An Agent-Based Prototype for Freight Trains Traffic Management

Alessio Cuppari[(1)] Pier Luigi Guida[(1)]
Maurizio Martelli[(2)] Viviana Mascardi[(2)] Floriano Zini[(2)]

(1) Sistemi Informativi, Divisione Infrastrutture - FS S.p.A
P.zza Croce Rossa 1, 00161 Roma, Italy
cuppari@disi.unige.it
pierluigi.guida@guidafs.inet.it

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

Abstract. The increasing amount of train traffic highlights the necessity of automated tools for decision support, mainly when the availability of tracks is known on a day-by-day basis and no long-term schedules can be made. The paper describes the use of CaseLP, a logic programming based environment for developing multi-agent system prototypes, to face the management of freight trains traffic between the Italian stations of Milano and La Spezia. This real case-study, developed within the framework of the EuROPE-TRIS Project [6], has been chosen for evaluating the benefits of prototyping and testing a decision support system following an agent-based approach. The choice of a logic programming paradigm as the basis for the prototyping environment is motivated and compared with other existing solutions.

1 Introduction

The increasing demand of short-term train schedules by Transport Operators highlights the necessity of automated tools for train traffic decision support. When the number of trains running on a railway line and the availability of tracks are known on a day-by-day basis, decision support systems can help in maximizing the demand granting and optimizing the traffic flow.

The railway line connecting Milano and La Spezia yards is strongly characterized by short-term train scheduling: La Spezia train traffic flow depends on a great number of variables, such as the average number of trains that La Spezia yard will contain, the average train flow from the yard to the port terminals and the number of loading authorizations already granted for a certain day. The possibility of delivering a new train on the Milano - La Spezia line depends both on local congestion indexes of railway sections and on global information about the train traffic situation. Coordination and cooperation between the involved entities is necessary to reach an agreement on granting the authorization to a loading or the permission of delivering a train on the main railway line.

In this complex scenario the adoption of some decision support mechanism can prove useful: the paper describes an agent-based approach for modeling the application, developing a working prototype, testing and refining it against the problem requirements. A real implementation of the prototype can be used as an effective decision support tool.

The environment used to develop the freight train traffic management application is CaseLP [12], a *Complex Application Specification Environment based on Logic Programming*. CaseLP provides the user with tools for specifying, implementing and testing multi-agent system (MAS) prototypes and with a method for facing these three steps.

The structure of the paper is the following: Section 2 introduces our prototyping environment and the method for developing a working prototype. Section 3 introduces our case study and shows how it has been prototyped using CaseLP. Finally, Section 4 presents some final considerations on related works and future directions of our research.

2 CaseLP, a tool for multi-agent systems prototyping

CaseLP is a framework for rapid prototyping of agent-based software applications. It is built upon the logic programming language ECLiPSe [1]. The environment provides both an iterative method for specification, implementation, execution and testing of MAS-based prototypes, and a set of tools that are used for this aim. The method allows the application developer to build the prototype following a sequence of clear steps, refining it against the client's requirements.

In CaseLP there are two kinds of agents, *logical agents* and *interface agents*. The former ones provide control and coordination among MAS components thanks to their complex reasoning capabilities. The latter ones provide an interface between external modules and the agents in the MAS. The final prototype can be built by combining existing software modules and new components. This approach furthers software integration and reusing, two aspects that are highly relevant for the success of new software engineering technologies.

CaseLP agents are characterized by a purely reactive, purely proactive or hybrid architecture. In the same MAS agents of different kinds (logical or interface) and with different architectures may coexist, thus allowing a great flexibility in the prototype definition. However, for the sake of this paper, only purely reactive and proactive agents are considered.

All the agents share some main components: an updatable set of *beliefs* defining the agent's *state*, a *mail-box* for incoming messages and a fixed set of rules that defines the *behavior*. Reactive behavior is given by *event-condition-action* rules, whereas proactive behavior is given by means of *condition-action* rules. Reactive interface agents embed an *interpreter* for accessing external software. Agents communicate via point-to-point asynchronous message passing. Agents' goals are not explicitly represented in their state: what the agents aim at is implicitly coded into the reactive or proactive behavior they are given. This means that agents reason about how they can achieve some hard-wired goal and not

about which goal is more convenient for them in a certain situation. In some sense, they have a constrained, problem-oriented autonomy.

The proposed prototyping method is fully described in [5]. Below, a 5-steps simplified version of it is presented. The final goal is an environment that provides automatic tools to perform each step. Already available tools are described in Section 2.2.

2.1 Prototyping method

The first stage in any prototyping method is the requirements gathering. To begin with, the client may describe the application features and the requirements of the final prototype in natural language, and after this phase a more formal prototyping method can be used. The method can be summarized as follows:

1. **Static description of the application architecture.** In this step the developer defines:
   - the classes of agents that the application needs: their *kind, architecture, interpreters* (for interface agents) and required and provided *services*;
   - the *instances* of each class that will form the MAS;
   - the *interconnections* between agents, that is, how a service provided by an agent is linked to a service required by another agent. Some agents can export services to "entities", either humans or other agencies, that are external to the MAS under construction. Furthermore, agents in the MAS can import external services.
2. **Description of communication between agents.** Each provided or requested service needs a specific *conversation* between the agent that provides the service and the one that requires it. This step allows to specify the sequence of exchanged messages, as well as their content. Some conversation can start some other (sub)conversation. For example, imagine agent a, requested for a service by agent b, that has to require an accessory service to agent c, in order to reply to agent b. Such situations are properly captured defining a relation sc (*sub-conversation*) between conversations. The *conversation model* of the MAS includes the set of all conversations and the relation sc.
3. **Description of initial state and behavior.** This step sets the initial beliefs of the agents in the MAS. Furthermore, each reactive agent is given a behavior expressed by *event-condition-actions* reactive rules. This step takes into consideration the conversation model defined in step 2. For each message (event) received by an agent, the appropriate behavior is given. Each rule tells the agent what to do when a particular message is received and *condition* is satisfied. For proactive agents, behavior is given instead by *condition-action* rules: depending on its internal state, the agent performs the appropriate actions.
4. **Prototype implementation.** In this step the MAS prototype is built: each agent is implemented as an ECLiPSe module. This module is obtained from the specification given in the previous steps. This step allows integration of external software modules.

5. **Execution and testing.** The obtained prototype is tested using the CaseLP Simulator. The developer can choose which events (communication events or state updates) have to be visualized. The CaseLP Visualizer permits both on-line and off-line visualization of the MAS execution.

2.2 Prototyping tools

MAS-adl: a simple architectural description language. In the first step of the method MAS-adl, a simple, customized, *architectural description language* [11] for MAS, is adopted. The syntax for MAS-adl is sketched below:

AgentClassDef ::= **agentclass** AgentClassName {ClassDef}
ClassDef ::= LogicalAgentDef | InterfaceAgentDef
LogicalAgentDef ::= **kind**: logical; **architecture**: reactive | proactive;
RequiredAndProvidedServices
InterfaceAgentDef ::= **kind**: interface; **architecture**: reactive;
interpreters: InterpretersList; RequiredAndProvidedServices
RequiredAndProvidedServices ::= **requires** ServicesList; **provides** ServicesList;
import ServicesList; **export** ServicesList;

AgentInstancesDef ::= **agentInstances** {AgentInstDefList }
AgentInstDef ::= NumberOfInstances AgentName AgentClassName

LinksDef ::= **link** { LinkList}
Link ::= OneToOneLink | OneToManyLink | ManyToOneLink
OneToOneLink ::= AgentNameAndService ← AgentNameAndService
OneToManyLink ::= AgentNameAndServiceList ← AgentNameAndService
ManyToOneLink ::= AgentNameAndService ← AgentNameAndServiceList
AgentNameAndService ::= AgentName.Service | AgentName*.Service

MAS-adl describes three main constructs: AgentClassDef defines the classes of agents used in the MAS under construction; AgentInstancesDef defines the instances of classes previously defined i.e., the agents that constitute the MAS; LinksDef defines the directed links between instances of agents, from a service provided by an agent to a service required by another agent. Obviously, links for exported or imported services are not defined. AgentClassName and AgentName are strings of characters. InterpretersList is a list of interpreters, each for any external domain to which the agent is interfaced. The *interpretation* approach to software integration [9] has been followed. Service is one of the services that have been individuated for the application, as defined in its ontology[1].

Tools for conversation model. The second step of the method describes the conversation model of the MAS. The agent communication language is a subset of KQML[13]. The conversation model is defined by choosing, for each service,

[1] All agents share a single domain ontology. Due to space constraints, in the case study in Section 3, it will be left implicit.

the sequence of messages (conversation), as well as their *performative* and the content of each message. The content language is a set of ECLiPSe terms that are part of the application ontology. A relation sc defines which conversations eventually start during other conversations. Let $c_1 = \{m_1, \ldots, m_k\}$ be a conversation composed by messages $m_1, \ldots, m_k$ and let c_2 be another conversation. $c_1 sc_{m_i} c_2$ denotes that c_2 must start after message m_i in c_1 has been handled by the receiving agent. $c_1 sc?_{m_i} c_2$ denotes that c_2 eventually starts after message m_i has been handled by the receiving agent. In the latter case, the decision about starting c_2 is up to the receiving agent.

ACLPL. The third step of the given method defines agents behavior and their initial state. The code for an agent is structured in three parts. They express respectively the initial set the agent's beliefs, the set of reactive or proactive rules defining its behavior and a set of auxiliary procedures. The language used in this step is ACLPL, whose syntax, limited to agent behaviour, is sketched below.

Behavior ::= **behavior** ReactiveRulesList | ProactiveRulesList **endbehaviour**
ReactiveRulesList ::= ReactiveRule | ReactiveRule; ReactiveRulesList
ReactiveRule ::= **on message** Msg **check** Condition **do** ActionsList
Msg ::= Performative { **content**: Content; **sender**: Sender; **receiver**: Receiver; }
Condition ::= StateCondition **and** AuxiliaryCondition
StateCondition ::= true | Goal
AuxiliaryCondition ::= true | Goal
Goal ::= A | A **and** Goal | A **or** Goal
ProactiveRulesList ::= ProactiveRule | ProactiveRule; ProactiveRulesList
ProactiveRule ::= **check** Condition **do** ActionsList
Action ::= assert_state(Belief) | retract_state(Belief) | send(Msg, Receiver) |
send(Msg, Receiver, high_priority)

The initial state of an agent is a (possibly empty) set of beliefs. A belief is an atomic ground formula. A reactive rule is fired by a message taken from the agent's mail-box, and if the condition is satisfied, corresponding actions are executed. Condition is actually formed by two distinct conditions. The former is about the agent state, and expresses what the agent has to believe in order to execute a sequence of actions. The latter is an auxiliary condition, and is actually a set of calls to auxiliary procedures. If these calls succeed, the auxiliary condition is satisfied and the sequence of actions is then executed. Single actions can be either state updates or messages sending. An auxiliary procedure is either an ECLiPSe clause defined in the agent code, or a built-in ECLiPSe procedure. assert_state and retract_state perform the update of the set of agent beliefs. Their semantics assure that the previous state is restored if the execution of some actions fails. The same happens with send: a message is effectively sent only if the execution of the actions succeeds.

It has to be noted that the same behavior can be re-used for all the agent in a same class, even if they have different initial states. In such a way, it can be defined only once.

Architectural task control. When the prototype is being implemented, in step 4 of the method, each agent is given a particular *task control*. According to its architecture, reactive or proactive, a different meta-interpreter is used as execution engine of the code of the agent. The task control can include several execution policies, for example selection of which fired rules are actually executed.

CaseLP Simulator and Visualizer. Execution and visualization of the MAS are performed in step 5 of the method by the CaseLP Simulator and Visualizer. Visualization provides documentation about events that happen at the agent level during MAS execution. According to the developer needs, the code of the agents is automatically *instrumented*. 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. In order to execute the MAS, the system user initializes some agent mail-boxes, and then simulation starts. 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. Instrumentation is completely independent from execution. This will not influence a possible future change of the execution support.

3 The case-study: freight trains traffic management

This section describes a prototype developed to study the freight trains traffic management along the railway line connecting the Italian stations of Milano and La Spezia. This demonstrator aims to be a base-line for a decision support system for *Train Dispatching* to be supplied to the *Traffic Co-ordinator* in Milano (COIM) and the *Production Deputy* in La Spezia (yard operative manager). The case-study has been developed within the framework of the EuROPE-TRIS Project [6], as a result of the co-operation between the Information Systems Division of Italian Railways and the Computer Science Department of Genova University.

3.1 Problem description

Line model. The railway line connecting Milano and La Spezia is formed by three main sections, named S_1, S_2, S_3. Sections are delimited by nodes, respectively *Voghera, Arquata* and *Genova*, that are accessed by minor lines serving several container terminals (see Figure 1).

Fig. 1. Railway line connecting Milano and La Spezia.

System actors. The following real entities operate in this scenario: $T_{[v,a,g]x}$ is a container terminal operating inside the railway network. It is connected to the node $[Voghera, Arquata, Genova]$ via a minor line. *COIM* is the line traffic manager, responsible for train dispatching on the overall line. *Production Deputy of La Spezia (PD)* manages trains arrival, recovery and departure operations in La Spezia yards. *Section Traffic Co-ordinator (STC_i)* is responsible for train dispatching on line section S_i.

Management of wagons loading authorization (LA) requests. The loading operation of train wagons directed to La Spezia is performed in a terminal, and has to be authorized by *PD* in La Spezia. The requesting terminal sends to *COIM* the LA request; the *COIM* updates the congestion index (number of trains per Km) of the concerned section in the indicated date, assuming the granting of the LA. If the new index is greater than the maximum limit, the COIM tells the terminal the request has been refused; otherwise, the COIM transmits the request to the PD.

PD grants or refuses a LA request based on: average number of trains that La Spezia yard will contain at wagons arrival date; average train flow from the yard to the port terminals; number of LAs already granted for the requested date.

Train dispatching authorization. A terminal asks its *STC* for dispatching a train along the main railway. The generic STC_i knows the congestion index of his section. The decision concerning the running of the train is based on the comparison between such index and a maximum limit value: if this value is not overcome, the train is sent and the congestion index of section S_i is updated. Otherwise, the decision is delegated to *COIM*, that is asked by STC_i. *COIM* knows the congestion indexes of all the line sections; the decision rule is based on the comparison between the average of these indexes and a reference parameter: only if this value is not overcome, the train is sent and the congestion index of section S_i is updated.

Model Assumptions. In order to realize the prototype, some assumptions have been made. Furthermore, some constraints related to actual structure of the railway has been taken into consideration. The system time-frame (day) is divided into four time bands, each elapsing six hours. Train progress between two different nodes is managed at the end of each band. Infrastructures in La Spezia are limited. The prototype considers only one yard, having the capacity (i.e., number of tracks) of all the real yards of La Spezia. Average wagon flow from La Spezia yards to the port terminals is considered, as well as train arrival forecasts in La Spezia. It is assumed that each node has infinite capacity i.e., it can contain an infinite number of trains.

3.2 Prototype realization

An agent based approach certainly suits the realization of a prototype for the problem described above. This description can be seen as an informal modeling of the services that the prototype has to provide, as well as a suggestion about the architectural structure of the MAS. In the following, we show how the CaseLP method has been used for the realization of a working prototype.

Static description of the application architecture. Below the architectural description of the case study, given by using MAS-adl, is depicted.

```
agentclass Terminal {
   kind: logical;
   architecture: reactive;
   requires: loading_auth,
             disp_auth;
   provides: nil;}

agentclass ProdDep {
   kind: logical;
   architecture: reactive;
   requires: nil;
   provides: port_loading_auth;}

agentclass Timer {
   kind: logical;
   architecture: proactive;
   requires: nil;
   provides: system_time;}

agentclass Coim {
   kind: logical;
   architecture: reactive;
   requires: port_loading_auth,
             sent_train_notification;
   provides: global_disp_auth,
             loading_auth;}

agentclass Section {
   kind: logical;
   architecture: reactive;
   requires: global_disp_auth,
             system_time,
             list_of_trains;
   provides: disp_auth,
             list_of_trains,
             sent_train_notification;}

agentInstances {
   2 terminal Terminal;
   1 coim Coim;
   1 pr_deputy ProdDep;
   3 section Section;
   1 timer Timer;}
```

```
link {
terminal*.loading_auth <- coim.loading_auth;
terminal*.disp_auth <- section1.disp_auth;
coim.port_loading_auth <- pr_deputy.port_loading_auth;
coim.sent_train_notification <- section*.sent_train_notification;
section*.global_disp_auth <- coim.global_disp_auth;
section*.system_time <- timer.system_time;
section2.list_of_trains <- section1.list_of_trains
section3.list_of_trains <- section2.list_of_trains;}
```

Five classes and eight instances of agents are defined. Four classes correspond to the four system actors previously described and define logical reactive agents. The fifth class defines logical agents that proactively check the system time and inform section agents when a time band is elapsed, in such a way they can manage train progress between sections. The prototype does non interact with external entities, so export and import fields have been omitted. Services and links specification reflect the informal description of the problem previously illustrated. Furthermore, links define communication channels among agents. All the service names are self explanatory, apart from *sent_train_notification* and *list_of_trains*. The former allows a section agent to inform COIM about the sending of a recovered train at the end of a time band. The latter permits a section agent to inform the following agent about a list of trains entering its section when the time band changes.

Description of communication between agents. A sample of the conversation model of the MAS is shown below. For each previously defined link, a conversation is given (only two of them are presented here). Furthermore, the conversation model defines relation *sc* on conversations.

```
Conversation c1 (terminal*.loading_auth <-- coim.loading_auth)

1) Message: terminal --> coim
   Performative: ask
   Content: la_request(section(S),terminal(Term),train(Tr),date(D))
2) Message: coim --> terminal
   Performative: reply
   Content: request(la(section(S),terminal(Term),train(Tr),date(D)), Answer)

Conversation c2 (coim.port_loading_auth <-- pr_deputy.port_loading_auth)

1) Message: coim --> pr_deputy
   Performative: ask
   Content: la_request(section(S),terminal(Term),train(Tr),date(D))
2) Message: pr_deputy --> coim
   Performative: reply
   Content: request(la(section(S),terminal(Term),train(Tr), date(Date)), Answer)

Relation sc: c1 sc?_m1 c2
```

Agent initial state and behavior Finally, part of the ACLPL code for COIM behavior is presented below. Some abbreviations are used for messages, the initial state is not shown. The code for the other agents in the system has a similar form.

```
on message ask(content(la_request(...)),sender(Terminal),)
check
   congest_limit(Section, Date, Limit)                    and
   new_congestion_index_la(Section, Date, Index)          and
   Index > Limit
do send(reply(content(request(la(...)),refused)),Terminal)

on message ask(content(la_request(...)),sender(Terminal),)
check
   congest_limit(Section, Date, Limit)                    and
   new_congestion_index_la(Section, Date, Index)          and
   Index <=  Limit
do send(ask(content(la_request(...))),pr_deputy,high_priority)

on message ask(content(send_the_train(...)),sender(Section))
check
   medium_limit(Medium_limit)                             and
   congest_index(Section, Date, Hour, Index)              and
   count_medium_index(Section, Date, Hour, Index, Medium) and
   (Medium >= Medium_limit and Response == refused        or
    Medium < Medium_limit and Response == granted)
do send(reply(content(permission(send_the_train(...),Response))),Section)
```

Execution and Testing. The final result of this project will be a decision support system. Thus the test of the prototype has been carried out running the system with various initial configurations and inputs. The resulting behavior was satisfactory. However it is too lengthy, here, to describe and give details of these runs. To give an idea of the used visualization tool Figure 2 is shown. It presents off-line visualization of the execution of the MAS defined in the previous steps. It shows how concurrent LA have been managed by the MAS. For each instrumented agent, both exchanged messages and state updates are depicted.

4 Discussion and future work

The tools and the methodology constituting CaseLP represent an *agent-based approach to software prototyping.*

The potential of such an approach has been demonstrated by the adoption of a MAS-based prototyping technology in the freight traffic management field. The developed prototype has been successfully presented to Project Officers in January, 21th, 1999, during the official Project Demonstration Phase [6]. During the next stages of the Project it will be integrated with other tools developed in the above mentioned project to give birth to an overall prototype.

Fig. 2. Case study: off-line trace of execution.

The *prototyping* approach represents a software engineering paradigm more flexible than the classical *waterfall model*. As it is well-known, the *waterfall model* lacks of support for *requirements refining* during the product development. Instead, the prototype serves as a mechanism for identifying software requirements and its life-cycle leaves room for an iterative tuning of the initial choices. A software prototype is generally a simplified and/or not very efficient version of an end product which can be realized using more formal, even if less efficient, technologies. A logic-based formalisms has been adopted to develop a working prototype of a complex application modeled as a multi-agent system. Obviously this choice is not the only possible one, but it has some advantages over the adoption of *structured methods* and *object-oriented methods*. In fact, as pointed out in [8], *structured methods* are either data-oriented or action-oriented and so they cannot capture the complexity of an agent-based system, where data (knowledge) and behavior (action) are strictly tied. *Object-oriented methods*, though offering important advantages as the differentiation between internal and external view of an agent-based system, do not address issues of autonomy, reactivity and proactiveness, and they fail to address interaction on a higher level.

The use of *formal languages*, in particular *logic-based formal languages*, seems to lead to more encouraging results. Many existing approaches for developing agent-based systems are based on logics.
Temporal Logic has proven useful for specifying agents which, on the basis of the past, *do* the future. Concurrent METATEM is an example of this approach [7].
Deontic Logic [14] fits the needs of representing the actions an agent may, may not, or must perform according to some conditions. This allows a quick and high-level description of the agents' state and behavior.
Linear Logic, as discussed in [5, 4], has connectives to express *concurrency* and *synchronization* primitives, which are fundamental in a MAS setting.
Even if it is often necessary to extend the logical frameworks to cope with all the MAS features, the benefits deriving from the use of formal languages, with

clear semantics and easy testing mechanisms, make these extensions worthwhile. Moreover, since interpreters exist for most of these languages, or at least for subsets of them, logic-based descriptions of MAS can be seen as *executable specifications*, and executable specifications are prototypes.

The paper focused on the adoption of *Logic Programming* (LP) [2] for developing MAS prototypes. Besides the already observed features common to all the logic-based frameworks, other characteristics make LP suitable for our purposes.

MAS execution: the evolution of a MAS consists of non deterministic succession of events; from an abstract point of view an LP language is a non deterministic language in which computation occurs via a search process.

Meta-reasoning capabilities: agents need to dynamically modify their behavior so as to adapt it to changes in the environment. Thus, the possibility given by LP of viewing programs as data is very important in this setting. This feature is useful also for integrating external heterogeneous software following some "wrapping" or "interpretation" approach [9].

Rationality and reactiveness of agents: the *declarative* and the *operational* interpretation of logic programs are strictly related to the main characteristics of agents, i.e., *rationality* and *reactiveness*. A *pure* logic program can be viewed as the specification of the rational component of an agent and the operational view of logic programs can be used to model the reactive behavior of an agent. The adoption of LP for combining reactivity and rationality is described in [10].

Since, as already remarked, traditional LP languages do not fulfill all requirements arising during the MAS development, ECLiPSe has been extended to tackle these issues. Classical concepts coming from the distributed software engineering field [11], such as the ability of developing communicating modular software components and combining them in a structure by means of an *Architectural Description Language*, have been taken into account.

The next extension of CaseLP will be the realization of an automatic ACLPL-ECLiPSe translator. It would allow the prototype developer to give an architectural description of the MAS, without worrying about implementative details. In fact the final aim is to provide the prototype developer with a set of high-level specification languages for describing agents and systems in a friendly fashion. A semi-automatic translation mechanism from the linear logic language $\mathcal{E}_{hhf}$ to ECLiPSe already exists, as described in [5], and it is planned to add Z to the specification languages available in the environment. Z should help in defining complex data structures which cannot be represented in $\mathcal{E}_{hhf}$ where only terms can be manipulated. For the implementation of a Z–ECLiPSe translator the experience of the Pipedream project [15] will be fundamental. In this project a Z specification is compiled into the logic-based language Mercury for animation purposes. Another important issue to consider is the integration of external software: by now it is possible to integrate C, Tcl/Tk and the ECLiPSe Data and Knowledge Base but it is planned to support other programming languages. Moreover it is under study how to extend the interface agents' capabilities to make them real *mediator agents*, in the spirit of HERMES [16] and IMPACT [3] approaches.

References

1. A. Abderrahamane, D. Chan, P. Dufresne, E. Falvey, H. Grant, A. Herold, G. Macartney, M. Meier, D. Miller, S. Mudambi, B. Perez, E. van Rossum, J. Schimpf, P. A. Tsahageas, and D. H. de Villeneuve. *ECLiPSe 3.5 User Manual*. European Computer Research Center, Munich, 1995.
2. K. R. Apt. *Introduction to Logic Programming*, volume B of *Handbook of Theoretical Computer Science*. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.
3. K. Arisha, S. Kraus, P. Ozcan, R. Ross, and V.S. Subrahmanian. IMPACT: The Interactive Maryland Platform for Agents Collaborating Together. Technical report, Department of Computer Science, University of Maryland, MD, 1997.
4. M. Bozzano, G. Delzanno, M. Martelli, V. Mascardi, and F. Zini. Logic Programming & Multi-Agent Systems: a Synergic Combination for Applications and Semantics. In *The Logic Programming Paradigm: a 25-Year Perspective*. Springer Verlag, 1999.
5. M. Bozzano, G. Delzanno, M. Martelli, V. Mascardi, and F. Zini. Multi-Agent Systems Development as a Software Engineering Enterprise. In G. Gupta, editor, *Proc. of First International Workshop on Practical Aspects of Declarative Langueges*, Texas, 1999. Springer Verlag.
6. P. L. Guida (Project Coordinator). EuROPE–TRIS Project — 4th European Union RDT Program Telematics Applications Programme. 1998. Transport (Rail) Contract DGXIII - TR 1022 (TR), Specifications and System Design, Deliverable D04.2.
7. M. Fisher. Concurrent METATEM – A Language for Modeling Reactive Systems. In *PARLE'93 Proceedings of Parallel Architectures and languages, Europe*. Springer Verlag, 1993. Lecture Notes in Computer Science.
8. M. Fisher, J. Mueller, M. Schroeder, G. Staniford, and G. Wagner. Methodological Foundations for Agent-Based Systems. *The Knowledge Engineering Review*, 12(3), 1997.
9. M. R. Genesereth and S. P. Ketchpel. Software agents. *Communications of the ACM*, 37(7):49–53, 1994.
10. R. Kowalski 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. Kramer. Distributed Software Engineering. In B. Fadini, editor, *Proc. of the 16th International Conference on Software Engineeri ng*, pages 253–266, Sorrento, Italy, 1994. IEEE Computer Society Press.
12. M. Martelli, V. Mascardi, and F. Zini. Towards Multi-Agent Software Prototyping. In *Proc. of PAAM'98*, London, UK, 1998.
13. J. Mayfield, Y. Labrou, and T. Finin. Evaluation of KQML as an Agent Communication Language. In *Intelligent Agents II*. Springer Verlag, 1995. Lecture Notes in Artificial Intelligence 1037.
14. J. J. C. Meyer and R. Wieringa (Eds.). *Deontic Logic in Computer Science*. Chichester et al. Wiley & Sons, 1993.
15. 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.
16. V.S. Subrahmanian, S. Adali, A. Brink, R. Emery, J.J. Lu, A. Rajput, T.J. Rogers, R. Ross, and C. Ward. HERMES: Heterogeneous Reasoning and Mediator System, 1995.