention.FailedInstances)*
else
let Applicable=GetApplInstances(Intention.RelevantInstances)

GetApplInstances(RelInst) returns all the instances *(Plan,Substitution)* obtained from *RelInst* such that the plan precondition instantiated with *Substitution* is a ground logical consequence of the agent's beliefs. By definition, for any desire there is at least one relevant plan instance (the one originated by one default plan), and this instance is also applicable (there is no precondition to check): when *GetApplInstances* is applied to a non-empty set of relevant plan instances it always returns at least one plan instance; it returns the empty set if and only if its argument is the empty set. In the specification given so far, *Applicable* may be empty only if the intention relevant instances minus the intention failed instances is the empty set, namely, all the relevant instances for the main desire have failed. In this case, the desire fails: both the desire and the intention must be removed by the corresponding sets and the relations involving them must be updated.

if Applicable=∅ then
TheDesires:=TheDesires\{Request.Desire}
TheIntentions:=TheIntentions\{Intention}
DesireIntention:=DesireIntention
{(Request.Desire,Intention.IntentionId}
IntentionRequest:=IntentionRequest\{(Intention.IntentionId,Request)}

If *Applicable* is not empty, a plan instance is selected from it and its execution is created by setting the instance field to the selected plan instance, the substitution field to *Substitution*, the current state to the root of the plan body and the next states to the children of the current state. The intention execution is pushed onto the intention stack and the intention's status specifier is set to *active*. The selected plan is discarded, added to the plan library or used to replace all the existing plans whose trigger unifies with the plan's trigger to which *Substitution* is applied according to the agent's plan acquisition policy.

else
let Instance=SelectInstance(Applicable)
let Execution=StartExec(Instance)
let UpdatedIntention=Intention
UpdatedIntention.Stack:=Push(Execution,Intention.Stack)
UpdatedIntention.Status:=active
TheIntentions:=(TheIntentions\{Intention})∪{UpdatedIntention}
if Instance.Plan≠∈TheSpecificPlans then
TheSpecificPlans:=AcquirePlan(Instance)

∀ *Plan,Subst,Instances*
(Plan,ComposedGroundingSubst)∈GetApplInstances(Instances)⇔
(Plan,Subst)∈Instances ∧
∃ *GroundingSubst s.t.*
GroundLogicalConsequence(GroundingSubst,
Apply(Subst,Plan.Precondition)) ∧
GroundingSubst≠null ∧
ComposedGroundingSubst=Compose(Subst,GroundingSubst)

∀ *Plan,Sustitution,InstanceExecution*
InstanceExecution=StartExec((Plan,Substitution))⇔
InstanceExecution.Instance=(Plan,Substitution)∧

$InstanceExecution.Substitution = Substitution \wedge$
$InstanceExecution.CurrentState = Root(Plan.Body) \wedge$
$InstanceExecution.NextStates = Children(Root(Plan.Body))$

$\forall\ Plan, Substitution, PlanSet$
$PlanSet = AcquirePlan((Plan, Substitution)) \Leftrightarrow$
$(TheCooperation.Acquisition = discard \wedge PlanSet = TheSpecificPlans) \vee$
$(TheCooperation.Acquisition = add \wedge PlanSet = TheSpecificPlans \cup Plan) \vee$
$(TheCooperation.Acquisition = replace \wedge Plan \in PlanSet \wedge$
$\forall\ P \in TheSpecificPlans$
$P \in PlanSet \Leftrightarrow P = Plan \vee$
$\nexists\ Mgu\ s.t.\ Mgu = Mgu(P.Trigger, Apply(Substitution, Plan.Trigger)))$

Processing Active Intentions. Active intentions are processed like in dMARS. If there are active intentions, one is selected and removed by the intention set (a modified copy will be added at the end of the management) and the topmost plan instance execution is taken. Three cases may arise. 1) The current state of the topmost plan instance execution is a leaf: the plan succeeds and the procedure for managing this situation is called. 2) The invariant condition of the plan associated to the plan instance instantiated with the execution substitution is not satisfied in the current agent's state or the current state of the topmost plan instance execution is not a leaf, and there are no more states reachable from it: the plan fails and the procedure for managing this situation is called. 3) The topmost plan instance execution neither fails nor succeeds: the action to perform, which labels the edge between the current state and the selected successive state, is retrieved. If the action is *add(Atom)*, *remove(Atom)*, *send(AgentId,Message)* or *query(SituationFormula)*, the calls to primitive operations and updates to the intention, belief base and event queue are performed almost like in dMARS. If the action is *achieve(Desire)* and the desire is not a logical consequence of the agent's current beliefs, *RetrieveRelInstances* is called.

4 Related Work and Conclusions

Starting from the BDI model, many extensions have been proposed in literature. Most of them add some attitude or ability to the basic BDI theory or provide a better formalization of the relationships between existing ones. Just to make some examples, [13] extends the theoretical BDI framework with the notion of *capability* and investigates how capabilities affect the agent's reasoning about intentions. In [2] the BOID (Beliefs, *Obligations*, Intentions, Desires) architecture is discussed. It contains feedback loops to consider all effects of actions before committing to them, and mechanisms to resolve the conflicts between the outputs of the B, O, I, D components. In [5], classical BDI agents are extended with *conditional mental attitudes* represented by interconnected components. Mental attitudes are considered to be functions of other mental attitudes and functional dependencies between them are analyzed. The paper [15] investigates the impact that *sociality* has on mental states and how it can be formalized within a BDI

model, while [14] focuses on actions and their formalization by adding three more operators to the basic BDI ones: *capabilities*, *opportunities* and *results*. All these works are more concerned with the logical formalization of the theory behind the extended BDI model than with the complete specification of an implementable system, as our work is.

More similar in spirit to our pragmatic approach are the attempts to extend BDI systems with *mobility*. In [3] the TOMAS (Transaction Oriented Multi Agent System) architecture is described: it combines the distributed nested transaction paradigm with the BDI model. An algorithm is presented which enables agents in TOMAS to become mobile. The JAM BDI-theoretic mobile agent architecture [8] provides plan and procedural representations, metalevel and utility-based reasoning over simultaneous goals and goal-driven and event-driven behavior. Mobility is realized by an *agentGo* primitive for agent migration that may appear in the body of plans.

To the best of our knowledge, however, there are no extensions to the BDI model or to BDI-based implemented systems that allow agents to cooperate by exchanging plans: in all the BDI-based systems, theories and languages we are aware of, the plan library is a static component and there are no means to extend the agent's procedural knowledge at runtime. Our proposal overcomes this limitation by extending the architecture interpreter. This means that the agent's developer may take advantage of the cooperativity of agents for free: all the hard work of providing and retrieving external plans is managed by the interpreter without the agent's developer needing to be aware of the details of this mechanism. A classic BDI agent can be modeled in Coo-BDI by setting the access specifier of all its plans to *private* and by setting the trusted agents set to the empty set. In this way, the agent will never look for external relevant plans, and will never share its own plans, exactly as it happens in the original BDI model.

As far as the future work on Coo-BDI is concerned, there are three main directions we will follow: 1) providing a better formalization on Coo-BDI using a standard specification language; 2) analyzing how exchanges of plans can be described as part of a more general speech-act based communication mechanism adopting standard languages such as FIPA ACL (`http://www.fipa.org/`) and KQML [11]; and 3) developing a working Coo-BDI interpreter and using it to implement one of the applications discussed in the introduction. This exercise will allow us to get feedbacks on the Coo-BDI formalization, (possibly) confirming the adequacy of the approach presented in this paper.

Acknowledgments. The authors thank Rafael Bordini, Mehdi Dastani, Mark D'Inverno and Michael Winikoff for the useful discussions on the "no applicable plans" problem and the anonymous referees for their thoughtful and constructive comments.

References

1. R. H. Bordini, A. L. C. Bazzan, R. de O. Jannone, D. M. Basso, R. M. Vicari, and V. R. Lesser. AgentSpeak(XL): efficient intention selection in BDI agents via decision-theoretic task scheduling. In *Proc. of the 1st Int. Joint Conf. on Autonomous Agents and Multiagent Systems*, Bologna, Italy, 2002.
2. J. Broersen, M. Dastani, J. Hulstijn, Z. Huang, and L. van der Torre. The BOID architecture: conflicts between beliefs, obligations, intentions and desires. In *Proc. of the 5th Int. Conf. on Autonomous Agents*, Montreal, Canada, 2001. ACM Press.
3. P. Busetta and K. Ramamohanarao. An architecture for mobile BDI agents. In *Proc. of SAC'98, the 1998 ACM Symposium on Applied Computing*, Atlanta, Georgia, USA, 1998.
4. P. Busetta, R. Ronnquist, A. Hodgson, and A. Lucas. JACK intelligent agents – components for intelligent agents in Java. *AgentLink News Letter*, 2, 1999.
5. M. Dastani and L. van der Torre. An extension of BDI_{CTL} with functional dependencies and components. In *Proc. of the 9th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'02)*, 2002.
6. M. d'Inverno, K. V. Hindriks, and M. Luck. A formal architecture for the 3APL agent programming language. In *In Proc. of the 1st Int. Conf. of B and Z Users (ZB2000)*, York, UK, 2000.
7. M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A formal specification of dMARS. In *Intelligent Agents IV*, Providence, Rhode Island, USA, 1997.
8. M. J. Huber. JAM: A BDI-theoretic mobile agent architecture. *AgentLink News Letter*, 5, 2000.
9. N. R. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1:7–38, 1998.
10. S. Kahrs, D. Sannella, and A. Tarlecki. The definition of Extended ML: a gentle introduction. *Theoretical Computer Science*, 1997.
11. J. Mayfield, Y. Labrou, and T. Finin. Evaluation of KQML as an agent communication language. In *Intelligent Agents II*. Springer Verlag, 1995. LNAI 1037.
12. K. L. Myers. User guide for the procedural reasoning system. Technical report, Artificial Intelligence Center, Menlo Park, CA, 1997.
13. L. Padgham and P. Lambrix. Agent capabilities: Extending BDI theory. In *Proc. of the 7th National Conference on Artificial Intelligence*, Austin, Texas, 2000.
14. V. Padmanabhan, G. Governatori, and A. Sattar. Actions made explicit in BDI. In *Proc. of the 14th Australian Joint Conference on AI*, Adelaide, Australia, 2001.
15. P. Panzarasa, T. Norman, and N. R. Jennings. Modeling sociality in the BDI framework. In *Proc. of the 1st Asia-Pacific Conf. on Intelligent Agent Technology*, 1999.
16. A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. V. de Velde and J. W. Perram, editors, *Agents Breaking Away*, pages 42–55. Springer-Verlag, 1996. LNAI 1038.
17. A. S. Rao and M. Georgeff. BDI agents: from theory to practice. In *Proc. of the 1st Int. Conf. on Multi Agent Systems (ICMAS'95)*, San Francisco, CA, USA, 1995.
18. A. S. Rao and M. Georgeff. Decision procedures for BDI logics. *Journal of Logic and Computation*, 8(3):293 – 342, 1998.
19. J. A. Robinson. Computational logic: The unification computation. *Machine intelligence*, 6:63–72, 1971.
20. M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

A Combined Logic of Expectation and Observation

A generalisation of BDI logics

Bình Vũ Trần, James Harland, and Margaret Hamilton

School of Computer Science and Information Technology
RMIT University, Australia
{tvubinh, jah, mh}@cs.rmit.edu.au

Abstract. Although BDI logics have shown many advantages in modelling agent systems, the crucial problem of having computationally ungrounded semantics poses big challenges when extending the theories to multi-agent systems in an interactive, dynamic environment. The root cause lies at the *inability of modal languages to refer* to the world states which hampers agent reasoning about the connection of its mental attitudes and its world. In this paper, following ideas in hybrid logics, we attempt to readdress the computational grounding problem. Then, we provide a formalism for observations – the only connection between mind and worlds – and expectations – the mental states associated with observations. Finally, we compare our framework with BDI logics.

1 Introduction

The most widely held view for practical reasoning agents is that they are *intentional systems* whose behaviour can be explained and predicted through the attribution of mental attitudes such as beliefs, desires, hopes, fears... Since the seminal work of Hintikka [14], formal analyses of mental attitudes are mainly carried out using *modal logics*. Among these models, Belief-Desire-Intention (BDI) model [7] and BDI logics [18] have been one of the most successful. Unfortunately, BDI logics are usually claimed as having *ungrounded semantics* [21], that is, there have been no work showing a one-to-one correspondence between mental models and any concrete computational interpretation. This results in a large gap between theory and practice [16]. The problem, however in our view, should be stated more precisely that since the relationship between mental and computational models realizing the same modal language is a many-to-many relation, it is unclear which computational model is the most suitable for simulating an agent's mental model in a dynamic interactive environment.

In this paper, we will demonstrate that the inability to find a concrete and useful computational model is due to the lack of expressive power in modal languages which are used as agent specifications. Since modal syntax offers no grip on worlds, from the local, internal perspective of modal logics, it is impossible to detect any difference between the structures of two models. Consequently,

agent reasoning capabilities are seriously impaired. For example, imagine an eagle is chasing a sparrow in a cave system where every cave appears identical. At any cave, there is only one identical unidirectional passageway to another cave. Modal language would express this by saying "All caves accessible from this cave are identical to it." But the eagle cannot tell whether it has flown through a cave before. So either the cave system has an infinite number of caves connected with each other, so that the eagle cannot visit a cave twice, or it has only one cave looping back on itself, the eagle would not be able to distinguish using orthodox modal language. If a distinction could be recognised, the eagle would be able to justify its expectation where the sparrow could be. Hence it would speed up to catch the sparrow in the former case, but it would stay still in the current cave waiting for the sparrow in the latter case.

It may become apparent that a mechanism to mark the visited worlds will be a significant advantage for the exploring agent allowing it to redraw a map of the real world in its mind. It would not only help the agent to differentiate between two different models, but also provide a tool to base its future predictions. Hybrid languages by Blackburn and Tzakova [6], [3] provide such a naming mechanism for modal languages by introducing a unique label of each world and an operator to jump and evaluate formulae in any world. We believe that such mechanism is strongly related to the concept of observation – the only connection between mind and world. Hence, a formalism that describes observation and its associated mental states, expectations will bridge the mental models and the computational models.

In this paper, following Blackburn and Tzakova we develop a formalism for expectation and observation in hybrid logics and compare this with the BDI model. The paper commences by elaborating in detail the computational grounding problem, and giving an overview of hybrid logics. We then describe the observation-expectation system's details in section §3 and its comparison with BDI logics in section §4. Finally, we briefly outline our approach towards the application of our framework in multi-agent systems.

2 Computational grounding problem

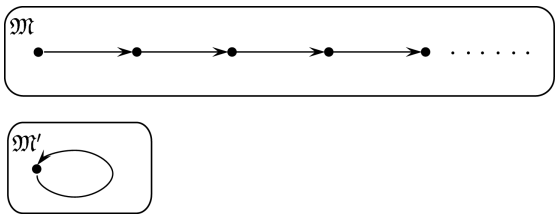

Fig. 1. Invariance and modal languages

The principal motivation for modal languages is to provide a local, internal perspective about world structures. The truth value of any formula is evalu-

ated inside the structure, at a particular (current) state. Even though modal operators also provide access to information at other states, only states that are directly accessible from the current are allowed. This property has attracted many scholars in various disciplines such as cognitive science, psychology, and artificial intelligence to use modal logic as a formal analytical tool for mental models.

The relationship between models in modal logic has been well studied under the notion of *bisimulation* [5]. It is revealed that with only restrictions on identical atomic information and matching accessibility relations to make modal languages invariant, bisimulations are *many-to-many* relations.

Definition 1 (Bisimulation) *Let $\Phi = \{p, q, \ldots\}$ be a set of atomic propositions. Given two models $\mathfrak{M} = (W, \sim, \pi)$ and $\mathfrak{M}' = (W', \sim', \pi')$, where W, W' are non-empty sets of possible worlds, $\sim \subseteq W \times W, \sim' \subseteq W' \times W'$ are two accessibility relations on W and W', and $\pi : \Phi \to Pow(W), \pi' : \Phi \to Pow(W')$ are two interpretation functions which respectively tell the sets of worlds of W and W' where each proposition holds.*

A bisimulation between two models $\mathfrak{M}$ and $\mathfrak{M}'$ is a non-empty binary relation $Z \subseteq W \times W'$ ($\mathfrak{M}$ and $\mathfrak{M}'$ are called bisimilar*) if the following conditions are satisfied:*

- ***(prop)** if wZw' and $w \in \pi(p)$, then $w' \in \pi'(p)$ for all $p \in \Phi$*
- ***(forth)** if wZw' and $w \sim v$, then there exists $v' \in W'$ such that vZv' and $w' \sim' v'$*
- ***(back)** if wZw' and $w' \sim' v'$, then there exists $v \in W$ such that vZv' and $w \sim v$*

Therefore, two models with completely different structures can be modally equivalent. Given an arbitrary model, there can be many other models bisimilar to it. For example, let's compare a model $\mathfrak{M}$, which has the natural numbers in their usual order as its frame ($W = \mathbb{N}$) and every propositional symbol is **true** at every world, with another model $\mathfrak{M}'$ which has a single reflexive world as its frame and all propositional symbols are **true** at this world (*see* **Fig. 1**). Apparently, one is infinite and irreflexive whilst the other is finite and reflexive. However, they both recognise the same modal language.

One may argue that this would be an advantage of modal language. So for example if the model $\mathfrak{M}$ above is a mental model, its identical structure would not be implementable on a computer due to the infinite set of mental states. However, the equivalent model $\mathfrak{M}'$ has a finite set of states, and hence would certainly be implementable. This argument is valid with the assumption that the original mental model is unchanged and fixed.

Unfortunately, we also have two crucial disadvantages, namely, the **inability to model agents in a dynamic environment** and the **verification** problems. Firstly, in a dynamic unpredictable environment, the agent is continuously updating its mental models. The properties of the models may very well be changed under such updates. The simple computational model $\mathfrak{M}'$ must also be updated to simulate exactly its mental model. However, for example, if we add one more

world v related to a particular world w in the above model $\mathfrak{M}$, where only some propositional symbols are true, it may be clear that a single simple addition to $\mathfrak{M}'$ to reflect the change is not easy. One may keep arguing that perhaps $\mathfrak{M}'$ was not the right choice. In an environment, where change is unpredictable an identical structure always guarantees the number of changes for the computational model is not greater than the number of changes in the mental model. For other non-identical structures, the possibility of more substantial changes does exist.

Secondly, whilst *axiomatic* verification is as hard as the complexity of a proof problem, *semantic* verification (model checking) is more efficient [22, p 296]. Unfortunately, semantic approaches have a serious problem: it is unclear how to derive appropriate accessibility relations from a given *arbitrary* concrete program. It may be clear how to use the relation between computational states of the computational model to construct a unimodal logic. However, for multi-modal logics such as BDI logics, the construction is usually ad hoc. There are two reasons for this difficulty.

1. *Partitioning problem*: It is unclear how to partition the relational structure between computational states, and use these partitions for their corresponding accessibility relations' constructions. It is as hard as an exhaustive search for all possible combinations.
2. *Interaction problem*: Assuming the first problem were solved, for any submodel of the original computational model $\mathfrak{M}'$, since bisimulation is a many-to-many relation, there can be many mental models with very different structures corresponding to it. It is also unclear how constraints between modalities can be derived from the sub-models.

The partitioning problem could be reduced if the relational structure can be classified at a meta level, i.e., if it is possible to classify formulae derived from the single relational structure into different sorts. Each sort corresponds to a mental attitude. The partition that realises that sort can then be used to derive the corresponding accessibility relation of the mental modality. This ultimately provides a two-tier approach. The first tier is totally based on a single accessibility relation with an additional sorting mechanism. The second tier with various accessibility relations is then constructed on the first tier. Unfortunately, such sorting mechanism is not provided in orthodox modal languages.

The above issues lead us to a definite conclusion: modal languages are not expressive enough for specifying agents in general and particularly in a dynamic, unpredictable environment. We are unable to update our computational model reliably and cheaply for changes in theory. More seriously, it is also expensive to verify correctness of a computational model. We call this problem *the computational grounding problem of modal languages* (cf. [21]). In other words, modal languages cannot describe the connection between minds and worlds.

Hybrid languages [3], [6], overcome the problem by two simple additions to modal languages: a new sort of formulae *nominals* and *satisfaction operators* @. Basically, nominals are just atomic propositions disjoint from the set of normal propositions. The crucial difference is that, each nominal is **true** only at a unique world in the possible worlds structure. Therefore, nominals can be considered

as the name, or label for worlds. Satisfaction operators @ are used to assert satisfaction of a formula at a certain world. They allow us to jump to the world where the formula is evaluated.

The additions are relatively simple, but they have significant contributions. Firstly, though hybrid bisimulation could be altered slightly by adding nominals to the set of proposition Φ in the **(prop)** rule, the addition of the **(@)** rule insists that nominals must be true at a unique world in each model.

- (@) for all nominals s, if $\pi(s) = \{w\}$ and $\pi'(s) = \{w'\}$, then wZw' where $w \in W, w' \in W'$.

The **(@)** rule guarantees every world $w \in W$ has a unique corresponding world $w' \in W'$. Therefore, though bisimulation Z is a relation, under this condition, it becomes a one-to-one function. In other words, a hybrid bisimulation is equivalent to an isomorphism [1]. With these results, two necessary conditions to solve the grounding problem are satisfied.

Secondly, in [1], it was also proved that the addition of nominals and satisfaction operators does not raise complexity. The satisfiability problem remains decidable in *PSPACE-complete*.

The path of hybrid logics that this paper follows is led by Blackburn [2]. Computational complexity and characterisation of hybrid logics were studied in [1]. The web site *http://www.hylo.net* provides further resources and development in hybrid logics.

3 Observation-expectation system

3.1 Agents, observations and expectations

The real world has its own structure and properties. An agent's mental model is only a reflected part of that structure in the agent's mind. The only means that the agent has to discover its environment is through its *observation*. Therefore, in order to formalise the connection between an agent's mind and its real world, we make observation our essential concept.

According to the *Merriam-Webster Unabridged* dictionary [13], an observation is "an act of recognising and noting a fact or occurrence..." or "an act of seeing or of fixing the mind upon anything." In our framework, observation is a bisimulation between the real world and an agent's mental model. A single object in the real world, e.g. the planet *Venus*, can have multiple images in an agent's mind, '*the morning star*' and '*the evening star*' through two different observations. Conversely, a single mental state can refer to various real world objects, e.g. a tiger refers to any individual which is a large carnivorous feline mammal having a tawny coat with transverse black stripes. This ability of rational agents is usually known as *abstraction*.

Unfortunately, this only connection to the real world is not always available due to various reasons, e.g. limitations of sensors, noises or disruptions from the environment. In such conditions, a rational agent is still able to continually

construct the world model in its mind using its inferential mechanisms and act upon this model accordingly. Thus in the above example, when chasing the sparrow, if the sparrow disappears into a passageway, the eagle would predict the sparrow's movement and keep flying to the other end of the passageway to catch the sparrow there, instead of stopping the chase. The images of the world in the agent's mind that are associated with observations as about to happen are called *expectations*.

In this section, firstly we describe how observations are linked in an observation system and the association of observation with expectation. Secondly, we introduce the formalism for expectation logic based on the observation system.

3.2 Observation system

We can now formally define an observation system with its corresponding real-world global states:

Definition 2 ***(Observation system)*** *Let $\mathfrak{I}$ be a set of agent identities, L_i be a set of local states for any agent a_i, where $i \in \mathfrak{I}$. An observation system is a quadruple $OS = \langle \mathbb{G}, \tau, \mathcal{O}, g_0 \rangle$ where*

- *$\mathbb{G} \subseteq \prod_{i \in \mathfrak{I}} L_i$ is the set of global states with each $g \in \mathbb{G}$ being an instantaneous global state.*
- *$g_0 \in \mathbb{G}$ is the initial state of the system.*
- *The environment of an agent a_i is $Env_i \subseteq \prod_{j \in \mathfrak{I} \setminus i} L_j$*
- *An agent a_i's collection of observations is a relation $Obs^i \subseteq \mathbb{G} \times \mathcal{E}_i$, where $\mathcal{E}_i \subseteq L_i$ is the set of* expectations *and each pair (g, ε) is called an* observation *taken by the agent a_i. ε is the agent's expectation about g through the observation.*
- *Let $\mathfrak{G} \subseteq \mathfrak{I}$. A group $\mathfrak{G}$'s collection of observations is a relation $Obs^{\mathfrak{G}} \subseteq \mathbb{G} \times \mathcal{E}_{\mathfrak{G}}$ where $\mathcal{E}_{\mathfrak{G}} = \bigcup_{i \in \mathfrak{G}} \mathcal{E}_i$. The pair $(g, \varepsilon) \in Obs^{\mathfrak{G}}$ is called a group observation.*
- *$\tau : \mathbb{G} \times \mathcal{O} \to \mathbb{G}$ is a system state transformer function that depicts how a global state transits to another when a set of observation methods from the observation method family $\mathcal{O}$ defined below are carried out by some or all agents in the system.*

Given a sensor set ($\mathbb{S} = \bigcup_{i \in \mathfrak{I}} \mathbb{S}_i$) and a effector set ($\mathbb{E} = \bigcup_{i \in \mathfrak{I}} \mathbb{E}_i$), where $\mathbb{S}_i$ and $\mathbb{E}_i$ are respectively the sets of sensors and effectors of an individual agent a_i. Let's look at the eagle and the cave system in the above example as an agent in an observation system. The eagle's sensors $\mathbb{S}_i$ (eyes, ears, skin, ...) and effectors $\mathbb{E}_i$ (wings, neck, ...) and any combination of them bring different observations (Obs_i) about the environment to the eagle. The eyes ($\xi \in \mathbb{S}_i$) bring visual images of the caves ($\varepsilon_1 \in \xi$), and sparrow ($\varepsilon_2 \in \xi$) to the eagle's brain $\mathcal{E}_i \subseteq L_i$. The wings ($e \in \mathbb{E}_i$) when flapping may bring an observation that its position would be closer to the sparrow ε_3. However, this can be verified by the eagle's eyes if $\varepsilon_3 \in \xi$.

In any observation system, sensors and effectors are the primary sources that generate observations. Each sensor or effector is associated with a set of observations. An important note is that observations associated with an effector are only *hypothetical*. That is, the agent is always *uncertain* about the consequences of its actions until it uses its sensors to verify the results. Thus, an observation of an effector is justified if and only if it is also associated with a sensor. Observing the world by obtaining observations directly from sensors and effectors is called *primitive observation method* $\mathbb{O}_0$ (e.g. $e, \xi \in \mathbb{O}_0$). A more complicated set of observation methods $\mathbb{O}_k$ would arrange the k results (expectations) of other observation methods in a systematic way to generate new expectations about the world. These expectations are also associated with global states to form more complex observations. Observation methods are formally defined as follows:

Definition 3 ***(Observation methods)*** *An observation method family is a set of observation method sets* $\mathcal{O} = \{\mathbb{O}_k\}_{k\in\mathbb{N}}$ *where* $\mathbb{O}_k$ *is a set of observation methods of arity* k *for every* $k \in \mathbb{N}^+$. $\mathbb{O}_0 = \mathbb{S} \cup \mathbb{E}$ *is called primitive observation method set.* $\mathbb{O}_k$ *is inductively defined as follows:*

- $\varepsilon \in \mathcal{E}$ *for all* $\varepsilon \in o_0, \forall o_0 \in \mathbb{O}_0$
- $o_k(\varepsilon_1, \ldots, \varepsilon_k) \subseteq Pow(\mathcal{E})$ *for all* $o_k \in \mathbb{O}_k$ *and* $\varepsilon_1, \ldots, \varepsilon_k \in \mathcal{E}$

Thus, if the eagle expects the sparrow would reach the end of the passageway, and it also expects with a flap it would get to the same place at the same time, a combination of the two expectations $o_2(\varepsilon_2, \varepsilon_3) \in \mathbb{O}_2$ provides a way of chasing the sparrow which generates an expectation that two birds would be at the same place (ε_4). However, if the eagle executed the "chasing the sparrow" method $o_2(\varepsilon_2, \varepsilon_3)$, and it got the expectation ε_5 that the sparrow was not at the same place, it would not be able to distinguish the two resulting expectations ε_4 and ε_5.

Intuitively, whilst adopting any observation method o_k, an agent may have various observations about a real world state g which bring various expectations to the agent's mind. We call these expectations *indistinguishable* to the observation method o_k about the world g. That is, we cannot tell by adopting observation method o_k what makes the generated observations different. Also, two sets of a group of agents' expectations will appear indistinguishable to an individual agent if its expectations in both sets do not change. Formally,

Definition 4 (Indistinguishability and transparency properties)

- *Two sets of expectations* $\mathcal{E}, \mathcal{E}'$ *are* indistinguishable *through an observation* o_k *if* $\mathcal{E}, \mathcal{E}' \in o_k(\varepsilon_1, \ldots, \varepsilon_k)$.
- *Two sets of expectations* $\mathcal{E} = \{\varepsilon_1, \ldots, \varepsilon_k\}, \mathcal{E}' = \{\varepsilon_1, \ldots, \varepsilon_j\}$ *are* indistinguishable *to an agent* a_i *if* $l_i(\mathcal{E}) = l_i(\mathcal{E}')$ *where* $l_i : Pow(\mathcal{E}) \to Pow(L_i)$ *is a function that extracts the local states of an agent from a set of expectations.*
- *An observation method* o_k *is considered as* transparent *if* $o_k(\varepsilon_1, \ldots, \varepsilon_k) = \{\mathcal{E}\}$ *where* $\mathcal{E} = \{\varepsilon_1, \ldots, \varepsilon_k\}$ *(cf. [23]). A transparent observation method can also be considered as a* compound sensor. *The family of transparent observation methods is denoted as* $\mathcal{O}^t$.

Indistinguishability of observation methods unfortunately also brings uncertainty to agents. Thus, if $o_2(\varepsilon_2, \varepsilon_3) = \{\{\varepsilon_4\}, \{\varepsilon_5\}\}$, the eagle would be unsure which result would occur. Transparent observation methods hence are more preferable. However, that also means the eagle must determine a correct set of observations and take sufficient observations for a transparent observation method. The eagle could combine other observations such as no way out of the cave, or no other eagle close to the sparrow to guarantee that its "chasing the sparrow" output could be more certain.

3.3 Expectation logic

We can now study how an agent generates its expectations from its observations about its environment by introducing an expectation logic $\mathcal{L}$ based on an observation system. Consider the set of agents identified by the identity set $\mathfrak{I}$. To view an observation system as a hybrid Kripke structure, we introduce the observation interpretation function $\pi : (\Phi \cup \Xi) \to Pow(\mathbb{G})$ which based on available set of observations to tell which expectation is associated with which global state. $\Phi = \{p, q, r, \ldots\}$ is called the primitive expectation proposition set and $\Xi = \{s, t, \ldots\}$ is called the observation naming set. The crucial difference from orthodox modal logic is that for every observation name s, π returns a singleton. In other words, s is **true** at a unique global state, and therefore tags this state. Yet, it is possible that a global state can have different observation names. We refer the couple $\mathfrak{M} = \langle OS, \pi \rangle$ as a model of expectation in an observation system.

Definition 5 *The semantics of expectation logic $\mathcal{L}$ are defined via the satisfaction relation $\models$ as follows*

1. $\langle \mathfrak{M}, g \rangle \models p$ *iff* $g \in \pi(p)$ *(for all* $p \in \Phi$*)*
2. $\langle \mathfrak{M}, g \rangle \models \neg\varphi$ *iff* $\langle \mathfrak{M}, g \rangle \not\models \varphi$
3. $\langle \mathfrak{M}, g \rangle \models \varphi \vee \psi$ *iff* $\langle \mathfrak{M}, g \rangle \models \varphi$ *or* $\langle \mathfrak{M}, g \rangle \models \psi$
4. $\langle \mathfrak{M}, g \rangle \models \varphi \wedge \psi$ *iff* $\langle \mathfrak{M}, g \rangle \models \varphi$ *and* $\langle \mathfrak{M}, g \rangle \models \psi$
5. $\langle \mathfrak{M}, g \rangle \models \varphi \Rightarrow \psi$ *iff* $\langle \mathfrak{M}, g \rangle \not\models \varphi$ *or* $\langle \mathfrak{M}, g \rangle \models \psi$
6. $\langle \mathfrak{M}, g \rangle \models \langle \mathcal{E}_i \rangle \varphi$ *iff* $\langle \mathfrak{M}, g' \rangle \models \varphi$ *for some* g' *such that* $g \sim_e^i g'$
7. $\langle \mathfrak{M}, g \rangle \models [\mathcal{E}_i] \varphi$ *iff* $\langle \mathfrak{M}, g' \rangle \models \varphi$ *for all* g' *such that* $g \sim_e^i g'$
8. $\langle \mathfrak{M}, g \rangle \models s$ *iff* $\pi(s) = \{g\}$ *(for all* $s \in \Xi$*),* g *is called the denotation of* s
9. $\langle \mathfrak{M}, g \rangle \models @_s \varphi$ *iff* $\langle \mathfrak{M}, g_s \rangle \models \varphi$ *where* g_s *is the denotation of* s*.*

where 1 – 7 are standard in modal logics with two additions of hybrid logics in 8 and 9.

We have introduced the modality $\mathcal{E}_i$ which allows us to represent the information of the environment resident in the agent a_i's mind about the output of its observation methods. The semantics of the $\mathcal{E}_i$ modality are given through the *expectation accessibility relation* defined as follows

Definition 6 *Given a binary* expectation accessibility relation $\sim_e^i \subseteq \mathbb{G} \times \mathbb{G}$ *then* $g \sim_e^i g'$ *iff* $\exists \mathcal{E} \subseteq g, \exists \mathcal{E}' \subseteq g'$*, such that* $\mathcal{E}, \mathcal{E}'$ *are indistinguishable to the agent* a_i *through an arbitrary existing observation method* $o_k(\varepsilon_1, \ldots, \varepsilon_k)$*, where* $\varepsilon_1, \ldots, \varepsilon_k \in g$ *and* $\varepsilon_1, \ldots, \varepsilon_k \in g'$*.*

Thus, if $[\mathcal{E}_i]\varphi$ is true in some state $g \in \mathbb{G}$, then by adopting observation method o_k, the local states of the agent a_i about the environment (its expectations) remains the same. An eagle expects to catch a sparrow in a cave, if and only if *wherever* it adopts the observation method "chasing the sparrow" above, the sparrow will appear close to it.

The last two lines in the **Definition 5.** hybridise expectation language $\mathcal{L}$. Satisfaction operator $@_s$ is considered as an observation operator. Thus, a formula such as $@_s\varphi$ says there is an observation about the world state labelled as s which makes φ **true** in the agent's mind. For example, by assigning the current cave to s, and "seeing the sparrow" to p $(= \varepsilon_2)$, $@_s p$ tells us the sentence "I am seeing a sparrow in cave s." This formula remains valid even though the eagle's current cave is no longer s.

Observation operators are in fact *normal modal operators* (i.e. $@_s(\varphi \Rightarrow \psi) \Rightarrow (@_s\varphi \Rightarrow @_s\psi)$). However, specially it is a *self-dual operator* $(@_s\varphi \Leftrightarrow \neg @_s \neg\varphi)$. This can be read as *for all* observations about s, φ holds in the agent's mind if and only if *there exists* no observation about s that brings $\neg\varphi$ to its mind. We can use '*for all*' and '*there exists*' in this sentence interchangeably. It also allows the expression of state equality "In cave s, it is also named Happy Cave", by $@_s u$ if u represents 'Happy Cave'.

A formula with both observation operator and expectation modality is more interesting. $@_s\langle\mathcal{E}_i\rangle t$ will tell us that the eagle expects one of the next caves from s will be t. In other words, there is an observation about the connection from the cave s to the cave t. $@_s[\mathcal{E}_i]t$ strongly asserts that t is the only subsequent cave the eagle expects (since t is true at a unique world).

3.4 Expectation reasoning – Observation logic

The construction of expectation logic is strongly dependent on two crucial factors: the set of observations, which provides a basis to observation interpretation function π for assigning truth values to formulae, and the set of observation methods which is the skeleton for constructing accessibility relation between expectations. Unfortunately, due to limitations of primitive sensors and effectors, an agent will not always be able to obtain all observations about the real world.

Therefore, in such conditions a rational agent should carefully select its observations in order to maximise the synchronisation between its mental models and the real world. Thus, when the sparrow flies into a dark passageway, the eagle can no longer take an observation of its prey. Yet, based on its existing expectations (mental images) and available observation methods at the current world, the eagle can still deliberate and determine the next observation to take. Such deliberation is possible since the eagle's reasoning now relies on another model – the *attention model*.

Definition 7 ***(Attention model)*** *A model of the mental states at world s is called an* attention model $\mathcal{A}_i(s) = \langle W_e, \sim^i_{@_s}, \rho_s \rangle$ *where W_e is the set of expectation worlds which are uniquely named by primitive propositions $p \in \Phi$, the function $\rho_s : \Xi \rightarrow Pow(W_e)$ interprets what are possible expectations for an observation at s, and $\sim^i_{@_s} \subseteq W_e \times W_e$ is an* observability accessibility relation.

The semantics of observation logic are defined via the satisfaction relation $\models_s$ as follows

- $\langle \mathcal{A}_i(s), w \rangle \models_s t$ *iff* $w \in \rho_s(t)$ *(for all $t \in \Xi$)*
- $\langle \mathcal{A}_i(s), w \rangle \models_s p$ *iff* $\rho_s(p) = \{w\}$ *(for all $p \in W_e$)*
- $\langle \mathcal{A}_i(s), w \rangle \models_s \neg\varphi$ *iff* $\langle \mathcal{A}_i(s), w \rangle \not\models \varphi$
- $\langle \mathcal{A}_i(s), w \rangle \models_s \varphi \vee \psi$ *iff* $\langle \mathcal{A}_i(s), w \rangle \models_s \varphi$ *or* $\langle \mathcal{A}_i(s), w \rangle \models_s \psi$
- $\langle \mathcal{A}_i(s), w \rangle \models_s \varphi \wedge \psi$ *iff* $\langle \mathcal{A}_i(s), w \rangle \models_s \varphi$ *and* $\langle \mathcal{A}_i(s), w \rangle \models_s \psi$
- $\langle \mathcal{A}_i(s), w \rangle \models_s \varphi \Rightarrow \psi$ *iff* $\langle \mathcal{A}_i(s), w \rangle \not\models_s \varphi$ *or* $\langle \mathcal{A}_i(s), w \rangle \models_s \psi$
- $\langle \mathcal{A}_i(s), w \rangle \models_s \langle \mathcal{O}_i \rangle \varphi$ *iff* $\langle \mathcal{A}_i(s), w' \rangle \models_s \varphi$ *for some w' such that* $w \sim^i_{@_s} w'$
- $\langle \mathcal{A}_i(s), w \rangle \models_s [\mathcal{O}_i] \varphi$ *iff* $\langle \mathcal{A}_i(s), w' \rangle \models_s \varphi$ *for all w' such that* $w \sim^i_{@_s} w'$
- $\langle \mathcal{A}_i(s), w \rangle \models_s @_p \varphi$ *iff* $\langle \mathcal{A}_i(s), w_p \rangle \models_s \varphi$ *where w_p is the denotation of p.*

Definition 8 ***(Observation accessibility relation)*** *An expectation q is observable (reachable) from an expectation p, $p \sim^i_{@_s} q$ iff there exists an observation method $o_k(\varepsilon_1, \ldots, \varepsilon_k)$ where*

- $\varepsilon_1, \ldots, \varepsilon_k$ *are valid at g_s*
- $p \in \{\varepsilon_1, \ldots, \varepsilon_k\}$
- $\exists \mathcal{E} \in o_k(\varepsilon_1, \ldots, \varepsilon_k)$*, such that* $q \in \mathcal{E}$*.*

Hence $[\mathcal{O}_i]\varphi$ says from the current expectation, φ will hold after any available observation method is carried out. So if the eagle is currently expecting that it will see the sparrow, it will expect the proposition q $(= \varepsilon_4)$– "the sparrow is caught" holds (i.e. observable) regardless of what observation methods "chasing the sparrow" or "staying in the cave" is taken. $\langle \mathcal{O}_i \rangle \varphi$ however says from the current expectation, there are only some observation methods that would bring φ into the agent's mind.

The expectation operator @ is defined similarly to the observation operator @. $@_p\varphi$ hence asserts that there is an expectation p where φ holds. Hence, if p is the expectation "seeing a sparrow", and r is "there is some light", $@_p r$ is read "There is some light whenever I expect to see a sparrow".

$$\frac{@_p\langle \mathcal{O}_i \rangle \varphi}{\begin{array}{c} @_p\langle \mathcal{O}_i \rangle a \\ @_a\varphi \end{array}} (\Diamond); \quad \frac{\neg @_p\langle \mathcal{O}_i \rangle \varphi; @_p\langle \mathcal{O}_i \rangle q}{\neg @_q\varphi} (\neg\Diamond); \quad \frac{@_p[\mathcal{O}_i]\varphi \quad @_p\langle \mathcal{O}_i \rangle q}{@_q\varphi} (\Box); \quad \frac{\neg @_p[\mathcal{O}_i]\varphi}{\begin{array}{c} @_p\langle \mathcal{O}_i \rangle a \\ \neg @_a\varphi \end{array}} (\neg\Box)$$

Table 1. Some modality inference rules for observation system

The eagle's deliberation can now be formalised using the rules in **Table 1**. Consider the case when the cave system consists of only one cave looping back on itself. The passageway is dark, but there is some light in the cave. The eagle

can only see the sparrow if there is some light $@_p r$. When the sparrow flies into the dark passageway, the eagle would say I expect whatever I do, I can only see the sparrow in this cave s, $@_p[\mathcal{O}_i]s$. Whenever I see the sparrow, I know a way to catch it $@_p\langle\mathcal{O}_i\rangle q$. Using the ($\Box$) rule the eagle will decide to stay in s to catch the sparrow $@_q s$.

However, if the cave system consists of an infinite number of caves linking by unidirectional passageways, the deliberation will be slightly changed. After some observations, the eagle would discover the next cave cannot be s ($\neg@_s\langle\mathcal{E}_i\rangle s$), but another cave t ($@_s\langle\mathcal{E}_i\rangle t$). Hence, when the sparrow disappears, the eagle would say, I expect whatever I do, I can only observe the sparrow in the next cave $@_p[\mathcal{O}_i]\langle\mathcal{E}_i\rangle t$. Whenever I see the sparrow, I know a way to catch it $@_p\langle\mathcal{O}_i\rangle q$. Using the ($\Box$) rule, we will be able to derive $@_q\langle\mathcal{E}_i\rangle t$, which suggests to capture the sparrow, the eagle should follow the sparrow into the passageway linking to the next cave t.

4 Labelled BDI logics

Rao and Georgeff's *Belief-Desire-Intention* (BDI) logics are one of the most successful theories for agent specification or verification languages in agent research community. Following the philosopher Bratman [7], a formalisation of the three mental attitudes *belief*, *desire*, *intention* and their interactions has been investigated as characterisations of an agent. Most work on BDI logics focused on possible relationships between these three mental attitudes by adding different constraints on their interactions. According to Bratman [7], assuming an eagle is a rational agent, the eagle will not intend to catch the sparrow if it believes the sparrow is uncatchable. But it still tries its best (intends) to catch the sparrow for hunger though it does not believe it can catch it (*asymmetry thesis*). Also, the eagle may believe it can catch the sparrow, but it is not necessary that it intends to catch a sparrow now (*non-transference principle*). Additionally, if the eagle intends to catch a sparrow for hunger, though it believes catching the sparrow is certainly energy burning, it will not intend to burn out its energy (*side-effect free principle*). Rao and Georgeff [17] formally put these constraints in the following proposition:

Proposition 1 *A rational agent a_i must satisfy the following principles:*

- *Asymmetry thesis: An agent cannot have beliefs inconsistent with intentions, but can have incomplete beliefs about its intentions.*
  - *(BI-ICN)* $\not\models [\mathcal{I}_i]\varphi \wedge [\mathcal{B}_i]\neg\varphi$
  - *(BI-ICM)* $\exists\mathfrak{M},\mathfrak{M} \models [\mathcal{I}_i]\varphi \wedge \neg[\mathcal{B}_i]\varphi$
- *Non-transference principle: An agent who believes φ should not be forced to intend φ.*
  - *(BI-NT)* $\exists\mathfrak{M},\mathfrak{M} \models [\mathcal{B}_i]\varphi \wedge \neg[\mathcal{I}_i]\varphi$
- *Side-effect free principle: if an agent intends φ and believes that $\varphi \Rightarrow \psi$, it should not be forced to intend the side-effect ψ.*
  - *(BI-SE)* $\exists\mathfrak{M},\mathfrak{M} \models [\mathcal{I}_i]\varphi \wedge [\mathcal{B}_i](\varphi \Rightarrow \psi) \wedge \neg[\mathcal{I}_i]\psi$

These constraints also apply for belief-goal*, and* goal-intention *pairs.*

The constraints between these mental attitudes are set based on the relationships between the accessibility relations. Three well known cases of these systems were studied by Cohen and Levesque in term of *realism* ($\mathbb{B}_i \subseteq \mathbb{G}_i$) [8], by Rao and Georgeff in terms of *strong realism* ($\mathbb{G}_i \subseteq \mathbb{B}_i$) [18] and *weak realism* ($\mathbb{G}_i \cap \mathbb{B}_i \neq \varnothing$) [17]. Rao and Georgeff [17] also concluded that weak realism is the only system that satisfies all desirable properties of a rational agent.

The real world model of BDI logics is viewed as a single past-branching time future tree. Each possible world of any belief, desire (goal) or intention models consists of a subtree of the above temporal structure. In other words, they are different images of the real world in the agent's mind. However, a major drawback of BDI logics is that it is very unclear which part of the real world temporal structure should be in an agent's mental attitudes, beliefs, desires, or intentions. There is no formal correspondence from the mental models to the world structure. Consider the eagle chasing the sparrow again. BDI language is unable to tell when a particular event would happen. Thus the sentence "eventually the sparrow will be caught" ($[\mathcal{B}_i]\Diamond q$) can be interpreted to be true at two different time points t_1 and t_2. Regardless of how many observations it can take, the eagle using BDI logics is unable to tell if t_1 and t_2 is a unique time point. A BDI agent would continue to seek for a sparrow after having one caught. Expectation-observation logic however allows us to tell if these points are equal by $@_{t_1}t_2$. Hence, if it is the case, the eagle will drop all subsequent goals to catch the sparrow in the future in the cave system.

A translation from BDI languages to our expectation-observation language can be useful to attain the new expressive power. Firstly, we can construct our observation system using similar temporal structure. Our expectation modality in the system becomes the expectation about the future, equivalently to future modality.

Definition 9 (Mental translation) *A mental translation taking BDI formulae to expectation-observation formulae is defined as follows:*

Expectation model	*Attention model*
$-\ \Box\varphi \stackrel{def}{=} [\mathcal{E}_i]\varphi$	
$-\ \Diamond\varphi \stackrel{def}{=} \langle\mathcal{E}_i\rangle\varphi$	$-\ [\mathcal{B}_i]\varphi \stackrel{def}{=} [\mathcal{O}_i^{\mathcal{B}}]\varphi$
$-\ \varphi\mathcal{U}\psi \stackrel{def}{=} \langle\mathcal{E}_i\rangle(s\wedge\psi)\wedge[\mathcal{E}_i](\langle\mathcal{E}_i\rangle s \Rightarrow \varphi)$	$-\ [\mathcal{G}_i]\varphi \stackrel{def}{=} [\mathcal{O}_i^{\mathcal{G}}]\varphi \wedge \neg\varphi$
$-\ \bigcirc\varphi \stackrel{def}{=} \langle\mathcal{E}_i\rangle(s \wedge \varphi) \wedge [\mathcal{E}_i](\neg\langle\mathcal{E}_i\rangle s)$	$-\ [\mathcal{I}_i]\varphi \stackrel{def}{=} \langle\mathcal{O}_i^{\mathcal{I}}\rangle p \wedge @_p\varphi$

where $W_e^{\mathcal{B}} = \{p \in \Phi \mid @_s p\}, W_e^{\mathcal{G}} = W_e^{\mathcal{I}} = \{p \in \Phi \mid @_s\langle\mathcal{E}_i\rangle p\}$

The crucial difference between $\mathcal{O}_i^{\mathcal{B}}, \mathcal{O}_i^{\mathcal{G}}, \mathcal{O}_i^{\mathcal{I}}$ is only based on which primitive expectations are selected into the sets of possible worlds $W_e^{\mathcal{B}}, W_e^{\mathcal{G}}, W_e^{\mathcal{I}}$. At a particular world named as s, beliefs are its mental states, where goals and intentions are its mental states about what it expects to happen next if it takes more observation.

By this translation, *beliefs* now are an agent's expectations of what will be observable. If there is no observation linking to the expectation, a belief may well

be false. *Goals* are what an agent expects to be observable (in future), but are not observable now. This definition satisfies a number of required properties for goals [20]. Observable also means 'achievable' or 'possible' – the agent only has goals that it believes achievable. However, '$\neg\varphi$' guarantees the goal is *unachieved* or at least expected to be unachieved. Consistency and persistence are just normal logical properties of goals. Although *intention* could be defined as $\langle\mathcal{O}_i^{\mathcal{I}}\rangle\varphi$, the above definition insists the agent has committed to a specific observation method o_k to achieve φ at the subsequent expectation p.

The definition also satisfies weak-realism constraint. The overlapping between beliefs and goals is the set of expectations that hold now and at some time points in the future ($@_s p$ and $@_s\langle\mathcal{E}_i\rangle p$). However, the agent does not have direct observation of the expectations at the current observation. The non-overlapped part of goals are what the agent does not expect to observe now ($@_s\neg p$), but it expects them to be observable in future ($@_s\langle\mathcal{E}_i\rangle p$). Conversely, a belief is not in the goal set if there is a direct observation now ($@_s p$) or the agent expects it will not happen at all ($\neg@_s\langle\mathcal{E}_i\rangle p$).

Similarly, the overlapping between beliefs and intentions is where the subsequent expectation p is also in the set of expectations of the current observation. An intentions will no longer be in the set of beliefs if the resultant expectation of the observation method o_k which is committed to the intention, does not hold at the current observation g_s. On the other hand, a belief is out of the intention set if there is no observation method o_k links to p. In other words, the agent believes φ is observable, but it has no way to observe φ now.

The overlapping between goals and intentions is where the expectation p is in the set of expectations of all possible observations about to occur in the whole system. An intention may not be a goal if it is already observable at the current observation. On the other hand, a goal will not be a specific intention if the agent expects φ can only be achieved by other observation methods not the one associated with the intention.

By this definition, it is clear that the following proposition holds:

Proposition 2 *The agent modelled by the above observation-expectation system is a rational agent. That is, it satisfies asymmetry thesis, side-effect free and non-transference principles.*

Proof. See **Appendix A**.

5 Conclusion and further work

In this paper we apply hybrid logic to address a well-known unresolved problem in the agent research community, the computational grounding problem and to introduce a formal correspondence between mental and computational models. It is certainly not yet another paper about hybrid logics. Hence, we do not show decidability, completeness results which have been deeply studied by other researchers [6], [1], [3], [4]. Instead, our crucial argument here is that any concrete computational models are extensional whereas any mental models are intensional. Agent specifications using orthodox modal languages can only express intensional aspects and therefore fail to make connection to extensional aspects.

Apart from BDI logics discussed above, a major strand of research led by the work of Fagin et al. [11], [10] has attempted to bring the external to the internal perspective using interpreted systems as the basis of epistemic logic. This approach tightly connects the internal to the external. The approach hence started from a perfectly synchronised mental model with the environment where everything is directly reflected to every agent's mind. Then, the connection is loosened to reflect the fact that agents are imperfect. A dilemma has arisen in this investigation [10, Chapter 11], simultaneity (time synchronisation) strongly affects the attainability of (common) knowledge, but true simultaneity cannot be attained in reality. Interestingly, the resolution of this paradox leads to the ability to record time points and the granularity of time – timestamped (common) knowledge. However, unlike hybrid languages, their naming mechanisms cannot be manipulated and hence reasoned as formulae. An extension for time observation in our framework to link with this work hence appears very promising.

An extension of Fagin et al.'s interpreted systems, $\mathcal{VSK}$ systems and $\mathcal{VSK}$ logic [23], provided another attempt to formalise the connection between the states of agents within a system and the percepts received by them. The imperfect situation is captured by using the notion of "partial observability" in POMDPs [15] through $\mathcal{V}$ and $\mathcal{S}$ modalities. Their knowledge modality $\mathcal{K}$ remains the same as modal epistemic logic [10]. There are two crucial drawbacks of this work. Firstly, *visibility* function is similar to *observation function* by van der Meyden [19] which is only capable of capturing the discrete states of an environment but not the relationships between them. Secondly, $\mathcal{VSK}$ fails to fully capture human perception which can be faulty. Our framework using the idea from hybrid logic overcomes the former problem by letting an observation about a relationship be mapped onto an expectation of named state (e.g. $@_s\langle\mathcal{E}_i\rangle t$). The second problem is resolved by adding hypothetical observations from agents' effectors into the concept of observability.

Finally, our chief further work is to show completeness and correspondence results. However it is also worth noting that our work is principally based upon fibring techniques by Gabbay [12] and analytic deduction via labelled deductive systems **LKE** by D'Agostino and Gabbay [9]. These works provide a potential approach towards a uniform way of combining logical systems, hence modelling cooperative reasoning in interactive dynamic environment.

References

1. C. Areces, P. Blackburn, and M. Marx. Hybrid logics: Characterization, interpolation and complexity. *Journal of Symbolic Logic*, 66(3):977 – 1010, 2001.
2. P. Blackburn. Nominal tense logic. *Notre Dame Journal of Formal Logic*, 34(1):56–83, 1993.
3. P. Blackburn. Internalizing labelled deduction. *Journal of Logic and Computation*, 10:137–168, 2000.
4. P. Blackburn. Representation, reasoning, and relational structures: a hybrid logic manifesto. *Logic Journal of the IGPL*, 8(3):339–625, 2000.

5. P. Blackburn, M. de Rijke, and Y. Venema. *Modal logic*. Cambridge University Press, 2001.
6. P. Blackburn and M. Tzakova. Hybrid languages and temporal logic. *Logic Journal of the IGPL*, 7:27–54, 1999.
7. M. E. Bratman. *Intention, Plans, and Practical Reason*. Harvard University Press, 1987.
8. P. R. Cohen and H. J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42:213–261, 1990.
9. M. D'Agostino and D. Gabbay. A generalization of analytic deduction via labelled deductive systems. Part I: Basic substructural logics. *Journal of Automated Reasoning*, 13:243–281, 1994.
10. R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about Knowledge*. The MIT Press, Cambridge, Massachusetts, 1995.
11. R. Fagin, J. Y. Halpern, and M. Y. Vardi. What can machines know? on the properties of knowledge in distributed systems. *Journal of the ACM*, 39(2):328–376, 1992.
12. D. Gabbay. *Fibring Logics*, volume 38 of *Oxford Logic Guides*. Oxford University Press, 1999.
13. P. B. Gove, editor. *Webster's Revised Unabridged Dictionary*. Merriam Webster Inc., 3rd edition, 2002.
14. J. Hintikka. *Knowledge and Belief: An Introduction to the Logic of The Two Notions*. Cornell University Press, Ithaca, New York, 1962.
15. L. P. Kaelbling, M. L. Littman, and A. R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101:99–134, 1998.
16. A. Rao. AgentSpeak(L): BDI Agents speak out in a logical computable language. In W. Van de Velde and J. Perram, editors, *Proceedings of the Seventh Workshop on Modelling Autonomous Agents in Multi-Agent World (MAAMAW'96)*, volume 1038 of *Lecture Notes in Artificial Intelligence*, pages 42–55, Eindhoven, The Netherlands, 1996. Springer-Verlag.
17. A. Rao and M. Georgeff. Asymmetry thesis and side-effect problems in linear-time and branching-time intention logics. In J. Myopoulos and R. Reiter, editors, *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 498–505, Sydney, Australia, 1991. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA.
18. A. Rao and M. Georgeff. Modelling rational agents within a BDI-architecture. In R. Fikes and E. Sandewall, editors, *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning*, pages 473–484, Cambridge (USA), 1991.
19. R. van der Meyden. Common knowledge and update in finite environments. *Information and Computation*, 140(2):115–157, 1998.
20. M. Winikoff, L. Padgham, J. Harland, and J. Thangarajah. Declarative and procedural goals in intelligent agent systems. In *Eighth International Conference on Principles of Knowledge Representation and Reasoning*, pages 470–481, 2002.
21. M. Woolridge. Computationally Grounded Theories of Agency. In E. H. Durfee, editor, *Proceedings of the Fourth International Conference on Multi-Agent Systems (ICMAS 2000)*, volume 9. IEEE Press, 2000.
22. M. Woolridge. *An introduction to MultiAgent System*. John Wiley & Sons, Chichester, England, 2002.
23. M. Woolridge and A. Lomuscio. Reasoning about visibility, perception, and knowledge. In N. Jennings and Y. Lespérance, editors, *Intelligent Agents VI*, volume Lecture Notes in AI Volume. Springer-Verlag, 2000.

A Proofs

	BDI language	Expectation-observation language
BI-ICN	$\not\models [\mathcal{I}_i]\varphi \wedge [\mathcal{B}_i]\neg\varphi$	$\not\models \langle\mathcal{O}_i^{\mathcal{I}}\rangle p \wedge @_p\varphi \wedge [\mathcal{O}_i^{\mathcal{B}}]\neg\varphi$
BI-ICM	$\exists\mathfrak{M},\mathfrak{M} \models [\mathcal{I}_i]\varphi \wedge \neg[\mathcal{B}_i]\varphi$	$\exists\mathfrak{M},\mathfrak{M} \models \langle\mathcal{O}_i^{\mathcal{I}}\rangle p \wedge @_p\varphi \wedge \langle\mathcal{O}_i^{\mathcal{B}}\rangle q \wedge @_q\neg\varphi$
GI-ICN	$\not\models [\mathcal{I}_i]\varphi \wedge [\mathcal{G}_i]\neg\varphi$	$\not\models \langle\mathcal{O}_i^{\mathcal{I}}\rangle p \wedge @_p\varphi \wedge [\mathcal{O}_i^{\mathcal{G}}]\neg\varphi \wedge \varphi$
GI-ICM	$\exists\mathfrak{M},\mathfrak{M} \models [\mathcal{I}_i]\varphi \wedge \neg[\mathcal{G}_i]\varphi$	$\exists\mathfrak{M},\mathfrak{M} \models \langle\mathcal{O}_i^{\mathcal{I}}\rangle p \wedge @_p\varphi \wedge \langle\mathcal{O}_i^{\mathcal{G}}\rangle q \wedge (@_q\neg\varphi \vee \varphi)$
BG-ICN	$\not\models [\mathcal{G}_i]\varphi \wedge [\mathcal{B}_i]\neg\varphi$	$\not\models ([\mathcal{O}_i^{\mathcal{G}}]\varphi \wedge \neg\varphi) \wedge [\mathcal{O}_i^{\mathcal{B}}]\neg\varphi$
BG-ICM	$\exists\mathfrak{M},\mathfrak{M} \models [\mathcal{G}_i]\varphi \wedge \neg[\mathcal{B}_i]\varphi$	$\exists\mathfrak{M},\mathfrak{M} \models ([\mathcal{O}_i^{\mathcal{G}}]\varphi \wedge \neg\varphi) \wedge \neg[\mathcal{O}_i^{\mathcal{B}}]\varphi$
BG-NT	$\exists\mathfrak{M},\mathfrak{M} \models [\mathcal{B}_i]\varphi \wedge \neg[\mathcal{G}_i]\varphi$	$\exists\mathfrak{M},\mathfrak{M} \models [\mathcal{O}_i^{\mathcal{B}}]\varphi \wedge (\neg[\mathcal{O}_i^{\mathcal{G}}]\varphi \vee \varphi)$
BI-NT	$\exists\mathfrak{M},\mathfrak{M} \models [\mathcal{B}_i]\varphi \wedge \neg[\mathcal{I}_i]\varphi$	$\exists\mathfrak{M},\mathfrak{M} \models [\mathcal{O}_i^{\mathcal{G}}]\mathcal{B} \wedge ([\mathcal{O}_i^{\mathcal{I}}]\neg p \vee \neg@_p\varphi)$
GI-NT	$\exists\mathfrak{M},\mathfrak{M} \models [\mathcal{G}_i]\varphi \wedge \neg[\mathcal{I}_i]\varphi$	$\exists\mathfrak{M},\mathfrak{M} \models ([\mathcal{O}_i^{\mathcal{G}}]\varphi \wedge \neg\varphi) \wedge ([\mathcal{O}_i^{\mathcal{I}}]\neg p \vee \neg@_p\varphi)$
BI-SE	$\exists\mathfrak{M},\mathfrak{M} \models [\mathcal{I}_i]\varphi \wedge$ $[\mathcal{B}_i](\varphi \Rightarrow \psi) \wedge \neg[\mathcal{I}_i]\psi$	$\exists\mathfrak{M},\mathfrak{M} \models (\langle\mathcal{O}_i^{\mathcal{I}}\rangle p \wedge @_p\varphi \wedge \langle\mathcal{O}_i^{\mathcal{B}}\rangle q \wedge @_q\neg\varphi \wedge @_p\neg\psi)$ $\vee(\langle\mathcal{O}_i^{\mathcal{I}}\rangle p \wedge @_p\varphi \wedge [\mathcal{O}_i^{\mathcal{B}}]\psi \wedge @_p\neg\psi)$
GI-SE	$\exists\mathfrak{M},\mathfrak{M} \models [\mathcal{I}_i]\varphi \wedge$ $[\mathcal{G}_i](\varphi \Rightarrow \psi) \wedge \neg[\mathcal{I}_i]\psi$	$\exists\mathfrak{M},\mathfrak{M} \models (\langle\mathcal{O}_i^{\mathcal{I}}\rangle p \wedge @_p\varphi \wedge \langle\mathcal{O}_i^{\mathcal{G}}\rangle q \wedge @_q\neg\varphi \wedge @_p\neg\psi)$ $\vee(\langle\mathcal{O}_i^{\mathcal{I}}\rangle p \wedge @_p\varphi \wedge \varphi \wedge @_p\neg\psi)$ $\vee(\langle\mathcal{O}_i^{\mathcal{I}}\rangle p \wedge @_p\varphi \wedge [\mathcal{O}_i^{\mathcal{G}}]\psi \wedge @_p\neg\psi)$
BG-SE	$\exists\mathfrak{M},\mathfrak{M} \models [\mathcal{G}_i]\varphi \wedge$ $[\mathcal{B}_i](\varphi \Rightarrow \psi) \wedge \neg[\mathcal{G}_i]\psi$	$\exists\mathfrak{M},\mathfrak{M} \models ([\mathcal{O}_i^{\mathcal{G}}]\varphi \wedge \neg\varphi \wedge \neg[\mathcal{O}_i^{\mathcal{B}}]\varphi \wedge (\neg[\mathcal{O}_i^{\mathcal{G}}]\psi \vee \psi))$ $\vee([\mathcal{O}_i^{\mathcal{G}}]\varphi \wedge \neg\varphi \wedge [\mathcal{O}_i^{\mathcal{B}}]\psi \wedge (\neg[\mathcal{O}_i^{\mathcal{G}}]\psi \vee \psi))$

Table 2. BDI constraints in expectation observation language

From **Table 2**, it is clear that (BI-ICN), (GI-ICN) and (BI-ICN) constraints are satisfied by our framework. (BI-ICM) happens when intention is not in the belief set. From the translation, it says p, q are indistinguishable to an observation method and they reside separately in the two sets intention and belief respectively. Similarly for goal-intention pair except that the agent can intend an achieved goal (i.e. no longer goal) for example to maintain its achievement. (BG-ICM) may look counter-intuitive. However, our translation insists the difference. Beliefs are based on the current observations only, where goals can come from different sources (from other agents – e.g. your boss). Hence, this constraint seems appropriate in a multi-agent system. The asymmetry thesis principle is preserved under the new language.

(BG-NT) appears obvious, since φ is observable now, the agent can hold a belief about φ without having φ as its goal. The emphasis of the commitment to an intention can now be used for (BI-NT) and (GI-NT). Commitment ties a specific mental state p to an intention. Therefore, the agent will not intend φ but it can still believe or have φ as goal. For example, a person does not intend war in Iraq, but believes war is there. This also seems intuitive in a multi-agent environment. The non-transference principle is hence preserved under expectation observation logic.

(BI-SE) can be rewritten as $([\mathcal{I}_i]\varphi \wedge \neg[\mathcal{B}_i]\varphi \wedge \neg[\mathcal{I}_i]\psi) \vee ([\mathcal{I}_i]\varphi \wedge [\mathcal{B}_i]\psi \wedge \neg[\mathcal{I}_i]\psi)$ which appears to be a restricted version of (BI-ICM) and (BI-NT). So if the agent's belief is incomplete about the intention and at the intended expectation p, ψ does not hold, the agent would not worry about the side effect. On the other hand, assuming the agent believes ψ, according to (BI-NT) it is not forced to intend ψ. The translation clarifies the situations where side-effect free can be satisfied. We can achieve similar results for (GI-SE) and (BG-SE).

Author Index