HEMASL: A Flexible Language to Specify Heterogeneous Agents

Simone Marini Maurizio Martelli Viviana Mascardi
Floriano Zini
D.I.S.I. - Università di Genova
Via Dodecaneso 35, 16146 Genova, Italy
marini@educ.disi.unige.it
{martelli,mascardi,zini}@disi.unige.it

Abstract

In the realization of agent-based applications the developer generally needs to use heterogeneous agent architectures, so that each application component can optimally perform its task. Languages that easily model the heterogeneity of agents' architectures are very useful in the early stages of the application development. This paper presents HEMASL, a simple meta-language used to specify heterogeneous agent architectures, and sketches how HEMASL should be implemented in an object-oriented commercial programming language as Java. Moreover, the paper briefly discusses the benefits of adding HEMASL to CaseLP, a LP-based specification and prototyping environment for multi-agent systems, in order to enhance its flexibility and usability.

1. Introduction

Intelligent agents and multi-agent systems (MASs) are more and more recognized as the "new" modeling techniques to be used to engineer complex and distributed software applications [12]. Agent-based software engineering is concerned with the realization of software applications modeled as MASs.

A two-phases iterative approach can be followed for agent-based software development, before implementing the final application.

1. *Specification of the MAS*. This phase concerns the description of the services that each agent in the MAS provides, the modeling of agent-agent, agent-human and agent-environment interactions, and the description of domain-dependent procedural knowledge, used by each agent to supply its services and to respond to stimuli from the environment.
2. *Proof of the specification correctness*. The specification properties are formally verified, and/or informal testing of the MAS behavior is performed by means of a working prototype.

Generally, the first phase does not explicitly take into account the *architecture* of each agent and profitably abstracts from the internal organization and structure of single agents. In this way, the specification is given at the *macro-level*, it is clearer and more modular and the application developer in not burdened with the modeling of too many details.

On the other hand, the services that an agent provides are usually variegated and complex and generally involve management of heterogeneous information using different behaviors. In other words, an application needs to incorporate agents having heterogeneous architectures, so that each application component can optimally perform its task [10]. For example, real-time control systems for industrial machineries could be implemented by *reactive* agents, whereas a long-term planner of the maintenance activity on all the control systems by a *deliberative* agent[1]. These agents could be incorporated into an application that monitor all the activities of the industry.

In general, a good approach is to provide the MAS developer with a pre-defined library of architectures from which he/she can choose the most appropriate ones. Obviously, the architectures in the library must be specified, verified and tested before being employed. Specification at the *micro-level* involves the description of internal components of the agent and interactions among them.

At the Computer Science Department of Genova University, the Logic Programming Group is working since 1997 on CaseLP, a specification and prototyping environment for MASs. CaseLP (*Complex Application Specification Environment based on Logic Programming*) [8, 7] provides a development method as well as tools and languages which support the various steps of the method. Target of the

[1] For a survey on agent architectures, see [9].

development is generally the realization of an executable MAS prototype, implemented in a logic programming language, encompassing all the main features of the final application and amenable to be animated and tested. Moreover, CaseLP is also suitable for micro-level agent development.

The declarative nature of logic programming languages makes them very suitable for interactive development and testing of agent applications: they can be used to specify agents and MASs at the right 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, for a widespread use of this technology, two considerations have to be done:

- 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 adoption even harder than adopting the "simple" and "user-friendly" Prolog!

For these reasons one of the starting point in the design of CaseLP was the idea to have an environment able to accommodate different specification languages, different implementation languages and legacy software, in such a way a MAS prototype can be built using a range of tools and techniques integrated into a common framework.

CaseLP has been designed to provide a library of agent architectures, implemented in a logic programming language, that can be picked up and used for the implementation of the prototype.

In our method the more formal and abstract specifications can be given using the executable linear logic programming language $\mathcal{E}_{hhf}$ [3], which provides constructs for concurrency and state-updating. Even if $\mathcal{E}_{hhf}$ has been successfully adopted as specification language for both micro-level [2] and macro-level [1] modeling, its use requires a deep knowledge in the linear logic syntax and semantics [4] and the language is definitely difficult to adopt.

In this paper we try to bridge the gap between the above mentioned users' habits and our tool for rapid prototyping. We present HEMASL [6], which can be seen as a simple meta-language for specifying agent architectures (micro-level) and which is much closer to widespread existing implementation and specification languages. HEMASL's imperative setting, as well as the hierarchy of abstractions it supports, should make its adoption quite easy for those users familiar with traditional programming paradigms.

The integration of HEMASL in CaseLP is currently under study and it is based on the translation of HEMASL specifications into $\mathcal{E}_{hhf}$ sets of clauses. $\mathcal{E}_{hhf}$ specifications can be tested and/or verified and refined in CaseLP and finally translated into Prolog implementations to be included in the CaseLP library of agent architectures. For these reasons and for its closeness to widespread specification and implementation languages, HEMASL can be seen as an "intermediate" language for rapidly developing a library of agents architectures to be integrated, executed and tested in CaseLP, without requiring any confidence with the logic programming paradigm.

After the correctness of HEMASL specifications has been verified by means of their integration in CaseLP, they can be mapped into commercial programming languages, in particular object oriented languages. This paper delineates how HEMASL specifications can be translated into Java programs. We aim at building a library of already-tested Java architectures which can be used as building blocks to implement agent-based applications.

In the following sections we expand some of the concepts outlined above. Section 2 introduces the main features of HEMASL and the hierarchy of abstraction levels. Section 3 deals with how developing HEMASL specifications. Section 4 outlines the mapping between HEMASL and Java. Finally, Section 5 concludes the paper individuating some future research directions.

2. A flexible language for agent architectures

HEMASL is a simple and easy-to-use imperative meta-language whose features make it suitable for specification of agent architectures.

- **Agent model.** The agent model adopted by HEMASL is an abstraction of many existing architectures. This makes it easier the development of heterogeneous architecture specifications. Moreover, the HEMASL agent model is the same we adopted in CaseLP, so the integration of HEMASL specifications into CaseLP is facilitated.

- **Hierarchy of abstraction levels.** HEMASL provides constructs for specifying an agent through four different levels of abstraction that support a modular and flexible representation based on the concepts of abstract and concrete architecture, agent class and agent instance.

- **Situatedness and social ability.** 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 modeling the message exchange among agents.

- **Semantics.** HEMASL has an operational semantics [6] that describes the execution of an agent as a tree of "agent configurations". This formal semantics can be exploited to prove that the translation of HEMASL into $\mathcal{E}_{hhf}$ or Java is correct.

In the following, we detail the first and second features, whereas aspects of HEMASL related to the third and fourth items are not deepened in this paper.

2.1. Agent model

Since we want to model agents having different architectures, we need a simple abstraction able to encompass most of the existing architectures. The agents we model with HEMASL are characterized by:

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

We do not propose yet another definition of agent (see [5]), but only a compact and simple characterization of an agent architecture.

The *state* includes data that may change during the execution of the agent. For example, the state of a BDI agent [11] contains its beliefs, goals and intentions. The *program* contains that part of information that does not change during the execution of the agent. The program of a BDI agent is determined by its plans. Finally, the *engine* controls the execution of the agent. A typical BDI engine should be characterized by percepting an event, individuating a set of plans which can be used to manage the percepted event, selecting one plan from the set, adding the selected plan to the intention set, selecting one intention and finally executing one instruction of the selected intention.

The engine and the program are situated at different abstraction levels: the engine is a meta-interpreter for the program and the data (state) on which the program operates. The behavior 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 and 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.

2.2. Abstraction hierarchy

A specification language for heterogeneous agents should ensure modularity and flexibility in the definition of their architectures. For this reason we introduce a hierarchy composed by four abstraction levels (see Fig. 1). HEMASL

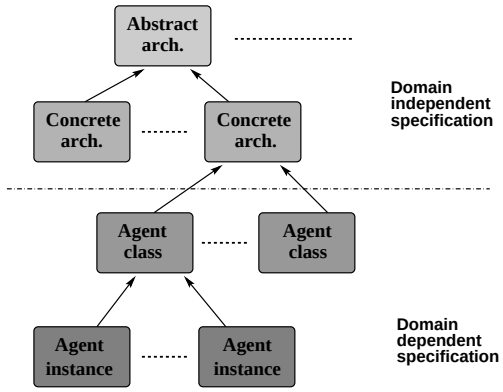

Figure 1. Abstraction hierarchy.

provides primitives for specifying every abstraction level.

- **Abstract architecture.** The abstract architecture defines the components in the architecture and the basic structure of the engine. For any "macro-instruction" (procedure) present in the engine, it is possible to sketch how it should be realized, without necessarily giving all the implementation details.
- **Concrete architecture.** A concrete architecture is defined starting from an abstract architecture:
  - each component is given a type chosen among the ones that HEMASL provides;
  - for each not completely defined macro-instruction of the engine an implementation is given.
- **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 individuates an agent instance.

Intuitively, an abstract architecture defines the components and the engine that organize agent internal activities without going into too many details. For example, in a BDI abstract architecture which kind of data structures are used to implement beliefs, goals, intentions and plans, or how an intention to be executed is selected are irrelevant details at this level.

An abstract architecture can originate several concrete architectures. This level of abstraction defines the data structures used for architecture components, as well as the detailed functioning of the agent.

```
abstract_architecture {bdi}{
    components{
        class plans_component;
        agent beliefs_component;
        agent goals_component
        agent intentions_component
    };
    procedures
        { ... definition of the engine's procedures ... }
    engine{
        decl event;
        while true do
            percept_event();
            plan_triggering();
            plan_selection();
            upgrade_intentions_component();
            exec_intention()
        endwhile
    }
};
```

Figure 2. BDI abstract architecture.

```
architecture {bdi1} is a {bdi}
{
    components{
      plans_component: stack; belief_component: set;
      goals_component: set; intention_component: queue };
    init_global_vars{
      ...; };
    procedures{
      percept_event()
      {
          decl sender;    decl e_1;    decl e_2;
          percept(e_1);    event := insqueue(event, e_1);
          get_belief_component("sender", !sender);
          receive(sender, e_2);    event := insqueue(event, e_2) };
     ...}};

architecture {bdi2} is a {bdi}
{
    components{
      plans_component: set; belief_component: set;
      goals_component: set; intention_component: stack };
    init_global_vars{
      ...; };
    procedures{
      percept_event()
      {
          decl e;
          percept(e);    event := insqueue(event, e) };
     ...}};
```

Figure 3. BDI concrete architectures.

The same concrete architecture can be employed for agents that work on various application domains. Domain dependent behavioral patterns are given defining the agent program, at the class level.

Finally, an agent behavior can be tested starting from different initial scenarios. This is captured by the presence, in our hierarchy, of the agent instance level.

3. Using HEMASL

To define an abstract architecture in HEMASL, its components, its engine and the engine procedures are declared. 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 Fig. 2. The keyword **class** means that the *plans_component* will be instantiated during the definition of the agent's class, and thus that the data contained in it represent the *program* of the agent. The keyword **agent** states 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-instruction defined as procedure calls. The body of a procedure may be only partially specified in the abstract architecture.

In the definition of a 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 possible implementation choices, consider two concrete BDI architectures, $bdi1$ and $bdi2$, obtained from the abstract BDI architecture. In the concrete architecture $bdi1$, *plans_component* is assigned a type `stack`, *beliefs_component* has type `set`, *goals_component* has type `set` and *intentions_component* has a type `queue`.

In the architecture $bdi2$, *plans_component*, *beliefs_component* and *goals_component* are `sets`, while *intentions_component* is a `stack`[2].

Besides assigning different types to the components, the two concrete architectures also implement the engine "macro-instructions" in different ways. In $bdi1$ the perception of events consists in retrieving an event taking place in the environment (procedure call **percept**(e_1)) and in retrieving a message coming from *sender* (procedure call **receive**(sender, e_2)). In $bdi2$, only events that occur in the environment are percepted and inserted in the event queue *event*. The $bdi2$ concrete architecture is suitable for those agents which do not possess social ability and are very reactive (for example, control systems), while $bdi1$ can be used for agents which combine reactivity to the environment with interaction with other agents. Fig. 3 sketches $bdi1$ and $bdi2$.

[2]Probably, these types are not the most reasonable to assign to BDI components. They have been chosen just to demonstrate the language flexibility.

4. Towards an OO implementation of HEMASL specifications

HEMASL specifications are hierarchies of abstraction levels: for this reason their implementation in an object oriented language is very appealing. In this section we outline some ideas about how HEMASL specifications should be implemented in Java.

Some considerations guide our mapping. Firstly, any agent is an autonomously executing entity, with its own thread of execution. This naturally leads to implement agents as concurrently executing Java *Thread*. A HEMASL abstract architecture declares the architecture components, the "macro-instructions" of the engine and eventually provides an implementation for some of them. An entity with these features finds a natural mapping in a Java `abstract class` where some methods can be left undefined and which extends the *Thread* class. As far as a HEMASL concrete architecture is concerned, it completes the information provided in the abstract architecture by defining those methods for which just the signature was given and by assigning a type to the architecture components. This HEMASL entity can be naturally mapped into a Java `class` which `extends` the Java abstract architecture. Finally, in a HEMASL agent instance both the program and the state components are filled with the proper information. Such an entity can be implemented as a Java object.
These considerations lead to following implementation of HEMASL entities.

HEMASL basic types. HEMASL provides the following basic types: *List*, *Queue*, *Set*, *Stack* and *Tuple*. We assume the existence of a Java class implementing each HEMASL type. All these classes inherit from *Hemasl_type*.

Abstract architecture. An HEMASL abstract architecture HAA can be implemented as a Java abstract class (JAA, *Java Abstract Architecture*) that extends the class *Thread*. The architecture components are implicitly declared by means of the methods for reading and writing them, which are declared `abstract`[3]. Also the procedures which are undefined in the HAA are implemented as `abstract` methods. The engine of the HAA can be implemented by overriding the method *run* of *Thread*. Fig. 4 sketches the mapping from HEMASL to Java of the abstract BDI architecture depicted in Section 2.

Concrete architecture. A HEMASL concrete architecture HCA can be implemented as a Java concrete class

[3]There are technical reasons for delaying the explicit declaration of the components when implementing the concrete architecture. Here do not discuss these subtle implementative details.

```
abstract class Bdi extends Thread {
        abstract Hemasl_type get_plans_component();
        abstract void set_plans_component(Hemasl_type plans);
        abstract Hemasl_type get_beliefs_component();
        abstract void set_beliefs_component(Hemasl_type plans);
        ...

        ...Auxiliary (abstract) methods...;

        abstract void percept_event();
        public void plan_triggering()
            { ...implementation of the method... };
        abstract void plan_selection();
        ...;

    public void run() {
        while (true) {
            percept_event();
            plan_triggering();
            plan_selection();
            upgrade_intentions_component();
            exec_intention() }
    }
};
```

Figure 4. BDI abstract architecture in Java.

```
class Bdi1 extends Bdi {
        Stack plans_component;
        Set beliefs_component;
        Set goals_component;
        Queue intentions_component;

        public void percept_event()
            { ... implementation of the method ... };
        ...;

        public Bdi1(
            Stack plans, Set beliefs, Set goals, Queue intentions)
            { implementation of the constructor };
    }
```

Figure 5. BDI concrete architecture in Java.

(JCA, *Java Concrete Architecture*) extending the JAA which implements the corresponding HAA. In the JCA each component is explicitly declared and is given a precise type among the ones supported by HEMASL. The constructor for JCA objects is also defined at this stage. It takes as arguments the architecture components, thus allowing the creation of an agent instance. In Fig. 5 the definition of the concrete architecture $bdi1$ is sketched.

Agent class. HEMASL classes have not a direct counterpart in a Java entity. Their implementation in Java is realized by defining the different Java objects which characterize the program components of the different classes. The program components of the agent instances belonging to the same agent class will all point to the same objects, which are

```
Stack diagnostic_agents_class_plans =
      new Stack(arguments characterizing
      the diagnostic agent program);

Stack robot_agents_class_plans =
      new Stack(arguments characterizing
      the robot agent program);
```

Figure 6. Programs for BDI diagnostic and robot agents in Java.

```
Bdi1 diagnostic_agents_instance1 = new Bdi1(
    diagnostic_agents_class_plans,
    new Set(...), new Set(...), new Queue(...));

Bdi1 diagnostic_agents_instance2 = new Bdi1(
    diagnostic_agents_class_plans,
    new Set(...), new Set(...), new Queue(...))

Bdi1 robot_agents_instance1 = new Bdi1(
    robot_agents_class_plans,
    new Set(...), new Set(...), Queue(...));
```

Figure 7. Instances of BDI diagnostic and robot agents in Java.

defined once for all for any agent class. As an example, if in HEMASL we have a $bdi1$ **diagnostic_agent** class and a $bdi1$ **robot_agent** class, we need to define the proper plans for both the classes, as depicted in Fig. 6. In this simple example we have just one "program component", namely the *plans_component*. In general, there should be more than one, and we should define all of them at this stage.

Agent instance. An HEMASL agent instance HAI is obtained from an agent class HAC by filling the components that corresponds to the agent state. In our mapping a *Java Agent Instance*, JAI, is an object whose *program* components are shared with all the agent instances of the same class and whose *state* components are "individual". In Fig. 7 two instances of diagnostic agents and one instance of a robot agent are defined.

5. Conclusions

The paper described HEMASL, a simple meta-language for specifying heterogeneous agent architectures. There are two main motivations behind our work. On the one hand, many existing specification languages for agents commit to a particular architecture, thus making it difficult their adoption when very different agents, ranging from reactive to rationale ones, are involved. A more flexible language, coping with the heterogeneity of agents' architectures, can prove useful in many contexts. On the other hand, the integration of HEMASL within CaseLP makes its use possible also for those users which are not familiar with the logic programming paradigm. Moreover, its mapping in the Java language represents a first step towards a real implementation of a system.

The future directions of our work are mainly involved with completing the mapping between HEMASL and Java, which by now is just outlined, and evaluating the possibility of mapping HEMASL in UML.

References

[1] A. Aretti. Semantica di sistemi multi-agente in logica lineare. Master's thesis, DISI – Università di Genova, Genova, Italy, 1999.

[2] M. Bozzano, G. Delzanno, M. Martelli, V. Mascardi, and F. Zini. Multi-Agent Systems Development as a Software Engineering Enterprise. In *Proc. of PADL'99*, San Antonio, Texas, January 1999. Springer-Verlag. LNCS 1551.

[3] G. Delzanno and M. Martelli. Proofs as Computations in Linear Logic. *Theoretical Computer Science*, 1999. To appear.

[4] J. Girard. Linear logic. *Theoretical Computer Science*, 50:1:1–102, 1987.

[5] 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.

[6] S. Marini. Specifica di sistemi multi agente eterogenei. Master's thesis, D.I.S.I. - Università di Genova, 1999.

[7] M. Martelli, V. Mascardi, and F. Zini. Towards Multi-Agent Software Prototyping. In H. S. Nwana and D. T. Ndumu, editors, *Proc. of PAAM'98*, pages 331–354, London, UK, March 1998.

[8] M. Martelli, V. Mascardi, and F. Zini. Specification and Simulation of Multi-Agent Systems in CaseLP. In *Proc. of Appia–Gulp–Prode 1999*, L'Aquila, Italy, September 1999.

[9] J. P. Müller. Control Architectures for Autonomous and Interacting Agents: A Survey. In W. W. Laurence Cavedon, Anand Rao, editor, *Intelligent Agent Systems: Teoretical and Pratical Issues*, number 1209 in Lecture Notes in Artificial Intelligence, Cairn Australia, August 1997. Berlin Springer.

[10] J. P. Müller. The Right Agent (architecture) to Do the Right Thing. In J. P. Müller, M. P. Singh, and A. S. Rao, editors, *Intelligent Agents V*, number 1555 in Lecture Notes in Artificial Intelligence, pages 211–225, July 1998.

[11] A. S. Rao and M. Georgeff. BDI Agents: from Theory to Practice. In *Proc. of ICMAS'95*, San Francisco, CA, 1995.

[12] M. Wooldridge. Agent-based Software Engineering. *IEE Proc. of Software Engineering*, 144(1), 1997.