Specification of Heterogeneous Agent Architectures

Simone Marini[1], Maurizio Martelli[1], Viviana Mascardi[1], and Floriano Zini[2]

[1] DISI - Università di Genova
Via Dodecaneso 35, 16146, Genova, Italy.
marini@educ.disi.unige.it
{martelli, mascardi}@disi.unige.it
[2] ITC-IRST
Via Sommarive 18, 38050 Povo (Trento), ITALY
zini@itc.it

Abstract. Agent-based software applications need to incorporate agents having heterogeneous architectures in order for each agent to optimally perform its task. HEMASL is a simple meta-language used to specify intelligent agents and multi-agent systems when different and heterogeneous agent architectures must be used. HEMASL specifications are based on an agent model that abstracts several existing agent architectures. The paper describes some of the features of the language, presents examples of its use and outlines its operational semantics. We argue that adding HEMASL to CaseLP, a specification and prototyping environment for MAS, can enhance its flexibility and usability.

1 Introduction

Intelligent agents and multi-agent systems (MAS) are increasingly being acknowledged as the "new" modelling techniques to be used to engineer complex and distributed software applications [17,9]. *Agent-based* software development is concerned with the realization of software applications modelled as MAS. A two-phase approach can be adopted to develop agent applications at the *macro-level* before implementing the final application.

1. *Specification of the MAS*:
   - describe the *services* each agent provides other agents or human beings with;
   - describe the environmental *events* that each agent can perceive;
   - describe agent-agent, agent-human and agent-environment interactions, abstracting from the agents' internal structure;
   - provide each agent with domain-dependent procedural knowledge, so it can supply its services and respond to stimuli from the environment.

   If a *prototype* of the final application is being developed additional specifications of the environment (and its evolution due to the agents' actions) as well as specifications of the communication media among agents can be given.
2. *Proof of the specification correctness*:
   - specification properties are formally verified, and/or
   - informal testing of the system behaviour by means of a working prototype is performed.

The first phase leaves the *architecture* of each agent implicit and profitably abstracts from the internal organization and structure of single agents. Thus, the application developer is not burdened with the modelling of too many details and the specification phase is clearer and more modular. On the other hand, the services that an agent provides are usually variegated and complex and generally involve management of heterogeneous information using different kinds of behaviour. This diversity has to be taken into account and the developer must identify "the right agent to do the right thing" [14]. In other words, an application needs to incorporate agents having heterogeneous architectures in order for each agent to optimally perform its task.

One good approach is certainly to provide the application developer with a pre-defined library of architectures from which he/she can choose the most appropriate ones. Obviously, the architectures in the library have to be specified, verified and tested before being employed. This process concerns *micro-level* agent development, i.e., building software systems that include some of the main features of an agent.

The declarative nature of *logic programming* makes it very suitable for the interactive development and testing of macro and micro-level agent applications. Logic programming languages can be used to specify agents and MAS at the proper level of abstraction. They can be executed, thus providing a working prototype "for free", and thanks to their well-founded semantics they can be used to formally verify properties of programs, which is fundamental when safety critical applications are developed. Nevertheless, two considerations have to be made:

- Industries and programmers mostly use implementation languages such as C, C++, Visual Basic or Java, and specification languages (mainly non-executable) such as UML or even less formal ones.
- Logic languages which are most suitable for formal verification of system properties are definitely not user-friendly. This makes their use even harder than the "simple" and "user-friendly" Prolog!

Since 1997 the Logic Programming Group at the Computer Science Department of Genova University has been working on the development of a specification and prototyping environment for MAS. CaseLP (*Complex Application Specification Environment based on Logic Programming*) [13,12] provides a macro and micro-level development method for agent applications, as well as tools and languages which support the development steps. In our methodology the more formal and abstract specification of the MAS can be given using the executable linear logic programming language $\mathcal{E}_{hhf}$ [4], which provides constructs for concurrency and state-updating. Even if $\mathcal{E}_{hhf}$ has been successfully adopted as a specification language both at the inter-agent (macro) [1] and intra-agent (micro) [2] levels of abstraction, its use requires a profound knowledge of the linear logic [7] syntax and semantics and the language is definitely difficult to adopt.

One of the initial ideas in the design of CaseLP was to create an environment which could accommodate various specification and implementation languages as well as legacy software so that a MAS prototype could be built using a range of techniques integrated into a common framework. In this paper we attempt to bridge the gap between the above mentioned users' habits and our tool for rapid prototyping.

We present HEMASL (*HEterogeneous Multi-Agent Systems Language*) which is a simple, imperative, meta-language for the specification of agent architectures and the configuration of MAS and which is much closer to widespread existing specification and implementation languages than logic languages. The basic features of HEMASL make it suitable for agent architecture specification and for incorporation of heterogeneous agents into the same MAS. The operational semantics of HEMASL is based on the concepts of "MAS configuration" and "configuration transition". It defines an abstract interpreter for the language that can be used to animate specifications within the CaseLP framework.

HEMASL can be considered as a first step towards defining an "intermediate" language which could make it much easier to animate and incorporate traditional specification languages into CaseLP. The possibility to formally verify MAS specifications using $\mathcal{E}_{hhf}$, a feature of CaseLP, can be obviously obtained by an implementation of HEMASL in $\mathcal{E}_{hhf}$. A formal mapping between HEMASL and $\mathcal{E}_{hhf}$ constructs is under study to make this possibility more valuable.

Some important characteristics of HEMASL are presented below.

Agent model. The agent model supported by HEMASL is an abstraction of many existing architectures. This facilitates the development of the architecture specification. Moreover, since this model is the same as the one we adopted to develop agents in CaseLP, integration of HEMASL specifications into CaseLP is facilitated.

Hierarchy of abstraction levels. HEMASL provides constructs for specifying a MAS through four different levels of abstraction which support a modular and flexible representation of the system based on the concepts of abstract and concrete architecture, agent class and agent instance. This helps to model the heterogeneity of agent services and architectures.

Situatedness and social ability. The ability of an agent to interact with the surrounding environment consists in perceiving the events which take place in the environment and in performing actions which modify it. HEMASL provides an explicit model of the environment as well as primitives that can be used by agents to sense and modify it. It also provides constructs for modelling the message exchange among agents.

In the following sections we expand the concepts outlined above. Section 2 describes the computational model of our agents and the hierarchy of abstraction levels. Section 3 shows the main features of HEMASL through the specification of two agent architectures and one MAS embedding agents having these architectures. Section 4 outlines the operational semantics of HEMASL. Sections 5 and 6 compare our approach to other proposals, and conclude the paper by identifying future research possibilities.

2 Agent Model and Structure of a MAS

Since we want to model MAS with agents having different architectures, we need a simple abstraction which can encompass several of the existing architectures. The agents we model with our language are characterized, from a computational point of view, by:

– a **state**,
– a **program** and
– an **engine**.

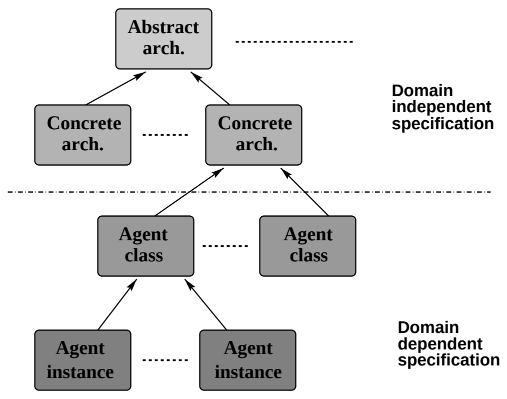

Fig. 1. Abstraction hierarchy.

The *state* includes data that may change during the execution of the agent. For example, the state of a BDI agent [15] contains its beliefs, goals and intentions. The *program* contains the information that does not change during the execution of the agent. The program of a BDI agent is determined by its plans. Lastly, the *engine* controls the execution of the agent. A typical BDI engine should be characterized by *a)* perceiving an event, *b)* identifying a set of plans which can be used to manage the perceived event according to the current beliefs, *c)* selecting one plan from the set, *d)* adding the selected plan to the intention set, *e)* selecting one intention and executing one instruction of the selected intention, and *f)* dropping successful goals.

The engine and the program belong to different abstraction levels: the engine is a meta-interpreter for the program and the data (state) on which the program operates. The behaviour of the agent is determined by the application of the agent program on the agent state by means of the agent engine.

The architecture of an agent is characterized by components which contain its state, components for the program, and an engine operating on them. The content of the agent components will be expressed using some architecture dependent object language for which the engine provides an interpreter. In this paper we do not commit to any particular object language.

A specification language for heterogeneous MAS should ensure modularity and flexibility in the definition of agent architectures. Therefore, we introduce a four abstraction level hierarchy, illustrated in Fig 1.

Abstract architecture. The abstract architecture defines the components in the architecture and the basic structure of the engine. It is possible to provide guidelines on the realization of any “macro-instruction” (procedure) present in the engine without necessarily giving all the implementation details.

Concrete architecture. Concrete architecture is defined by starting from an abstract architecture: each component is assigned a type chosen among the ones the language provides; each macro-instruction of the engine is implemented.

Agent class. A class is defined by instantiating the components in the concrete architecture which contain the program of the agent.

Agent instance. Starting from an existing class, the initialization of the architecture components containing the state identifies an agent instance.

HEMASL allows the definition of all these abstraction levels. Intuitively, an abstract architecture defines the data structures and the engine that organize agent internal activities without going into too many details. For example, the way data structures are implemented or how an intention is selected in a BDI architecture are irrelevant details at this level. An abstract architecture can give rise to several concrete architectures. This level of abstraction makes the data structures used for architecture components as well as the detailed functioning of the agent concrete. The same concrete architecture can be employed for agents that work on various application domains. Domain dependent behavioural patterns are given by defining the agent program at the class level. Finally, a MAS may require several instances of the same agent class that work by starting from different initial states. This is captured in our hierarchy by defining an agent instance level in which state components are filled.

Besides its internal representation, an agent is also characterized by the ability to perceive the surrounding environment and to interact with other agents in the system. In HEMASL the *environment* (env) is modelled as a collection of facts representing the physical environment features which are relevant to the application domain. Agents are able to directly perceive the environment, but they can only modify it by interacting with an "environment agent". Suppose that an agent *robot* is able to perform the action `turn_the_engine_on`. There are several consequences of the action on the environment, for example, noise, temperature and pollution will increase. It is not realistic that the robot would know all the consequences of its actions, thus it could not directly modify the environment. Therefore we employ a domain-dependent "environment agent" whose task is to evaluate all the relevant consequences of primitive agent actions, and to update the environment consequently.

As far as communication is concerned, an agent interacts with the "environment agent" and with other agents by means of the *ether* (eth), a data structure where messages are collected, and from which they are retrieved. HEMASL provides primitives for communication.

3 Syntax and Examples of Use

3.1 Primitive Instructions and Statements

HEMASL provides primitive instructions for managing the information contained in the architecture components, for delivering and receiving messages, for perceiving events from the environment, for generating events which modify it, and for performing actions in it. These basic instructions can be composed with the statements: *variable declaration* and *assignment*; *procedure call*; *deterministic choice* (**if-then-else**); *loop* (**while**); *concatenation* (**;**); *non deterministic choice* (|); *concurrent execution* (||).

The instructions which operate on the agent's internal components are

– **get_comp**(c, m, v),
– **put_comp**(c, m, e), and
– **del_comp**(c, m)

where c stands for the name of the component, v is a variable, e is an expression and m is the mode of insertion, extraction or deletion of an element into/from a component, that depends on the component type. For example, if the component is a list, m can assume `head` and `tail` as values.

The instructions for message exchange are

– **send**(r, m),
– **rec**(s, m), and
– **block_rec**(s, m)

where r, m, and s represent respectively the receiver, the message, and the sender. When a blocking reception is performed, execution of the agent program is blocked until agent s sends a message. In the not-blocking reception, the execution goes on even if no message coming from s may be retrieved from the ether.

The instructions for perception and generation of an event e in the MAS environment are

– **perceive**(e),
– **put_event**(e), and
– **remove_event**(e).

Insertion and deletion of an event into/from the environment are reserved operations that can be executed only by the "environment agent". Perception operations can be performed by all the agents in the system.

Lastly, any agent can execute an action a using the primitive

– **exec**(a).

The effect of this primitive is to send a message with content a to the "environment agent". This agent evaluates the consequences of performing a in the environment and modifies it by means of **put_event**(e) and **remove_event**(e) primitives.

3.2 Abstract Architecture Definition

To define an abstract architecture, we declare its components, its engine and the engine procedures. As an example, consider a BDI-like architecture. As previously described, it is characterized by four components: one for beliefs, one for goals, one for intentions and one for plans.

The definition of an abstract BDI architecture is depicted in Figure 2. The keyword **class** means that the *plans_component* will be instantiated during the definition

```
abstract_architecture {bdi} {
    components {
        class plans_component;
        agent beliefs_component, goals_component, intentions_component;
    };
    procedures
        { ... $definition\ of\ the\ engine's\ procedures$ ... }
    engine {
        decl event, selected_event, triggered_plans, selected_plans;
        while true do
            perceive_event();
            plan_triggering();
            plan_selection();
            upgrade_intentions_component();
            exec_intention();
            drop_succesful_goals();
        endwhile
    }
}
```

Fig. 2. BDI abstract architecture.

```
upgrade_intentions_component() {
    decl plan, empty;
    is_empty_selected_plans(!empty);
    if not(empty) then
        select_plan(!plan);
        put_intentions_component(gettupla(plan,3))
    else skip
}
```

Fig. 3. A partially defined procedure.

of the agent class, and thus that the data contained in it represent the *program* of the agent. The keyword **agent** suggests that *beliefs_component*, *goals_component* and *intentions_component* will contain information representing the agent's state.

The engine consists in a "while" loop continuously executing a sequence of macro-instructions defined as procedure calls. *Global variables* declarations can be included in the engine body. The body of a procedure may be only partially specified in the abstract architecture. This means that the implementation details of the macro-instruction defined by the procedure will be completely described when defining the concrete architecture. A macro-instruction can also be left completely undefined. In this example, *perceive_event()* is not defined at all, while *upgrade_intentions_component()* is partially defined, as depicted in Figure 3.

The procedure body contains declarations of *local variables* at the beginning. The procedures *is_empty_selected_plans(!empty)*, *select_plan(!plan)*[1], and *put_intentions_component(***gettupla***(plan,3))* are not defined at the abstract architecture level.

3.3 Concrete Architecture Definition

In the definition of the concrete architecture, all the components are assigned a type, the global variables are initialized, and the definitions of partially specified procedures are completed. To illustrate two different implementation choices, we consider two concrete BDI architectures, bdi_1 and bdi_2, obtained from the previously defined abstract BDI architecture bdi.

In concrete architecture bdi_1, *plans_component* is assigned a type `stack`, *beliefs_component* has type `set`, *goals_component* has type `set` and *intentions_component* has a type `queue`. External events are either events generated by the environment or messages sent by other agents in the system. An agent which is implemented using this architecture will have both reactivity and social ability.

In the architecture bdi_2, *plans_component*, *beliefs_component* and *goals_component* are `sets`, while *intentions_component* is a `stack`[2]. The only perceived events are those generated by the environment. This architecture gives origin to strongly reactive agents without the ability to receive messages. In both concrete architectures, perceived events are collected in the global variable *event* which has type `queue`.

The implementation of the BDI concrete architectures is depicted in Figure 4 and 5. In bdi_1, an event is perceived from the environment by means of the **perceive**(e_1) procedure. The global variable *event* is updated by inserting the perceived event into it. A message from *sender* is received in parallel with the perception of the environment, and the received message is also put into the event queue. In bdi_2, only events taking place in the environment are perceived and inserted in the event queue.

3.4 Definition of Agent Classes and Instances

After the concrete architectures have been defined the MAS is instantiated (Figure 6). The classes of agents are defined, the environment *env* and the ether *eth* are initialized and finally the instances of the agents are created. Due to space constraints we will not go into further detail on this aspect of the language.

4 Operational Semantics

The operational semantics of HEMASL specifications is given by a tree that represents the transitions between tuples of *MAS configurations*. A full account of the semantics can be found in [10], while a brief account is given here. A MAS configuration has the form

[1] The symbol "!" in the procedure call means that the argument is passed by reference.

[2] These types are probably not the most reasonable to assign to BDI components. They have been chosen to demonstrate language flexibility.

```
architecture {$bdi_1$} is a {bdi} {
  components {
      plans_component: stack;
      belief_component, goals_component: set;
      intention_component: queue
  };
  init_global_vars { ...};
  procedures {
      perceive_event() {
           decl sender, e_1, e_2;
           ( perceive(e_1); event := insqueue(event, e_1)) ||
           ( get_belief_component("sender", !sender); rec(sender, e_2);
           event := insqueue(event, e_2) )
      };
      ...
  }
}
```

Fig. 4. Definition of concrete architecture bdi_1.

$$[env,\ eth,\ s_{a_1},\ \ldots,\ s_{a_n},\ s_{env\_agt}]$$

where env and eth represent respectively the state of the MAS environment and the state of the MAS ether, $s_{a_1},\ \ldots,\ s_{a_n}$ represent the states of the "common" agents and s_{env_agt} is the state of the "environment agent".

The environment env is a set of pairs $\langle fact,\ value\rangle$, where $fact$ is a string representing a relevant fact characterizing the MAS environment (for example, "temperature"), and $value$ is a string representing the current value of the observed fact (for example, "25"). The ether eth is a set of triples of strings $\langle sender,\ receiver,\ content\rangle$. The term $sender$ must be instantiated by the name of the sender and $receiver$ may be instantiated by an agent name or with the string "all", to model broadcast communication. The ether contains all the messages that have been delivered but have yet to be received by an agent.

The state of the agent a is a pair $\langle ex\_env_a,\ cmp_a\rangle$. It contains information about the architecture components content (cmp_a) and about the execution environment of the agent engine (ex_env_a)[3]. Coherently with the meaning assigned to the execution environment in the imperative languages semantics, ex_env_a is a function which associates values to variables and local execution environments to procedure identifiers.

The relation $\overset{MAS}{\longmapsto}$ defines the transitions between MAS configurations. The actions of the "environment agent" may affect its state, the ether and the MAS environment, and thus the relation $\overset{env\_agt}{\longmapsto}$ is defined over *environment agent configurations* of the form $(s_{env\_agt},\ eth,\ env)$. Conversely, the actions of a "common agent" a cannot

[3] Some confusion could arise due to the presence of the agents' execution environment ex_env_a and the MAS environment env. The context always clarifies the meaning of the term "environment".

```
architecture {$bdi_2$} is a {bdi} {
  components {
      plans_component, belief_component, goals_component: set;
      intention_component: stack
  };
  init_global_vars { ...};
  procedures {
      perceive_event() {
          decl e;
          perceive(e); event := insqueue(event, e)
      };
      ...
  };
}
```

Fig. 5. Definition of concrete architecture bdi_2.

```
MAS {
  class_agent {$arch\_name_1,\ classagent\_name_1$} {
      init_comp $comp\_name_1$ [$elem_{1_1},\ \ldots,\ elem_{1_i}$]; ...
  };
  ⋮
  init_ENV [$event_1,\ \ldots,\ event_e$];
  init_ETH [$msg_1,\ \ldots,\ msg_m$];
  create_agent($classagent\_name_{t_1}$, $agent\_name_1$) {
      init_comp $comp\_name_1$ [$elem_{1_1},\ \ldots,\ elem_{1_i}$]; ...
  };
   ⋮
}
```

Fig. 6. MAS definition schema.

directly modify the MAS environment, and thus $\overset{agt}{\underset{a}{\longmapsto}}$ is defined over *agent configurations* $(s_a,\ eth)$. Let $\mathcal{A}$ be the set of names of the agents in the MAS. The above concepts are formalized by the meta-rules

$$\frac{(s_{env\_agt},\ eth,\ env) \overset{env\_agt}{\longmapsto} (s'_{env\_agt},\ eth',\ env')}{[env,\ eth,\ S,\ s_{env\_agt}] \overset{MAS}{\longmapsto} [env',\ eth',\ S,\ s'_{env\_agt}]}$$

$$\frac{(s_a,\ eth) \overset{agt}{\underset{a}{\longmapsto}} (s'_a,\ eth')}{[env,\ eth,\ S,\ s_{env\_agt}] \overset{MAS}{\longmapsto} [env,\ eth',\ S',\ s_{env\_agt}]} \quad a \in \mathcal{A}$$

where S' is obtained by substituting s_a with s'_a in S.

To give an overview of the language semantics we will describe some meta-rules governing the execution of "common" agents. The meta-rules for the "environment agent" execution are similar to the following ones; in addition, they also define the semantics of **put_event** and **remove_event**. The relation $\overset{agt}{\underset{a}{\longmapsto}}$ is defined by the meta-rule

$$\frac{(\langle ex\_env_a, cmp_a\rangle,\ eth) \overset{ins}{\underset{a,i}{\longmapsto}} (\langle ex\_env'_a, cmp'_a\rangle,\ eth')}{(\langle ex\_env_a, cmp_a\rangle,\ eth) \overset{agt}{\underset{a}{\longmapsto}} (\langle ex\_env'_a, cmp'_a\rangle,\ eth')} \quad \begin{array}{l} a \in \mathcal{A} \\ N\_I(a) = i \end{array}$$

The function $N\_I(a)$ returns the next instruction which agent a must execute. The relation $\overset{ins}{\underset{a,i}{\longmapsto}}$ is defined over agent configurations and depends on the agent performing the instruction and on the instruction itself. We give its definition for some HEMASL basic constructs.

Event perception

$$\frac{f \overset{e\_exp}{\underset{ex\_env_a}{\longmapsto}} f'}{(\langle ex\_env_a, cmp_a\rangle,\ eth) \overset{ins}{\underset{a,\mathbf{perceive}((f,x))}{\longmapsto}} (\langle ex\_env_a[v/x],\ cmp_a\rangle,\ eth)} \quad \begin{array}{l} a \in \mathcal{A} \\ (f', v) \in env \\ x \in\ d(ex\_env_a) \end{array}$$

The relation $\overset{e\_exp}{\underset{ex\_env_a}{\longmapsto}}$ evaluates an expression according to the current execution environment of agent a. The function $d(ex\_env_a)$ returns the domain of the function ex_env_a and $ex\_env_a[v/x]$ is obtained by composing ex_env_a with the function which associates v to x. The effect of a perception instruction is to modify the agent's state by creating a new association between the variable argument of **perceive** and the value associated to the perceived fact.

Message delivery

$$\frac{r \overset{e\_exp}{\underset{ex\_env_a}{\longmapsto}} r' \qquad c \overset{e\_exp}{\underset{ex\_env_a}{\longmapsto}} c'}{(\langle ex\_env_a, cmp_a\rangle,\ eth) \overset{ins}{\underset{a,\mathbf{send}(r,c)}{\longmapsto}} (\langle ex\_env_a, cmp_a\rangle,\ eth \cup \{(a,\ r',\ c')\})} \quad a \in \mathcal{A}$$

The effect of a **send** is the insertion of a new triple in the ether.

Message reception

$$\frac{s \overset{e\_exp}{\underset{ex\_env_a}{\longmapsto}} s'}{(\langle ex\_env_a, c_a\rangle,\ eth) \overset{ins}{\underset{a,\mathbf{rec}(s,x)}{\longmapsto}} (\langle ex\_env_a[c/x], c_a\rangle,\ eth/\{(s',\ a,\ c)\})} \quad \begin{array}{l} a \in \mathcal{A} \\ (s',\ a,\ c) \in eth \\ x \in\ d(ex\_env_a) \\ a \neq\ ''all'' \end{array}$$

This meta-rule can be applied when the receiver of the message is a particular agent and not the string "all". The effect of a **rec** is to associate the content of the received message to the variable argument of **rec** and to remove the read message from the ether. If the receiver is "all", the message is not removed from the ether until all the agents in the system have read it[4].

[4] The ether takes care that the receivers of a broadcast message read it only once, and that the message is removed when all the recipients have read it.

The same rules are also given for the blocking reception instruction. The difference between not-blocking and blocking reception semantics is that for non-blocking reception a third meta-rule exists. It can be applied when the desired message is not present in the ether. In this case the effect of a **rec** is to create an association between the variable argument of **rec** and the string "null", with no side-effects on the ether. Conversely, the semantics of the blocking reception is undefined if the message is not present in the ether thus forcing the agent to block its execution.

Action execution

$$\frac{act \overset{e\_exp}{\underset{ex\_env_a}{\longmapsto}} act'}{(\langle ex\_env_a, cmp_a\rangle,\ eth) \overset{ins}{\underset{a,\mathbf{exec}(act)}{\longmapsto}} (\langle ex\_env_a, cmp_a\rangle,\ eth \cup \{\ (a,\ env\_ag,\ act')\ \}} \quad a \in \mathcal{A}$$

The semantics of an **exec** instruction is to send a message to the "environment agent" containing the action to be performed. This is achieved by modifying the ether.

5 Comparison

In this section we compare our approach to the specification of heterogeneous agent architectures with other proposals: [5], by Fisher; [8], by Hindriks et al.; [3], by Treur et al., [6] by De Giacomo et al. and [16] by van Eijk et al.. The proposals will be compared with respect to their capabilities to specify architectures that comply with the model of the agent presented in Section 2, i.e., in terms of state, program and engine.

In [5], Fisher presents the specification of an "abstract agent architecture" using Concurrent MetateM, a specification language based on temporal logics. The architecture model allows us to encompass different kinds of *behaviour*, performed by *groups* of sub-agents. Each behaviour is described as a set of temporal logics rules that specify how the future state of agent computation should be obtained by starting from its present state. Examples are presented on the specification of *reactive*, *deliberative*, and *social* behaviour and on the composition of these types of behaviour in different kinds of layered architectures. This approach seems highly suitable to represent the agent program thanks to the expressiveness of temporal logic. On the other hand, this approach does not give explicit representation of agent state and engine which are implicitly maintained in the *interpreter* for Concurrent MetateM, that executes Concurrent MetateM specifications.

Hindriks et al. follow a different approach. They mainly focus on specifications of agent engines that can be used as meta-interpreters for many different object level languages used to give agent programs. They assume that agent computation may be well expressed using programming languages based on the concept of *beliefs*, *goals* and *rules* and define an imperative meta language that is used to describe a "standard" engine cycle which includes *sensing*, *rule application*, and *goal execution*. They argue that a *glass box* approach for defining agent engines which makes internal functioning of engines visible to the MAS developer, is the "right" approach. In such a way the developer can directly program the control of the agent's internal activities. Following the approach by Hindriks et al., the development of agent engines is natural and immediate. However, it constrains the agent state to be composed by beliefs and goals, and the agent program to be composed by rules. Even if most of the agent architectures are designed in terms of these abstractions, the approach could not be as general as needed to fit some agent applications.

The DESIRE research focuses on the study of compositional MAS for complex tasks and development methods for these systems. The structure of DESIRE specifications is based on the notion of compositional architecture: an architecture composed of components with hierarchical relations among themselves. This approach flexibly supports the definition of different heterogeneous architectures. Some architecture components may contain the agent's state and some others the agent's program. The different purposes of these two kinds of components do not arise, however, either from a syntactical or from a semantical point of view. The flow of information among the components is described by the specification of communication links. There is not an engine governing the execution flow inside the agent's components. Even if the DESIRE framework does not provide explicit support for defining agent architectures in terms of state, program and engine, it has been successfully adopted to develop a library of heterogeneous architectures ranging from reactive and proactive ones, to reflective and BDI ones.

ConGolog specifies the agent's behaviour based on the actions an agent can execute. The language adopted for this purpose is the *situation calculus*, a first-order language (with some second order features) for representing dynamic domains. ConGolog also includes facilities for prioritizing the execution of concurrent processes, interrupting the execution when certain conditions become true, and dealing with exogenous actions. These features make adoption of the language for implementing reactive agents possible. On the other hand, the ascription of mental attitudes to agents can easily be achieved by adapting the possible worlds model of knowledge to the situation calculus. Communicative capabilities can also be modelled in the framework. ConGolog provides the means for describing both the declarative and the procedural knowledge of agents with very different capabilities ranging from reactivity to rationality. Nevertheless, this is always done at the "program" level. The engine for these programs is always the same, as is the agent architecture.

Finally, [16] defines a multi-agent programming language in which concepts from the object-oriented paradigm are adapted and generalized in the light of communication among agents. An agent class $\mathcal{A}$ is a tuple (C, Q, D, S, ϕ). C is a first-order system describing the language and operators the agents in the class employ to represent and process information. Q is a set of question templates the agents in the class can answer, and represents the interface of the class. S denotes a programming statement in which procedures that are declared in D can be invoked: upon its creation each agent of the class will start to execute this statement. $\phi \in C$ constitutes the initial information store of the agents in the class. There are many similarities between HEMASL and this language: as far as the agent model is concerned, C resembles the *architecture engine*, which can be different for different classes, S corresponds to the *agent program* and ϕ to the *agent state*. The language for composing statements provides, like HEMASL does, atomic operations for updating the information store and for communication, as well as nondeterministic choice and parallelism operators. The semantics of the language is given in terms of transition rules quite similar to the ones given for HEMASL.

HEMASL lets the architecture designer choose the most suitable components and agent programming languages, in order for the agent state and program to encompass a great variety of representations, and for the architecture to provide a high degree of flexibility. Furthermore, the HEMASL primitives allow the architecture designer to build the engine as he/she wants. We argue that an "opaque box" approach is the best solution for agent-based software development. The MAS developer should know the main characteristics of the architectures he/she can use for the applications, but he/she is not usually an expert and does not need to be burdened with implementation details about architecture control. By giving a great deal of freedom in architecture development, we can obtain a library including architectures with several data structures and control flows. In such a way, the MAS developer can (hopefully) find the architectures that fits his/her needs in the library.

6 Conclusions

The realization of MAS often involves the choice of agents with heterogeneous architectures, so that each agent can optimally provide its services. HEMASL is a simple meta-language which is used to specify heterogeneous agent architectures, in terms of the components that form the agent state and program, and the engine that implements the mechanisms concerning control of the agent's internal activities. These specifications can be subsequently implemented into a library of architectures that MAS developers can use.

Adding HEMASL to CaseLP, our specification and prototyping environment for MAS, enhances its flexibility and usability. In fact, HEMASL can be the "intermediate language" through which non-executable specification languages (both formal and commercial) can be integrated into CaseLP and thus animated using CaseLP working prototypes. Moreover, we think that HEMASL can be used as a specification language for the development of agents which will be implemented into commercial, object-oriented programming languages. Translation of HEMASL specifications into Java programs is presently being investigated [11].

Acknowledgments. The authors thank Valeria Perricone for her helpful contribution in improving the presentation of the paper, and the anonymous referees for their useful comments.

This research was carried out while Floriano Zini was at DISI, Università di Genova, Italy.

References

1. A. Aretti. Semantica di Sistemi Multi-Agente in Logica Lineare. Master's thesis, DISI – Università di Genova, Genova, Italy, 1999. In Italian.
2. M. Bozzano, G. Delzanno, M. Martelli, V. Mascardi, and F. Zini. Multi-Agent Systems Development as a Software Engineering Enterprise. In G. Gupta, editor, *Proc. of First International Workshop on Practical Aspects of Declarative Languages (PADL'99)*, number 1551 in Lecture Notes in Computer Science. Springer-Verlag, 1999.

3. F. Brazier, B. Dunin Keplcz, N. R. Jennings, and J. Treur. Formal Specification of Multi-Agent Systems: a Real-World Case. In *Proc. of International Conference on Multi Agent Systems (ICMAS'95)*, San Francisco, CA, USA, 1995.
4. G. Delzanno and M. Martelli. Proofs as Computations in Linear Logic. *Theoretical Computer Science*. To appear.
5. M. Fisher. Representing Abstract Agent Architectures. In M. P. Singh J. P. Mueller and A. S. Rao, editors, *Intelligent Agents V*, number 1555 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1999.
6. G. De Giacomo, Y. Lespérance, and H. J. Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.
7. J. Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1:1–102, 1987.
8. K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J. C. Meyer. Control Structures of Rule-Based Agent Languages. In M. P. Singh J. P. Mueller and A. S. Rao, editors, *Intelligent Agents V*, number 1555 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1999.
9. N. R. Jennings, K. Sycara, and M. Wooldridge. A Roadmap of Agent Research and Development. *Autonomous Agents and Multi-Agent Systems*, 1:7–38, 1998.
10. S. Marini. Specifica di Sistemi Multi-Agente Eterogenei. Master's thesis, DISI - Università di Genova, Genova, Italy, 1999. In Italian.
11. S. Marini, M. Martelli, V. Mascardi, and F. Zini. HEMASL: A Flexible Language to Specify Heterogeneous Agents. In A. Corradi, A. Omicini, and A. Poggi, editors, *WOA 2000. Dagli Oggetti agli Agenti*, Parma, Italy, 2000.
12. M. Martelli, V. Mascardi, and F. Zini. Towards Multi-Agent Software Prototyping. In H. S. Nwana and D. T. Ndumu, editors, *Proc. of The Third International Conference and Exhibition on The Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM'98)*, London, UK, 1998.
13. M. Martelli, V. Mascardi, and F. Zini. Specification and Simulation of Multi-Agent Systems in CaseLP. In M. C. Meo and M. Vilares Ferro, editors, *Proc. of Appia–Gulp–Prode 1999*, L'Aquila, Italy, 1999.
14. J. P. Müller. The Right Agent (Architecture) to Do the Right Thing. In M. P. Singh J. P. Mueller and A. S. Rao, editors, *Intelligent Agents V*, number 1555 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1999.
15. A. S. Rao and M. Georgeff. BDI Agents: from Theory to Practice. In *Proc. of International Conference on Multi Agent Systems (ICMAS'95)*, San Francisco, CA, USA, 1995.
16. R. M. van Eijk, F. S. de Boer, W. van der Hoek, and J. C. Meyer. Generalised Object-Oriented Concepts for Inter-Agent Communication. In C. Castelfranchi and Y. Lespérance, editors, *Intelligent Agents VII*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 2001. In this volume.
17. M. Wooldridge. Agent-based Software Engineering. *IEE Proc. of Software Engineering*, 144(1), 1997.