A Logic-Based Infrastructure for Reconfiguring Applications [⋆]

Marco Castaldi Stefania Costantini Stefano Gentile Arianna Tocchio

Università degli Studi di L'Aquila
Dipartimento di Informatica
Via Vetoio, Loc. Coppito, I-67010 L'Aquila - Italy
{castaldi, stefcost, gentile, tocchio}@di.univaq.it

Abstract. This paper proposes the DALI Multiagent System, which is a logic programming environment for developing agent-based applications, as a tool for component-based software management based on coordination. In particular we show the usefulness of the integration between DALI and the agent-based Lira system, which is a Light-weight Infrastructure for Reconfiguring Applications. We argue that using intelligent agents for managing component-based software systems makes it possible to: (i) perform monitoring and supervision upon complex properties of a system, such as for instance performance; (ii) perform global reconfigurations dynamically through the cooperation of intelligent agents.

1 Introduction

After a long predominance of imperative and object oriented languages in the software development process, declarative languages, thanks to new efficient implementations, have recently regained attention as an attractive programming paradigm for the development of complex applications, in particular related to the Internet, or more generally to distributed application contexts. Declarative methods exhibit important well known advantages: (i) the reduction, also in terms of "lines of code", of the effort required for solving a problem, (ii) the actual reduction of errors introduced in the application, (iii) the fast prototyping of complex applications, that reduces "time to market" and development costs of business applications. In many cases, these applications are better implemented by using agents technology: declarative languages make the implementation of intelligent agents easier and effective. In our opinion, agent-based distributed applications give really a chance to Artificial Intelligence to show its usefulness in practical contexts.

In this paper we show how DALI, a new logic-based declarative language for agents and multi-agent systems, supports the development of innovative agent-based applications. We consider the topic of distributed component management in the context of

[⋆] We acknowledge the support of MIUR 40% project *Aggregate- and number-reasoning for computing: from decision algorithms to constraint programming with multisets, sets, and maps* and by the *Information Society Technologies programme of the European Commission, Future and Emerging Technologies* under the IST-2001-37004 WASP project.

Large Scale Distributed Component Based Applications (LSDCBA). The role of agents is that of monitoring and reconfiguring the system, in order to dynamically maintain some critical non-functional properties such as performance, high availability or security. The possibility of keeping a complex system under control while running is often a key factor for the success of complex and expensive systems. Several so-called "infrastructures" are being proposed to this purpose. In particular, as an interesting example we consider Lira, an agent-based infrastructure created for dynamic and automatic reconfigurations. As a case study, we have integrated Lira and DALI to create agents able to manage heterogeneous software components.

The DALI language [8] [9] is a Prolog-like logic programming language, equipped with reactive and proactive capabilities. The definition of DALI formalizes in a declarative way different basic patterns for reactivity, proactivity, internal "thinking", and "memory". The language introduces different classes of events: external, internal, present and past. Like Prolog, DALI can be useful and effective for rapid prototyping of light applications.

Lira [5] [11] [7] [6] has been recently defined (and fully implemented as a prototypical version in Java) to perform application reconfiguration in critical domains, such as for instance mobile and wireless systems and networks, risk management, e-government, environment supervision and monitoring. Each application/component of a complex system is managed by an attached agent, and there is a hierarchy of agents performing different tasks. Reconfigurations are dynamic and automatic: they are performed while the application is running and as a reaction to some specified events. However, Lira only allows the agents to execute the orders of a Manager, without any kind of internal reasoning, preventing *a priori* any autonomous decision. Moreover, the hierarchical structure of Lira makes the coordination and cooperation among the agents very difficult to implement, thus reducing the applicability in many real contexts.

The problems that we have found when trying to use Java for implementing agents in the context of a critical system (ensuring security of a bank application) suggested us the idea of using DALI instead of Java. In fact, the features of DALI are suitable for enhancing Lira agents by implementing a form of intelligence in the agents. Thus enhanced, Lira may perform reconfigurations in a more flexible and adaptable fashion: not only under predefined conditions, but also proactively, in order to reach some kind of objective, for instance to ensure some properties of the managed system. To this aim, DALI agents managing different components can communicate and cooperate. The Intelligent agents (IAs), created by integrating DALI and Lira, are able to learn, interact and cooperate in order to: (i) perform monitoring and supervision upon complex properties of a system, such as performance; (ii) perform global reconfigurations through the cooperation of intelligent agents in order to fulfill the required properties.

We argue that Lira/DALI agents are light, easy to write, and independent of any specific agent architecture. We support our argument by presenting as a Case Study a practical experience of use of the enhanced infrastructure. In particular, we show how to implement in Lira+DALI the remote management of web sites in a web hosting provider that runs on a Windows 2000 Server. By using the features provided by Lira/DALI agents, the web sites management becomes easier, and it is possible to perform supervision and automatic reconfiguration in order to optimize bandwidth and space usage. The Case Study demonstrates how a declarative language like DALI has a significant impact in the developing process of complex applications in the practical and real context of LSDCBA. The agents created by the integration of DALI and Lira constitute in our opinion a significant advance in terms of supported functionalities, readability, modifiability and extensibility.

The paper is organized as follows: we start by describing the features of both DALI and Lira, respectively in Sections 3 and 4. In Section 4.5 we elicit some problems found when using Lira infrastructure. Then, in Section 5 we discuss the motivations of using DALI, an then we introduce the general architecture of the Lira/DALI agents. Section 6 describes the Case Study. Finally, we summarize the results in Section 8.

2 Motivations

Many approaches of dynamic reconfiguration for component based applications have been proposed in the last years

A dynamic reconfiguration is defined as any change in the component configuration or application topology performed while the system is running [**?**]. Dynamic reconfiguration comes in many forms, but two extreme approaches can be identified: *internal* and *external* [**?**].

Internal reconfiguration relies on the programmer to build into a component the facilities for its reconfiguration. For example, a component might observe its own performance and switch from one algorithm or data structure to another when some performance threshold has been crossed. This form of reconfiguration is sometimes called “programmed” or “self-healing” reconfiguration [**?**,**?**].

External reconfiguration, by contrast, relies on some entity external to the component to determine when and how the component is reconfigured. For example, an external entity might monitor the performance of a component and perform a wholesale replacement of the component when a performance threshold has been crossed.

An approach of self-adaptation which uses internal reconfiguration presents different problems: firstly, the reconfiguration policies are programmed within the components, then they cannot be modified or extended without changing the component itself, reducing the reusability of such policies [17]. Secondly, in the presence of component based applications with many heterogeneous components, each component likely has a different reconfiguration/adaptation policy, sometimes incompatible or conflicting with the policies of other components.

It seems clear that an approach of self-adaptation where an external infrastructure provides reconfiguration features to the application presents many points of interest to provide non functional properties such as performance, dependability or fault tolerance.

3 DALI Multiagent System

DALI [8] [9] is an Active Logic Programming language, designed for executable specification of logical agents. DALI allows the programmer to define one or more agents, interacting either among themselves, or with an external environment, or with a user. A DALI agent is a logic program containing special rules and classes of events (represented by special atoms) which guarantee the *reactive* and *proactive* behavior of the agent. The kinds of events are: external, internal, present, past.

Proactivity makes an agent able to initiate a behavior according to its own internal reasoning, and not only as a reaction to some external event. Reactivity determines actions that the agent will perform when some kind of event happens. Actions can be messages to other agents and/or interaction with the environment.

An agent is able to manipulate its knowledge base, to have temporary memory, to perceive an environment and consequently to make actions. Moreover, the system provides a treatment of time: the events are kept or "forgotten" according to suitable conditions.

DALI provides a complete run-time support for development of Multiagent Systems. A DALI Multiagent System is composed by communicating environments, and each environment is composed by one server and more agents. Each agent is defined by a *.pl* file, containing the agent's code written in DALI.

The new approach proposed by DALI is compared to other existing logic programming languages and agent architectures such as ConGolog, 3APL, IMPACT, METATEM, BDI in [9]. However, it is useful to remark that DALI is a logic programming language for defining agents and multi-agent systems, and does not commit to any agent architecture. Differently from other significant approaches like, e.g., DESIRE [10], DALI agents do not have pre-defined submodules. Thus, different possible functionalities (problem-solving, cooperation, negotiation, etc.) and their interactions are specific to the particular application. DALI is in fact an "agent-oriented" general-purpose language that provides, as discussed below, a number of primitive mechanisms for supporting this paradigm, all of them within a precise logical semantics.

The declarative semantics of DALI is an *evolutionary semantics*, where the meaning of a given DALI program P is defined in terms of a modified program P_s, where reactive and proactive rules are reinterpreted in terms of standard Horn Clauses. The agent reception of an event is formalized as a program transformation step. The evolutionary semantics consists of a sequence of logic programs, resulting from these subsequent transformations, together with the sequence of the Least Herbrand Model of these programs. Therefore, this makes it possible to reason about the "state"of an agent, without introducing explicitly such a notion, and to reason about the conclusions reached and the actions performed at a certain stage. Procedurally, the interpreter simulates the program transformation steps, and applies an extended resolution which is correct with respect to the Least Herbrand Model of the program at each stage.

DALI is fully implemented in Sicstus Prolog [14]. The implementation, together with a set of examples, is available at the URL http://gentile.dm.univaq.it/dali/dali.htm.

3.1 Events Classes

In DALI, events are represented as special atoms, called *events atoms*. The corresponding predicates are indicated by a particular prefix.

– **External Events.** When something happens in the "external world" in which the agent is situated, and the agent can perceive it, this is an *external event*. If an agent receives an external event, it can decide to react to it. In order to define rules that specify the reaction, the external event is syntactically indicated by the prefix *eve*. For instance, $eve(alarm\_clock\_rings)$ represents an external event to which the agent is able to respond. When the event happens, the corresponding atom becomes true and, if in the DALI logical program that defines the agent there is a rule with this atom in the head, then the reaction defined in the body of the rule is triggered. The external events are recorded, in the arrival order, in a list called **EV** and are consumed whenever the correspondent reactive rule is activated (i.e., upon reaction the event is removed from **EV**).
  In the implementation, events are time-stamped, and the order in which they are "consumed" corresponds to the arrival order. The time-stamp can be useful for introducing into the language some (limited) possibility of reasoning about time. The head of a reactive rule can contain several events: in order to trigger reaction, they must all happen within an amount of time that can be set by a directive.
  Attached to each external event there is also the indication of the agent that has originated the event For events like $rainsE$ there will be the default indication $environment$. Then, an event atom can be more precisely seen as a triple:
  $$Sender : Event\_Atom : Timestamp$$
  The $Sender$ and $Timestamp$ fields can be omitted whenever not needed.
– **Internal Events.** The internal events define a kind of "individuality" of a DALI agent, making it independent of the environment, of the user and of the other agents, and allowing it to manipulate and revise its knowledge. An internal event is indicated by the prefix *evi*. For instance, $evi(food\_is\_finished)$ is a conclusion that the prefix evi interprets as an internal event, to which the agent may react, for instance by going to buy food. Internal events are attempted with some frequency (customizable by means of directives in an initialization file). Whenever one of them becomes true, it is inserted in a set **IV**. Similarly to external events, internal events are extracted from this set to trigger reaction. In more detail, the mechanism is the following: if goal G has been indicated to the interpreter as an internal event by means of a suitable directive, from time to time the agent attempts the goal (at the given frequency). If the goal succeeds, it is interpreted as an event, thus determining the corresponding reaction. I.e., internal events are events that do not come from the environment. Rather, they are goals defined in some other part of the program.
  There is a default frequency for attempting goals corresponding to internal events, that can be customized by the user when the agent is activated. Also, priorities among different internal events that could be attempted at the same time can be specified. At present, this frequency cannot be dynamically changed by the agent itself, but a future direction is that of providing this possibility, so as the agent will be able to adapt to changing situations.
– **Present Events.** When an agent perceives an event from the "external world", it doesn't necessarily react immediately: it has the possibility of reasoning about the event, before (or instead of) triggering reaction. Reasoning also allows a *proactive* behavior. In this situation, the event is called *present event* and is indicated by the

prefix *en*. For instance, $en(alarm\_clock\_ring)$, represents a present event to which the agent has not reacted yet.

– **Past Events.** Past events represent the agent's "memory", that makes it capable to perform its future activities while having experience of previous events, and of its own previous conclusions. A past event is indicated by the prefix *evp*. For instance, $evp(alarm\_clock\_ring)$ is an event to which the agent has reacted and which remains in the agent's memory. Memory of course is not unlimited, neither conceptually nor practically: it is possible to set, for each event, for how long it has to be kept in memory. The agent has the possibility to keep events in memory either forever or for some time or until something happens, based on directives. In fact, an agent cannot keep track of *every* event and action for an unlimited period of time. Moreover, sometimes subsequent events/actions can make former ones no more valid.
  In the implementation, past events are kept for a certain default amount of time, that can be modified by the user through a suitable directive. The user can also express a condition of the form:

  $keep$ *evp(A)* $until$ *HH:MM*.

  The past event will be removed at the specified time. Alternatively, one can specify the terminating condition. As soon as the condition is fulfilled (i.e. the corresponding goal is proved) the event is removed.

  $keep$ *evp(A)* $until$ *Cond*.

  In particular cases, an event should never be dropped from the knowledge base, like in the example below:

  $keep$ *evp(born(daniele)) : 27/Aug/1993* $forever$.

Implicitly, if a second version of the same past event arrives, with a more recent timestamp, the "older" event is overridden, unless this violates a directive.

3.2 Actions: Reactivity in DALI

Actions are the agent's way of affecting its environment, possibly in reaction to an external or internal event. In DALI, actions can have or not *preconditions*: in the former case, the actions are defined by *actions rules*, in the latter they are just action atoms. In **actions rules**, success of preconditions determine the execution of the action: only when all preconditions are verified, then the corresponding action is performed. These preconditions are indicated by the prefix *cd*. In the case of **action atoms**, the actions always succeed. An action is indicated by prefix *a*. An example:

```
         eve(saturday) :- a(go_to_the_supermarket).
            fridge_full :- evp(go_to_the_supermarket).
       evi(fridge_full) :- a(prepare_a_snack).
eve(child(I), we_are_hungry)) :- assert(children_are_hungry).
  cd (prepare_a_snack) :- children_are_hungry..
```

When the external event *eve(saturday)* occurs, the agent reacts by performing the action $go\_to\_the\_supermarket$. Since the reaction is recorded as a past event (indicated by $evp(go\_to\_the\_supermarket)$), the recollection triggers the proactive rule and allows the internal event ***evi(fridge_full)***. The action $a(prepare\_a\_snack)$ is executed if the precondition $cd(children\_are\_hungry)$ is true. This is conditioned by the external event $eve(we\_are\_hungry)$ coming from agent $child(I)$. As soon as it is observed, DALI executes the subgoal in the body of the rule, that consists in a predefined predicate (namely **assert**) that records the event.

Similarly to events, actions are recorded as *past actions*, with prefix pa. The following example illustrates how to exploit past actions. In particular, the action of opening (resp. closing) a door can be performed only if the door is closed (resp. open). The window is closed if the agent remembers to have closed it previously. The window is open if the agent remembers to have opened it previously.

```
a(open_the_door) :- door_is_closed.
  door_is_closed :- pa(close_the_door).
a(close_the_door) :- door_is_open.
    door_is_open :- pa(open_the_door).
```

External events and actions are used also for expressing communication acts. An external event can be a message from another agent, and, symmetrically, an action can consist in sending a message. Presently we do not commit to any particular agent communication language, that we consider as a customizable choice that can be changed according to the application domain.

4 Lira

In the context of Large Scale Distributed Systems we usually deal with: (i) thousands of components that are part of one or more Applications; (ii) single Applications that are part of bigger systems, distributed over a wide area network. A basic objective of remote control is that of making the Java managed system flexible, highly modifiable at run time and stable with respect to many different faults. To these aims, remote (re)configuration should be dynamic: i.e., should be performed while a system is running, possibly as an automatic reaction when some event happens.

Lira (Light-weight Infrastructure for Reconfiguring Applications) [5] [11] [7] [6] is a system that performs remote control and dynamic reconfigurations [13] [3] over single components or applications. It uses and extends the approach of Network Management [12] architectures and protocols, where an agent controls directly the managed device and a Manager orders the reconfigurations. The decision maker could be an Administration Workbench with a graphical interface, or, in a more interesting case, a program that has the necessary knowledge to decide, when a specified precondition is verified, what kind of reconfigurations must be performed.

With component reconfiguration we mean any allowed change in the component's parameters (*component re-parametrization*): the addressed components are usually black-boxes, so Lira is able to dynamically change the values of the provided parameters. An Application reconfiguration [2] can be: (i) any change of the Application in terms of number and location of components; (ii) any kind of architectural modification [16].

Lira has been designed *light-weight* [5], given that components can be very small in size and might be run on limited-resource devices such as mobile phones or PDA. It provides a general interface for interacting with components and applications: this interface is realized using a specified architecture and a very simple protocol that allows one *to set and get variable values* and *to call functions*.

Lira has been created to provide the minimal amount of functionalities necessary to perform components reconfiguration and deployment. There are many others approaches of components reconfiguration, based on heavy weight infrastructures that manage also application dependencies and consistence. A complete description of the existing infrastructures with respect to the Lira approach is provided in [5].

The Lira architecture specifies three main actors: the **Reconfiguration Agent**, which performs the reconfiguration; the **MIB**, which is a list of variables and functions that an agent exports in order to reconfigure the component; the **Management Protocol**, that allows agents to communicate.

There are different kinds of agent, depending of their functionalities: the **Component Agent** is associated to the reconfigurable component; the **Host Agent** manages installation and activation of components and agents on the deployment host; the **Application Agent** is a higher-level agent able to monitor and reconfigure a set of components or a subsystem (for details see [6]); finally the **Manager** is the particular agent providing the interface with the decision maker, having the role to order reconfigurations to other agents.

In the next subsections we will describe the Lira features relevant for the proposed integration.

4.1 Component Agent

The Component Agent (CompAgent) is the most important part of the Lira infrastructure: it directly controls and manages the component. To keep the system general, Lira does not specify how the component is attached to the agent, but it only assumes that the agent is able to act on the component. The CompAgent is composed by a generic part (called *Protocol Manager*) which manages the agent communication, and by a local part (called *Local Agent*) which is the actual interface between the agent and the component. This interface is component-specific and it implements the following functions for the component's life-cycle management:

– `void start(compParams)`: starts the component.
– `void stop()`: stops the component.
– `void suspend()`: suspends the component.
– `void resume()`: resumes the component.
– `void shutdown()`: stops the component and kills the agent.

Moreover, each CompAgent exports the variables:

– `STATUS`: maintains the current status of the component. It can assume one of the following values: **starting, started, stopping, stopped, suspending, suspended, resuming**.
– `NOTIFYTO`: contains the address of the agent that has to be notified when a specified event happens.

All the variables exported for the specific component must be declared in the MIB (Section 3.3).

A very important property of a Lira-based reconfiguration system is the *composability* of the agents: they may be composed in a hierarchical way [15], thus creating a higher level agent which performs reconfigurations at application level, by using variables and functions exported by lower level agents. The Application Agent is a Manager for the agents in the controlled components, but it is a reconfigurations actuator for the global (if present) Manager.

4.2 Manager

The Manager orders reconfigurations on the controlled components through the associated CompAgents. The top-level manager of the hierarchy constitutes the Lira interface with the Decision Maker. It exports the `NOTIFYTO` variable, like every other agent. The Manager is allowed to send Lira messages, but may also receive SET, GET, CALL messages: it means that different Managers can communicate with each other.

4.3 MIB

This description represents the agreement among agents that allows them to communicate in a consistent way.

The description provides the list of variables and functions exported by the agent. In particular, the MIB contains the variables and functions always exported by the agent, such as the STATUS or the start() ones, as well as variables and functions specific for the managed components, that are component dependent.

Finally, the MIB specifies constraints to bind declared variables and performed actions to obtain the specified behavior [7].

4.4 Management Protocol

The management protocol has been designed to be as simple as possible, in order to keep the system *light*. Based on TCP/IP, it specifies seven messages, of which six are synchronous, namely:

– SET(*variable_name, variable_value*) / ACK(*message_text*)
– GET(*variable_name*) / REPLY(*variable_name, variable_value*)
– CALL(*function_name, parameters_list*) / RETURN(*return_value*)

and one is asynchronous, namely:

– NOTIFY(*variable_name, variable_value, agent_name*)

4.5 Some problems with using Lira

The current Lira version specifies a very clean, powerful and effective architecture. The Java prototype works well in the test examples proposed in [4] [5] [11] [6]. Nevertheless, there are still problems to solve, related to both the specification and the implementation.

From the implementation point of view, an object-oriented language such as Java allows one to easily create every kind of Lira agent by inheritance from specified classes, thus encouraging agent's reuse. Also, it provides a direct interface with the managed component. However, it is not so immediate to implement in Java mechanisms to provide agents with some kind of intelligence, such as "internal thinking" or "memory". This is demonstrated by the fact that Java-based frameworks for agent development like JADE [1] have built-in reactive capabilities, but do not directly provide proactivity.

Moreover, the hierarchical structure of Lira inherited by the network management architecture model is useful and powerful but very strict. In fact, a hierarchical management is effective for rigidly structured domains, while it makes agents implementation very hard when coordination and cooperation is needed. In this way, the applicability of Lira is reduced.

5 The integration

In this research, we have tried to overcome Lira problems by implementing a part of Lira agents using DALI.

There are several motivations to propose DALI as a formalism for the implementation of intelligent reconfiguration Lira agents. Firstly, DALI's proactive capabilities allow the agents to timely supervise component -and application- behavior, by using *Internal events*. Secondly, by using *External events* the agents are able to communicate with each other so that they can synchronize and adapt their behavior in a changing environment (e.g., in case of applications oriented to mobile devices). Thirdly, by using *Past Events* the agents have a *memory* and can perform actions automatically whenever a well-known situation occurs. Therefore, the resulting infrastructure is flexible and allows run-time event-driven reconfiguration.

A good reason to keep a Java part of Lira agents is that DALI infrastructure cannot act directly on the component to perform reconfiguration. In fact, DALI does not provide high level mechanisms to interact with the components, while Lira is specified with that purpose.

The agents created by integrating DALI and Lira, that we have called Intelligent Agents (IA) for dynamic reconfiguration, have an intelligent part provided by DALI and a managing part provided by Lira. In other words, we can say that DALI constitutes the "mind", and Lira the "hand" for performing reconfigurations.

Fig. 1. The architecture of the Intelligent agents

The general architecture of the IA is shown in Figure 1. The interface between DALI and Lira is provided by a SICTUS Prolog library called *Jasper*, which allows one to call the specified Java method inside Prolog code.

Lira loses the TCP message management, but it still provides the access to the exported variables and functions through the following methods:

$ACKmsg$	$msgSET(varName, varValue)$
$REPLYmsg$	$msgGET(varName)$
$RETURNmsg$	$msgCALL(funcName, parList)$
$void$	$msgNOTIFY(varName, varValue)$

The communication among IAs is implemented by using DALI messages, and is managed by its run time support. The Java methods are called by the DALI environment through the Jasper interface whenever the corresponding DALI message is received.

In DALI, the reception of a Lira message is implemented by using an *external event*. When the event is received, the agent performs a specific action which hides the Jasper predicate. For example, the reception of the Lira message *CALL(”STOP”, ””)* is implemented as:

$$eve(CALL(\text{''}STOP\text{''}, void) : -a(daliCALL(stop, void))$$

where *daliCALL* is a macro which hides all the steps (objects creation, method call etc) necessary to actually invoke the Java method.

In the sample DALI code that we will present in the next subsections, all the operations for getting and setting values or more generally for affecting the supervised component are implemented in a similar way.

The reactive capabilities provided by DALI make the IA’s able to dynamically perform reconfigurations either upon certain conditions, or upon occurrence of significant events. In some cases reconfigurations can be decided and performed locally (on the controlled component), whenever the managing agent has sufficient knowledge, otherwise they can be decided by means of a process of cooperation and negotiation among the different agents.

Also, the Lira/DALI IA’s can manage and exchange meta-information about system configuration and functionality. In perspective, they may have knowledge and competence to detect critical situations, and to activate dynamic security processes in order to ensure system consistency also in presence of faults or attacks.

Finally, by using DALI primitives the agents are able to learn from past situations, for example to repeat the same kind of reconfiguration upon the same conditions, or to retry the same kind of negotiation.

6 The Case Study

The case study proposed here is remote management of web sites in a web hosting provider that runs on a Windows 2000 Server. This particular environment manages single web sites as independent components, allowing the administrator to start and stop web sites independently from the actual web server that hosts them.

The features of the example are the following: we have a general server W that manages the web sites W_i through the IAs IA_i. Each agent can communicate with the other agents and with the Manager. In particular, for each web site we are interested to supervise the disk space and the bandwidth.

Fig. 2. The hosting web provider

In order to show the flexibility of these new IA's, we propose (a sketch of) the implementation of two different policies of reconfiguration, aimed at optimizing space and bandwidth usage.

The space is managed by using a hierarchical model, where a Manager maintains the global knowledge about the space usage, and eventually orders the reconfigurations to the agents.

A high quality of service for each web site is guaranteed through a dynamic distribution of the available bandwidth. We employ to this purpose a cooperative model, where an agent that needs more bandwidth asks other agents for obtaining the possibility to increase its own usage.

The details of these policies are described in the Sections 6.1 and 6.2.

In order to act on the component (web site) and perform the reconfigurations, the IA exports the following variables and functions. USED_SPACE, that contains the used space; MAX_SPACE, i.e., the max space allowed; USED_BAND, i.e., the band used; MAX_BANDWIDTH, i.e., the max bandwidth allowed; STATUS, which is the state of the web site; NOTIFYTO, i.e., the agent that must be notified. ERASE(fileType, space) erases the specified files, thus freeing some space on the disk. COMPRESS(files, space) compresses the specified files thus making available a larger space quota on the disk.

6.1 Space Management

DALI definition of the Manager agent

Implementation of Site Maintenance
The manager starts the maintenance of a web site managed by an IA whenever a certain timeout has expired. The exact mechanism in DALI is that the predicate $activate\_maintenance$ is automatically attempted at a predefined (customizable) frequency, and succeeds as soon as the timeout is expired. Since this predicate is an internal event, its success triggers the proactive clause with head $evi(activate\_maintenance(IA))$, thus executing the body, and sends messages to IA to stop the web site, and perform the maintenance. At the end, the site is restarted.

$activate\_maintenance(IA) :- timeout\_expired(IA).$
$evi(activate\_maintenance(IA)) :- a(message(IA, CALL(''STOP'', void))),$
$\quad a(message(IA, perform\_maintenance(IA))).$
$eve(maintenance\_terminated(IA)) :- a(message(IA, CALL(''START'', void))).$

Remote Reconfiguration
If an IA that manages a web site asks for more disk space, the manager assigns more space to this web site if available. Only as *extrema ratio* the Manager eliminates old web sites with expired life time.

$eve(space\_not\_recovered(IA)) :- once(find\_space(IA)).$
$find\_space(IA) :- a(message(IA, SET(MAX\_SPACE, New\_space))).$
$cd(message(IA, SET(MAX\_SPACE, New\_space))) :- space\_available(New\_space).$
$find\_space(IA) :- once(check\_accounts(IA)).$
$check\_accounts(IA) :- a(erase\_expired\_web\_site).$
$cd(erase\_expired\_web\_site) :- \ldots$

DALI Intelligent agent definition for the Web Sites

Site Maintenance
The following piece of code defines how IA becomes aware of the orders by the manager of stopping/starting the site, and of performing maintenance. Notice that IA knows that maintenance is finished as soon as $perform\_maintenance$ becomes a past event (prefix evp), i.e., as soon as action $a(perform\_maintenance)$ has been done. If so, $end\_maintenance$ becomes true, and, since it is an internal event, it triggers a reaction that sends a message to the manager to signal that maintenance is over.

$eve(CALL(''STOP'', void)) :- a(daliCALL(stop, void)).$
$eve(CALL(''START'', void)) :- a(daliCALL(start, void)).$
$eve(perform\_maintenance) :- a(perform\_maintenance).$
$end\_maintenance(IA) :- evp(perform\_maintenance).$
$evi(end\_maintenance(IA)) :- a(message(M, maintenance\_over(IA))).$

Managing lack of space
As an example of adaptive behavior, the following piece of code included in the definition of a local agent specifies that if the used space of the managed web site is close to MAX_SPACE, then IA tries to find more space. First, the agent tries to recovery space locally, by either erasing or compressing files. If this is impossible, then it asks the manager. These local attempts of reconfigurations can be done only if they have not been performed recently, i.e., only if the corresponding past events (prefix evp) are not present (notice that the past events expire after a pre-set, customizable amount of time). Otherwise, the manager is informed by sending the message $space\_not\_recovered$.

$$\begin{aligned}&more\_space\_needed :- a(daliGET(MAX\_SPACE)),\\&\qquad a(daliGET(USED\_SPACE)),\\&\qquad MAX\_SPACE - USED\_SPACE \leq threshold.\\&evi(more\_space\_needed) :- recovery\_space(IA).\\&recovery\_space(IA) :- a(erase\_useless\_files(IA)).\\&cd(erase\_useless\_files) :- not(evp(erase\_useless\_files(IA))).\\&recovery\_space(IA) :- a(compress\_files(IA)).\\&cd(compress\_files) :- not(evp(compress\_files(IA))).\\&recovery\_space(IA) :- a(message(M, space\_not\_recovered(IA))).\end{aligned}$$

Updating space limit
If asked, the manager can give three different answers, corresponding to the following external events: (i) send an enlarged value new_space of MAX_SPACE; (ii) order to erase all files; (iii) stop the web site.

$$\begin{aligned}&eve(SET(MAX\_SPACE, new\_space)) :- a(daliSET(MAX\_SPACE, new\_space)).\\&eve(CALL(ERASE, all\_files)) :- a(daliCALL(ERASE, all\_files)).\\&eve(CALL(KILL, void)) :- a(daliCALL(KILL, void)).\end{aligned}$$

6.2 Bandwidth management

In order to exhibit a good performance to the end user, the Intelligent Agents cooperate for a dynamic band distribution according to the component needs. In particular, when an IA detects that the available band is less than the bandwidth needed, the internal event $seek\_band$ triggers a reaction: the agent checks which agents are present in the system and creates a list. Then it takes the first one and sends a request for a part of the band. If the agent receives the external event that indicates that more band is available, it sets the Lira variable MAX_BANDWIDTH, while the giving agent reduces its max bandwidth by taking off the given value. If the bandwidth is still insufficient, the agent keeps asking for band to the other agents are present in the system.

$$\begin{aligned}&seek\_band :- band\_insufficient.\\&band\_insufficient :- \ldots\\&evi(seek\_band) :- findallagents(Askable\_agents\_list), askfb(Askable\_agents\_list).\\&askfb(Askable\_agents\_list) :- member(IA1, Askable\_agents\_list),\\&\qquad a(message(IA1, ask\_for\_band(IA, IA1))).\\&\ldots\end{aligned}$$

Symmetrically, if IA is asked for some band, it checks if it is actually in the condition to give it. When the external event ask_for_band arrives, the agent checks its bandwidth. If it has some unused band (the USED_BAND is minor of the MAX_BANDWIDTH) it keeps 80% of its band, and offers the remaining amount Bt to the other agent. Otherwise, the agent sends the message $impossible\_to\_transfer\_band$.

$eve(ask\_for\_band(IA, IA1)) : -check\_available\_band(Bt).$
$check\_available\_band(Bt) : - \ldots$
$evi(check\_available\_band(Bt)) : -Bt \neq 0, a(message(IA1, offer\_band(Bt, IA)).$
$evi(check\_available\_band(0)) : -a(message(IA1, impossible\_to\_transfer\_band)$
$\ldots$

7 Discussion and Related Work

In [17], Garlan et al. proposed a methodology and a framework for performance optimization of a network component based application. In the proposed approach the software architecture of the application plays a central role: the managed application is continuously monitored and, when necessary, it is reconfigured at architectural level. The new configuration is chosen as result of the online evaluation of a components-connectors models. As well as our intelligent agents, the *gauges* proposed in [17] provide the monitored information to the manager, but, differing from the IAs, they are not provided of intelligence, and they cannot take any form of decision. The IAs, instead, have internal reasoning and memory, which allow the agents to activate a *local* reconfiguration without asking the manager.

In [11], the decisions about reconfiguration are taken by using the feedback provided by the online evaluation of a Petri Nets model representing the system. Even if effective in the context of dependability provision, the impossibility for agents and manager to *remember* the previous decision taken in similar situations forces an expensive model evaluation at every step of the decision. The memory mechanisms provided by DALI, instead, allows the agents to learn from the past configurations, adapting their behavior to the new situations.

In the current version of our framework, the Intelligent Agents and the Manager decide the best reconfiguration for the managed system by applying he defined reconfiguration policies, in a purely reactive way. Following the approach proposed by Son et al. in [18], the DALI/Lira framework can be modified to use planning mechanisms for the choice of the best reconfiguration. Each reconfiguration can be specified in terms of a plan to reach a specified goal, and the best one is chosen among the not failed plans. Currently, we are working on adding planning capabilities to the DALI language, and we foresee to upgrade the functionalities of the manager to have also this *plan-based* decision making.

The framework described in this paper is fully implemented, but it is a preliminary version. The current implementation suffers of many problems due to the technologies used for its development. Firstly, the Sictus prolog [14] used to implement DALI has a very heavy runtime support (a 10 Mbytes process for each agent), which creates many problems for the actual deployment of the Intelligent Agents within a limited resource device. Moreover, the library used for the integration between the Lira and the DALI agents, namely Jasper, does not provide a confortable support for many Java types, making the implementation of the Prolog-Java interface not easy at all. Then, we are planning to use tuProlog [20] as the base for DALI: tuProlog is characterised by a minimal yet efficient Prolog engine, created for infrastructures where software thickness and overloading is simply unacceptable [19]. tuProlog is written in Java, thus allowing an easy interface with the Lira infrastructure.

8 Concluding Remarks

In this paper we have proposed our practical experience of using the logic programming language DALI for enriching the functionalities of Lira, an infrastructure for managing and reconfiguring Large Scale Component Based Applications.

The advantage of Lira is that of being lightweight, although able to perform both component-level reconfigurations and scalable application-level reconfigurations. The key design choice of Lira has been that of providing a minimal basic set of functionalities, while assuming that advanced capabilities are implemented in the agents, according to the application at hand. This has allowed us to gracefully integrate Lira with DALI, by replacing Java agents with DALI agents. To the best of our knowledge, this is the first running prototype of a logic-based infrastructure.

We have argued that DALI brings practical advantages under several respects. (i) The task of developing agents with memory and reasoning capabilities becomes easier, and the resulting agent programs are easier to understand, extend and modify. (ii) Reactivity, proactivity and learning capabilities of logical agents make the system more powerful through the intelligent cooperation among logical agents that can supervise and solve critical situations. (iii) Intelligent agents with reasoning abilities can cooperatively perform many tasks and reach overall objectives, also by means on suitable forms of delegation and learning. The coordination and cooperation among agents that are difficult to implement with Lira because of its hierarchical architecture can be easily realized by using Lira/DALI intelligent agents. This makes the resulting infrastructure powerful and effective, especially in real-time contexts.

Both DALI and Lira are fully implemented, and the Intelligent Agents have been successfully experimented. Future applications are being specified, in challenging contexts such as system security in critical applications.

References

1. F. Bellifemine, A. Poggi and G. Rimassa. "JADE A FIPA-compliant agent framework". *Proceedings of PAAM'99*, held in London, April 1999, pp.97-108. Project URL: http://sharon.cselt.it/projects/jade/
2. L. Bellissard, N. de Palma and M. Riveill. "Dynamic Reconfiguration of agent-Based Applications". *Proceedings of European Research Seminar on Advances in Distributed systems, April 1999.*
3. *C. Bidan, V. Issarny, T. Saridakis and A. Zarras. "A Dynamic Reconfiguration Service for CORBA". In* Proc. of the 4th International Conference on Configurable Distributed Systems*, May 1998, Annapolis, Maryland, USA, pp. 35–42.*
4. *M. Castaldi. "Lira: a practitioner approach".* Technical Report*, University of L'Aquila, July 2002.*

5. *M. Castaldi, A. Carzaniga, P. Inverardi and A. L. Wolf. "A Light-weight Infrastructure for Reconfiguring Applications".* Proceedings of 11th Software Configuration Management Workshop*, Portland, USA, May 2003.*
6. *M. Castaldi, G. De Angelis and P. Inverardi. "A Reconfiguration Language for Remote Analysis and Application Adaptation".* Proceedings of ICSE Workshop on Remote Analysis and Measurement of Software Systems*, Portland, USA, May 2003.*
7. *M. Castaldi and N. D. Ryan. "Supporting Component-based Development by Enriching the Traditional API".* Proceedings of 4th European GCSE Young Researchers Workshop 2002*, in conjunction with NoDE, to be held in Erfurt, Germany, 7-10 October 2002.*
8. *S. Costantini. "Towards active logic programming". In A. Brogi and P. Hill, editors,* Proc. of 2nd International Workshop on component-based Software Development in Computational Logic (COCL'99)*, PLI'99, Paris, France, September 1999. http://www.di.unipi.it/ brogi/ResearchActivity/COCL99/ proceedings/index.html.*
9. *S. Costantini and A. Tocchio. "A logic programming language for multi-agent systems".* Proceedings of JELIA02, 8th European Conference on Logics in Artificial Intelligence, held in Cosenza, Italy, September 23-26, 2002. LNCS 2424, Springer-Verlag.
10. C. M. Jonker, R. A. Lam and J. Treur. "A Reusable Multi-Agent Architecture for Active Intelligent Websites". *Journal of Applied Intelligence*, vol. 15, 2001, pp. 7-24.
11. S. Porcarelli, M. Castaldi, F. Di Giandomenico, P. Inverardi and A. Bondavalli. "An Approach to Manage Reconfiguration in Fault-Tolerant Distributed Systems". *Proceedings of ICSE Workshop on Software Architectures for Dependable Systems*, Portland, USA, May 2003.
12. M. T. Rose. "The Simple Book: An Introduction to Networking Management". Prentice Hall, April 1996.
13. S.K. Shrivastava and S.M. Wheater. "Architectural Support for Dynamic Reconfiguration of Large Scale Distributed Applications". *Technical Report 645, pp. 1-14, Department of Computing Science, University of Newcastle upon Tyne, 1998.*
14. *SICStus home page.* http://www.sics.se/sicstus/.
15. *M. Wermelinger. "A Hierarchical Architecture Model for Dynamic Reconfiguration". In* Proc. of the 2nd Intl. Workshop on Software Engineering for Parallel and Distributed Systems*, IEEE Computer Society Press, 1997, pp. 243–254.*
16. *M. Wermelinger, A.Lopes and J.Fiadeiro. "A Graph Based Architectural (Re)configuration Language". In* Proc. of ESEC/FSE'01*, ACM Press, 2001.*
17. *David Garlan, Bradley Schmerl and Jichuan Chang. "Using Gauges for Architecture-Based Monitoring and Adaptation". In* Proc. of Working Conference on Complex and Dynamic Systems Architecture*, Brisbane, Australia, December 2001.*
18. *C. T. Son, E. Pontelli, D. Ranjan, B. Milligan and G. Gupta. "An Agent-based Domain Specific Framework for Rapid Prototyping of Applications in Evolutionary Biology". In* Workshop Notes of Declarative Agent Languages and Technologies, First International Workshop (DALT 2003)*. Melbourne, Victoria, July 15th, 2003.*
19. *tuProlog web page. In* http://www.lia.deis.unibo.it/Research/2P/.
20. *Enrico Denti, Andrea Omicini and Alessandro Ricci. "tuProlog: A Light-Weight Prolog for Internet Applications and Infrastructures". In* Proc. of Third International Symposium on Practical Aspects of Declarative Languages (PADL)*, Las Vegas, Nevada, March 2001.*