Combining Logical Agents with Rapid Prototyping for Engineering Distributed Applications

Philip Dart Ed. Kazmierczak Leon Sterling
Department of Computer Science & Software Engineering
University of Melbourne, Parkville 3052, Australia
{philip,edleon}@cs.mu.oz.au

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

V.S. Subrahmanian
Department of Computer Science, University of Maryland,
A.V. Williams Building, College Park, MD 20742 USA
vs@cs.umd.edu

Abstract

The realization of new distributed and heterogeneous software applications is a challenge that software engineers have to face. Logic Programming and Multi-Agent Systems can play a very effective role in the rapid prototyping of new software products. The paper proposes a general approach to the prototyping of complex and distributed applications modelled as Multi-Agent Systems and outlines the autonomous research experiences of different research groups from which the approach originates. All the experiences have Logic Programming as the common foundation and deal with different aspects of the problem: integration of heterogeneous data and reasoning systems, animation of formal specifications and development of agent based software. The final goal is joining the diverse experiences into a unique open framework.

1 Software Engineering Challenges

Despite thirty years of research and experience and many successful results, software systems are still difficult to engineer to guarantee correctness and reliability. This is particularly true for distributed software systems where a set of entities have to cooperate and coordinate in order to exchange information. Many distributed systems must make use of existing software modules and consequently must integrate information from a potentially large number of diverse sources. Thus, integration and reuse of different kinds of information and software tools is an urgent necessity that new software products must address.

The agent-oriented paradigm [27, 19] is an emerging approach which permits a high level model of applications in which many autonomous, intelligent and interacting entities (a Multi-Agent System or MAS) cooperate to achieve a common goal or compete to satisfy personal needs. Many applications in which legacy software must be integrated into a new distributed system can be modelled as Multi-Agent Systems.

Our interest in agents is driven by the firm belief that they meet the demands of complex interaction between components of application. We assume a loose “declarative” definition of an agent as an autonomous, social, reactive and proactive piece of software that provides some services and is able to communicate with other agents using a common agent communication language.

Due to the inherent complexity of the applications modelled as MAS, it is important to build them following a well-established method. Rigorous processes, such as formal methods, are recognized as essential for correctness, but their key problem is the difficulty in determining the client’s needs quickly and accurately. Further, they usually force the developer to give a (too) detailed system specification, and also fail to take into account the modelling and verification of existing software modules if they are not formally
specified. Prototyping and animating specifications are recognized as an aid to requirement clarification and preliminary design, and can be also used for integrating existing software into the model. Moreover, they enhance an iterative approach to software development, much more suitable than the classical waterfall model to handle complex applications.

In this paper we propose a method and an environment which allow software engineers to interactively prototype specifications of distributed systems in order to facilitate the development of heterogeneous software. In particular we concentrate on two issues which nowadays seem to be fundamental for the success of applications: distribution and integration. The former concerns chiefly the organization of the interaction of the different components that make up the application. The latter is instead related to the integration and reuse of legacy software.

We believe that an essential aspect of a method for the development of new software products is to be open with respect to existing software and its distribution. More precisely, an appropriate method should be able to take into account both new components and old legacy systems that will form the final applications, and provide modelling techniques and tools by means of which (partial) specification and prototypes at various level of details can be tested, verified and refined.

We present a general approach to the prototyping of complex distributed and heterogeneous applications. We propose a rigorous software development method and a supporting environment which integrate prototyping and formal specification. In the proposed method, the activities of specification and prototyping would take place concurrently, each towards the goal of iteratively and incrementally developing a validated formal specification. The environment would support the complementary techniques of animating parts of the specification, as well as providing for specification-based testing and formal verification of correctness (e.g. model checking).

The base for our open multi-agent framework is Logic Programming (LP) [11], a high level programming paradigm which blurs the distinction between specifications and code. We believe that Logic Programming can be very useful in the definition of our method and environment, mainly because it can provide the right mix between formalism and experimentation. It is a high level language, amenable to formal reasoning. A logic program is an executable specification and this encourages rapid prototyping. A logical language can naturally be the target language for the animation of many not-executable specification languages [24]. On the side of integration of heterogeneous systems, Logic Programming can profitably address semantic integration that is the process of specifying methods to resolve conflicts, pool information together and define new operations based on operations belonging to different domains [12]. BDI frameworks [20] can use logic to represent goals and plans. Logic programming is effective for representing ontologies. Finally, meta-programming features of Logic Programming can be flexibly exploited to define agents with different architecture and control [10, 3], in such a way that many kinds of agents, suitable for different tasks, can be encompassed in an application.

The paper is structured as follows. The next section gives the main ideas as to how our open multi-agent framework should be structured, as well as a method for applications development. Section 3 presents different projects which fit the framework depicted in Section 2 and describes some results obtained by the different contributors. Finally, Section 4 concludes the paper.

2 An Open Multi-Agent Framework

2.1 Our general approach

As we have already sketched in the previous section, our aim is to develop an open architecture based on Logic Programming which supports the prototyping and the integration of existing components.

We will assume that we have at our disposal the components depicted in Fig. 1. At the top of the picture there are some specification languages; each of them is suitable for specifying a particular aspect of the MAS architecture and behaviour. The application developers are not necessarily “experts” of all the specification languages at their disposal but choose the most useful ones with respect to their needs.

For example, there could be a non-executable specification language $A$ based on $Z$ [23] and an executable specification language $eC$ based on Linear Logic, which can be respectively used either for modelling the high level interactions between agents or the global evolution of the system. A specification language $D$, based on algebraic models, should be suitable for describing the static structure of the agents, while another executable specification language $eB$ should help in modelling the dynamic behaviour of a single agent. The development of a specification is described in the picture by $S$ arrows. The top layer in Fig. 1 contains executable specification languages $eA$ to $eZ$ and non-executable specification languages $A$ to $Z$. For any specification language, both executable and non-executable, we assume a (semi-) automatic compiler which can produce an executable form of any specified agent, in a logic programming target language: the $C$ arrows on the left of the picture represent this compilation step. If the specification does not describe architectural details for the agents (for example, using Petri nets it would be quite easy to describe how the agents interact, but it would be tedious to describe how the agents are structured internally), the (semi-)automatic tool enriches
this specification with all the default mandatory details for building the executable prototype.

Once a prototype of the MAS has been created by compiling the various components of the specification we can execute the prototype and so conduct experiments which will detect problems with the specification itself. Animation provides a powerful means of testing a model interactively; even if it can never prove that the formulation of a model is consistent, or correct, or complete, it can be used to obtain a similar level of confidence in the consistency, correctness and completeness of the model as that given by program testing: it trades off completeness for speed and ease of use. Besides the other well-known advantages of animation, our approach has two more advantages:

1. We will mix the positive aspects of a formal approach with the satisfaction of needs closer to the real world problems: as explained below, we assume the existence of interface agents which actually integrate external software. The agents specified in a high level language, when compiled, will interact with these interface agents, thus providing an integration between existing software and the compiled form of a high level specification.

2. We will mix executable forms of agents specified in different languages: since the target language in which the agents are compiled is the same, and since all the details necessary for developing an executable agent are either explicitly specified or added by the compiler, all the compiled agents have an homogeneous form, and are able to interact.

If an executable specification language has been used, it is possible to directly interpret the given specification before compiling and animating it. Both direct execution and animation have the purpose of providing a better understanding of the system, thus facilitating the refinement of the specification, until the obtained executable prototype behaves exactly as expected. In Fig. 1, the R arrows represent the refinement stage. If the specification language is executable, it is possible to analyze the specification’s behaviour and to refine it without developing the executable MAS prototype. Whether or not the specification language is executable, a formal approach can be used to verify properties of the specification, and refinement can also be done formally. The looping R arrows describe this refinement cycle. Using compilation will bring us closer to the actual behaviour of the final system because it incorporates the legacy

Figure 1. Our approach.
components.

The bottom rightmost part of Fig. 1 represents legacy software and data that are integrated into the prototype. As well as this kind of module, external specifications can be taken into consideration. Thus, specification languages can be used not only for defining agents, but also for describing the behaviour of an external module\(^1\). Using an executable specification language it is possible to develop an executable specification of an external module, which can be treated as any other legacy software. If the language is not-executable, an ad-hoc compilation process can produce an animatable form of the specification (see the C arrow in the right part of the picture).

Finally, the executable MAS prototype will be implemented in a logic programming target language, for example, ECLiPSe [6], Mercury [22], SICStus Prolog [8] or HERMES [26]. At least two types of agents are necessary for accomplishing the complex tasks typical of MAS applications: (1) logical agents, which show capabilities of high-level reasoning, carried out through symbolic manipulation and theorem proving, and (2) interface agents, which provide an interface towards the external software modules (the access to external software is represented by the E lines in Fig. 1). A third kind of agent, hybrid agents, combine the skills of the other two types of agents. Communication between agents occurs in an agent communication language (communication channels are depicted by the I lines in Fig. 1).

To better understand the nature and roles of the components in Fig. 1 we have the following example. Consider a hypothetical scenario where the application being modelled involves a Job-finding agency (see Fig. 2), which must cooperate with industries to find jobs for as many candidates as possible. The agency’s main components are a Public Relation Office (PRO), modelled by a logical agent, and a Candidate Processing Office (CPO), modelled by an interface agent. The PRO engages in dialog with industries, modelled as logical agents, to find the best solution for all the parties involved. The CPO processes the images of the candidates’ resumés, contained in an existing multimedia database, using an existing C module which recognizes type-written characters. The module which indexes the resumés according to their content may use standard text indexing techniques [26], implemented in a full text indexing system accessible through the HERMES mediation system [4, 13]. The interface agent modelling the CPO interfaces the existing legacy software (multimedia DB, character recognizer and Resumés sorter).

According to industries’ requests, or autonomously at

\(^1\)We distinguish between agents and external modules since the latter are pieces of software which provide functionality without communication capability, autonomy, or intelligence; they can be inserted into a multi-agent system only if there are agents interfaced to them.

regular intervals of time, the PRO interacts with the CPO to find out from the database the resumés of people who match certain requirements. Since the PRO wants to find a job for as many persons as possible, it tries to combine the industries’ requests to optimize the number of engagements. It interacts with industries to propose candidates and to adjust its plans according to the industries’ answers; it needs to adopt some high-level reasoning mechanism to behave rationally. The CPO has the task of interacting with the external modules and integrating their answers for providing a final answer to PRO. Its behaviour is more reactive, but the task it performs is complex.

We have started our example by describing the components belonging to the executable MAS prototype, but the execution of the prototype comes only after a specification stage. In the Job-finding agency example, this step should start by choosing one specification language, then modelling the interactions between CPO, PRO and industries, and finally animating this specification to analyze the behaviour of the system. After this stage, it would be possible to refine the specification by choosing another language and modelling lower-level details or aspects different from the ones described before.

The animation of this refined specification will lead to new improvements to the system; after some refinement steps the behaviour of the implemented prototype will match all the initial requirements.

In the following, we formalize these ideas: we provide a six step method for realizing the final prototype; Fig. 1 describes steps 3 through 6, while the first two steps are not in the picture.

2.2 A method for realizing prototypes

The realization of a MAS-based software prototype for a complex application can be performed according to the schema depicted in Fig. 3 and explained by the following steps.

1. Static architectural description of the prototype.

The developer decides the static structure of the MAS. This phase lists the components, the services they provide, which services they require, and the communication channels that exist among agents. Further, the developer chooses the most appropriate architecture of each agent in the system (e.g. if the agents are purely reactive or proactive or more complex BDI agents). The chosen architecture can belong to a library of predefined architectures, or can be new. In the latter case it has to be further specified in the next steps.

2. Components interactions description.

This step specifies how a service is provided/requested by means
of a particular conversation between each pair of connected agents. Each conversation can be performed using different communication models, such as synchronous or asynchronous message passing.

3. Specification of the system. In this step the specifications languages come into play, and we can identify at least three different levels of modelling:

(a) specification of interactions among agents (external concurrency), abstracting from their architecture and taking into account the interaction model specified in step 2;

(b) specification of the (new) architectures chosen in step 1, that is, modelling of the interactions between agents’ internal components (internal concurrency);

(c) specification of the agents’ behaviour that is, how they operate to provide their services.

For any of these levels the system developer can choose the most suitable specification language in the set of available specification languages. The whole process given by steps 1 through 3 may be repeated more than once, either because the next phase (step 4) reveals some flaws in the initial choices, or because the developer wishes to refine the specification by providing more detail and/or using a different specification language. For example, as a first stage only the specification 3a above could be given with 3b and 3c being defined later.

4. Analysis, verification and testing of the specification. This phase concerns testing and/or analyzing and verifying the specification in order to check how much the built prototype corresponds to the wanted requirements. Testing can take place if the specification given in the previous step is executable. Analysis, verification and testing can give feedback to some of the previous steps.

5. Implementation of the prototype. This step transforms the (more) abstract specifications defined in step 3 into a prototype. This prototype is closer to the final implementation because it incorporates external software, data or specifications as well as support for message exchange among agents. The choice of a logic programming language as the target language makes compilation easier. Each of the specifications 3a, 3b and 3c has to be translated into executable code, for example, the specifications in 3a may be translated into different protocols for message exchange; the specifications in 3b may be implemented based on suitable data structures (realizing different architecture parts) and a meta-program that implements the architecture’s flow of control. The architecture-dependent behavioural specification 3c may be translated into rules written in the target language. Finally, the agents that form the prototype are joined into a unique executable specification, in such a way the system can be further tested.

6. Testing of the prototype. The last step further tests the system with respect to “real” communication among agents and actual interaction with legacy programs, data or specifications. Any error or misbehaviour discovered in this step may require a revision of the choices made in the previous steps.

3 Some Components of the Open Framework

Some systems suitable for being integrated into the open framework already exist: we first describe these systems, and then we show which role they should play in our open framework.
3.1 HERMES

In this section, we will briefly describe the basic theory behind mediated systems proposed in [12, 1]. Illustration is provided via the HERMES implementation.

A domain, \( D \), is an abstraction of databases and software packages and consists of three components: (1) a set \( \Sigma \) whose elements may be thought of as the data-objects that are being manipulated by the package in question, (2) a set \( \mathcal{F} \) of functions on \( \Sigma \) – these functions take objects in \( \Sigma \) as input, and return, as output, objects from their range (which needs to be specified). The functions in \( \mathcal{F} \) may be thought of as the predefined functions that have been implemented in the software package being considered, (3) a set of relations on the data-objects in \( \Sigma \) – intuitively, these relations may be thought of as the predefined relations in the domain, \( D \).

A constraint \( \Xi \) over \( D \) is a first order formula where the symbols are interpreted over \( D \). \( \Xi \) is either true or false in \( D \), in which case we say that \( \Xi \) is solvable, or respectively unsolvable in \( D \). The key idea behind a mediated system is that constraints provide the link to external sources, whether they be databases, object bases, or other knowledge sources. This idea is developed extensively in [12, 1] and we do not elaborate on them here. For example in HERMES, a domain call is a syntactic expression of the form

\[
\text{domainname: (domainfunct)((arg1, ..., argn))}
\]

where \( \text{domainfunct} \) is the name of the function, and \( (\text{arg1}, ..., \text{argn}) \) are the arguments it takes. Intuitively, a domain call may be read as: in the domain called \( \text{domainname} \), execute the function \( \text{domainfunct} \) defined therein on the arguments \( (\text{arg1}, ..., \text{argn}) \). The result of executing this domain call is coerced into a set of entities that have the same type as the output type of the function \( \text{domainfunct} \) on the arguments \( (\text{arg1}, ..., \text{argn}) \).

A domain-call atom (DCA-atom) is of the form

\[
\text{in}(\text{X}, \text{domainname}: (\text{domainfunct})((\text{arg1}, ..., \text{argn})))
\]

where \( \text{in} \) is a constraint that is satisfied only if the entity \( \text{X} \) is in the set returned by the domain call in the second argument of \( \text{in}(-,-) \). In other words, \( \text{in} \) is the polymorphic set membership predicate. More concretely, \( \text{in}(\text{A}, \text{ingles: select('criminals', 'name', 'smith')}) \) is a DCA-atom that is true just in case \( \text{A} \) is a tuple in the result of executing a selection operation (finding tuples where the \text{name} field is \text{smith}) on a relation called \text{criminals} maintained in a INGRES database system.

A mediator/constrained database is a set of rules of the form

\[
\text{A} \leftarrow \text{D}_1 \& \ldots \& \text{D}_m \& \text{A}_1, \ldots, \text{A}_n,
\]

where \( \text{A}, \text{A}_1, ..., \text{A}_n \) are atoms, and \( \text{D}_1, ..., \text{D}_m \) are DCA-atoms. Note that for simplicity, we restrict constraints to DCA-atoms of the form described above. This does not,
however, detract from the generality of the techniques described here [12].

3.2 IMPACT

IMPACT (Interactive Maryland Platform for Agents Collaborating Together) builds upon HERMES. In IMPACT, we have two kinds of entities.

Agents, which are software programs (legacy or new) that are augmented with several new interacting components constituting a wrapper. Agents may be created by either arbitrary human beings or by other software agents (under some restrictions).

IMPACT Servers, which are programs that provide a range of infrastructural services used by agents. IMPACT Servers are created by the IMPACT developers, rather than by arbitrary individuals.

An IMPACT agent may be built on top of an arbitrary piece of software, defined in any programming language whatsoever. The structure of IMPACT agents is presented below.

Application Program Interface: Each IMPACT agent has an associated application program interface (API) that provides a set of functions which may be used to manipulate the data structures managed by the agent in question. The API of a system consists of a set of procedures that enable external access and utilization of the system, without requiring detailed knowledge of system internals such as the data structures and implementation methods used. Thus, a remote process can use the system via procedure invocations and gets results back in the form defined by the output of the API procedure.

Service Description: Each IMPACT agent has an associated service description that specifies the set of services offered by the agent. These service descriptions are written in a specific HTML-like language.

Message Manager: Each agent has an associated module that manages incoming and outgoing messages.

Actions, Constraints and Action Policies: Each agent has a set of actions that it can physically perform. The actions performed by an agent are capable of changing the data structures managed by the agent and/or changing the message queue associated with another agent (if the action is to send a message to another agent). Each agent has an associated action policy that states the conditions under which the agent may, may not, or must do some actions.

In addition, IMPACT agents contain components to handle data security, metaknowledge, temporal reasoning, and reasoning with uncertainty.

3.3 PipeDream

The idea of Logic Programming arose from the realization that how you express axioms in logic affects computational properties when proving theorems. The task of expressing logical axioms in an appropriate form is akin to programming. This has proved very powerful in the thousands of applications developed in the language Prolog [25]. Good Prolog programmers rapidly learn, however, that it is very rare for people to write down statements of logic that are right first time. Execution and debugging, i.e. prototyping, is necessary. It should be noted that such prototyping can proceed very quickly and productively. Furthermore understanding how the program, i.e. statements of logic, will be run influences the software design, yet clear logical statements can be produced. Passing between good logic and good Prolog was investigated in [24].

The PipeDream (Prototyping Specifications, Design and Requirements At Melbourne) project of the Computer Science Department of the University of Melbourne, aims to improve the outcomes of requirements analysis by using formal methods, or more precisely mathematical modelling, to determine, analyze and verify requirements. Logic programming can provide the basis for a light weight approach to achieving better analysis of specifications. The PipeDream analysis encompasses exploring specifications through animation, and a limited form of theorem proving supported within the logic programming model. The PipeDream approach contrasts with the heavy weight approach of using a general purpose interactive theorem prover, which requires the developer to have detailed knowledge of underlying mathematical theories and proof strategies. While a light weight approach may not give the same levels of assurance as an automated reasoning system, levels of assurance are provided which are adequate for most projects, with significantly less overhead.

Animating specifications is particularly promising for developing formal specifications. Animation can be highly automated and thus cheap to perform, with static analysis of the specification providing important information about the model. Animation can be very effective at detecting problems with the specification because animating a specification provides a means of testing a model and its properties interactively. These two properties of animation make it very suitable for early lifecycle when the model is more likely to be incorrect or incomplete and still evolving.

In [9] the use of animation to perform verification of a simple dependency management system is illustrated.

3.4 $E_{hh}$

Linear Logic [7] enriches the operational interpretation of classical logic in that formulas can be treated as resources.
This idea has been incorporated in recent linear logic extensions of logic programming (e.g. [2, 17]) that have originated the new paradigm of Linear Logic Programming (LLP). Such paradigm has successfully been applied to formalize important programming aspects such as state-based computations, object-orientation, data management and aspects of concurrency. These features make LLP a suitable framework for specifying distributed systems and agent systems in particular. The notion of state in LLP has a natural correspondence with the notion of state and beliefs of an agent; the possibility of using resources during a computation is a natural means for supporting dynamic changes in the behaviour of an agent.

$\mathcal{E}_{hhf}$ [5] is a concrete linear logic programming language, developed at the Department of Computer and Information Science of Genova University (Italy). It is based on a particular subset of Forum [17], a presentation of higher-order linear logic in terms of goal-driven proofs. $\mathcal{E}_{hhf}$ extends the previous proposals with aspects derived by the general purpose logic defined by Forum. $\mathcal{E}_{hhf}$ is a multiset-based logic which combines features peculiar of extensions of logic programming languages like $\lambda$Prolog [18], e.g. goals with implication and universal quantification, with the notion of formulas as resources at the basis of linear logic. Furthermore, $\mathcal{E}_{hhf}$ is defined in a higher-order setting, thus facilitating the development of applications based on meta-programming.

A specification written in $\mathcal{E}_{hhf}$ has a natural mapping into a logic program, and may be easily translated in LP, the automation of this process is still under study.

### 3.5 CaseLP

CaseLP (Complex Application Specification Environment Based on Logic Programming) [14] is a MAS-based framework for prototyping applications involving heterogeneous and distributed entities. It furnishes tools for describing both the architecture of the MAS under development and the behaviour of agents that compose the system. Furthermore, it provides simulation tools for animating the MAS execution.

In CaseLP agents communicate via point-to-point message passing, with messages written in KQML [16]. The types of agents supported by CaseLP are the same outlined in Section 2.1: logical agents, interface agents and hybrid agents. The main components of all these agents are: an updatable set of facts, defining the state of the agent; a fixed set of rules, defining the behaviour of the agent; a mail-box for incoming messages. Interface agents also possess a user-defined interpreter which describes how to interface to the external module to which they are linked.

The structure of the MAS is given using MAS-adl, a simple, customized, architectural description language for MAS. It is used to define classes of agents (type, internal architecture, services they require and/or provide), as well as class instances and links between services provided and required by instances.

Agents’ initial beliefs and behavioural rules are given using the language ACLPL, (Agent Constraint Logic Programming Language). ACLPL enriches the Constraint Logic Programming Language ECLiPSe with with primitives for communication between agents and for agent state update. ACLPL supports asynchronous sending of messages and two types of reception: non-blocking reception of a message through inspection of the mail-box and blocking reception; the agent blocks until a particular message coming from a particular sender enters its mail-box. The primitives $\text{assert}_\text{state}(\text{Fact})$ and $\text{retract}_\text{state}(\text{Fact})$ ensure a semantically clear management of state changes: updates are committed only if they are part of a successful rule. The same happens with $\text{send}$: a message is effectively sent only if the execution of the enclosing rule succeeds.

Execution and visualization of the MAS are performed by the CaseLP Simulator and Visualizer. They provide a GUI for loading, initializing and tracing the agents’ behaviour in a graphical user-friendly manner. Visualization documents events that happen at the agent level during MAS execution. Instrumentation adds probes to agents code; events related to state changes and/or exchanged messages can be recorded and collected for on-line and/or off-line visualization. The CaseLP Simulator is based on a round-robin scheduler that activates in turn all the agent in the MAS. During the simulation, views related to instrumented agents are shown. At the end of the simulation a more complete trace of all the instrumented events can be visualized.

### 3.6 Some ideas for the integration of components

We sketch how the described components can become part of the open framework described in Section 2. As far as the specification languages are concerned, $Z$ will belong to the set of not-executable languages, while $\mathcal{E}_{hhf}$ can be an element of the executable set.

The PipeDream approach represents a way of animating a $Z$ specification. By now the specification is compiled into Mercury, but it would be easy to compile it into any other logical language.

It does not yet exist an automated compiler from $\mathcal{E}_{hhf}$ into a commercial logical language, but there is a working interpreter: referring to steps 3 and 4 of the previously given methodology, we should think to develop a first system specification using $\mathcal{E}_{hhf}$, testing it using the interpreter, refining it, and then, when sure that the $\mathcal{E}_{hhf}$ system specification works, compiling it into the target language.

As target language for the MAS executable prototype we can think of any logical language extended with com-
munication capabilities. Moreover, for realizing a real application instead of a prototype, we can also think about a mapping of the logical implementation into some more widespread languages like C or Java.

Logical agents described in the previous section will be CaseLP logical agents. IMPACT agents can be instead seen as hybrid agents. Also CaseLP hybrid agents can be seen in a similar manner.

Finally, the most suitable components for playing the role of the interface agents are HERMES mediators. As already described, mediators can semantically integrate results coming from heterogeneous data sources. The query language for mediators is logic-based, thus allowing an easier integration in our logic framework. CaseLP interface agents can be used for accessing external software too, even if the definition of the interpreter for this purpose is always left to the user without much support from the system.

3.7 Some sample applications

The authors have had many successful experiences in using Logic Programming as a specification language; some of these simply used Prolog-like languages for defining an executable prototype, while other ones used the more complex systems described above for building, in a simpler way, a logic MAS. We describe some samples of these applications.

3.7.1 HERMES–based applications.

The HERMES system [4] was used in the past to integrate a wide variety of packages including Ingres, Oracle, Object-Store, UM Nonlin (a nonlinear planner), a Terrain Reasoning System, the AVIS Video Information System, a Full Text Indexing System, flat files, GIS data structures (quadtrees), a Face Recognition System, linear programming and integer programming algorithms, to name a few.

3.7.2 FLiPSiDE.

A financial portfolio manager was prototyped by one of the authors and a graduate student using the approach advocated here [21]. A logical layer was rapidly developed using Bim-Prolog which integrated a range of diverse systems including a data filter of stock ticks, a neural network for forecasting, and a rule-based expert systems which controlled the data filter. It was easy to integrate new services and the logic language made clarity of the integration transparent.

3.7.3 CaseLP–based applications.

CaseLP has been adopted for developing applications in various areas: two applications were related to transport and logistic problems; in particular one has been developed in collaboration with FS (the railway Italian company) for solving train scheduling problems in the La Spezia – Milano track, and another one has been developed with Elsag s.p.a., an international company which provides service automation, for planning goods transportation. Another application concerned the retrieval of medical information contained in distributed databases; in this case CaseLP has been successfully adopted for a reverse engineering process. Finally the combination of agent-oriented and constraint logic programming techniques to solve the distributed transaction management problem has been faced in [15]; CaseLP has been used as MAS prototyping environment.

4 Conclusions

In this paper we have presented a proposal for an open MAS framework whose aim is to put in evidence the usefulness of a logic programming based approach in the realization of open, heterogeneous, distributed systems. We have sketched our approach and how a multi-agent system prototype should be developed following the given methodology.

Our intention is to integrate the authors’ different research experiences based on Logic Programming into a common joint project which will lead to the development of the general open framework ARPEGGIO (Agent based Rapid Prototyping Environment Good for Global Information Organization), for the specification, rapid prototyping and engineering of agent-based software.

As previously described the authors’ experiences face different aspects of the realization of complex and distributed applications. The ARPEGGIO framework will take contributions from works on integration of multiple data sources and reasoning systems by means of mediators and/or agents by the Department of Computer Science at the University of Maryland (USA), work on animation of specifications included in the PipeDream project at the Department of Computer Science at the University of Melbourne (Australia) and research on MAS and LP-based software prototyping by the Department of Computer and Information Science at the University of Genova (Italy).

References


