# Combining Logical Agents with Rapid Prototyping for Engineering Distributed Applications

Philip Dart[1] Ed. Kazmierczak[1] Maurizio Martelli[2] Viviana Mascardi[2]
Leon Sterling[1] V.S. Subrahmanian[3] Floriano Zini[2]

(1) Department of Computer Science, University of Melbourne
Parkville 3052, Australia
email: {philip,ed,leon}@cs.mu.oz.au

(2) D.I.S.I. - Università di Genova
via Dodecaneso 35, 16146 Genova, Italy
email: {martelli,mascardi,zini}@disi.unige.it

(3) Department of Computer Science, University of Maryland,
A.V. Williams Building, College Park, MD 20742 USA
email: 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 proposal 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 systems, where a set of entities have to cooperate and coordinate for exchanging information coming from diverse existing sources. Hence, integration and reusing of different kinds of information and software tools is an urgent necessity that new software products have to cope with.

The agent-oriented paradigm [25, 19] is an emerging technology 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.

Our interest in intelligent 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 that an agent is 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 some well-established method. Rigorous processes as formal methods are recognized as essential for correctness, but their key problem is the difficulty to match client's needs quickly and accurately. Further, they usually force the developer to give a (too) detailed system specification, without taking into account modelling and verification of applications integrating existing software modules. Prototyping and animating specifications are recognized as an aid to

requirement clarification and preliminary design, and can be also used for performing integration of existing software. 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 for facilitating the development of correct heterogeneous software. In particular we concentrate on two issues that 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 reusing 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 proposal of 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 for 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 [22]. 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]. Further, 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 into an application.

The paper is structured as follows. The next section gives the main ideas of 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 where existing components will be integrated.

We suppose to have at our disposal the components depicted in Fig. 1. On 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 developer is not necessarily an "expert" of all the specification languages at his/her disposal but chooses the most useful ones with respect to his/her needs. For example, there could be a not executable specification language `A` based on *Petri Nets* 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

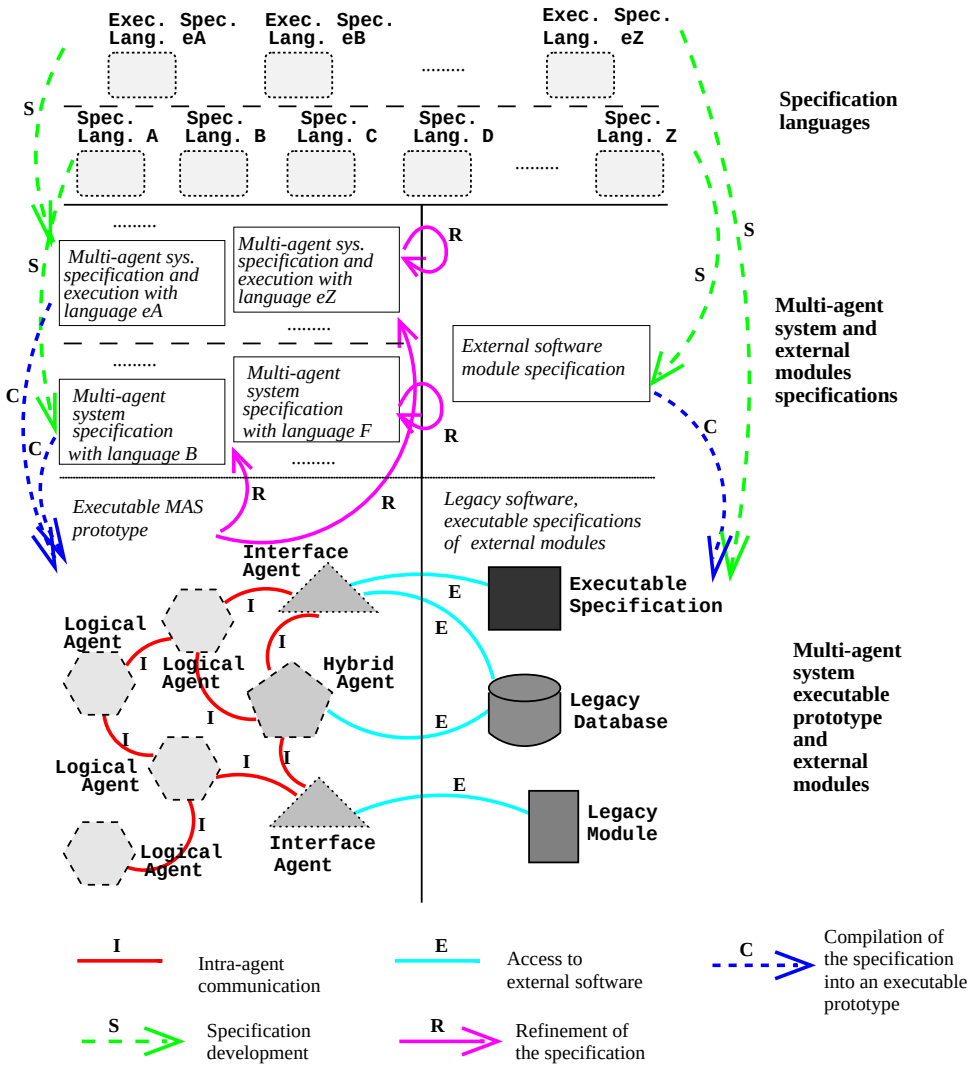


**Fig. 1.** Our approach.

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. For any language we assume a (semi-) automatic compiler which can produce an executable form of any specified agent, in a logical *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 the *Petri Net* specification language it would be quite easy to describe how the agents interact, but it would be boring 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.

When the executable MAS prototype has been created via compilation, its execution animates the given specification, and allows the MAS developer to 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

the same level of confidence in the consistency, correctness and completeness of the model as that given by 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 some executable specification languages are available, it is possible to execute the given specification before animating it. Both execution and animation have the purpose of providing a better understanding of the system, thus facilitating the MAS developer to refining the specification, animating and testing it again, and so on, until the obtained executable prototype behaves exactly as expected. In the picture, the $R$ arrows represent the refinement stage. If the specification language is executable, it is possible to execute it, analyze its behaviour and refine the specification without developing the executable MAS prototype. Either if the language is executable or not, a formal approach can be used to verify properties of the specification, and refinement can be done on the basis of the verification. The looping $R$ arrows describe this refinement cycle. Other than testing or verification of an abstract specification, compilation can be used to produce a more concrete executable MAS prototype. After the MAS prototype has been executed and tested, a refinement of the specification can happen.

The bottom rightmost part of the picture represents legacy software and data that are integrated into the prototype. As well as this kind of module, external specification 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 compiler can produce an animable form of the specification (see the $C$ arrow in the right part of the picture).

Finally, as far as the *executable MAS prototype* is concerned, it will be implemented in a logical *target language*, for example ECLiPSe [6], or Mercury [21], or SICStus Prolog [8] or HERMES [24]. At least two types of agents are necessary for exploiting the complex tasks typical of MAS applications: *logical agents*, which show capabilities of high-level reasoning, carried out through symbolic manipulation and theorem proving, and *interface agents*, which provide an interface towards the external software modules (the access to external software is represented by the $E$ lines in the picture). A third kind of agents, the *hybrid* ones, combine the skills of the two other types of agents. Communication between agents occurs in an agent communication language (communication channels are depicted by the $I$ lines in the picture).

In the following we present an example for better understanding the meaning of the components in Fig. 1. Consider a hypothetical scenario where the complex application being modelled involves a *Finding-job agency* (see Fig. 2), which must cooperate with industries to find a job 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

[1] We distinguish between *agents* and *external modules* since the last are pieces of software which provide a set of functionalities without having communication capability, autonomy, intelligence; they can be inserted into a multi-agent system only if there are agents interfaced to them.

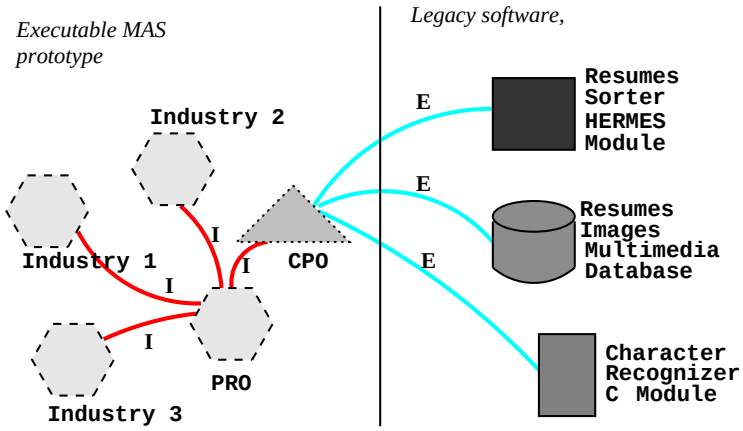


**Fig. 2.** The Finding-job agency example.

into 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 [24], 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*, *C character recognizer* and *Resumés sorter*).

According to industries' requests, or autonomously at 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 *Finding-job 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 steps 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 sets the components, which services they provide and require, which communication channels exist among agents. Further, (s)he chooses the most appropriate architecture of each agent in the system (e.g., only reactive or proactive agents may be

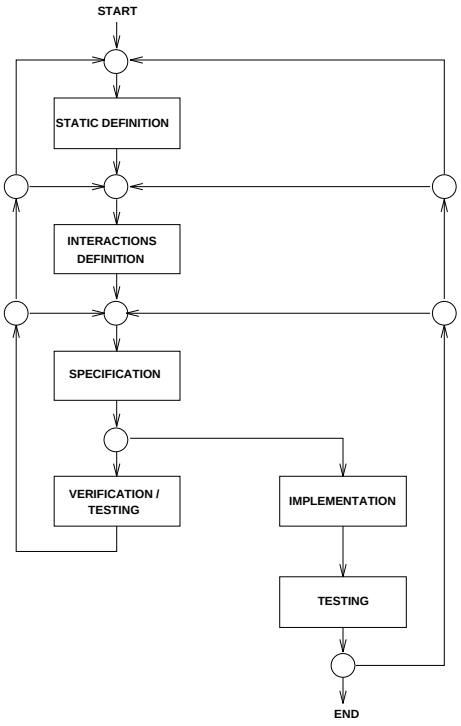


**Fig. 3.** The development method.

needed). 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 specification 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 using a greater degree of granularity and/or different specification languages. For example, at a first stage only the specification 3a above could be given, 3b and 3c could be defined afterwards.
4. **Analysis, verification and testing of the specification**. This phase concerns testing and/or analyzing and/or 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 specification(s) defined in step 3 into a prototype much closer to the final implementation. In particular

interfaces towards external software, data or specifications are provided, as well as support for messages exchange among agents. Either if the specification given in step 3 is executable or not, it is possible to compile it into the executable "low-level" prototype. The choice of a logic programming language as the target language for the prototype realization makes the compilation step easier. Each of the specification 3a, 3b and 3c has to be translated into executable code. 3a corresponds to different implementations of message exchange; 3b is translated into suitable data structures (realizing different architecture parts) and into a meta-program that implements the architecture's flow of control. The architecture-dependent behavioural specification 3c can 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 a "real" communication among agents and actual interaction with legacy programs, data or specifications. Any error or misbehaviour discovered in this step may imply a revision of the choices performed 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*, $\mathcal{D}$, 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, $\mathcal{D}$.

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 $D$, where the reference to $D$ will be eliminated if it is clear from context. 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

$$\texttt{domainname} : \langle \texttt{domainfunction} \rangle (\langle \texttt{arg1}, \ldots, \texttt{argn} \rangle)$$

where `domainfunction` is the name of the function, and $\langle \texttt{arg1}, \ldots, \texttt{argn} \rangle$ are the arguments it takes. Intuitively, a domain call may be read as: in the domain called `domainname`, execute the function `domainfunction` defined therein on the arguments $(\texttt{arg1}, \ldots, \texttt{argn})$. 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 `domainfunction` on the arguments $(\texttt{arg1}, \ldots, \texttt{argn})$.

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

$$\texttt{in}(\texttt{X}, \texttt{domainname} : \langle \texttt{domainfunction} \rangle (\langle \texttt{arg1}, \ldots, \texttt{argn} \rangle))$$

where `in` is a constraint that is satisfied just in case the entity `X` is in the set returned by the domain call in the second argument of `in(-,-)`. In other words, `in` is the polymorphic set membership

predicate. More concretely, `in(A,ingres:select_eq('criminals',"name","john smith"))` is a DCA-atom that is true just in case `A` is a tuple in the result of executing a selection operation (finding tuples where the `name` field is `john smith`) on a relation called `criminals` maintained in a `INGRES` database system.

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

$$A \leftarrow D_1 \,\&\, \ldots \,\&\, D_m \,\&\, A_1, \ldots, A_n.$$

where $A, A_1, \ldots, A_n$ are atoms, and $D_1, \ldots, D_m$ 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 expressed axioms in logic affected its computational properties to prove 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 [23]. Good Prolog developers rapidly learn, however, that it is very rare for people to write down statements of logic that are right first time. They need execution and debugging which can go very quickly and productively. Furthermore understanding how the code will be run, i.e. the software design, influences the code, yet clear logical statements can be produced. Passing between good logic and good Prolog was investigated in [22].

The PipeDream (*PrototypIng sPEcifcations, 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 this. In particular, the PipeDream approach involves exploring specifications through animation. Such an approach can be contrasted 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 does not give the same levels of assurance that an automated reasoning system would, it does give levels of assurance which are adequate for most projects and 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 earlier iterations 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 $\mathcal{E}_{hhf}$

*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.

At the Department of Computer and Information Science of Genova University (Italy) the language $\mathcal{E}_{hhf}$ [5] has been developed. It is a concrete linear logic *programming* language, 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 the behaviour of agents that compose the system by means of a simple rule-based logic language, as well as 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 an user-defined *interpreter* which describes how to interface to the external module.

The language for defining the agent's behaviour is called *ACLPL (Agent Constraint Logic Programming Language)*, and is the Constraint Logic Programming Language ECLiPSe enriched with primitives for updating the agent state (`assert_state(Fact)`, `retract_state(Fact)`) and for communicating (`send(Message, Receiver)`, `async_receive(Message)` and `sync_receive(Message, Sender)`).

ACLPL supports two types of reception: `async_receive(Message)` for non-blocking asynchronous reception of a message through inspection of the mail-box, and `sync_receive(Message, Sender)`, which is the blocking reception primitive; the agent blocks until a *Message* coming from *Sender* enters the mail-box.

As far as updating is concerned, the primitives `assert_state(Fact)` and `retract_state(Fact)` ensure a semantically clear management of state changes: updates are committed only if they are part of a successful computation. The user of the system can however force the commitment of updates belonging to failing computations, if (s)he wants.

When the code of all agents for the application has been written in ACLPL, it is possible to insert them in a context in which other agents are present (`load` primitive), and to start the simulation of the system (`initialize` primitive).

While the simulation is running, the user can follow the exchange of messages between agents, and their state evolution. A more detailed off-line check is possible after the simulation stops.

The CaseLP Visualizer, yet under development, provides a GUI for loading, initializing and tracing the agents' behaviour in a graphical user-friendly manner.

### 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.

For $\mathcal{E}_{hhf}$ it does not yet exist an automated compiler 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 communication 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 sampling 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.

**HERMES–based applications.** The HERMES system [4] was used in the past to integrate a wide variety of packages including Ingres, Oracle, ObjectStore, 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.

**FLiPSiDE.** A financial portfolio manager was prototyped by one of the authors using the approach advocated here [20]. A logical layer was rapidly developed using BimProlog 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.

**CaseLP–based applications.** CaseLP has been adopted for developing applications in very different 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 Bailey, 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


1. S. Adali and V.S. Subrahmanian. Intelligent Caching in Hybrid Knowledge Bases. In N. Mars, editor, *Proc. 1995 Intl. Conf. on Very Large Knowledge Bases*, pages 247–256, Twente, The Netherlands, May 1995. IOS Press.
2. J. M. Andreoli. Logic Programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.
3. M. Bozzano, G. Delzanno, M. Martelli, V. Mascardi, and F. Zini. Multi-Agent Systems Development as a Software Engineering Enterprise. Submitted to PADL'99.
4. A. Brink, S. Marcus, and V.S. Subrahmanian. Heterogeneous Multimedia Reasoning. *IEEE Computer*, 28(9):33–39, September 1995.
5. G. Delzanno. *Logic & Object–Oriented Programming in Linear Logic*. PhD thesis, Università di Pisa, Dipartimento di Informatica, March 1997. Technical report TD 2/97.
6. A. Abderrahamane et al. *ECLiPSe 3.5 User Manual*. European Computer Research Center, Munich, December 1995.
7. J. Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.
8. Programming Systems Group. *SICStus Prolog User's Manual*. Swedish Institute of Computer Science, Kista, Sweden, June 1995.
9. E. Kazmierczak, M. Winikoff, and P. Dart. Verifying Model Oriented Specifications through Animation. To appear in Proc. of APSEC'98.
10. R. Kowalsky and F. Sadri. Towards a Unified Agent Architecture that Combines Rationality with Reactivity. In *Proc. of International Workshop on Logic in Databases*, San Miniato, Italy, 1996. Springer-Verlag.
11. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.
12. J. Lu, A. Nerode, and V.S. Subrahmanian. Hybrid Knowledge Bases. *IEEE Transactions on Knowledge and Data Engineering*, 8(5):773–785, October 1996.
13. S. Marcus and V.S. Subrahmanian. Foundations of Multimedia Database Systems. *Journal of the ACM*, 43(3):474–523, 1996.
14. M. Martelli, V. Mascardi, and F. Zini. Towards Multi-Agent Software Prototyping. In H. S. Nwana and D. T. Ndumu, editors, *Proc. of the 3rd International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM98)*, pages 331–354, London, UK, March 1998.
15. V. Mascardi and E. Merelli. Agent-Oriented and Constraint Technologies for Distributed Transaction Management. Submitted to IIA'99.
16. J. Mayfield, Y. Labrou, and T. Finin. Evaluation of KQML as an Agent Communication Language. In M. Wooldridge, J. P. Müller, and M. Tambe, editors, *Intelligent Agents II*, pages 347–360. Springer-Verlag, 1995. LNAI 1037.
17. D. Miller. Forum: A Multiple-Conclusion Specification Logic. *Theoretical Computer Science*, 165(1):201–232, 1996.
18. G. Nadathur and D. Miller. Higher-Order Horn Clauses. *Journal of the ACM*, 37(4):777–814, 1990.
19. D. T. Ndumu and H. S. Nwana. Research and development challanges for agent–based systems. *IEE Proceedings of Software Engineering*, 144(1):2–10, February 1997.
20. D. Schwartz. Cooperating Heterogeneous System. *Kluwer International Series in Engineering and Computer Science*, 1995.
21. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, October–December 1996.

22. L. Sterling, P. Ciancarini, and T. Turnidge. On the Animation of "not Executable" Specifications by Prolog. *International Journal of Software Engineering and Knowledge Engineering*, 6(1):63–87, 1996.
23. L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, 1994.
24. V.S. Subrahmanian. *Principles of Multimedia Database Systems*. Morgan Kaufman Press, January 1998.
25. M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.