**Reasoning about Bang3 Multi-agent Systems in KR-Hyper**

Beuster, G.
2006

**Institute of Computer Science**
**Academy of Sciences of the Czech Republic**

# Reasoning about Bang3 multi-agent systems in KR-Hyper

Gerd Beuster, Roman Neruda, Björn Pelzer

Technical report No. 955

January 2006

Pod Vodárenskou věží 2, 182 07 Prague 8, phone: +420 266 051 111, fax: +420 286 585 789,
e-mail:ics@cs.cas.cz

**Institute of Computer Science**
**Academy of Sciences of the Czech Republic**

# Reasoning about Bang3 multi-agent systems in KR-Hyper

Gerd Beuster, Roman Neruda, Björn Pelzer

Technical report No. 955

January 2006

Abstract:

Bang3 is a Multi-Agent-System platform focusing on soft computing. KR-Hyper is an automatic theorem proofer for First Order Logic. In this paper, we describe how KR-HYPER can be connected to Bang3 in order to allow reasoning about Multi-Agent-Systems.

Keywords:
MAS, Bang, KR-HYPER, Logic, Deduction, Description Logics, First Order Logic, Agents

# Chapter

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

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

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

# Formal Reasoning

## 1 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 standardization of agent communications.

For the purpose of describing multi-agent systems, description logics is sometimes to weak. Therefore, we decided to use both Description Logics and First Order Logic. The description logic $ACL$ is used to describe the ontology of agent types. FOL is used to describe multi-agent-systems. In order to reason about MAS, the ontology is transferred from DL to FOL. The automatic theorem proofer KR-HYPER is used for reasoning.

## 2 Ontologies

Description logics are subsets of FOL. There are standard methods for transferring DL formulas to FOL formulae. Our approach is to transfer DL to FOL and then use the FOL theorem proofer KR-HYPER for automatic deduction.

# Chapter

# The Bang3-Ontology in KRHyper

KRHyper is a theorem prover and model generator for first order logics (FOL). It uses an input syntax similar to Prolog and which conforms to that of the Protein prover; the KRHyper Reference Manual provides a detailed specification[1]. Therefore KRHyper requires a compatible FOL-ontology in order to perform any reasoning tasks regarding the multi-agent systems (MAS) of *Bang3*. The current *Bang3*-distribution already includes a MAS-ontology, however, it is intended for the RACER Description Logic Reasoner[2]. Therefore it was necessary to translate the ontology from description logics (DL) into first order logics.

## 1 Translation

In principle the transformation algorithm described by Beuster in [3] was applied to the RACER ontology. The result received some further modifications in order to accommodate to peculiarities of the KRHyper system, and also to achieve an ontology which is more consistent with the specific needs.

Basically, the transformation algorithm creates a FOL-ontology with only two predicates, `concept_instance/2` and `role_instance/3`. All role identifiers and primitive concepts from the DL-ontology become atoms in the FOL version. `concept_instance/2` assigns an instance to a concept identifier, and `role_instance/3` assigns two instances to a role identifier. While this approach produces functional input for KRHyper, it renders some of the subsequent tasks impractical and lengthy due to the way KRHyper operates. KRHyper implements the hyper resolution calculus, and therefore it works in a bottom-up manner, always deriving all possible facts from an input set. This means that KRHyper will usually generate more output than needed to solve a specific task, making the computation time-consuming and the output unwieldy. KRHyper provides means to alleviate the problem: the computation can be limited to applying only those clauses relevant for a single given predicate (using the command `run/1`), and similarly the output can also be limited to facts of one predicate (using the command `print_extent/1`).

Hence it is helpful to introduce additional predicates, as these form the distinguishing features that enable a more refined model generation. Accordingly, roles from the DL-ontology are preserved as predicates in the FOL-version. While the transformation algorithm would turn a role expression with the role `r` into a clause `role_instance(r,X,Y)`, the final translation would instead contain the clause `r(X,Y)`. Four roles have been translated this way:

- `has_gate(X,Y)`: Every agent of the class `X` has a gate of the class `Y`.
- `has_interface(X,Y)`: Every agent of the class `X` has an interface of the class `Y`.
- `has_message_type(X,Y)`: Every gate or interface of the class `X` has the message type `Y`.
- `hide(X,Y)`: Every agent of the class `X` hides its gates of the class `Y`.

Translating role identifiers into predicates is also more consistent with further additions to the FOL-ontology that go beyond the original description logics. The syntax of KRHyper allows for predicates

with more than two arguments, and hence it is possible to express roles with multiple role fillers. These would not fit into the `role_instance/3`-scheme, and therefore they would have to be expressed in a way inconsistent to the roles from the DL-ontology.

Primitive concept identifiers are translated into atoms as by the transformation algorithm. The predicate `concept_instance/2` is essentially kept as well, although it is differentiated into multiple predicates to match the *Bang3*-terminology:

- `agent_instance(X,Y)`: `Y` is an agent of the class `X`.
- `gate_instance(X,Y)`: `Y` is a gate of the class `X`.
- `interface_instance(X,Y)`: `Y` is an interface of the class `X`.

In addition three predicates have been introduced, `agent_class/1`, `gate_class/1` and `interface_class/1`, which merely serve to indicate the semantics of the atomic identifiers for agent, gate and interface classes. The three unary predicates are not really necessary for the reasoning operations, and in theory they could be derived for their respective atoms from the three instantiating predicates above. However, they make the ontology a bit more intuitive by preserving a resemblance to the hierarchy of *Bang3*, and they also provide a shortcut when extending the ontology with new clauses. Analogously the predicate `message_type/1` identifies the various message types.

Regarding class inheritance, the transformation algorithm expresses a DL-rule $Class \sqsupseteq SubClass$as the following clause, where `class` and `subClass` are the atoms representing the concepts *Class* and *SubClass*:

```
concept_instance(class,X):-concept_instance(subClass,X).
```

For the KRHyper ontology a more extensive method was chosen. The reasoning tasks may involve both queries on the abstract class level ("Which class of agents has gate X?") as well as on the level of agent instances ("Which particular agent has gate X?"). It is necessary to ensure inheritance both between class names and between instances. For this reason inheritance is expressed with several clauses using the predicate `is_subclass_of/2`, where `is_subclass_of(X,Y)` holds if `X` is a subclass of `Y` in *Bang3*. The clause ensuring inheritance between agent instances for example is as follows:

```
agent_instance(Class,Agent):-
        agent_instance(SubClass,Agent),
        is_subclass_of(SubClass,Class).
```

Class identifiers inherit from each other with the following clauses:

```
has_gate(X,Z):-
        is_subclass_of(X,Y),
        has_gate(Y,Z).
has_interface(X,Z):-
        is_subclass_of(X,Y),
        has_interface(Y,Z).
hides(X,Z):-
        is_subclass_of(X,Y),
        hides(Y,Z).
```

This lengthy way of inheritance specification was chosen because it emphasizes the class structure of *Bang3*, and it allows traversal of the class hierarchy on the abstract level, whithout requiring the existence of instances for the classes.

On several occasions the translation of the RACER-ontology made it necessary to deal with non-primitive concepts on the left side of concept subsumptions:

- $C_1 \sqcap ... \sqcap C_n \sqsupseteq D$: In the case of conjunctions the rule is treated as a set of subsumption rules, i.e. $C_1 \sqsupseteq D, ..., C_n \sqsupseteq D$, which are then translated one by one. As KRHypers syntax allows conjunctions in clause heads, the translated clauses are then again merged when possible.

- $\exists r : C \sqsupseteq D$: Existentially quantified roles are translated as facts of the type `r(c,d)`, where `c` and `d` are the atoms representing the concepts $C$ and $D$. This simplistic translation scheme is possible because all DL-concepts $C$ and $D$ occurring in such rules in the RACER-ontology are primitive concepts, and primitive concepts become atoms in the FOL-ontology.

- $\forall r : C \sqsupseteq D$: Universally quantified roles are translated into clauses of the type `concept_instance(c,X):-r(d,X)`, where in addition `concept_instance/2` is replaced by one of the more specific instantiation clauses `agent_instance/2`, `gate_instance/2` and `interface_instance/2`. A second clause is required to ensure inheritance of $C$ to $r$-role fillers of subclasses of $D$:

  ```
  concept_instance(c,X):-
       is_subclass_of(SubClass,d),
       r(SubClass,X).
  ```

- $\forall r : (C_1 \sqcup ... \sqcup C_n) \sqsupseteq D$: In the special case of a disjunction as a role filler KRHypers ability to handle disjoint clause heads is used. The resulting clauses are:

  ```
  concept_instance(c_1,X);
  ...;
  concept_instance(c_n,X):-r(d,X).
  concept_instance(c_1,X);
  ...;
  concept_instance(c_n,X):-
       is_subclass_of(SubClass,d),
       r(SubClass,X).
  ```

  As in the previous case, concept_instance/2 is replaced as appropriate.

# 2 Usage and Examples

The agent class *SimpleTaskManager* has the following DL-definition in the RACER-ontology:

```
(implies SimpleTaskManager (and Taskmanager
                        (some gate igCommonCompControl)
                        (all gate igCommonCompControl)
                             )
)
```

Its translation in the FOL-ontology for KRHyper:

```
agent_class(simpleTaskManager).
is_subclass_of(simpleTaskManager,taskManager).
has_gate(simpleTaskManager,simpleTaskManager_gate).
gate_instance(igCommonCompControl,simpleTaskManager_gate).
gate_instance(igCommonCompControl,X)
:-has_gate(simpleTaskManager,X).
gate_class(igCommonCompControl).
```

With first order logics it is possible to express roles/relations with more than one role filler. For example, the predicate `c_connection/4` can be used to find out which classes of agents can communicate using which gate:

```
c_connection(SClass,RClass,SGate,SGateClass):-
        agent_class(SClass),
        agent_class(RClass),
        has_interface(RClass,RInterface),
```

```
interface_instance(RInterfaceClass,RInterface),
has_message_type(RInterfaceClass,MessageType),
has_gate(SClass,SGate),
gate_instance(SGateClass,SGate),
has_message_type(SGateClass,MessageType).
```

`c_connection/4` holds iff the sending agent class `SClass` has a gate `SGate` of the gate class `SGateClass`, whose message type matches that of an interface of the receiving agent class `RClass`. As KRHyper derives all possible facts from its input, the extent of `c_connection/4` shows all possible connections in a given agent class hierarchy.

For reasoning over existing agents a set of instantiation facts equivalent to an ABox should be provided, for example:

```
agent_instance(dataSource,lexiconBot).
agent_instance(aDecisionTree,dt543).
agent_instance(simpleTaskManager,commandoBot).
```

Now KRHyper can find possible connections between particular agent instances with the predicate `i_connection/4`, which is analogous to `c_connection/4` except for instance identifiers being used instead of class identifiers.

More importantly, KRHyper can also be used to generate multi-agent systems from the existing agents. A MAS is an assemby of several agents from specific classes, which communicate with each other. There are different classes of multi-agent systems, depending on the classes of their component agents. A computational MAS for example consists of three agents, one of the class *DataSource*, one from *SimpleTaskManager*, and one which inherits both from *Computation* and from *Model*. The following clause allows KRHyper to generate a computational MAS:

```
mas_instance(computationalMAS,
  mas([DataSourceAgent,ComputationAgent,      SimpleTaskManagerAgent])),
has_interface(mas([DataSourceAgent,
ComputationAgent,SimpleTaskManagerAgent]),
  Interface):-
    agent_instance(computation,ComputationAgent),
    agent_instance(model,ComputationAgent),
    agent_instance(dataSource,DataSourceAgent),
    agent_instance(simpleTaskManager,SimpleTaskManagerAgent),
    has_interface(SimpleTaskManagerAgent,Interface),
    i_connection(SimpleTaskManagerAgentAgent,ComputationAgent,_,_),
    i_connection(ComputationAgent,DataSourceAgent,_,_).
```

If KRHyper finds a set of agents matching the criteria in the ABox, then it creates a new instance in the form of a Skolem function `mas/1` with the list of agent identifiers. Given the ABox above KRHyper would hence derive these fact:

```
mas_instance(computationalMAS,mas([lexiconBot,dt543,commandoBot])).
has_interface(mas([lexiconBot,dt543,commandoBot]),
  taskManager_interface)
```

This means that the three agents `lexiconBot`, `dt543` and `commandoBot` together form a MAS, which in turn is an instance of the class of computational multi-agent systems. The newly created MAS-instance also appropriates the interface of its SimpleTaskManager-agent as the interface for the entire MAS.

# Chapter

# 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 $X$ is part of MAS $Y$ |

Table 0.1: 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 1.1: 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.*

# 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 0.1.

An example agent class description is given in Figure 1.1. 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 first order logic:

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

Constraints on MAS can be described in Description Logics, in First Order Logic, or in a combination of both. As an example, the following concept description requires the MAS my mas to contain a decision tree agent:

$$\mathrm{d}t\_MAS \sqsupseteq \mathrm{m}as \sqcap \exists \mathrm{h}as\_agent.(\exists \mathrm{a}gent\_instance.\mathrm{d}ecision\_tree)$$

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 in FOL. If we are only interested in checking if a connection satisfies this property, the rule is very simple:

```
connection(S,R,G) :-
     role_instance(agent_instance,R,RC),
     role_instance(has_interface,S,SC),
     role_instance(has_gate,RC,MT),
     role_instance(message_type_of_gate,G,MT),
```

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 checks if gate G sends messages of type MT.

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),
     role_instance(has_agent,MAS,CA),
     role_instance(agent_instance,DS,data_source),
     role_instance(has_agent,MAS,DS),
     connection(CA,DS,G).
```

The first three lines of the clause body ensure that the MAS has an agent whose class 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.

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

```
interactive_MAS(MAS) :-
     has_agent(MAS,CA),
     role_instance(agent_instance,CA,CAC),
     role_instance(agent_class_type,CAC,computational),
     has_gui_or_cli_agent(MAS,I)
     connection(I,CA).

has_gui_or_cli_agent(MAS,I) :-
     has_agent(MAS,I),
     role_instance(agent_instance,I,IC),
     role_instance(agent_class_type,IC,gui)

has_gui_or_cli_agent(MAS,I) :-
     has_agent(MAS,I),
     role_instance(agent_intance,I,IC),
     role_instance(agent_class_type,IC,cli)
```

# Chapter

# 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 FOL. 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.

A combination of Description Logics and First Order Logics has been used reason about and to generate MAS configurations. In order to use the First Order Logic reasoner KR-HYPER in combination with Bang, the Description Logics parts of agent specifications have been transferred to First Order Logic.

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.

## Cmplete specification

```
% TBox:

interface_class(igToYellowPages).
has_message_type(igToYellowPages, yellowPageRequest).
message_type(yellowPageRequest).


interface_class(iAgentStdIface).
has_message_type(iAgentStdIface, agentLifeManagement).
message_type(agentLifeManagement).

interface_class(igData).
has_message_type(igData, requestData).
message_type(requestData).

agent_class(dataSource).
is_subclass_of(dataSource, father).
```

```
is_subclass_of(dataSource, classInBang).
has_interface(dataSource, dataSourceInterface).
interface_instance(igData, dataSourceInterface).
interface_class(igData).
interface_instance(igData, X):-has_interface(dataSource, X).
interface_instance(igData, X):-
    is_subclass_of(SubClass, dataSource),
    has_interface(SubClass, X).

agent_class(dataSourceConsumer).
has_gate(dataSourceConsumer, dataSourceConsumer_gate).
gate_instance(igData, dataSourceConsumer_gate).
gate_class(igData).
gate_instance(igData, X):-has_gate(dataSourceConsumer, X).
gate_instance(igData, X):-
    is_subclass_of(SubClass, dataSourceConsumer),
    has_gate(SubClass, X).

interface_class(igCommonCompControl).
has_message_type(igCommonCompControl, requestSolveTask).
message_type(requestSolveTask).
has_message_type(igCommonCompControl, requestConfig).
message_type(requestConfig).
has_message_type(igCommonCompControl, queryCompState).
message_type(queryCompState).

agent_class(father).
has_interface(father, father_interface).
interface_instance(iAgentStdIface, father_interface).
interface_instance(iAgentStdIface, X):- has_interface(father, X).
interface_instance(iAgentStdIface, X):-
    is_subclass_of(SubClass, father),
    has_interface(SubClass, X).
interface_class(iAgentStdIface).
has_gate(father, father_gate).
gate_instance(igToYellowPages, father_gate).
gate_instance(igToYellowPages, X) :- has_gate(father, X).
gate_instance(igToYellowPages, X) :-
    is_subclass_of(SubClass, father),
    has_gate(SubClass, X).
gate_class(igToYellowPages).

has_message_type(igCommonFromSolver, informState).

is_subclass_of(igIterativeCompControl, igCommonCompControl).
has_message_type(igIterativeCompControl, requestController).


has_message_type(igToMonitor, informMonitor).

is_subclass_of(igIterativeToMonitor, igToMonitor).
has_message_type(igIterativeToMonitor, informMonitorIter).

agent_class(computation).
has_interface(computation, computation_interface).
interface_instance(igCommonCompControl, computation_interface).
interface_class(igCommonCompControl).
interface_instance(igCommonCompControl, X):-has_interface(computation, X).
interface_instance(igCommonCompControl, X):-
```

```
    is_subclass_of(SubClass, computation),
    has_interface(SubClass, X).
has_gate(computation, computation_gate_1).
gate_instance(ifCommonFromSolver, computation_gate_1).
gate_class(ifCommonFromSolver).
has_gate(computation, computation_gate_2).
gate_instance(igToMonitor, computation_gate_2).
gate_class(igToMonitor).
gate_instance(igToMonitor, X);
gate_instance(ifCommonFromSolver, X) :- has_gate(computation, X).
gate_instance(igToMonitor, X);
gate_instance(ifCommonFromSolver, X) :-
    is_subclass_of(SubClass, computation),
    has_gate(SubClass, X).

agent_class(iterativeComputation).
is_subclass_of(iterativeComputation, computation).
has_interface(iterativeComputation, iterativeComputation_interface).
interface_instance(igIterativeCompControl, iterativeComputation_interface).
interface_class(igIterativeCompControl).
interface_instance(igIterativeCompControl, X):-has_interface(iterativeComputation, X).
interface_instance(igIterativeCompControl, X):-
    is_subclass_of(SubClass, iterativeComputation),
    has_interface(SubClass, X).
has_gate(iterativeComputation, iterativeComputation_gate).
gate_instance(igIterativeToMonitor, iterativeComputation_gate).
gate_class(igIterativeToMonitor).
gate_instance(igIterativeToMonitor, X):-has_gate(iterativeComputation, X).
gate_instance(igIterativeToMonitor, X):-
    is_subclass_of(SubClass, iterativeComputation),
    has_gate(SubClass, X).
hide(iterativeComputation, igToMonitor).

agent_class(function).
has_interface(function, function_interface).
interface_instance(igFunction, function_interface).
interface_class(igFunction).
interface_instance(igFunction, X):-has_interface(function, X).
interface_instance(igFunction, X):-
    is_subclass_of(SubClass, function),
    has_interface(SubClass, X).
has_message_type(igStoreModel, modelStorage).

has_message_type(igQueryModel, queryModel).

agent_class(model).
is_subclass_of(model, funtion).
is_subclass_of(model, dataSourceConsumer).
has_interface(model, model_interface_1).
has_interface(model, model_interface_2).
interface_instance(igStoreModel, model_interface_1).
interface_instance(igQueryModel, model_interface_2).
interface_class(igStoreModel).
interface_class(igQueryModel).
interface_instance(igStoreModel, X);
interface_instance(igQueryMdel, X):-has_interface(model, X).
interface_instance(igStoreModel, X);
interface_instance(igQueryModel, X):-
    is_subclass_of(SubClass, model),
```

```
    has_interface(SubClass, X).

agent_class(approximator).
is_subclass_of(approximator, model).

agent_class(classifier).
is_subclass_of(classifier, model).

is_subclass_of(igSolveRepresentatives, igCommonCompControl).
has_message_type(igSolveRepresentatives, solveRepresentatives).

agent_class(classifierWithRepresentatives).
is_subclass_of(classifierWithRepresentatives, classifier).
has_interface(classifierWithRepresentatives, classifierWithRepresentatives_interface).
interface_instance(igSolveRepresentatives, classifierWithRepresentatives_interface).
interface_class(igSolveRepresentatives).
interface_instance(igSolveRepresentatives, X):-has_interface(classifierWithRepresentatives, X).
interface_instance(igSolveRepresentatives, X):-
    is_subclass_of(SubClass, classifierWithRepresentatives),
    has_interface(SubClass, X).

agent_class(taskManager).
has_interface(taskManager, taskManager_interface).
interface_instance(igCommonFromSolver, taskManager_interface).
interface_class(igCommonFromSolver).
interface_instance(igCommonFromSolver, X):-has_interface(taskManager, X).
interface_instance(igCommonFromSolver, X):-
    is_subclass_of(SubClass, taskManager),
    has_interface(SubClass, X).

agent_class(simpleTaskManager).
is_subclass_of(simpleTaskManager, taskManager).
has_gate(simpleTaskManager, simpleTaskManager_gate).
gate_instance(igCommonCompControl, simpleTaskManager_gate).
gate_instance(igCommonCompControl, X) :- has_gate(simpleTaskManager, X).
gate_instance(igCommonCompControl, X) :-
    is_subclass_of(SubClass, simpleTaskManager),
    has_gate(SubClass, X).
gate_class(igCommonCompControl).

is_subclass_of(igManageTaskComplex, igIterativeCompControl).

agent_class(complexTaskManager).
is_subclass_of(complexTaskManager, taskManager).
gate_class(igManageTaskComplex).
has_gate(complexTaskManager, complexTaskManager_gate_2).
gate_instance(igFunction, complexTaskManager_gate_2).
gate_instance(igFunction, X);
gate_instance(igManageTaskComplex, X) :- has_gate(complexTaskManager, X).
gate_instance(igFunction, X);
gate_instance(igManageTaskComplex, X) :-
    is_subclass_of(SubClass, complexTaskManager),
    has_gate(SubClass, X).
gate_class(igFunction).

agent_class(aVectorQuantization).
is_subclass_of(aVectorQuantization, father).
is_subclass_of(aVectorQuantization, classifierWithRepresentatives).
is_subclass_of(aVectorQuantization, iterativeComputation).
```

```
is_subclass_of(aVectorQuantization, function).
is_subclass_of(aVectorQuantization, classInBang).

has_message_type(igSolveLinEqSystem, solveLinEqSystem).

agent_class(aLinearSolver).
is_subclass_of(aLinearSolver, father).
is_subclass_of(aLinearSolver, function).
is_subclass_of(aLinearSolver, classInBang).
has_interface(aLinearSolver, aLinearSolver_interface).
interface_instance(igSolveLinEqSystem, aLinearSolver_interface).
interface_instance(igSolveLinEqSystem, X) :- has_interface(aLinearSolver, X).
interface_instance(igSolveLinEqSystem, X) :-
    is_subclass_of(SubClass, aLinearSolver),
    has_interface(SubClass, X).
interface_class(igSolveLinEqSystem).

agent_class(aDecisionTree).
is_subclass_of(aDecisionTree, father).
is_subclass_of(aDecisionTree, classifier).
is_subclass_of(aDecisionTree, iterativeComputation).
is_subclass_of(aDecisionTree, classInBang).

is_subclass_of(neuralNetwork, approximator).

agent_class(rBFNetworkAI).
is_subclass_of(rBFNetworkAI, father).
is_subclass_of(rBFNetworkAI, neuralNetwork).
is_subclass_of(rBFNetworkAI, iterativeComputation).
is_subclass_of(rBFNetworkAI, classInBang).
is_subclass_of(rBFNetworkAI, simpleTaskManager).
hide(rBFNetworkAI, igCommonCompControl).
has_gate(rBFNetworkAI, rBFNetworkAI_gate_1).
gate_instance(igSolveRepresentatives, rBFNetworkAI_gate_1).
has_gate(rBFNetworkAI, rBFNetworkAI_gate_2).
gate_instance(igSolveLinEqSystem, rBFNetworkAI_gate_2).
gate_instance(igSolveLinEqSystem, X);
gate_instance(igSolveRepresentatives, X) :- has_gate(rBFNetworkAI, X).
gate_instance(igSolveLinEqSystem, X);
gate_instance(igSolveRepresentatives, X) :-
    is_subclass_of(SubClass, rBFNetworkAI),
    has_gate(SubClass, X).
gate_class(igSolveRepresentatives).
gate_class(igSolveLinEqSystem).
has_interface(rBFNetworkAI, rBFNetworkAI_interface).
interface_instance(igRunNetworkDemo, rBFNetworkAI_interface).
interface_instance(igRunNetworkDemo, X) :- has_interface(rBFNetworkAI, X).
interface_instance(igRunNetworkDemo, X) :-
    is_subclass_of(SubClass, rBFNetworkAI),
    has_interface(SubClass, X).
interface_class(igRunNetworkDemo).

agent_class(multiLayerPerceptron).
is_subclass_of(multiLayerPerceptron, neuralNetwork).
is_subclass_of(multiLayerPerceptron, iterativeComputation).
is_subclass_of(multiLayerPerceptron, classInBang).
is_subclass_of(multiLayerPerceptron, father).

agent_class(multiLayerPerceptronAI).
```

```
is_subclass_of(multiLayerPerceptronAI, neuralNetwork).
is_subclass_of(multiLayerPerceptronAI, iterativeComputation).
is_subclass_of(multiLayerPerceptronAI, classInBang).
is_subclass_of(multiLayerPerceptronAI, father).

is_subclass_of(simpleTaskManagerAI, simpleTaskManager).
is_subclass_of(simpleTaskManagerAI, classInBang).
is_subclass_of(simpleTaskManagerAI, father).

has_message_type(iGUI, guiMessages).

agent_class(aTanya).
is_subclass_of(aTanya, classInBang).
is_subclass_of(aTanya, father).
has_interface(aTanya, aTanya_interface).
interface_instance(igGUI, aTanya_interface).
interface_instance(igGUI, X) :- has_interface(aTanya, X).
interface_instance(igGUI, X) :-
    is_subclass_of(SubClass, aTanya),
    has_interface(SubClass, X).

interface_class(igGUI).

has_message_type(igFitnessFunction, computeOperators).
has_message_type(igOperatorsPack, computeOperators).
has_message_type(igSelectionFunction, computeSelectionFunction).
has_message_type(igTunerFunction, computeTunerFunction).
has_message_type(igShaperFunction, computeShaperFunction).
has_message_type(igGetGAParameters, igGetGAParameters).

agent_class(aGenetics).
is_subclass_of(aGenetics, father).
is_subclass_of(aGenetics, classInBang).
is_subclass_of(aGenetics, iterativeComputation).
has_interface(aGenetics, aGenetics_interface).
interface_instance(igGetGAParameters, aGenetics_interface).
interface_instance(igGetGAParameters, X) :- has_interface(aGenetics, X).
interface_instance(igGetGAParameters, X) :-
    is_subclass_of(SubClass, aGenetics),
    has_interface(SubClass, X).
interface_class(igGetGAParameters).
has_gate(aGenetics, aGenetics_gate_1).
has_gate(aGenetics, aGenetics_gate_2).
has_gate(aGenetics, aGenetics_gate_3).
has_gate(aGenetics, aGenetics_gate_4).
has_gate(aGenetics, aGenetics_gate_5).
has_gate(aGenetics, aGenetics_gate_6).
gate_instance(igFitnessFunction, aGenetics_gate_1).
gate_instance(igOperatorsPack, aGenetics_gate_2).
gate_instance(igSelectionFunction, aGenetics_gate_3).
gate_instance(igTunerFunction, aGenetics_gate_4).
gate_instance(igShaperFunction, aGenetics_gate_5).
gate_instance(igGetGAParameters, aGenetics_gate_6).
gate_instance(igFitnessFunction, X);
gate_instance(igOperatorsPack, X);
gate_instance(igSelectionFunction, X);
gate_instance(igTunerFunction, X);
gate_instance(igShaperFunction, X);
gate_instance(igGetGAParameters, X):- has_gate(aGenetics, X).
```

```
gate_instance(igFitnessFunction, X);
gate_instance(igOperatorsPack, X);
gate_instance(igSelectionFunction, X);
gate_instance(igTunerFunction, X);
gate_instance(igShaperFunction, X);
gate_instance(igGetGAParameters, X):-
    is_subclass_of(SubClass, aGenetics),
    has_gate(SubClass, X).
gate_class(igFitnessFunction).
gate_class(igOperatorsPack).
gate_class(igSelectionFunction).
gate_class(igTunerFunction).
gate_class(igShaperFunction).
gate_class(igGetGAParameters).

agent_class(aFitness).
is_subclass_of(aFitness, classInBang).
is_subclass_of(aFitness, father).
is_subclass_of(aFitness, function).
has_interface(aFitness, aFitness_interface).
interface_instance(igFitnessFunction, aFitness_interface).
interface_instance(igFitnessFunction, X) :- has_interface(aFitness, X).
interface_instance(igFitnessFunction, X) :-
    is_subclass_of(SubClass, aGenetics),
    has_interface(SubClass, X).
interface_class(igFitnessFunction).

agent_class(aFloatOperators).
is_subclass_of(aFloatOperators, classInBang).
is_subclass_of(aFloatOperators, father).
is_subclass_of(aFloatOperators, function).
has_interface(aFloatOperators, aFloatOperators_interface).
interface_instance(igOperatorsPack, aFloatOperators_interface).
interface_instance(igFitnessFunction, X) :- has_interface(aFloatOperators, X).
interface_instance(igFitnessFunction, X) :-
    is_subclass_of(SubClass, aFloatOperators),
    has_interface(SubClass, X).
interface_class(igOperatorsPack).

agent_class(aSelection).
is_subclass_of(aSelection, classInBang).
is_subclass_of(aSelection, father).
is_subclass_of(aSelection, function).
has_interface(aSelection, aSelection_interface).
interface_instance(igSelectionFunction, aSelection_interface).
interface_instance(igSelectionFunction, X) :- has_interface(aSelection, X).
interface_instance(igSelectionFunction, X) :-
    is_subclass_of(SubClass, aSelection),
    has_interface(SubClass, X).
interface_class(igSelectionFunction).

agent_class(aTuner).
is_subclass_of(aTuner, classInBang).
is_subclass_of(aTuner, father).
is_subclass_of(aTuner, function).
has_interface(aTuner, aTuner_interface).
interface_instance(igTunerFunction, aTuner_interface).
interface_instance(igTunerFunction, X) :- has_interface(aTuner, X).
interface_instance(igTunerFunction, X) :-
```

```
    is_subclass_of(SubClass, aTuner),
    has_interface(SubClass, X).
interface_class(igTunerFunction).

has_message_type(igM_lListener, iDoNotKnow).
has_message_type(igCurses, textOutput).

agent_class(aCurses).
is_subclass_of(aCurses, classInBang).
is_subclass_of(aCurses, father).
has_gate(aCurses, aCurses_gate).
gate_instance(igM_lListener, aCurses_gate).
gate_instance(igM_lListener, X):- has_gate(aCurses, X).
gate_instance(igM_lListener, X):-
    is_subclass_of(SubClass, aCurses),
    has_gate(SubClass, X).
gate_class(igM_lListener).
has_interface(aCurses, aCurses_interface).
interface_instance(igCurses, aCurses_interface).
interface_instance(igCurses, X) :- has_interface(aCurses, X).
interface_instance(igCurses, X) :-
    is_subclass_of(SubClass, aCurses),
    has_interface(SubClass, X).
interface_class(igCurses).

has_message_type(igMan, msgToManPages).

agent_class(aManualPage).
is_subclass_of(aManualPage, classInBang).
is_subclass_of(aManualPage, father).
has_interface(aManualPage, aManualPage_interface).
interface_instance(igMan, aManualPage_interface).
interface_instance(igMan, X) :- has_interface(aManualPage, X).
interface_instance(igMan, X) :-
    is_subclass_of(SubClass, aManualPage),
    has_interface(SubClass, X).
interface_class(igMan).

has_message_type(gateTerm, accessToTerminal).
agent_class(debuggerShell).
has_gate(debuggerShell, debuggerShell_gate_1).
has_gate(debuggerShell, debuggerShell_gate_2).
has_gate(debuggerShell, debuggerShell_gate_3).
gate_instance(igCurses, debuggerShell_gate_1).
gate_instance(gateTerm, debuggerShell_gate_2).
gate_instance(igMan, debuggerShell_gate_3).
gate_instance(igCurses, X);
gate_instance(gateTerm, X);
gate_instance(igMan, X):- has_gate(debuggerShell, X).
gate_instance(igCurses, X);
gate_instance(gateTerm, X);
gate_instance(igMan, X):-
    is_subclass_of(SubClass, debuggerShell),
    has_gate(SubClass, X).
gate_class(igCurses).
gate_class(gateTerm).
gate_class(igMan).
has_interface(debuggerShell, debuggerShell_interface_1).
has_interface(debuggerShell, debuggerShell_interface_2).
```

```
has_interface(debuggerShell, debuggerShell_interface_3).
has_interface(debuggerShell, debuggerShell_interface_4).
interface_instance(controlOtherAgents, debuggerShell_interface_1).
interface_instance(eavesdropOtherAgents, debuggerShell_interface_2).
interface_instance(bangStatus, debuggerShell_interface_3).
interface_instance(gdbMessages, debuggerShell_interface_4).
interface_instance(controlOtherAgents, X);
interface_instance(eavesdropOtherAgents, X);
interface_instance(bangStatus, X);
interface_instance(gdbMessages, X) :- has_interface(debuggerShell, X).
interface_instance(controlOtherAgents, X);
interface_instance(eavesdropOtherAgents, X);
interface_instance(bangStatus, X);
interface_instance(gdbMessages, X) :-
    is_subclass_of(SubClass, debuggerShell),
    has_interface(SubClass, X).
interface_class(controlOtherAgents).
interface_class(eavesdropOtherAgents).
interface_class(bangStatus).
interface_class(gdbMessages).

agent_class(aYellowPages).
is_subclass_of(aYellowPages, father).
is_subclass_of(aYellowPages, classInBang).
has_interface(aYellowPages, aYellowPages_interface).
interface_instance(igToYellowPages, aYellowPages_interface).
interface_instance(igToYellowPages, X):-has_interface(aYellowPages, X).
interface_instance(igToYellowPages, X):-
    is_subclass_of(SubClass, aYellowPages),
    has_interface(SubClass, X).
interface_class(igToYellowPages).



% Inheritance relations:
has_interface(X,Z):-
    is_subclass_of(X,Y),
    has_interface(Y,Z).
has_gate(X,Z):-
    is_subclass_of(X,Y),
    has_gate(Y,Z).
hides(X,Z):-
    is_subclass_of(X,Y),
    hides(Y,Z).
is_subclass_of(X,Z):-
    is_subclass_of(X,Y),
    is_subclass_of(Y,Z).
is_superclass_of(X,Y):-is_subclass_of(Y,Z).
agent_instance(Class, Agent):-
    agent_instance(SubClass, Agent),
    is_subclass_of(SubClass, Class).
has_gate(Agent, Gate):-
    agent_instance(AgentClass, Agent),
    has_gate(AgentClass, Gate).
has_interface(Agent, Interface):-
    agent_instance(AgentClass, Agent),
    has_interface(AgentClass, Interface).

% Connection between two agent instances (Sender and Receiver) using Gate.
```

```
i_connection(SAgent, RAgent, SGate, SGateClass):-
    agent_instance(SClass, SAgent),
    agent_instance(RClass, RAgent),
    has_interface(RClass, RInterface),
    interface_instance(RInterfaceClass, RInterface),
    has_message_type(RInterfaceClass, MessageType),
    has_gate(SClass, SGate),
    gate_instance(SGateClass, SGate),
    has_message_type(SGateClass, MessageType).

% Connection between two agent classes (Sender and Receiver) using Gate.
c_connection(SClass, RClass, SGate, SGateClass):-
    agent_class(SClass),
    agent_class(RClass),
    has_interface(RClass, RInterface),
    interface_instance(RInterfaceClass, RInterface),
    has_message_type(RInterfaceClass, MessageType),
    has_gate(SClass, SGate),
    gate_instance(SGateClass, SGate),
    has_message_type(SGateClass, MessageType).

% Which particular agent instances offer gate X?
offers_gate(Agent, X):-
        agent_instance(Class, Agent),
        has_gate(Class, X).

% Which particular agent instances offer interface X?
offers_interface(Agent, X):-
        agent_instance(Class, Agent),
        has_interface(Class, X).

% Example for a predicate that would be generated for a specific request:
% "Which classes of agents offer the gate igToYellowPages?"
answer(X):-
    has_gate(X, igToYellowPages).



% ABox:
agent_instance(aYellowPages, phonebook).
agent_instance(aDecisionTree, dt_instance).
agent_instance(father, big_daddy).
agent_instance(aCurses, cursor).
agent_instance(aSelection, selector).
agent_instance(aSelection, anotherSelector).
agent_instance(dataSource, lexiconBot).
agent_instance(function, mathBot).
agent_instance(complexTaskManager, commander).
agent_instance(simpleTaskManager, lieutenant).
agent_instance(multiLayerPerceptron, neuroBot).
agent_instance(multiLayerPerceptronAI, artificial_NeuroBot).
agent_instance(aLinearSolver, deep_thought).
agent_instance(aLinearSolver, hal).
agent_instance(aLinearSolver, skynet).
agent_instance(dataSource, mcDonalds).
agent_instance(dataSourceConsumer, hungryBot).
agent_instance(dataSourceConsumer, greedyBot).
agent_instance(dataSourceConsumer, thirstyBot).
agent_instance(comp, calculon).
```

```
contains_agent(xMas, phonebook).
contains_agent(xMas, deep_thought).
contains_agent(xMas, mcDonalds).
contains_agent(xMas, lieutenant).
contains_agent(xMas, dt_instance).
contains_agent(xMas, calculon).

mas_class(computationalMAS).
mas_instance(computationalMAS, MAS),
has_gate(MAS, Gate),
has_interface(MAS, Interface):-
    contains_agent(MAS, ComputationAgent),
    contains_agent(MAS, DataSourceAgent),
    contains_agent(MAS, SimpleTaskManagerAgent),
    has_interface(SimpleTaskManagerAgent, Interface),
    agent_instance(computation, ComputationAgent),
    agent_instance(model, ComputationAgent),
    agent_instance(dataSource, DataSourceAgent),
    agent_instance(simpleTaskManager, SimpleTaskManagerAgent),
    i_connection(SimpleTaskManagerAgentAgent, ComputationAgent, _, _),
    i_connection(ComputationAgent, DataSourceAgent, _, _).

mas_instance(computationalMAS,mas([DataSourceAgent, ComputationAgent, SimpleTaskManagerAgent])),
contains_agent(mas([DataSourceAgent, ComputationAgent, SimpleTaskManagerAgent]), ComputationAgent),
contains_agent(mas([DataSourceAgent, ComputationAgent, SimpleTaskManagerAgent]), DataSourceAgent),
contains_agent(mas([DataSourceAgent, ComputationAgent, SimpleTaskManagerAgent]), TaskManagerAgent),
has_interface(mas([DataSourceAgent, ComputationAgent, SimpleTaskManagerAgent]), Interface):-
    agent_instance(computation, ComputationAgent),
    agent_instance(model, ComputationAgent),
    agent_instance(dataSource, DataSourceAgent),
    agent_instance(simpleTaskManager, SimpleTaskManagerAgent),
    has_interface(SimpleTaskManagerAgent, Interface),
    i_connection(SimpleTaskManagerAgentAgent, ComputationAgent, _, _),
    i_connection(ComputationAgent, DataSourceAgent, _, _).
```

# Bibliography

[1] Christoph Wernhard. KRHyper Reference Manual. http://www.uni-koblenz.de/~wernhard/krhyper/doc/MANUAL.TXT

[2] RACER. http://www.racer-systems.com

[3] Gerd Beuster. Formal Reasoning in the Bang3 Multi-Agent System. In *Technical Report No. V-889*, Institute of Computer Science, Academy of Sciences of the Czech Republic, 2003