SVR 2001 - 4th SBC Symposium on Virtual Reality, p. 127 - 138. Florianópolis, Brazil. SBC, 2001.

A COMPONENT-BASED INFRASTRUCTURE FOR THE COORDINATION OF COLLABORATIVE ACTIVITIES IN VIRTUAL ENVIRONMENTS

Alberto B. Raposo[1], **Christian M. Adriano**[1],
Adailton J. A. da Cruz[1,2], **Léo P. Magalhães**[1]

[1] Department of Computer Engineering and Industrial Automation
School of Electrical and Computer Engineering – State University of Campinas
CP 6101 – 13083-970 – Campinas, SP, Brazil
[2] Universitary Center of Dourados – Federal University of Mato Grosso do Sul
CP 322 – 79804-970 – Dourados, MS, Brazil
{alberto, medeiros, ajcruz, leopini}@dca.fee.unicamp.br

Abstract. *In this paper we introduce an approach based on software components to coordinate the behavior of virtual environments from the collaboration point-of-view, in which human or virtual participants interact by means of interdependent tasks. The coordination components communicate with tasks, controlling their execution and ensuring that interdependencies are not violated. By the use of coordination components, we not only separate the coordination model from the implementation of virtual environments, but also open the possibility to create a library of reusable components. The library implements the last part of a threesome coordination infrastructure for virtual environments, which is based on an abstract model, its mathematical counterpart, and its executable elements (the components).*

Keywords: Collaborative Virtual Environments, Coordination, Software Components, Modeling of Behavior, Computer Supported Cooperative Work.

1. Introduction

A networked virtual environment is a simulation of a real or imaginary world where multiple users may interact with each other in real time, share information, and manipulate objects in the shared environment [Singhal 99]. Due to these characteristics, virtual environments (VEs) appear as potential tools for the support of collaborative activities. Nevertheless, the development of VEs has been dominated by leisure activities, enabling basically navigation through virtual scenarios and communication with remote users. This kind of activity is what we call “loosely coupled collaborative activity” and is well coordinated by the social protocol, characterized by the absence of any explicit coordination mechanism, trusting the participants’ abilities to mediate interactions.

On the other hand, “tightly coupled collaborative activities” require sophisticated coordination mechanisms in order to be efficiently realized in VEs. This is a kind of activity whose tasks depend on each other to start, to be performed, and/or to finish. Examples of tightly coupled collaborative activities include collaborative authoring, workflow procedures, computer games, among others. A core concern in collaborative VEs is the risk that actors get involved in conflicting or repetitive tasks. The coordination

needed in this context, is defined as “the act of managing interdependencies between activities performed to achieve a goal” [Malone 90] and is managed by coordination mechanisms [Schmidt 96], which are software devices that interact with the application to control the behavior of the VE.

We argue here that an effective solution for the coordination issue resides in an infrastructure-like initiative. Such initiative consists of thinking the solution as free of application domain details. Instead, the infrastructure should be viewed as three abstract aspects inherent to coordination dependent domains, namely a generic coordination model and its mathematical and executable counterparts. The reason for this abstraction resides on two characteristics of coordination. First, many coordination features are generic and apply to domains out-side VR research, such as time sharing, process scheduling, locking policies, race condition analysis, deadlock prediction, etc. Due to all these computer pervasive facts, any solution for the coordination issue has necessarily a considerable generic dimension. Therefore, an infrastructure-like solution is convenient, since its essence is to provide some set of general functionality. As second point, we aimed at a solution to be as independent as possible of the specific VE semantics. Such feature would enable decoupling tradeoffs between a coordination apparatus and the whole VE.

The coordination infrastructure we propose in this paper is based on the following tripod: task-interdependency model, a mathematical formalism, and a component-based framework. The task-interdependency model abstracts the aspects necessary to coordination reasoning and actuating. The formalism validates and implements the mathematical aspects of the model, providing tools for prediction and optimization. The component-based framework implements the executable aspects of the model.

The paper is organized as follows. Section 2 describes the coordination model and briefly depicts the mathematical formalism. Section 3 explains the framework of coordination control. An implementation using the presented infrastructure was crafted in a form of videogame prototype (Section 4). The last section presents the conclusions.

2. Coordination Model

The required model must represent the VE in a manner that a coordination control could reason on it and actuate effectively. The coordination model presented here stems from two previous results accomplished in earlier work [Raposo 00a].

The coordination model, which constitutes the first element of the infrastructure, defines a collaborative activity as an interdependent set of tasks realized by multiple actors to achieve a common goal. Also according to this model, tasks are the building blocks of activities and may be atomic or composed of subtasks. Tasks are connected to one another through interdependencies, and the management of such interdependencies is realized by coordination mechanisms. Interdependencies in the proposed model are divided into two types, *temporal* and *resource management* [Raposo 00a].

Temporal interdependencies establish the execution order of tasks. The set of temporal interdependencies is based on temporal relations defined by Allen [Allen 84]. According to him, there is a set of primitive and mutually exclusive relations that could be applied over time intervals (and not over time instants). This characteristic made these relations suited for task coordination purposes, because tasks are generally non-instantaneous operations. This temporal logic, however, is defined in a context where it is essential to have properties

such as the definition of a minimal set of basic relations, and the mutual exclusion among them. Temporal interdependencies between collaborative tasks, on the other hand, are inserted in a different context. For this reason, it was necessary to adapt Allen's basic relations, adding a couple of new relations and creating variations of those originally proposed. The main difference in the context of collaborative activities is that it is possible to relax some restrictions imposed by the original relations. The result of the adaptation for the context of collaborative activities is a set of 13 temporal interdependencies presented elsewhere [Raposo 00a]. To illustrate some of these interdependencies, consider two tasks T1 and T2 that occur, respectively, in intervals $[t1_i, t1_f)$ and $[t2_i, t2_f)$.

T1 equals T2 ($t1_i = t2_i$ and $t1_f = t2_f$): This dependency establishes that two tasks must occur simultaneously.

T2 afterA T1 ($t1_{f,n} < t2_{i,n}$, $\forall n > 0$, where *n* means the n^{th} execution of the task): T2 may only be executed if T1 has already finished. In this case, T2 may be executed only once after each execution of T1.

T2 afterB T1 ($t1_{f,1} < t2_{i,n}$, $\forall n > 0$): Variation of the previous dependency, in which T2 may be executed several times after a single execution of T1.

T1 meets T2 ($t1_f = t2_i$): T2 must start immediately after the end of T1.

Resource management interdependencies are complementary to temporal ones and may be used in parallel to them. This kind of interdependency deals with the distribution of resources among the tasks. By "resource" we mean not only the agent that performs the task, but also any artifact needed to the execution of the task. Three basic resource management dependencies were defined:

Sharing: A limited number of resources must be shared among several tasks. It represents a situation that occurs, for example, when several users want to edit a document.

Simultaneity: A resource is available only if a certain number of tasks request it simultaneously. It represents, e.g., a machine that may only be used with two operators.

Volatility: Indicates whether, after the use, the resource is available again.

The second element of our threesome infrastructure is the mathematical formalization of coordination mechanisms that control the above interdependencies. The models of the coordination mechanisms are used for the analysis and simulation in order to verify the correctness and validate the efficiency of the collaborative environment before its implementation, which is an important step in the design of complex environments.

The coordination mechanisms have been modeled using three distinct approaches, conventional [Raposo 00a], high level [Raposo 00b], and fuzzy [Raposo 01] Petri Nets (PNs). The choice of PNs as modeling tool is justified because they are a well-established theory (there are numerous applications and techniques available) and can capture the main features of VEs, such as concurrency, synchronization of asynchronous processes and non-determinism. Moreover, PNs accommodate models at different abstraction levels and are amenable both to simulation and formal verification.

The goal of the present work is to give the next step on the coordination infrastructure, defining a software component framework to implement those coordination mechanisms as "black boxes" connecting the collaborative tasks. The coordination components are capable of receiving and generating events from/to collaborative tasks, controlling their execution. Thus, events related to tasks (e.g., the beginning of a task) may generate output events that affect the execution of interdependent tasks (e.g., blocking the execution of another task).

The components have been implemented as a set of JavaBeans [JavaBeans 01], which is a framework consisting of a Java API that defines rules by which to implement software components (beans) and to have them integrated through an event notification scheme.

3. Coordination Components Framework

Software components [Szyperski 98] were idealized based on an analogy with electronic components, characterized by reusable modules with well-defined functions. Today, issues such as autonomy, interconnection and integration complement the component approach, in addition to reuse and modularization. The paradigm states that components are integrated in order to provide complete functions, but neither keep references to one another nor communicate directly–the communication is achieved by means of messages. The decoupling compromise satisfies a number of necessities, such as the exchange of a component for another and the immediate insertion of new components.

Therefore, the component model is a satisfactory solution for the design of VEs that follow the task/interdependency model for tightly coupled collaborative activities. Allied with the component paradigm is the framework design approach, which together perform the executable aspect of the proposed coordination infrastructure.

The component framework is presented under two documentation approaches, a pattern-based [Johnson 92] and hook methods and hotspots [Albrecht 97]. The former explains the framework design as problem/solution pairs. The latter establishes the effective bridges between requirements and design solutions. The framework will be described as follows: requirements met, internal flow of control, and integration and extension.

3.1 Requirements

We idealized the coordination level as a software layer loosely coupled to the scene (VE). The advantages of this approach are twofold. It isolates the scene from changes made exclusively in the coordination control architecture, and also provides a clear test bench to investigate coordination control issues. A number of requirements should be supported by the coordination control layer: *(i)* implementation separated from the scene; *(ii)* accommodation of changes in the scene, such as removal or inclusion of a task; *(iii)* possibility of changes in the coordination logic, which should be localized; *(iv)* mixing of different coordination logic within the same coordination control and *(v)* attachment of monitoring tools to gather performance related to the coordination control strategy (potential evaluation indices are throughput, deadlock risk, and load balance).

The architectural decoupling between components is the solution for requirements *i-iii*. The encapsulation implementation of complete services within each component is the solution for requirement *iv*. The event-oriented communication is the basis to attach external tools (requirement *v*).

3.2 Internal Flow of Control

The flow of control is basically an algorithm. Since a framework is a kind of incomplete application, it also encapsulates an algorithm. Here we depict this algorithm from a high level vision of components involved in the coordination between two tasks.

Figure 1 illustrates such high level vision. In the coordination level the figure shows three components, a coordinator and two tasks. The coordinator component implements the

coordination mechanisms, both for temporal and resource management dependencies. The task component, instantiated for each task, is responsible for maintaining the task's schedule. The components of our architecture were designed to communicate by means of the sequence of events described in Table 1.

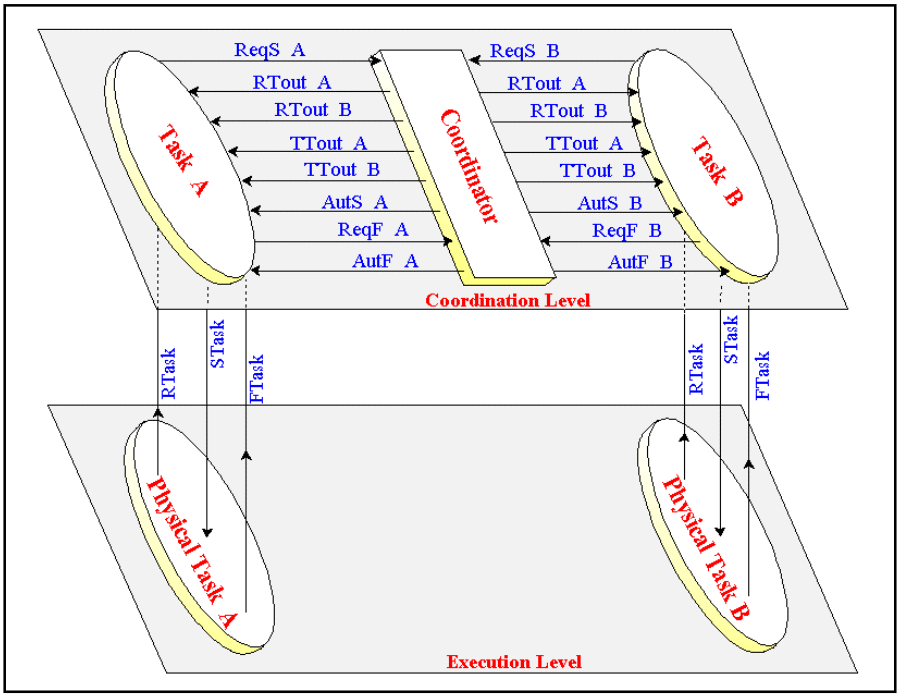

Figure 1: High-level vision of the coordination component acting between two tasks.

In the model of the coordinator component, a task has to wait until the temporal and resource management coordination mechanisms authorize its execution. A consequence of this fact is that the task may wait indefinitely if either of these conditions is not satisfied. In order to avoid such situations, the coordinator sends timeout signals, described in Table 2. The treatment of timeouts is left to the task components. This gives more flexibility to the architecture, because it does not risk the coordinator component reusability.

A possible consequence of timeouts is that the non-execution of an expected task may invalidate interdependent tasks previously executed. For this reason, the coordinator component sends timeout signals to all interdependent tasks, and not only to that which had its execution unauthorized.

Up to this point we have centered the discussion in the coordination level, where only the logical execution of the tasks is considered. Figure 1 also illustrates the execution level that represents the "physical" execution of the tasks in the VEs. The connection between both levels is left to the task components, reducing the responsibilities of the coordinator components, what contributes for their reuse. The physical tasks communicate with their respective task components by means of the events described in Table 3.

An advantage of this two-level approach is the modularization of the architecture. The coordination model of the VE may be developed independently of its implementation and vice versa. The only compromise between both levels is the communication by means of the three signals described in Table 3.

EVENT	DESCRIPTION
REQUEST START (ReqS)	When a task is demanded, for example due to a user action, it must first contact the coordinator to request an authorization for execution. At this time, the coordinator checks if there is any resource management interdependency and, if so, it consults the resource management mechanism to verify if the resource is available (if not, it waits until the resource becomes available). Once the resource is available or in the case of having no resource dependency, the coordinator checks if there is any temporal interdependency. If so, it consults the temporal coordination mechanism to verify if the conditions to the task's beginning are satisfied (if not, it waits until these conditions are satisfied). Once all conditions are satisfied, the signal AutS is sent to the task component.
AUTHORIZE START (AutS)	This signal is the authorization given by the coordinator to enable the beginning of a task's execution.
REQUEST FINISH (ReqF)	Once the task wants to finish its execution, it sends this signal to the coordinator, which verifies if the temporal interdependency (whether it exists) enables the end of the task. If so, it sends the signal AutF to the task and, in the case of having a resource management dependency, releases the assigned resource. Otherwise, the coordinator waits until the temporal coordination mechanism authorizes the task's finish.
AUTHORIZE FINISH (AutF)	This signal indicates to the task that it may finish.

Table 1: Events between coordination and task components.

EVENT	DESCRIPTION
RESOURCE TIMEOUT (RTout)	If the resource is not assigned to the task after a certain waiting time, the coordinator sends this signal.
TEMPORAL TIMEOUT (TTout)	If another task does not offer the conditions to the beginning of the task that requested it, the coordinator sends this signal after a certain waiting time. In this case, if the resource has already been assigned to the task, the coordinator also releases it.

Table 2: Timeout signals.

EVENT	DESCRIPTION
REQUEST TASK (RTask)	This signal is sent from the physical task to the respective component when an event in the VE (user action, solicitation of another task, elapsed time, etc.) asks for its beginning. The task component forwards this information to the coordinator in the coordination level via ReqS signal.
START TASK (STask)	When the coordinator authorizes the beginning of a task (AutS), the task component forwards this authorization to the physical task via this signal. At this moment the task really starts its execution in the VE.
FINISH TASK (FTask)	The end of the task's execution in the VE generates this signal that is sent to the respective component in the coordination level. This information is forwarded to the coordinator via ReqF signal. It is important to clarify that the task's logical end (i.e., from the coordination point-of-view) occurs when the task component receives the AutF signal. The logical end of a task guarantees that temporal interdependencies are not violated and may be delayed in relation to the physical end in specific situations.

Table 3: Events between the physical task and the task component.

Concerning the coordination component internals, it encapsulates two PN simulators, one for the temporal and another for the resource management coordination mechanism.

Each of these simulators interacts with their associated events, received or sent by the component. Events arriving at the coordinator alter the state of the simulator, while certain states of the simulator generate output events. Figure 2 depicts the coordinator functioning.

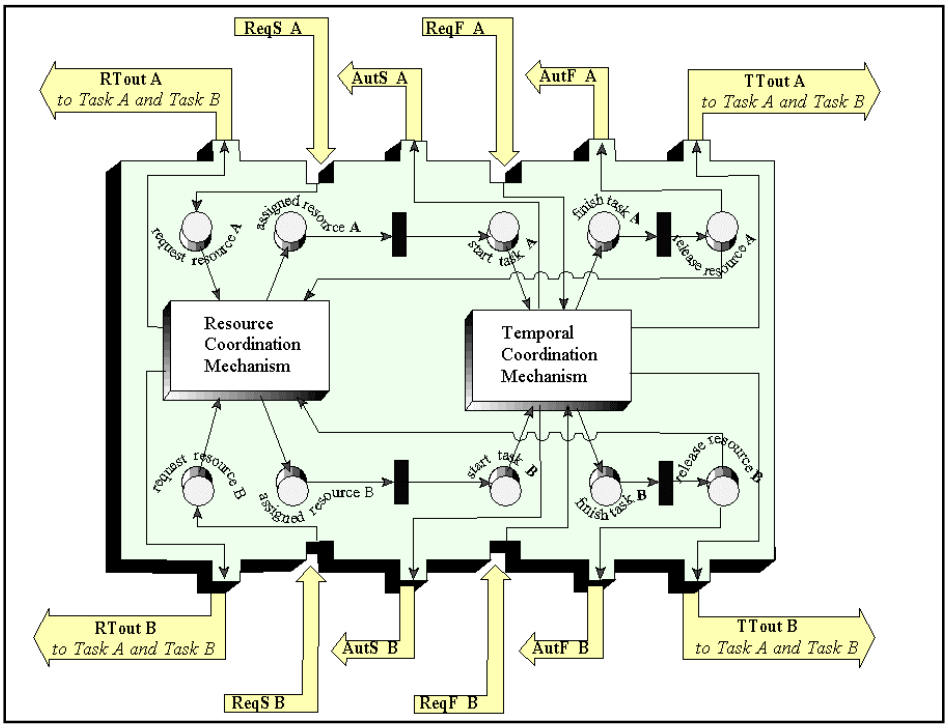

Figure 2: The coordinator functioning.

When the coordinator receives a ReqS signal, it puts a token in the respective request_resource place, starting the resource management mechanism. When the resource is available, the resource management mechanism puts a token in the assigned_resource place, which sends it to start_task. The start_task place starts the temporal coordination mechanism, which sends the respective AutS signal when the task may begin. When the physical task is finished, the temporal mechanism receives the ReqF signal. If the logical end of the task is authorized, a token is sent to finish_task, and then to release_resource, which indicates to the resource management mechanism that the resource is free. Finally, the coordinator sends the AutF signal, indicating the logical end of the task. The PN models for several temporal and resource management mechanisms have been shown elsewhere [Raposo 00a]. Although belonging to the same component, both internal coordination mechanisms have independent behaviors, running different PN simulators.

3.3 Integration and Extension

Although the only compromise between the coordination and execution levels is the communication by means of the signals discussed above, the integration issue is not an easy task, specially due to the tight relation between scene semantics and its coordination. The idea is that the decoupled integration resides essentially on communication patterns between the scene and coordination components. A communication pattern imposes different kinds of externalization of scene internal elements (i.e., actors, resources, tasks,

etc.). In the following, we discuss three externalization possibilities (Figure 3).

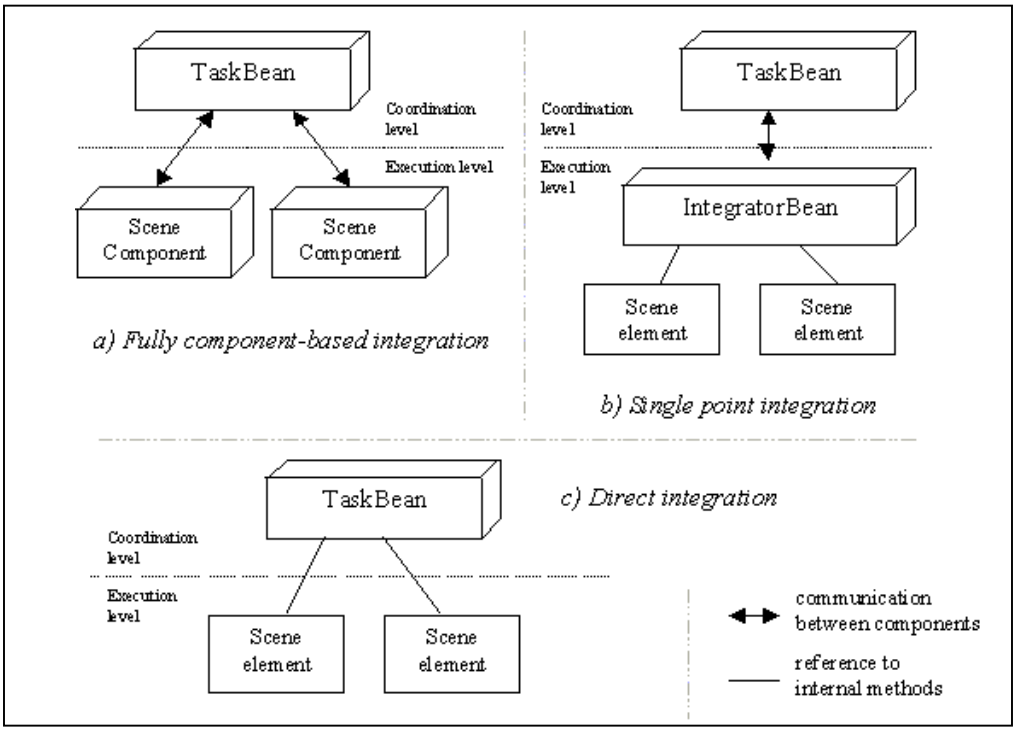

Figure 3: Approaches for the integration of scene-coordination components.

The first possibility is to implement the VE under the software component paradigm (i.e., actors, tasks, and resources implemented as components), which would lead to a straightforward integration with the coordination control. Each task component in the coordination level would be connected to a set of components in the scene. In this approach connections are loosely coupled because TaskBean (Figure 3a) does not need to import any scene library/package. This fully component-based integration approach is the ideal one from the software architecture point-of-view, but it requires a component-based VE.

A second alternative would be to encapsulate the whole scene in a single IntegratorBean, which would be basically responsible to forward coordination commands to the appropriate scene elements and to return scene events to respective task components at the coordination level (Figure 3b). This single point integration approach solution has the advantage of not imposing a reorganization of scene elements, enabling different scene technological implementations. Its drawback is a less decoupled and robust solution, since even a slight change in the scene would impact the IntegratorBean.

A third approach is to have each task component accessing their respective scene elements (Figure 3c). The advantage of this direct integration approach is that there is not a unique integration point and, therefore, no need to translate JavaBeans events to method calls in scene elements. Moreover, there is no need to restrict or adapt the scene to the coordination components. Scene elements may even be implemented as simple structured and not object-oriented code. The drawbacks, however, are numerous. The sense of layer is wicked, because coordination components need to import scene libraries. Furthermore, even a single change in the scene could impact the coordination components in a less

uniform and localized manner than in previous integration approaches.

Another framework design concern is the extension issue. There are three concepts needed here, hotspot subsystem and its two methods, hook and template. Hotspots represent a framework's adaptable aspects [Pree 99] and must be extended by the application in order to be active (though default implementations must also be provided). In concrete means, the hotspot is a set of concrete and abstract classes. The hook method is defined in one of the abstract classes, while its implementation is in application code. Furthermore, the hook method is called by a framework private method, namely a template method.

Two hotspot subsystems with default extensions were designed and are shown in Figure 4. The first hotspot (Figure 4a) consists of the TaskComponent, which implements the integration choice between coordination and execution layers. The TaskBean component, presented in the previous integration discussion, appears here as the default implementation for the TaskComponent. The second hotspot subsystem (Figure 4b) consists of class CoordinationComponent. This hotspot effectively enables the logic to be adapted, which is done by inheriting the CoordinationComponent. The default implementation uses a PN logic. Additionally, the classes in the diagram define protected (#) properties needed by the inter-component message exchange and declare the public (+) hook methods that are called by private (-) template methods. These last ones are not shown here.

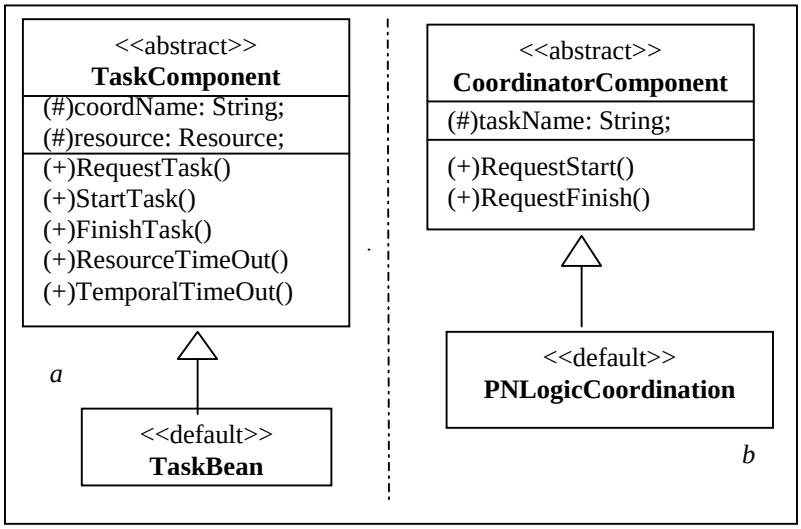

Figure 4: UML diagrams for the two hotspots.

4. Example: A Multiuser Videogame

In this section it is presented a case study of a VE where a user interacts with an autonomous agent that represents a second user. The example implements a kind of videogame based on the second "task" (activity) of Heracles, from the Greek mythology. According to the legend, Heracles had to kill the Hydra of Lerna, a monster with nine heads that are regenerated after being severed. In order to achieve his goal, Heracles needs the collaboration of his nephew Iolaus, who cauterizes the monster's wounds after Heracles cuts off each head. However, the last head may not be severed by any weapon. The solution was to bury the monster in a deep hole and cover it with a huge stone.

Figure 5 illustrates an abstract PN-like model of the videogame (open rectangles indicate interdependent tasks). There are two identical nets, one representing the user's and

the other representing the agent's sequence of tasks. Each net has two alternative paths, indicating that each "actor" (user or agent) may assume either role (Heracles or Iolaus). The upper part of the nets represents Heracles' sequence of tasks. He must get the sword, sever eight of the Hydra's heads, throw the beast into the hole and cover it with a stone. The lower part of the nets represents Iolaus' sequence of tasks. He must get the torch, cauterize the wounds after Heracles has severed the heads and dig the hole. The boxes with interdependency names are substituted by the respective coordination mechanism models for simulation and analysis purposes. In the implementation, these boxes represent the internal mechanisms of the coordination components to be used.

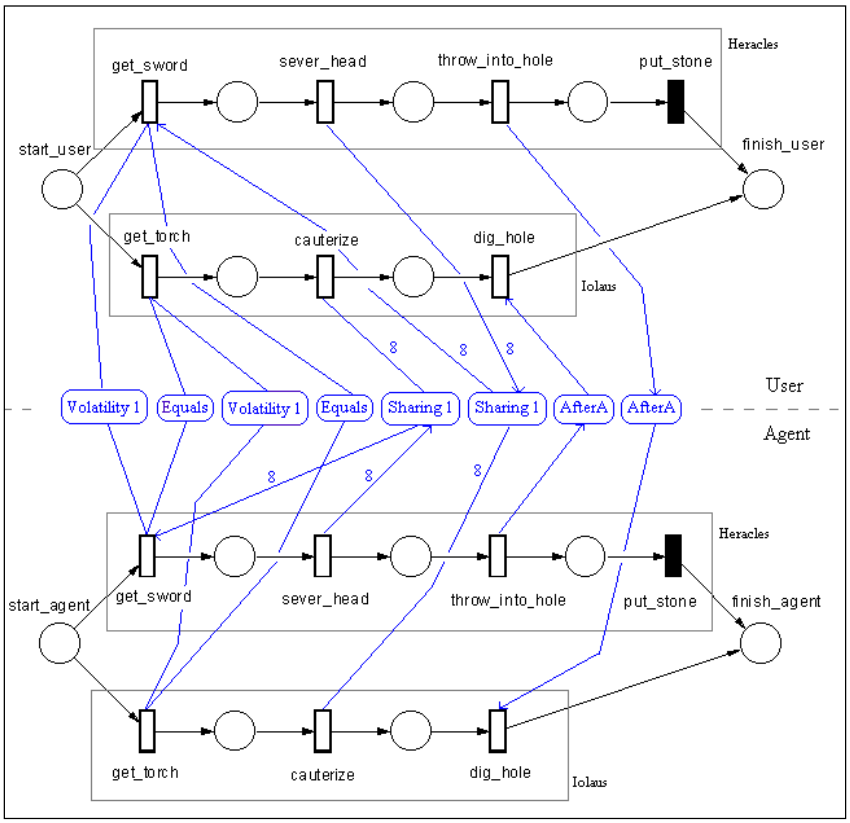

Figure 5: Model of Heracles videogame.

The definition of which actor will assume which role is given by the interdependencies volatility *1* between tasks get_sword and get_torch. Since there are only one sword and one torch available, the weapon's choice determines that each actor will assume a different role. There is also an *equals* interdependency between get_sword and get_torch of different actors, forcing the agent to choose the other weapon when the user chooses his/her weapon.

The interdependency *sharing 1* among the tasks get_sword, sever_head and cauterize is the "core" of the game. When Heracles gets the sword, he is also assigned eight resources that may be thought as "abstract authorizations" to cut Hydra's heads. After he has severed each head, a resource is released, indicating to Iolaus that he may cauterize that wound. If the head wound is not cauterized within a certain period after it has occurred, the timeout signal sent by the coordination mechanism causes the return of the task to its initial state and also reassigns the resource to Heracles, indicating that he must sever that head again

(the head is regenerated). There is also an interdependency *afterA*, indicating that Heracles may only throw the monster in the hole if Iolaus already has dug it.

The videogame was implemented using the blaxxun Contact [Blaxxun 00], a client for multimedia communication that provides resources for VRML (Virtual Reality Modeling Language) visualization, chats, message boards, avatars, etc. In this implementation we used the VRML visualization and the avatars with pre-defined movements.

The interaction with the user occurs by means of buttons defined in a Java applet that interacts with the VRML world via EAI (External Authoring Interface), an interface that enables external programs to interact with objects of a VRML scene. By clicking on the applet's buttons, the user orders the execution of a task in the virtual world. Therefore, the coordination mechanisms act on the interface's buttons, enabling or disabling them if their respective tasks are enabled or not. In order to give more dynamism for the game, the agent has an aleatory behavior, taking a variable time to start the execution of the tasks imputed to it. For example, when the user assumes the role of Heracles, the agent (Iolaus) may not cauterize a head wound before it is regenerated. Figure 6 shows some frames of the videogame (the nine heads of Hydra are represented by nine monsters). Frame *a* shows Heracles' interface and frame *b* shows Iolaus' interface.

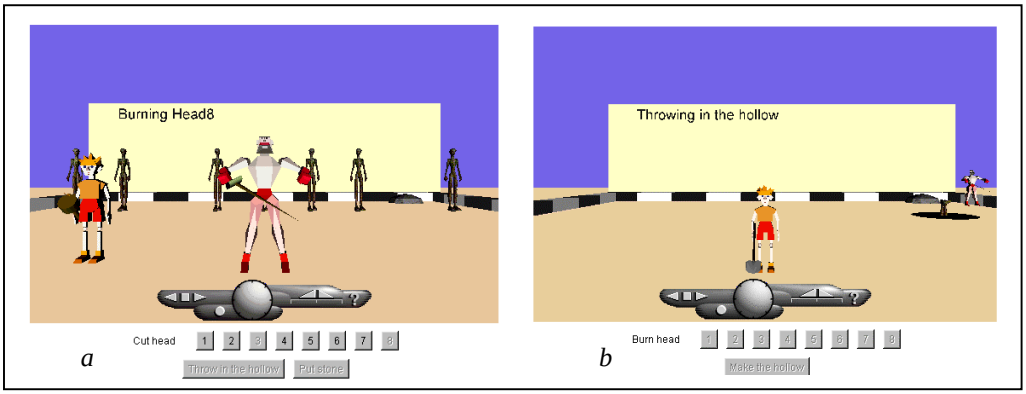

Figure 6: Frames of Heracles videogame.

5. Conclusion

The main goal of this work was to create facilities for the design and implementation of VEs. Due to the threefold infrastructure, we established the basis to intertwine the abstract coordination model with the mathematical formalism and the executable code. Concerning the executable part, the choice for software components and for structuring them in a framework incurred in two main benefits, the concrete realization of the coordination layer as idealized, and the flexibility to change the coordination logic that was initially crafted with a fixed mathematical formalism (PNs).

Our coordination approach follows a multi-abstraction levels hierarchy, which has the advantage of isolating the parts of coordination design. For instance, if the VE designer wants to implement a videogame such as the one presented in Section 4, it is not necessary to consider the mathematical model of the system. Designers may simply use pre-defined coordination components, which encapsulate the formal logic of the coordination mechanisms. On the other hand, in initial phases of the design, only the mathematical model may be used, not considering implementation details.

Currently, collaborative VEs have been mainly developed based on a trial and error approach. We believe that the presented infrastructure is a step towards offering a systematic approach for the development of this kind of environment. One of the few work results that propose another systematic approach for that is [Kim 98], which proposed the use of CASE tools for VE's development.

As future research, we plan to use the infrastructure not only for the implementation of more complex VEs but also for the implementation of other kinds of collaborative applications, making the necessary adjustments.

Acknowledgments. A. Raposo is sponsored by FAPESP (00/10247-3), and A. Cruz, by CAPES/PICD. Thanks also to Tecgraf / PUC-Rio for the expressive support.

References

[Albrecht 97] H. Albrecht. Systematic Framework Design, *Communications of the ACM*, 40(10): 48-51, Oct 1997.

[Allen 84] J. F. Allen. Towards a General Theory of Action and Time. *Artificial Intelligence*, 23: 123-154, 1984.

[Blaxxun 00] blaxxun interactive. *blaxxun Contact 4.4*. <http://www.blaxxun.com/products/contact>, Aug 2000.

[JavaBeans 01] Sun Microsystems. *JavaBeans*. <http://java.sun.com/products/javabeans>, May 2001.

[Johnson 92] R. Johnson. Documenting Frameworks with Patterns, *Proc. of OPSLA'92*, pp. 63-76, 1992.

[Kim 98] G. J. Kim et al. Software Engineering of Virtual Worlds. *Proc. of VRST´98*, pp. 131-138, 1998.

[Malone 90] T. W. Malone and K. Crowston. What is Coordination Theory and How Can It *Help* Design Cooperative Work Systems? *Proc. of CSCW'90*, pp. 357-370, 1990.

[Pree 99] W. Pree. Hot-Spot-Driven Development. In *Building Application Frameworks, Object-Oriented Foundations of Framework Design*, M. E. Fayad, D. Schmidt, and R. Johnson (Ed.). Wiley, 1999.

[Raposo 00a] A. B. Raposo, L. P. Magalhães and I. L. M. Ricarte. Petri Nets Based Coordination Mechanisms for Multi-Workflow Environments. *Int. J. Computer Systems Science & Engineering*, 15(5): 315-326. Sep 2000.

[Raposo 00b] A. B. Raposo, L. P. Magalhães and I. L. M. Ricarte. Coordinating Activities in Collaborative Environments: A High Level Petri Nets Based Approach. *Proc. of SCI'2000, Vol. I - Information Systems*, pp. 195-200, 2000.

[Raposo 01] A. B. Raposo, A. L. V. Coelho, L. P. Magalhães and I. L. M. Ricarte. Using Fuzzy Petri Nets to Coordinate Collaborative Activities. *Joint 9th IFSA World Congress and 20th NAFIPS Int. Conf.*, pp. 1494-1499, 2001.

[Schmidt 96] K. Schmidt and C. Simone. Coordination mechanisms: Towards a conceptual foundation of CSCW systems design. CSCW, 5(2-3): 155-200, 1996.

[Singhal 99] S. Singhal and M. Zyda. *Networked Virtual Environments: Design and Implementation*, Addison-Wesley, 1999.

[Szyperski 98] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.