Building User Interfaces thanks to Eco-Resolution

David JULIEN
LIP6 - Pôle IA
8 rue du Capitaine Scott
75015 PARIS - FRANCE
Email: david.julien@lip6.fr

Zahia GUESSOUM
MODECO/CReSTIC, IUT de Reims
Rue des Crayères
51000 REIMS - FRANCE
Email: zahia.guessoum@lip6.fr

Mikal ZIANE
Université René Descartes
12 rue de l'école de médecine
75005 PARIS - FRANCE
Email: mikal.ziane@lip6.fr

***Abstract*— The development and maintenance of user interfaces, especially adaptable interfaces, is too complex. Model-based approaches are promising but not as successful as was expected. We thus propose to encapsulate interface models with software agents and to rely on eco-resolution, a distributed problem-solving algorithm, to reach the objectives of the interface designer or users. This will lead to the development of a proactive tool to actively assist the designer to build interface models.**

I. Introduction

Despite many efforts to improve the specification of user-interfaces, their development and maintenance are still too difficult.

Most of these efforts focus on the presentation part, such as the selection of the best widget to display a data, while few efforts deal with other aspects which remain tedious to design. For instance, to ensure the consistency between the application state and the interface one, developers have to work directly at the level of source code instead of describing this kind of constraints at an abstract level.

Moreover, generated user interfaces are still too rigid and do not easily adapt to new hardware, new software libraries or changing users preferences.

To address these issues, developers and users should be allowed to express their choices declaratively, to postpone their choices until run-time, and to modify these choice at any time. Model-Based Environments (see [10] for a survey) aim at supporting some of these requirements. Models allow to describe the most important aspects of user interfaces and are then transformed into an operational user interface.

However, model-based environments are known to be difficult to use and their behavior difficult to predict [8]. So we suggest to encapsulate models by software agents. Each agent will monitor a model element and will rely on knowledge attached to this element to transform it into executable code as well as to adapt it to varying constraints.

This paper gives an overview of the eco-resolution algorithm in a model-based approach to build user interfaces. GOLIATH, a model-based environment to build user interfaces, is described first. Then we present agents and the eco-resolution principle, and we justify why they are suitable to design models. Afterwards we describe how eco-resolution is applied to GOLIATH. Then we briefly describe the implementation and detail an example. Finally we give a short overview of related work.

II. Goliath

When using GOLIATH [6], user interfaces are described using a set of declarative models.

The *application model* describes application data, functions and their pre-conditions, notifications, and consequences of function-calls on application data.

The *dialogue model* describes user interfaces in terms of abstract containers and relationships between these containers. An abstract container defines views (what application data are displayed), operations (what functions may be triggered by users) and local variables. Each abstract container is linked to a presentation element. A view defines mainly a link between an application data (that is the result of a function call which does not modify the application state) and a presentation data. An operation defines mainly a side-effects function call when an event occurs. Parameters of a function-call may be presentation data, application data or local variables. These details will be usefull to understand the example.

The *presentation model* describes native elements provided by toolkits and presentation elements defined by developers.

The application model is defined by the application designer. The goal of the interface designer is to build a dialogue model and a presentation model to describe the interface.

GOLIATH is implemented and is able to generate a runnable user interface described with models without using any heuristics.

III. Agents, eco-resolution and user interfaces

In this section we describe what is an agent and the eco-resolution principle, and why they are suitable to build user interfaces.

A. Agents

A multi-agent system is defined as a set of interacting agents, capable of organizing themselves dynamically and of adapting their own behavior to their environment. This dynamic and adaptive approach eases the development, the maintenance and the dynamic evolution of systems [4]. An agent has the following properties:

- *autonomy:* each agent encapsulates its own state and makes decisions about what to do based on this state;
- *reactivity:* agents are situated in an environment (for instance a collection of agents), are able to perceive this

environment, and are able to respond to changes that occur in it;
- *pro-activity:* agents are able to exhibit goal-oriented behaviour and take initiatives;
- *social ability:* agents interact with other agents trough an agent communication language and are able to engage in social activities, such as cooperation or negociation, to achieve their goals.

Multi-agent systems are considered as an important new direction in software engineering for several reasons [5]:

- *natural metaphor:* many domains can be conceived as a set of active agents interacting to reach some personal goals;
- *distribution of data and control:* many systems do not have a single locus of control to solve a problem but many. In that case it is more efficient to distribute this control through different autonomous agents;
- *legacy systems:* existing systems may be encapsulated in agents to use them in a new system. Agents facilitate interoperability.
- *open systems:* many systems are open because it is not possible to know at design time all their components and interactions between these components. Autonomy and sociability are useful properties to allow future yet-unknown evolutions.

Many solutions are available to design a multi-agent system to solve a problem. In our case we choose the eco-resolution principle.

B. Eco-resolution

Our multi-agent system relies on the eco-resolution algorithm [2]. When possible, eco-resolution transforms a global problem into a set of local problems. The local problems are solved by a set of interacting agents. Under certain conditions, the individual behavior of the agents globally converge towards a stable state corresponding to a solution of the problem.

Contrary to traditional approaches where a program defines the different steps to solve a problem, a program based on eco-resolution defines agents and their interactions from an organisational model. Agents interact in an environment and a solution emerges from these interactions.

Systems based on eco-resolution exhibit two important properties:

- They are able to find a solution without exploring all the global state space. From an initial configuration, they evolve until they reach a stable state corresponding to a possible solution;
- They resist well to noise. A local perturbation induces a local adaptation to reach a globally stable state.

An interesting point of view to design a system based on eco-resolution consists to identify the different components of this sytem, and then to associate an agent to each component [3]. It allows to:

- describe a complex system with a set of simple interacting models instead of a complex model;
- describe different levels of abstraction homogeneously while offering possibilities to provide different point of views and expertise;
- intervene at different levels to introduce local modifications;
- observe emergences of new design solutions, and consequences of local modifications on these solutions.

C. Agents and Eco-resolution for user interfaces

Multi-agent systems provide interesting properties to design GOLIATH's models:

- *natural metaphor:* abstract containers, views, presentation elements may be components encapsulated in agents. Each agent will have specific knowledge to manage its component. Components become pro-active.
- *distribution of data and control:* each component has its own tasks and goals which are independent of other components.
- *legacy systems:* agents may encapsulate existing toolkits or functional cores. Therefore they allow to build user interfaces for existing applications without modify them.
- *open systems:* to introduce new concepts with new components only consists to introduce new agents and possibilities of interactions.

Moreover eco-resolution allows to imagine a new way to build models:

- Finding a solution from a partial description allows the designer to describe the main aspects of the interface and let the system evolves to a first solution.
- Perturbations allow the designer to "push" the agents to propose another solution when the current solution is not satisfactory.
- Agents may be kept at run-time to allow final users themselves to update the interface.

In the next section we will describe how we use eco-resolution with models of GOLIATH.

IV. ECO-RESOLUTION IN GOLIATH

Building user interfaces using eco-resolution targets two objectives: to automatically generate a first solution from few details and to improve easily the proposed solution.

Our first objective is to generate a user interface (that is a dialogue model and a presentation model) from an application model and some information about the targeted interface like application data to display. In order to do that, we associate an agent to each component defined in the models of GOLIATH. These components correspond for instance to abstract containers, views, operations or presentation elements.

An agent uses knowledge specific to its encapsulated component to solve local problems linked to this component. For instance a view displays application data and thus has to insure that the presentation is updated when the value of this data is modified in the application; a function-call agent requires data for each parameter of the function and thus contacts other agents to get them.

Agents communicate thanks to the Contract-Net protocol [9] (see figure 1). When an agent accepts a contract (for instance “call a function”), a strategy is defined to fulfill it. A strategy is composed of activities (find an operation, find the parameters), and for each activity is defined a set of available techniques to carry it out (use a presentation data or an application data as a parameter). The selected technique may induce new relationships with other agents.

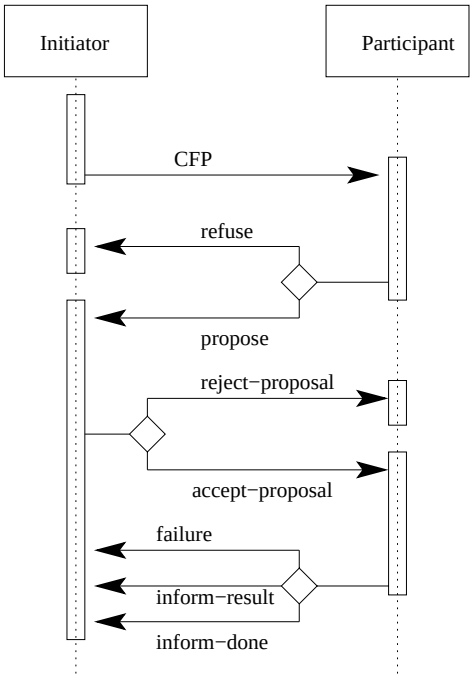

Fig. 1. The Contract-Net protocol.

Figure 2 represents a set of interacting agents encapsulating components. These components describe a part of a dialogue model and presentation model. For instance the function-call agent is linked to an operation agent, a local-variable agent and a presentation-data agent. Its describes which operation triggers the function call and what parameters are used to make the call.

Our *second objective* is to allow the designer to easily update the generated interface. For that, (s)he just has to identify which agent has made an incorrect decision, that is which agent has selected an inappropriate technique to solve a problem. After disabling this technique or suggesting an other, the agent cancels the consequences of the previous decision (like relationships with other agents) and puts in place new relationships to propose an other interface.

Figure 3 represents the agents after the designer identifies that choosing a presentation data as parameter is not the better solution. Disabling this decision leads the agent to cancel the relationship with the presentation-data agent and afterwards search an other date source, for instance an application data. Consequently the sub-element which displayed the presentation data becomes useless and is therefore rejected by the presentation element.

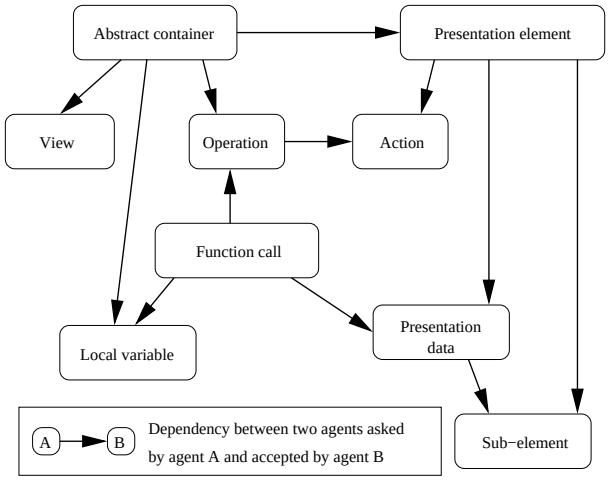

Fig. 2. Set of interacting agents managing user-interface components.

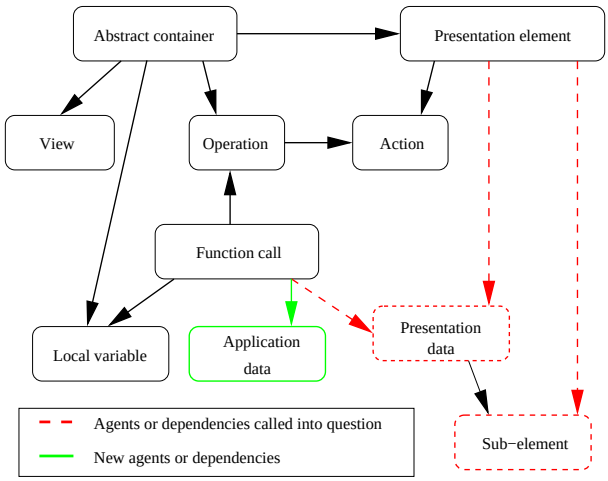

Fig. 3. New organization when the function-call agent is not authorised to use a presentation data as a parameter.

V. Implementation

Our approach has been implemented in Caml. Each agent class is described through a module. Each module defines, for each activity of an agent, an entry point to introduce techniques to fulfill this activity. Thus we are able to add or remove knowledge at any time.

The designer interacts with the multi-agent system through a user interface (see figure 4). This interface is described and runned thanks to Goliath. Our interface does not display the whole interactions between agents. It allows to introduce new agents, for instance by selecting through the interface which functions the final user may trigger. Moreover it allows to select an agent, to view its activities and selected techniques, and to change its decisions (see figure 13).

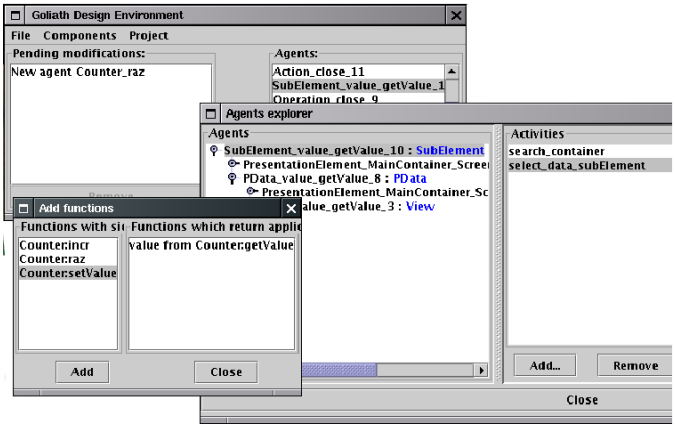

Fig. 4. GOLIATH's design tool.

VI. THE "COUNTER" EXAMPLE

To explain the behaviour of our eco-system, we choose a simple example which consists to build a user interface for a "Counter" application. This application defines four functions: `incr()`, `raz()`, `getValue(OUT Integer value)`, `setValue(IN Integer value)`.

Building the "Counter" interface requires two steps. First the designer selects interesting functions and tells the environment to generate a first version of the user interface. Second the designer corrects bad decisions to improve the solution.

In this example, 28 agents are used and the contract-net protocol is initiated in the neighbourhood of 350 times during the first phase. However the run time is less than 0.1s on a computer at 1Ghz. Obviously we give only a partial description of these interactions by considering only what happens around the "setValue" function call. To simplify too, we give a sequencial description but in reality all activities occur in parallel when possible.

A. First generation of the user interface

The designer wants to display the counter value and to allow the final user to increment, reset and set this value. Thus (s)he selects this four functions defined in the application model. We will focus on what happens around the "setValue" agent.

- The environment creates an "abstract container" agent since every interface has at least one container.
- The environment creates an agent for each selected function and sends a contract to each one to specify which function has to be encapsulated (see figure 5).
- When a function-call agent receives a contract to manage a function call (for instance `setValue`), it follows a strategy composed of two activities (see figure 6) :
  1) Determining the data source of each parameter of the function call, in our case the integer `value`. Many techniques are available for that: get the data from the presentation, get the data from the functional core, use a local variable. Agents are configured by default to privilege techniques involving the participation of the final user. Thus the function-call agent decides to get the data from the presentation since it is possible. Consequently the "setValue" agent creates a presentation-data agent and sends it a contract to get an integer.
  2) Finding an operation which will trigger the function call. For that, it creates an operation agent and sends it a contract to manage the triggering.

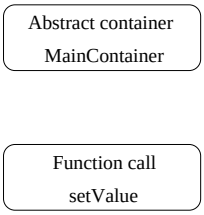

Fig. 5. Initial organisation.

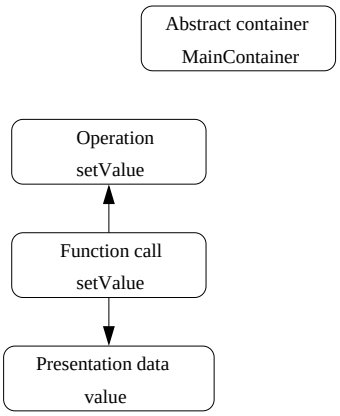

Fig. 6. The `setValue` agent has found a data source for the parameter `value` and an operation to control the triggering of function.

- When an operation agent receives a contract to manage a function call, it follows a strategy composed of two activities (see figure 7):
  1) Determining when the function call is triggered. For that it has to identify at least one event like an other function call, an application notification, an action fired by the final user or a navigation signal. If it selects an action, it creates an action agent and sends it a contract to manage an action which triggers an operation.
  2) Finding an abstract container: search an existing abstract container or create a new one. Here our agent contacts the default abstract-container agent.
- When an abstract-container agent receives a contract to integrate operations (whose linked to `incr`, `raz`, `setValue`) and views, it follows a strategy composed of three activities (see figure 8):
  1) Verifying that the container may be closed. For that a new operation agent is created with a contract to

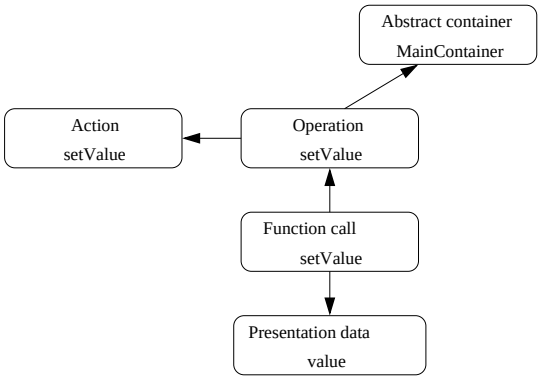

Fig. 7. The operation agent creates an action agent to trigger the operation and contacts the abstract-container agent.

close the container. This agent will search an event to control the closure (for instance a presentation action).

2) Looking for a presentation element if views or operations depend on presentation data or actions. For that a new presentation-element agent is created. It receives a contract to manage all presentation data and actions defined in views or operations linked to the abstract container.
3) Verifying that the abstract container may be triggered. For that a navigation-link is created to allow the opening (for instance when the interface is launched).

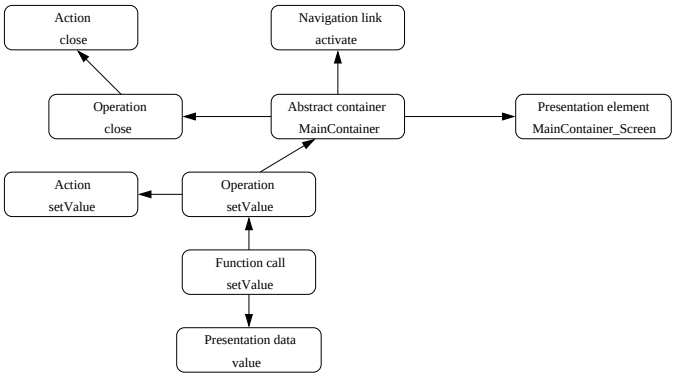

Fig. 8. New agents are created by the abstract-container agent.

A similar behaviour allows to create the presentation part linked to this dialogue (see figure 9). These two descriptions correspond to a dialogue model and a presentation model. They can be interpreted to obtain the user interface of the figure 10.

B. Improving the user interface

This first solution may be improved. For that, the designer just has to identify incorrect decisions, that is which agent has selected an inappropriate technique to solve a problem. The inappropriate techniques are then disabled and the designer

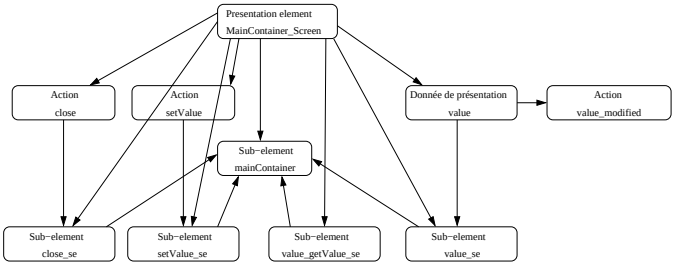

Fig. 9. Agents linked to the presentation part.

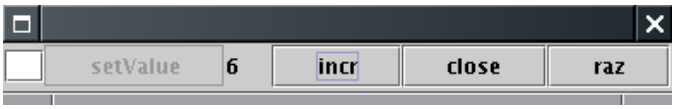

Fig. 10. The first solution suggested by the agents. A final user may increment the counter, reset it, set its value when an integer is specified in the text field. The presentation is updated each time the counter value changes. Moreover, the `setValue` button is disabled when the text field does not contain an integer.

may choose directly the best one or let the system tries an other solution. Modifications are then taken into account to generate a new solution. Here are some examples of corrections:

- Figure 11. Texts may be replaced: Main commands are capitalized and the "close" button is translated in french. For that, the designer just has to select the agent which manages a button, select the activity which associates a text to this button, and imposes a new text to replace the default one.

Fig. 11. The designer corrects some texts.

- Figure 12. Layout may be improved. For instance to move the "Fermer" button on the third line, the designer just has to select the agent which manages this button, search the layout activity to identify which agent manages the container. Then, the designer identifies which activity layouts the button and corrects the solution. For instance (s)he updates coordinates and dimensions.

Fig. 12. The designer corrects layout.

- Figure 13. A widget may be replaced by an other. For instance, it is possible to choose an other widget to get the value assigned to the counter. For that the designer just has to select the agent which manages the widget and changes the widget selected.

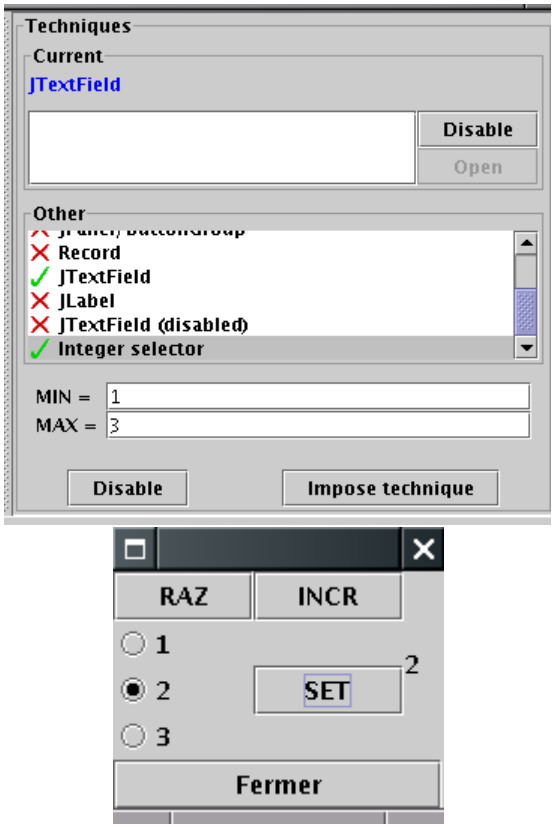

Fig. 13. The designer imposes an other solution for a widget.

- Figure 14. The designer may decide that the final user does not have to enter the value used to set the counter. For that, (s)he just has to start from the widget agent, identify that this agent has been created by a presentation-data agent, which has been created by the "setValue" agent. Then the designer modifies the technique selected to choose the function-call parameter. For instance (s)he may specify directly which value will be used instead of asking the final user.

Fig. 14. A default value defined by the designer is used when the final user clicks on the SET button.

VII. Related work

Agents have already been used to design a user interface, but only at run-time. Chiron-1 [11] defines "artists" to encapsulate decisions of presentation and dialogue linked to each application data. These artists create an initial graphical representation, manage events modifying their handled data, and communicate with a server responsible of the presentation. These agents are able to update the presentation when an event modifies an application data and to process user actions on the representation of the data. Vesuf [1] suggests to use agents to design adaptative interface. Agents select a dialogue model and a presentation model according to the run-time context like the display screen.

Using knowledge to facilitate the construction of user interfaces is a well-known idea. ITS [14] defines transformation rules to improve a dialogue model and to generate a presentation model. But according to the authors, it is difficult to maintain a consistent set of rules. TRIDENT [13] and then SEGUIA [12] use rules and heuristics to select widgets and layout them in a window. However knowledge is used to solve well defined problems and is not applied in the overall interface. In Just-UI [7], the designer may provide a partial description of the desired user interface through instances of interface patterns. This description is then automatically completed like us. However it is not possible to know alternatives to correct the produced solution. Moreover the patterns are simple and predefined, and the generation of the interface relies on hard-coded principles. Consequently all the generated user interfaces have the same visual aspect.

VIII. Conclusion

Multi-agent systems and eco-resolution allow to generate interface models from a very simple description. Moreover the produced models may be improved afterwards by modifying incorrect decisions. The tedious tasks are fully managed by the agents (for instance the consistency of the presentation regarding the application state) and the designer just has to correct some decisions made by the agents.

Therefore, eco-resolution paves the way to *proactive design tools*. This kind of tools goes further than basic assistants which advise the designer. They make the best part of the work and get the designer a position of expert who evaluates the result and corrects the decisions.

REFERENCES

[1] L. Braubach, A. Pokahr, D. Moldt, and W. Lamersdorf. Using a model-based interface construction mechanism for adaptable agent user interfaces. In T. Finin and Z. Maamar, editors, *Proceedings of AAMAS Workshop 16 - Ubiquitous Agents on Embedded, Wearable, and Mobile Devices*. Facoltà di Ingegneria Bologna, 7 2002.

[2] Jacques Ferber. *Multi-Agent System: An Introduction to Distributed Artificial Intelligence*. Addison Wesley, 1999.

[3] Rémy Foisel, Alexis Drogoul, Olivier Cayrol, Mondher Attia, and Nicolas Chauvat. Des écosystèmes artificiels d'aide à la conception : l'exemple du projet CAROSSE. In *Actes des JFIADSMA99. Gleizes M.-P. et Marcenac P. (eds)*, pages 313–326. Hermés, 1999.

[4] Nicholas R. Jennings and Stefan Bussmann. Agent-based control systems: Why are they suited to enginneering complex systems. *IEEE Control Systems Magazine*, 23(3):61–74, 2003.

[5] Nicholas R. Jennings, Katia Sycara, and Michael Wooldridge. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.

[6] David Julien, Mikal Ziane, and Zahia Guessoum. GOLIATH: An extensible model-based environment to develop user interfaces. In *Proceedings of the Fourth International Conference on Computer Aided Design for User Interfaces (CADUI'2004), sponsored by ACM and jointly organized with IUI'2004*, pages 95–106. Kluwer Academics Publishers, January 2004.

[7] Pedro J. Molina, Santiago Melia, and Oscar Pastor. Just-ui: A user interface specification model. In Ch. Kolski and J. Vanderdonckt (Editors), editors, *Proceedings of the 3rd International Conference on Computer- Aided Design of User Interfaces CADUI'02*, pages 63–74. Kluwer Academics Publisher, May 2002.

[8] Brad A. Myers, Scott E. Hudson, and Randy Pausch. Past, Present, and Future of User Interface Software Tools. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 7(1):3–28, 2000.

[9] James Odell, H. Van Dyke Parunak, and Bernhard Bauer. Representing agent interaction protocols in UML. In *First international workshop, AOSE 2000 on Agent-oriented software engineering*, pages 121–140. Springer-Verlag New York, Inc., 2001.

[10] Paulo Pinheiro da Silva. User Interface Declarative Models and Development Environments: A Survey. In Ph. Palanque and F. Paternò, editors, *Proceedings of DSV-IS2000*, volume 1946 of *LNCS*, pages 207–226, Limerick, Ireland, June 2000. Springer-Verlag.

[11] Richard N. Taylor, Kari A. Nies, Gregory Alan Bolcer, Craig A. MacFarlane, and Kenneth M. Anderson. Chiron-1: A Software Architecture for User Interface Development, Maintenace, and Run-Time Support. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 2(2):105–144, June 1995.

[12] Jean Vanderdonckt. Assisting designers in developing interactive business oriented applications. *Proc. of 8th Int. Conf. on Human-Computer Interaction of HCI International'99*, 1:pp. 1043–1047, 1999.

[13] Jean M. Vanderdonckt and François Bodart. Encapsulating knowledge for intelligent automatic interaction objects selection. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 424–429. ACM Press, 1993.

[14] C. Wiecha, W. Bennett, s. Boies, and J. Gould. Generating highly interactive user interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 277–282. ACM Press, 1989.