národní
úložiště
šedé
literatury

Formal Reasoning in the Bang3 Multi-Agent System

Beuster, Gerd
2003

Institute of Computer Science
Academy of Sciences of the Czech Republic

Formal Reasoning in the Bang3 Multi-Agent System

Gerd Beuster

Technical report No. V-889

Pod Vodárenskou věží 2, 182 07 Prague 8, phone: (+4202) 6605 1111, fax: (+4202) 86 585 789,
e-mail:ics@cs.cas.cz

Institute of Computer Science
Academy of Sciences of the Czech Republic

Formal Reasoning in the Bang3 Multi-Agent System

Gerd Beuster[1]

Technical report No. V-889

Abstract:

Bang3 is a Multi-Agent-System platform focusing on soft computing. In this paper, we describe the formal logical reasoning component of *Bang3*. We describe how formal logics are applied to *Bang3* Multi-Agent Systems, and the reasoning procedures used. We present a method to transform Description Logics into Horn rules.

Keywords:
MAS, Bang, Logic, Deduction, Description Logics, Agents

[1] gb@cs.cas.cz

Chapter 1

Introduction

The use of distributed Multi-Agent Systems (MAS) instead of monolithic programs has become a popular topic both in research and application development. Autonomous agents are small self-contained programs that can solve simple problems in a well-defined domain.[19] In order to solve complex problems, agents have to collaborate, forming Multi-Agent Systems (MAS). A key issue in MAS research is how to generate MAS configurations that solve a given problem.[10] In most Systems, an intelligent (human) user is required to set up the system configuration. Developing algorithms for automatic configuration of Multi-Agent Systems is a major challenge for AI research.

Bang3 is a platform for the development of Multi-Agent Systems.[16] Its main areas of application are soft computing methods (genetic algorithms, neural networks, fuzzy controllers) on single machines and clusters. As Multi-Agent Systems, *Bang3* applications require a number of cooperating agents to fulfill a given task. So far, MAS are created and configured manually.[1] In this paper, we introduce a logical reasoning component for *Bang3*. With this component, *Bang3* system configurations can be created automatically and semi-automatically. The logical description of MAS opens *Bang3* for interaction with ontology based distributed knowledge systems like the Semantic Web.[14]

The description of *Bang3* by formal logics enhances the construction, testing, and application of *Bang3*-MAS in numerous ways:

- System Checking

  A common question in Multi-Agent System design is whether a setup has certain properties. By the use of formal descriptions of the agents involved in a MAS and their interactions, properties of the MAS can be (dis-)proofed.[18]

- System Generation

  Starting with a set of requirements, the reasoning component can be used to create a MAS. The formal logical component augments evolutionary means of agent configuration that are already present in *Bang3*.[5]

- Interactive System Generation

  The reasoning component can also be used to create agents in semi-automated ways. Here, the reasoning component acts as a helper application that aids a user in setting up MAS by making suggestions.

- Interaction with ontology based systems

  There is a growing interest in creating common logical frameworks (ontologies) that allow the interaction of independent, distributed knowledge based system. The most prominent one is the Semantic Web, which attempts to augment the World Wide Web with ontological knowledge. Using formal logics and reasoning in *Bang3* allows to open this world to *Bang3*.

In order to satisfy these requirements, the logical formalism must fulfill the following requirements:

1. It must be expressive enough to describe *Bang3* MAS.

[1] Simple MAS for the approximation of polynomial functions can also be developed evolutionary.[5]

2. There must be efficient reasoning methods.
3. It should be suitable to describe ontologies
4. It should interface with other ontology based systems.

There is a lot of research in how to use formal logics to model ontologies. The goal of this research is to find logics that are both expressive enough to describe ontological concepts, and weak enough to allow efficient formal reasoning about ontologies. Description Logics are widely accepted as a family of languages that fulfill both requirements.[9] In short, description logics are equivalent to subsets of predicate logic restricted to unary and binary predicates.[2] Although this restriction is acceptable to describe ontologies, it is too limited to describe configurations of MAS, as we will show in chapter 3.1. Therefore, we combine description logics with traditional logical programs written in Prolog.

This technical report is split into two parts. Next, we describe the logical formalism used to reason about agents and MAS, and the reasoning methods. In the second part, we describe how these methods are practically applied to the *Bang3* MAS platform.

Chapter 2

Formal Reasoning

The most natural approach to formalize ontologies is the use of First Order Predicate Logics (FOL). This approach is used by well known ontology description languages like Ontolingua[11] and KIF[13].

The disadvantage of FOL-based languages is the expressive power of FOL. FOL is undecidable[8], and there are no efficient reasoning procedures. A lot of effort is put into the research of subset of FOL that are decidable and allow efficient reasoning algorithms. Nowadays, the de facto standard for ontology description language for formal reasoning is the family of description logics. Description logics are equivalent to subsets of first order logic restricted to predicates of arity one and two.[7] They are known to be equivalent to modal logics.[1]

Description logics are used in the Semantic Web, a project of the Internet standardization body W3C. "The Semantic Web is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation."[4] Description logics is also a main topic of interest in other projects dealing with the standarization of agent communications.

For the purpose of describing multi-agent systems, description logics is sometimes to weak. An example for this is given in chapter 3.1. In these cases, we want to have a more expressive formalism. We decided to use Prolog-style logic programs for this. In the following chapters, we describe how both approaches can be combined.

2.1 Ontologies

Description logics and Horn rules are orthogonal subsets of first order logic.[7] During the last years, a number of approaches to combine these two logical formalisms in one reasoning engine have been proposed. Most of these approaches use tableaux-style reasoners for description logics and combine them with Prolog-style Horn rules. In [15], Hustadt and Schmidt examined the relationship between resolution and tableaux proof systems for description logics. Baumgartner, Furbach and Thomas propose a combination of tableaux based reasoning and resolution on Horn logic [3]. Vellion [22] examines the relative complexity of SL-resolution and analytic tableau. The limits of combining description logics with horn rules are examined by Alon Y. Levy and Marie-Christine Rousset [17]. Borgida [6] has shown that Description Logics and Horn rules are orthogonal subsets of first oder logic.

Our approach differs insofar as we are not using a tableau-algorithm for reasoning, but Prolog's build in resolution-algorithm. This approach allows to freely mix description logics with Horn rules and reason about them in a common environment. To do this, description logics expressions are transformed into Horn rules and added to the "native" Horn rules. See figure 2.1.

Another advantage of our approach is that no special reasoning engine beside Prolog's build-in mechanisms is required after the description logics part have been transformed into Prolog rules. In this section, we describe how description logic statements are transformed into Prolog rules.

2.2 Transforming a subset of $\mathcal{ALC}$ into Horn rules

We start with a subset of the basic description logic $\mathcal{ALC}$ that does not contain all-quantifiers, negation, and the $\sqcup$-operator. After that, the subset is extended by the missing parts until all $\mathcal{ALC}$-constructs can be transformed.

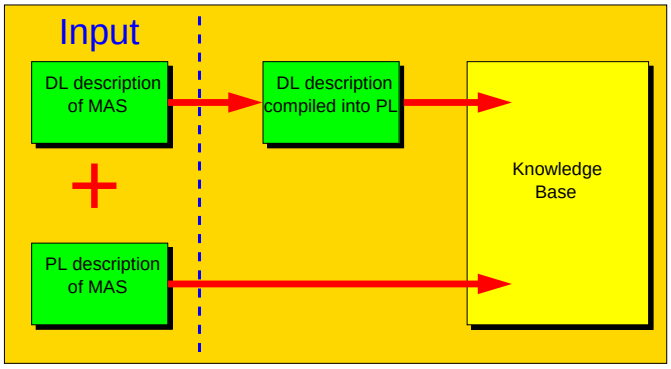

Figure 2.1: Description logics expressions are transformed into Horn rules and added to the "native" Horn rules.

Construct name	Syntax	Semantics
negation	$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
conjunction	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
disjunction	$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
existential restriction	$\exists r.C$	$\{x \in \Delta^{\mathcal{I}} \mid \exists y : (x,y) \in r^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$
value restriction	$\forall r.C$	$\{x \in \Delta^{\mathcal{I}} \mid \forall y : (x,y) \in r^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$

Table 2.1: Semantics of $\mathcal{ALC}$. See [2].

Concept descriptions in this subset of $\mathcal{ALC}$ can be defined inductively. Let C be a set of concept names and R be a set of role names:

- All elements of C are concept descriptions.
- If c is a concept description, so is $\neg c$.
- If c and d are concept descriptions, so are $c \sqcup d$ and $c \sqcap d$.
- if c is a concept description and r is a role name, $\exists c.r$ and $\forall c.r$ are concept description.

The semantic of concept descriptions can be defined via an interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$. $\Delta^{\mathcal{I}}$ is the domain. Interpretation function $\cdot^{\mathcal{I}}$ maps concept names to subsets of $\Delta^{\mathcal{I}}$ and role names to binary relations on $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. The semantics of composed concept descriptions is given in table 2.1.[2]

Concept definitions of the form $C := D$ can be removed by *unfolding* if they are acyclic. In the following, we assume all concept descriptions are in negative normal form (NNF), i.e. negation appears only in front of atomic concepts. Concept subsumptions are written as $C \sqsupseteq D$, meaning concept D is a subset of concept C. A finite set of concept subsumptions is called a *T-Box*. For the left side of a concept subsumption, we allow only primitive concept definitions.

Thus, all concept subsumptions (i.e. the T-Box) are of the following form: $C \sqsupseteq C_1 \sqcap \ldots \sqcap C_n$ with C a primitive concept. The *A-Box* is a set of assertional axioms $C(a)$ and $R(a,b)$, asserting that a is an instance of concept C, resp. that a, b are in relation R.

Algorithm 1 shows how T-Boxes can be transformed into Horn rules, and algorithm 2 show the transformation of A-boxes.

Proof of correctness for algorithms 1 and 2 We have to show that a) the transformation of concept definitions into predicate logic is correct and b) the resulting rules are Horn. a) follows directly from the definition of description logics in FOL. b) follows from the fact that only once a positive literal, concept_instance(C_{i_j}, X), is added to each clause. In the following for-loop, only negative literals are added. Therefore, all rules are horn.■

Algorithm 1 Transforming a T-Box into Horn rules

Require: A T-Box of DL concept definitions $DL = \{D_0, \ldots D_n\}$
with $D_i = C_{i_0} \sqsupseteq C_{i_1} \ldots C_{i_m}$. An empty set H.

1: **for all** $D_i \in DL$ **do**
2: $N = \{\text{concept\_instance}(C_{i_0}, X)\}$
3: **for all** $C_{i_j} \in D_i, \quad j \geq 1$ **do**
4: **if** C_{i_j} is an atomic concept definition **then**
5: Add $\neg$concept_instance(C_{i_j}, X) to N.
6: **end if**
7: **if** C_{i_j} is of the form $\exists r_{i_j}.C'_{i_j}$ **then**
8: Add $\neg$role_instance(r_{i_j}, X, Y) to N.
9: Add $\neg$concept_instance(C'_{i_j}, Y) to N.
10: **end if**
11: **end for**
12: Add N to H.
13: **end for**

Ensure: H is a transformation of DL into Horn rules.

Proof of completeness and soundness (SLD-Resolution) Completeness and soundness of SLD-resolution on Horn rules has been proven a number of times, e.g. in [20].■

The aim of transforming DL expressions into Horn rules is to use the Horn rules as Prolog programs. Since Horn rules are directly transferable into Prolog, we can use algorithms 1 and 2 directly to create Prolog programs. It is a well known fact that Prolog's reasoning algorithm is neither complete nor sound. In the next paragraphs, we will show that the Prolog programs produced by these algorithms are sound, but not complete.

Proof of soundness (Prolog program) The unsoundness of Prolog is due to the lack of a *occurs check* in Prolog. The statement `p(X,X), p(X,f(X))` is true in Prolog, (X is unified with the *infinite term* `f(f(f(...)))`) although there is no model for it. However, this can not happen for the programs generated by algorithm 1, because in synthesized Prolog program, variables are instantiated only with other uninstantiated variables (T-Box) or with atomic concepts names (A-Box).■

Algorithm 2 Transforming a subset of an $\mathcal{ALC}$ A-Box into Horn rules

Require: An A-Box of DL concept and role assertions $DL = \{D_0, \ldots D_n\}$
with $D_i = C(x)$ or $D_i = R(x, y)$. An empty set H.

1: **for all** $D_i \in DL$ **do**
2: **if** $D_i = C(x)$ **then**
3: Add $\{\text{concept\_instance}(C, x)\}$ to H.
4: **end if**
5: **if** $D_i = R(x, y)$ **then**
6: Add $\{\text{role\_instance}(R, x, y)\}$ to H.
7: **end if**
8: **end for**

Ensure: H is a transformation of DL into Horn rules.

Proof of (in-)completeness (Prolog program) Prolog's reasoning mechanism is incomplete, because it's depth-first search strategy may get lost in infinite branches of the search tree. Infinite branches are possible if we allow cycles in concept or role definitions, e.g. $A \sqsupseteq A \sqcap \exists B.C$. When we do not allow cycle definitions, all search trees are finite and thus Prolog's reasoning procedure is complete.■

Limitations Before extending this algorithm to handle full $\mathcal{ALC}$, it should be noted that the algorithm is limited in the way that it requires primitive concepts on the left side of concept subsumptions.

2.3 Expanding to full $\mathcal{ALC}$

From this limited description logic, it is easy to expand to a more expressive one. The following additional concepts are needed for the language $\mathcal{ALC}$: Negation, $\sqcup$-operator, nested role fillers, and all quantifiers. We also add inverse roles, resulting in the language $\mathcal{ALCI}$.

Nested role fillers Nested role fillers, e.g. concept descriptions of the form $\exists r.(C_1 \sqcap \ldots \sqcap C_n)$ can be handled by replacing lines 8 and 9 in algorithm 1 by algorithm 3. Algorithm 3 replaces nested subterms $C_{i_j} = (C_{i_{j_1}} \sqcap \ldots \sqcap C_{i_{j_k}})$ by a new unique concept γ which is defined as $\gamma \sqsupseteq C_{i_{j_1}} \sqcap \ldots \sqcap C_{i_{j_k}}$.

Algorithm 3 Additions to algorithm 1 to allow nested role fillers.

Add $\neg r_{i_j}(X, Y)$ to N.
if C_{i_j} is of the form $(C_{i_{j_1}} \sqcap \ldots \sqcap C_{i_{j_k}})$ **then**
  Let γ be a new unique concept name.
  Let H_γ be the result of applying algorithm 1 to $\gamma \sqsupseteq C_{i_j}$.
  Add H_γ to H.
  Add $\neg$concept_instance(γ, Y) to N.
else
  Add $\neg$concept_instance(C_{i_j}, Y) to N.
end if

Inverse roles Extending our algorithm for handling inverse roles is trivial. We add algorithm 4 to algorithm 1 after line 10.

Algorithm 4 Additions to algorithm 1 to allow inverse roles.

if C_{i_j} is of the form $\exists r_{i_j}^{-1}.C'_{i_j}$ **then**
  Add $\neg r_{i_j}(Y, X)$ and $\neg C_{i_j}(Y)$ to N.
end if

Adding All-quantifier The all-quantifier $\forall$ can be added by using Prolog's predicate `findall`, which generates all instantiations of a goal as shown in algorithm 5.

Algorithm 5 Additions to algorithm 1 to allow inverse roles.

if C_{i_j} is of the form $\forall r_{i_j}.C'_{i_j}$ **then**
  Add `findall`$(Y, r_{i_j}(X, Y), Z) \wedge$ `all_satisfy`(Z, C'_{i_j}) to N.,
end if

The new predicate `all_satisfy(X,C)` checks if all elements of X satisfy concept C. This predicate is also defined via `findall` and added to N after all terms have been transformed, as shown in algorithm 6. Note that in algorithm 6 we use the Prolog notation of referring to empty lists by `[]`, and in the term `[F|R]` F is the first element of a list and R the rest of the list. Although we are using a cyclic definition in algorithm 6, this does not affect completeness, because predicate `all_satisfy` always terminates.

The $\sqcup$-Operator In order to handle the $\sqcup$-operator, the set of formulae DL is transformed as described in algorithm 7 before applying algorithm 1.

Negation Negation requires three transformation of D_i:

- Replace all terms $\neg(C_{i_{j_1}} \sqcap \ldots \sqcap C_{i_{j_k}})$ by $(\neg C_{i_{j_1}} \sqcup \ldots \sqcup \neg C_{i_{j_k}})$
- Replace all terms $\neg\exists r_{i_j}.C'_{i_j}$ by $\forall r_{i_j}.\neg C'_{i_j}$

Algorithm 6 Predicate `all_satisfy`.

Add $\mathtt{all\_satisfy}([], C)$ to N.
Add $\mathtt{all\_satisfy}([F|R], C) \leftarrow$ concept_instance$(F, C) \wedge \mathtt{all_satisfy}(R, C)$ to N.

Algorithm 7 Eliminating the $\sqcup$-operator.

for all $D_i \in DL$ of the form $C_{i_0} \sqsupseteq \alpha \sqcup \beta$ **do**
replace D_i by $D_i' = C_{i_0} \sqsupseteq \alpha$ and $D_i'' = C_{i_0} \sqsupseteq \beta$
end for

- Replace all terms $\neg\forall r_{i_j}.C_{i_j}'$ by $\exists r_{i_j}.\neg C_{i_j}'$

After this transformation, the algorithm is extended to accept negated concept and role terms by adding algorithm 8 to the inner *for all* loop.

The drawback of this algorithm is that the resulting rules are not be Horn, because negated roles or concepts will show up as additional non-negated element in the clause. Therefore, Prolog's SLD resolution is incomplete and unsound for theses formulae.[21] It should be noted that Prolog is complete for these clauses if we add system elimination to Prolog's reasoning rules.[21] The unsoundness is due to Prolog's closed world assumption and negation as failure.

Algorithm 8 Addition to the inner loop of algorithm 1 to allow negation.

if $\neg C_{i_j}$ is an atomic concept definition **then**
Add concept_instance(C_{i_j}, X) to N.
end if

2.4 Adding System Generation

The algorithms we have shown so far do *system checking*. Given a T-Box and an A-Box description, they decide if a given term is an instance of a concept. In section 1, we gave the reasons for adding logical deduction capabilities to *Bang3*. System checking is only one of them. We also want *system generation*. System generation allows not only to ask questions like "Does Multi-Agent System X fulfills concept Y?", but also "How does Multi-Agent System X has to be changed in order to fulfills concept Y?".

The idea behind adding system creation capabilities is the following:

- Whenever a concept can not be fulfilled because something is not an instance of an atomic concept or an atomic role, make it an instance of this concept resp. role.
- Keep track of all concepts and roles created this way.

The list of these additional concepts and roles yields an extension of the A-Box that fulfills the concept in question. Algorithm 1 transforms concept and role definitions into horn rules of the form

$$\text{concept\_instance}(C, X) \leftarrow \text{concept\_instance}(C_1, X) \wedge \ldots \wedge \text{concept\_instance}(C_n, X)$$

In order to add system generation, the terms on the left side are extended with another parameter Y. This parameter contains all additional concept and role assertions required to fulfill the concept:

$$\text{concept\_instance}(C, X, [\cup_{i=1\ldots n} Y_i]) \leftarrow \text{concept\_instance}(C_1, X, Y_1) \wedge \ldots \wedge \text{concept\_instance}(C_n, X, Y_n)$$

Finally, for atomic concepts and roles we add the rules

$$\text{concept\_instance}(C, X, [\text{concept\_instance}(C, X, [])])$$

and

$$\text{role\_instance}(R, X, Y, [\text{role\_instance}(R, X, Y, [])])$$

The meaning of the latter is that X is an instance of concept C when the A-Box assertion $\text{concept\_instance}(C, X, [])$ is added, and X and Y are in role-relationship R if $\text{role\_instance}(R, X, Y, [])$ is added to the A-Box assertions. In order to work correctly, it is necessary that the SLD selection function chooses these rules only after all other atomic concept and role definitions have been used. Otherwise, it may happen that an atomic concept or role definition that is already present in the A-Box is added to the list of additional definitions. These predicates also have to cause some side-effects: Whenever a new concept or role instance is created, it has to be added to the database. Upon backtracking, these new facts have to be removed again. This behavior can be easily implemented with a stack and Prolog's `asserta` and `retractall` commands.

Avoiding redundancies The algorithm may mistakenly add facts to the system which are already present, because the system generating statement $\text{concept\_instance}(C, A_x, [\text{concept\_instance}(C, A_x, [])])$ does not check whether the concept is already in the fact base. This can be avoided by changing the rule to

$$\text{concept\_instance}(C, A_x, [\text{concept\_instance}(C, A_x, [])]) \leftarrow \\ \neg\text{concept\_instance}(C, A_x, [])$$

The all-quantifier rules present another kind of redundant fact generation. The new predicate $\texttt{all\_satisfy}(X, C)$ as defined in algorithm 6 checks if all elements of X satisfy concept C. Because of the fact generating rules, even if no additional rules are necessary to form a system, an instance of the role with a free variable is always added to the list of results. This problem can be avoided by creating a special variant of the `forall`-predicate that exempts fact generating rules.

2.5 Summary

We have shown a method to convert description logic $\mathcal{ALC}$ to Prolog clauses. This clauses can be combined with hand-written Prolog clauses to form logical programs. The combination of description logics and traditional Prolog programs provides a powerful way to describe Multi-Agent Systems in an ontological sound way. The Prolog programs resulting from the conversion can be used to check if a given MAS fulfills certain characteristics. It can also be used to generate MAS according to a given set of constraints. Next, we show how the formalisms described in this chapter are applied to *Bang3* MAS.

Chapter 3

Formal logics in Bang3

An agent is an entity that has some form of perception of its environment, can act, and can communicate with other agents. It has specific skills and tries to achieve goals. A Multi-Agent System (MAS) is an assemble of interacting agents in a common environment.[12]

In order to use automatic reasoning on a MAS, the MAS must be described in formal logics. For the *Bang3* system, we define a formal description for the static characteristics of the agents, and their communication channels. We do not model dynamic aspects of the system yet.

Bang3 agents communicate via *messages* and *triggers*. In order to identify the receiver of a message, the sending agent needs the message itself and a *link* to the receiving agent. A conversation between two agents usually consists of a number of messages. For example, when a neural network agent requests training data from a data source agent, it may send the following messages: "Open the data source located at XYZ", "Randomize the order of the data items", "Set the cursor to the first item", "send next item". All these messages belong to a common category: Messages requesting input data from a data source. In order to abstract from the actual messages, we subsume all these messages under a *message type* when describing an agent in formal logics.

Definition 1 (Message type) *A* message type *identifies a category of messages that can be send to an agent in order to fulfill a specific task. We refer to message types by unique* identifiers*.*

The set of message types understood by an agent is called its *interface*. For outgoing messages, each link of an agent is associated with a message type. Via this link, only messages of the given type are sent. We call a link with its associated message type a *gate*.

Definition 2 (Interface) *An* interface *is the set of message types understood by a class of agents.*

Definition 3 (Gate) *A* gate *is a tuple consisting of a message type and a named link.*

Now it is easy to define if two agents can be connected: Agent A can be connected to agent B via gate G if the message type of G is in the list of interfaces of agent B. Note that one output gate sends messages of one type only, whereas one agent can receive different types of messages. This is a very natural concept: When an agent sends a message to some other agent via a gate, it assigns a specific role to the other agent, e.g. being a supplier of training data. On the receiving side, the receiving agent usually should understand a number of different types of messages, because it may have different roles for different agents.

Definition 4 (Connection) *A* connection *is described by a triple consisting of a sending agent, the sending agent's gate, and a receiving agent.*

Next we define *agents* and *agent classes*. *Bang3* is object oriented. Agents are created by generating instances of classes. An agent derives all its characteristics from its class definition. Additionally, an agent has a *name* to identify it. The static aspects of an agent class are described by the interface of the agent class (the messages understood by the agents of this class), the gates of the agent (the messages send by agents of this class), and the type(s) of the agent class. *Types* are nominal identifiers for characteristics of an agent. The types used to describe the characteristics of the agents should be ontological sound.

Concepts	
mas(C)	C is a Multi-Agent System
agent_class(C)	C is the name of an agent class
gate(C)	C is a gate
message_type(C)	C is a message type
Roles	
agent_class_type(X,Y)	Agent class X is of type Y
has_gate(X,Y)	Agent X has gate Y
message_type_of_gate(X,Y)	Gate X emits messages of type Y
has_interface(X,Y)	Agent class X understands messages of type Y
agent_instance(X,Y)	Agent X is an instance of agent class Y
has_agent(X,Y)	Agent Y is part of MAS X

Table 3.1: Concepts and roles used to describe MAS.

```
agent_class(decision_tree);
agent_class_type(decision_tree, computational_agent);
gate(gate_for_data_agent);
has_gate(decision_tree, gate_for_data_agent);
message_type_of_gate(gate_for_data_agent, training_data);
has_interface(decision_tree, computational_agent_control);
```

Figure 3.1: Example agent class definition.

Definition 5 (Agent Class) *An* agent class *is defined by an interface (a set of message types, a set of gates, and a set of types.*

Definition 6 (Agent) *An* agent *is an instance of an agent class. It is defined by its name and its class.*

Multi-Agent Systems are assembles of agents. For now, only static aspects of agents are modeled. Therefore, a Multi-Agent System can be described by three elements: The set of agents in the MAS, the connections between these agents, and the characteristics of the MAS. The characteristics (constraints) of the MAS are the starting point of logical reasoning: In *MAS checking* the logical reasoner deduces if the MAS fulfills the constraints. In *MAS generation*, it creates a MAS that fulfills the constraints, starting with a partial MAS.

Definition 7 (Multi-Agent System) Multi-Agent Systems (MAS) *consist of a set of agents, a set of connections between the agents, and the characteristics of the MAS.*

3.1 Describing Multi-Agent Systems in Description Logics

Description logics know concepts (unary predicates) and roles (binary predicates). In order to describe agents and Multi-Agent Systems in description logics, the definitions 1 to 7 are mapped onto description logic concepts and roles as shown in table 3.1.

An example agent class description is given in figure 3.1. It defines the agent class `decision_tree`. This agent class accepts messages of type `computational_agent_control`. It has one gate called `gate_for_data_agent` and emits messages of type `training_data`.

In the same way, A-Box instances of agent classes are defined:

```
agent_instance(decision_tree, dt_instance)
```

An agent is assigned to a MAS via role `has_agent`. In the following example, we define dt_instance as belonging to MAS my_mas:

```
has_agent(my_mas, dt_instance)
```

Since connections are relations between three elements, a sending agent, a sending agent's gate, and a receiving agent, we can not formulate this relationship in traditional description logics. It would be possible to circumvent the problem by splitting these triples into two relationships, but this would be counter-intuitive to our goal of defining MAS in an ontological sound way. As we have described in section 2.1, we combine description logics with traditional logic programs. Connections are described in traditional Prolog:

```
connection(dt_instance, data_source_instance, gate_for_data_agent, [])
```

(The empty last element of the predicate means that no additional facts have to be generated in order to make this fact true. See explanations about system generation in section 2.4.)

Constraints on MAS can be described in Description Logics, in Prolog clauses, or in a combination of both. As an example, the following concept description requires the MAS my_mas to contain a decision tree agent:

$$\text{dt\_MAS} \sqsupseteq \text{mas} \sqcap \exists \text{has\_agent}.(\exists \text{agent\_instance}.\text{decision\_tree})$$

An essential requirement for a MAS is that agents are connected in a sane way: An agent should only connect to agents that understand its messages. According to definition 4, a connection is possible if the message type of the sending agent's output gate matches a message type of the receiving agent's interface. With the logical concepts and descriptions given in this section, this constraint can formulated as a Prolog style horn rule. If we are only interested in checking if a connection satisfies this property, the rule is very simple:

$$\begin{aligned}
&\text{connection}(S, R, G, []) \leftarrow \\
&\quad \text{role_instance}(\text{agent_instance}, R, RC, []) \wedge \\
&\quad \text{role_instance}(\text{agent_instance}, S, SC, []) \wedge \\
&\quad \text{role_instance}(\text{has_interface}, RC, MT, []) \wedge \\
&\quad \text{role_instance}(\text{has_gate}, SC, G, []) \wedge \\
&\quad \text{role_instance}(\text{message_type_of_gate}, G, MT, [])
\end{aligned}$$

The first two lines of the rule body determine the classes SC and RC of the sending (S) and receiving (R) agent. The next two lines unify MT and G with a message type (MT) understood by the receiving agent's class and a gate (G) of the sending agent's class. The last line check if gate G sends messages of type MT.

As described in chapter 2.4, the last argument of a predicate is used for system generation purposes: This argument is unified with a list of additional assertions required to make the predicate true. In the connection-rule, this argument has been set to the empty list [] for all predicates. Therefore, this predicate can be used for system checking, but not for system generation. Extending it for system generation is easy:

$$\begin{aligned}
&\text{connection}(S, R, G, AC) \leftarrow \\
&\quad \text{role_instance}(\text{agent_instance}, R, RC, AC0) \wedge \\
&\quad \text{role_instance}(\text{agent_instance}, S, SC, AC1) \wedge \\
&\quad \text{role_instance}(\text{has_interface}, RC, MT, []) \wedge \\
&\quad \text{role_instance}(\text{has_gate}, SC, G, []) \wedge \\
&\quad \text{role_instance}(\text{message_type_of_gate}, G, MT, []) \wedge \\
&\quad \text{flatten}([AC0, AC1, \text{connection}(S, R, G, [])], AC)
\end{aligned}$$

With this definition, it is possible to generate new class instances, while all other predicates remain the same. The flatten-predicate in the last line concatenates the lists of newly generated facts to one list. Note that we allow the creation of new instances only at certain points in this predicate: It is possible to create new agents ($AC0$ and $AC1$) and a new connection, but it is not possible to create new interfaces, new gates, or to change the message type of a gate.

The following paragraphs show some more examples for logical descriptions of MAS. It should be noted that these MAS types can be combined, i.e. it is possible to query for an interactive, computational MAS, or for a computational MAS with graphical output.

Computational MAS A `computational_MAS` can be defined as a MAS with a computational agent and a data source agent which are connected:

```
computational_MAS(MAS, AC) ←
    role_instance(agent_class_type, CAC, computational_agent, []) ∧
    role_instance(agent_instance, CA, CAC, AC0) ∧
    role_instance(has_agent, MAS, CA, AC1) ∧
    role_instance(agent_instance, DS, data_source, AC2) ∧
    role_instance(has_agent, MAS, DS, AC3) ∧
    connection(CA, DS, G, AC4) ∧
    flatten([AC0, AC1, AC2, AC3, AC4], AC)
```

The first three lines of the clause body ensure that the MAS has an agent CA whose class CAC is of type computational. The next two lines make sure that the MAS has an agent DS of class data_source. Line six ensures that these agents are connected via a gate G.

Trusted MAS A MAS is trusted if all of its agents are trusted. This examples uses the Prolog predicate `findall`. `findall` returns a list of all instances of a variable for which a predicate is true. In the definition of predicate `all_trusted` the usual Prolog syntax for recursive definitions is used.

```
trusted_MAS(MAS, []) ←
    findall(A, role_instance(has_agent, MAS, A), A) ∧
    all_trusted(A)
all_trusted([]) ← true
all_trusted([F|R]) ←
    role_instance(agent_class, FC, F, []) ∧
    role_instance(agent_class_type, FC, trusted, [])
```

Interactive MAS A MAS is interactive if it contains a computational agent and the computational agent is connected to a GUI (Grapical User Interface) agent or a CLI (Command Line Interface) agent.

```
interactive_MAS(MAS, AC) ←
    has_agent(MAS, CA, AC0) ∧
    role_instance(agent_instance, CA, CAC, AC1) ∧
    role_instance(agent_class_type, CAC, computational, []) ∧
    has_gui_or_cli_agent(MAS, I, AC2)
    connection(I, CA, G, AC3) ∧
    flatten([AC0, AC1, AC2, AC3], AC)
has_gui_or_cli_agent(MAS, I, AC) ←
    has_agent(MAS, I, AC0) ∧
    role_instance(agent_instance, I, IC, AC1) ∧
    role_instance(agent_class_type, IC, gui, [])
```

$$\begin{aligned}&\text{has\_gui\_or\_cli\_agent}(MAS, I, AC) \leftarrow \\ &\quad \text{has\_agent}(MAS, I, AC0) \wedge \\ &\quad \text{role\_instance}(\text{agent\_instance}, I, IC, AC1) \wedge \\ &\quad \text{role\_instance}(\text{agent\_class\_type}, IC, \text{cli}, [])\end{aligned}$$

Chapter 4

Conclusion

We have shown how formal logics can be incorporated in a MAS. We presented both a logical formalism for the description of MAS, and reasoning procedures to draw conclusions from the logical descriptions. In this, we combined Description Logics with traditional Prolog rules. The system we implemented allows the practical application of these technologies. So far, we only describe static aspects of MAS. Further research will be put in the development of formal descriptions of dynamic aspects of MAS.

The reasoning component uses a traditional resolution based method to generate MAS configurations. A further field of research will be different reasoning methods, e.g. tableaux-methods.

The hybrid character of the system, with both a logical component and soft computing agents, also makes it interesting to combine these two approaches in one reasoning component. In order to automatically come up with feasible hybrid solutions for specific problems, we plan to combine two orthogonal approaches: a soft computing evolutionary algorithm with a formal ontology-based model. We expect synergy effects from using formal logics to aid evolutionary algorithms and vice versa. Testing the fitness of an evolutionary bred Multi-Agent System can be an expensive operation, because multiple agents have to be created and connected in possibly complex ways. Testing the system consumes even more time. Determining the characteristics of a Multi-Agent System by formal logics can help avoiding or at least reducing these expensive operations. On the other hand, formal logic inference algorithms are geared to find optimal, non-redundant solutions to a given problem, at the expense of unfavorable complexity problems. In the field of Multi-Agent System configuration, good but non-optimal results are often acceptable. By the combination with evolutionary methods, complexity issues of formal logic inferences can be alleviated, while still producing adequate results.

Acknowledgments

This work has been supported by a DAAD postgraduate grant in the framework of the common special academia program III of the federal states and the federal government of Germany.

Bibliography

[1] F. Baader. Logic-based knowledge representation. In M. J. Wooldridge and M. Veloso, editors, *Artificial Intelligence Today, Recent Trends and Developments*, number 1600, pages 13–41. Springer Verlag, 1999.

[2] F. Baader and U. Sattler. Tableau algorithms for description logics. In R. Dyckhoff, editor, *Proceedings of the International Conference on Automated Reasoning with Tableaux and Related Methods (Tableaux 2000)*, volume 1847, pages 1–18, St Andrews, Scotland, UK, 2000. Springer-Verlag.

[3] Peter Baumgartner and Ulrich Furbach. Model-based deduction for knowledge representation. In *Proceedings of the International Workshop on the Semantic Web*, Hawaii, 2002.

[4] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, May 2001.

[5] Gerd Beuster, Pavel Krušina, Roman Neruda, and Pavel Rydvan. Towards building computational agent schemes. In *Artificial Neural Nets and Genetic Algorithms — Proceedings of the International Conference in Roanne, France*, SpringerWienNewYork, 2003.

[6] Alexander Borgida. On the relationship between description logic and predicate logic. In *CIKM*, pages 219–225, 1994.

[7] Alexander Borgida. On the relative expressiveness of description logics and predicate logics. *Artificial Intelligence*, 82(1-2):353–367, 1996.

[8] M. Davis, editor. *The Undecidable*. Raven Press, 1965.

[9] Stefan Decker, Dieter Fensel, Frank van Harmelen, Ian Horrocks, Sergey Melnik, Michel C. A. Klein, and Jeen Broekstra. Knowledge representation on the web. In *Description Logics*, pages 89–97, 2000.

[10] J. E. Doran, S. Franklin, N. R. Jennings, and T. J. Norman. On cooperation in multi-agent systems. *The Knowledge Engineering Review*, 12(3):309–314, 1997.

[11] A. Farquhar, R. Fikes, and J. Rice. Tools for assembling modular ontologies in ontolingua, 1997.

[12] Jacques Ferber. *Multi-Agent System: An Introduction to Distributed Artificial Intelligence*. Harlow: Addison Wesley Longman, 1999.

[13] Michael R. Genesreth and Richard E. Fikes. Knowledge interchange format, version 2.2. Technical report, Computer Science Department, 1991.

[14] J. Hendler. Agents and the semantic web, 2001.

[15] U. Hustadt and R. A. Schmidt. On the relation of resolution and tableaux proof systems for description logics. In D. Thomas, editor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence IJCAI'99*, volume 1, pages 110–115, Stockholm, Sweden, July 31-August 6, 1999. Morgan Kaufmann.

[16] Pavel Krušina, Roman Neruda, and Zuzana Petrová. More autonomous hybrid models in bang. In *International Conference on Computational Science (2)*, pages 935–942, 2001.

[17] Alon Y. Levy and Marie-Christine Rousset. The limits on combining recursive horn rules with description logics. In *AAAI/IAAI*, pages 577–584, 1996.

[18] Robert Meolic, Tatjana Kapus, and Zmago Brezocnik. Model checking: A formal method for safety assurance of logistic systems.

[19] H. S. Nwana. Software agents: An overview. *Knowledge Engineering Review*, 11(2):205–244, 1995.

[20] Robert F. Stärk. A direct proof for the completeness of SLD-resolution. In *CSL: 3rd Workshop on Computer Science Logic*. LNCS, Springer-Verlag, 1990.

[21] Mark E. Stickel. A Prolog technology theorem prover: Implementation by an extended Prolog compiler. In J. H. Siekmann, editor, *Proceedings of the Eighth International Conference on Automated Deduction*, volume 230, pages 573–587, Berlin, 1986. Springer-Verlag.

[22] André Vellino. The relative complexity of sl-resolution and analytic tableau. *Studia Logica 52*, 2:323–337, 1993. Kluewer.