Journal of Visual Languages and Computing 21 (2010) 136–156

Contents lists available at ScienceDirect

Journal of Visual Languages and Computing

journal homepage: www.elsevier.com/locate/jvlc

A methodology to specify three-dimensional interaction using Petri Nets

Rafael Rieder [a,*], Alberto Barbosa Raposo [b,1], Márcio Sarroglia Pinho [a,2]

[a] Pontifícia Universidade Católica do Rio Grande do Sul – PUCRS, Faculdade de Informática – FACIN, Avenida Ipiranga, 6681, Prédio 32, Sala 607, CEP 90619-900 Porto Alegre, RS, Brazil

[b] Pontifícia Universidade Católica do Rio de Janeiro, PUC-Rio Grupo de Tecnologia em Computação Gráfica, TECGRAF Rua Marquês de São Vicente, 225, Prédio Belisário Velloso, CEP 22453-900 Rio de Janeiro, RJ, Brazil

ARTICLE INFO

Article history:
Received 5 November 2007
Received in revised form
28 October 2009
Accepted 14 January 2010

Keywords:
Interaction tasks
Petri Nets
Specification
Code generation

ABSTRACT

This work presents a methodology to formally model and to build three-dimensional interaction tasks in virtual environments using three different tools: Petri Nets, the Interaction Technique Decomposition taxonomy, and Object-Oriented techniques. User operations in the virtual environment are represented as Petri Net nodes; these nodes, when linked, represent the interaction process stages. In our methodology, places represent all the states an application can reach, transitions define the conditions to start an action, and tokens embody the data manipulated by the application. As a result of this modeling process we automatically generate the core of the application's source code. We also use a Petri Net execution library to run the application code. In order to facilitate the application modeling, we have adapted Dia, a well-known graphical diagram editor, to support Petri Nets creation and code generation. The integration of these approaches results in a modular application, based on Petri Nets formalism that allows for the specification of an interaction task and for the reuse of developed blocks in new virtual environment projects.

1. Introduction

The development process of Virtual Reality (VR) applications still uses *ad-hoc* modeling and implementation techniques, with very little standardization and almost no formalism. This can be noticed particularly in efforts devoted to scientific applications, leading, in most cases, to code rewriting and hindering application analysis before its implementation.

In order to better understand a VR application, especially its possibly intricate interaction flow, it is very helpful to use some kind of formal method like Petri Nets (PN), the Unified Modeling Language (UML), or Finite State Machines (FSM), which can describe the system's function and components. These methods provide not only a better understanding but also a preliminary evaluation of each phase of the system's operation. Moreover, a model-based description facilitates the automatic generation of the core application code from graphical representations. The model-based code generation approach already has produced interesting results in other domains, such as the development of user interfaces for mobile computing [17,35], and the development of web applications [5,21].

Besides formal specification tools, some researchers have sought to develop taxonomies to document and specify virtual environments (VEs) at an abstraction level closer to the user's view instead of the programmer's concept of the application. VEs use nontraditional devices and techniques for three-dimensional interactions that need to be carefully planned, since different physical cues

* Corresponding author. Tel.: +55 51 3320 3611; fax: +55 51 3320 3621.
E-mail addresses: rafael.rieder@pucrs.br (R. Rieder), abraposo@tecgraf.puc-rio.br (A.B. Raposo), pinho@pucrs.br (M.S. Pinho).
[1] Tel.: +55 21 2512 5984; fax: +55 21 3527 1848.
[2] Tel.: +55 51 3320 3611; fax: +55 51 3320 3621.

1045-926X/$ - see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.jvlc.2010.01.002

are mapped during a computer simulation to allow the user's tasks to be performed directly in a three-dimensional spatial context. Some taxonomies seek, for example, to identify the interaction process phases [6], to classify interaction techniques [7,18,25], and to organize the system's control [10]. These approaches split the systems into smaller parts, identifying behavior patterns and allowing to encapsulate them into classes that are capable of executing relevant functionalities. They also allow reusing these classes in other projects and combining them to build a new interaction technique, for example.

The use of both formalisms and taxonomies aims to better define the interaction processes, reducing the time spent for the design and implementation of VEs. Therefore, an integration of both approaches can join the best of each: system specification according to the user's level of expertise, evaluation in the early stages of the development process, and detailed information on each phase of the software development process.

This paper describes a methodology that supports the design and implementation of software modules, which represent the interaction process phases. Our methodology integrates three modeling approaches: PN formalism [19], the interaction technique decomposition taxonomy created by Bowman and Hodges [6] and object-oriented programming concepts. The combination of these elements allows for the description of interaction tasks, the sequence of interaction processes being controlled by PNs with the codes generated automatically.

The PNs are used to graphically and formally represent the VE behavior patterns, based upon the phases of the interaction process according to Bowman's taxonomy. The adoption of these approaches provides a model that can be easily coded using a set of C++ classes. A PN simulator is used to control the program's execution flow.

The choice of PNs to specify tasks in VEs emerges when we start using Bowman's methodology because the interaction tasks can be easily understood as transitions, while the states reached by the application can be understood as places in the PN. With this approach, the definition of independent modules to represent the system's functionalities is straightforward. The logical separation into modules is important, for example, to develop interaction technique frameworks, or to facilitate an automatic code generation from a tested and validated model.

By integrating a taxonomy for interaction techniques, the formalism of PN and automatic code generation, the present work addresses the entire development cycle of a three-dimensional interaction. This cycle begins at the design stage, based on Bowman's interaction taxonomy, then moves to validation and debugging using PN simulation, and ends up with automatic code generation. This approach is different from existing works because the design process is focused on the user's view and on task decomposition, which allows analyzing the system in different levels of detail, making the communication between development teams and end-users more effective during the entire project. Existing approaches are specialized in specific parts of the cycle, as will be discussed in the section about related work.

This text is organized as follows: first we present a literature review in Section 2. Section 3 describes the developed methodology. In Section 4 we present a case study applying the methodology, and we show the necessary steps from application modeling to code generation for a VE. This section also discusses the possibility of hierarchical modeling using our methodology. In Section 5, we describe two case studies that illustrate the use of our approach in realistic settings, such as cooperative manipulation and innovative three-dimensional interaction applications. Section 6 presents a brief discussion about the use of the developed methodology, whereas Section 7 describes the goals we want to achieve with future work. Section 8 concludes the paper highlighting the potential of our approach.

2. Related work

Smith and Duke [28] point out that the lack of formal descriptions during the development process of virtual environments inhibits the identification of similarities among different interaction techniques, leading to the "reinvention" of existing techniques. Furthermore, according to Navarre et al. [20], informal descriptions are prone to ambiguities during the implementation process.

Different mechanisms have been proposed by the VR community to describe and implement interaction techniques, seeking to understand the dynamic behavior patterns of the applications and allowing the standardization of important functions.

This section presents an overview of the methodologies used to specify and implement the interaction process in VEs. Techniques and frameworks used to define the base of this work are briefly mentioned, focusing on the goals, advantages and disadvantages of each one of them.

2.1. Approaches for interaction technique specification

Interaction technique specification is an important task from the perspective of both users and designers. Users require interaction techniques that allow them to complete interaction tasks in a particular application, and designers have to build systems that make the required interactions possible [27]. The existing approaches for interaction technique specification that are more closely related to this work are presented in the following paragraphs.

HyNet [31] is a specification methodology for interaction techniques that integrates three modeling approaches. High-level PNs represent the formal base for the specification, defining the application's semantics and allowing a graphical representation of the application's events (the discrete part of the application). Differential Algebraic Equations handle the continuous behavior pattern of the application, and Object-Oriented Concepts help enhance the expressiveness of the methodology, generating concise and compact models.

Based on HyNet, the Flownet methodology [32] was developed to describe dynamic behavior patterns in VEs.

It provides a different graphical notation that allows for the specification of both the discrete and continuous behavior patterns of the application.

Interactive Cooperative Objects (ICO) is a formal notation for the specification of interactive systems [23]. It borrows concepts from Object-Oriented Programming to describe the structural or static aspects of systems, and uses high-level Petri Nets to describe their dynamic aspects. According to the authors, the specification created using ICO can be simulated, providing the possibility to quickly prototype and test an application before it is fully implemented.

The above specification approaches provide systematic methods for interaction technique design, testing and refining, facilitating the description of systems. However, neither HyNet/Flownet nor ICO are related to any interaction technique taxonomy or provide any support for the automatic generation of code, which requires a deeper knowledge of the formalisms used, both by the designers and the developers, especially in the implementation phase.

2.2. Taxonomies for interaction techniques

We can also view the development process of VEs under the perspective of the interaction techniques used, with the aim of classifying them in order to better understand their components, and therefore the possibilities of software reuse in new applications.

Lindeman [18], for example, presents a taxonomy that classifies interaction techniques according to the type of manipulation technique (direct or indirect), the system actions (discrete or continuous) and the degrees of freedom controlled by the interaction technique. This approach helps identify the parameters involved in each interaction technique, facilitating the development of new forms of interaction.

Bowman et al. [7] present two complementary taxonomies to classify interaction techniques. The first is a metaphor-based taxonomy that seeks to facilitate the user's understanding of the interaction techniques used in a particular VE. The idea is to take a given real action or situation as frame of reference for the user to understand how to interact in the VE. The second taxonomy, based on task decomposition, aims to perform a detailed analysis of the interaction process. According to the authors, the separation of tasks into simpler modules allows each of them to be analyzed and tested independently, serving as a tool for evaluating the usability and effectiveness of an interaction technique in a particular context or VE.

Another advantage in the use of a task-decomposition taxonomy is the possibility of reusing or combining interaction technique components in new projects. Besides flexibility, this feature allows for the conception of new techniques that are adequate for specific situations. Taxonomy and categorization are good approaches to understand the details of interaction techniques and to formalize the differences between them [6].

Csisinko and Kaufmann [8] present an approach to standardize the development of three-dimensional user interaction techniques. The authors propose to implement the techniques directly in the driver that controls the tracking device, using Python scripts as an extension of OpenTracker Framework [26]. This solution uses a variation of the Bowman's taxonomy, introducing an orthogonal property to describe the level of support provided by the driver: full, partial or not implemented in the tracking middleware.

According to the authors, this script-based approach enables testing, studying and evaluating new techniques without changing software components already done, in contrast with the monolithic or even modular approaches. A central repository of interaction techniques is being created to ease the use of this approach in different setups. The authors also suggest how this approach could be used to decouple the menu system control from the application with the final goal of helping to establish standards for three-dimensional interaction. However, this solution depends on the resources provided by a specific framework, which makes the implemented techniques hard to reuse in other platforms.

One way to exploit the use of standardization in VR projects, independently of device or platform, is to integrate code generation into the system model, implemented in design time. Thus, the model preserves the particular features of each interaction technique, allowing its components to become independent from system characteristics, such as programming language, devices or framework.

2.3. Frameworks and tools for code generation

VR frameworks have the purpose of separating functions into modules, allowing for the abstraction of the complexities of some system actions, as well as the reuse of these software modules in different projects.

Figueroa et al. [14] propose an interaction technique architecture development based on pipes and filters, where information sources such as physical devices generate a flow of data that are propagated through interconnected filters. This work presents the InTML markup language, based on X3D,[3] to act as front-end for VR development libraries. Using this methodology, interaction techniques can be built and used as external components, independently from the application. This approach allows for the integration of existing interaction techniques and the creation of new ones, facilitating software reuse and reducing the VE's complexity.

Similar to InTML, the Unit framework [22] also inserts an abstraction layer between the applications and its devices, and introduces application units into a data flow. Each unit has many different properties and can be interconnected to many other units. Moreover, the framework also allows the replacement of interaction techniques at run-time, as they have been specified during the project phase.

The code generation processes, offered by InTML and Unit, result in interpreted code, which may jeopardize the

[3] Web3D Consortium is available at http://www.web3d.org.

quality of the interaction if the hardware does not support the application demands. Moreover, InTML and Unit only provide code generation in Java, while most graphics/scene graph libraries used by VR applications are written in C/C++. This characteristic restricts their use to some specific libraries for Java.

Wingrave and Bowman [33] propose an architecture based on hierarchical state machines named CHASM, which is responsible for the communication between the designer and the programmer and between the programmer and the three-dimensional interface, making it easier to manage and reuse code. According to the authors, brief descriptions of interaction techniques (named "concepts") are used to compose complex interaction tasks. These concepts can be integrated into other concepts, which the authors call "component concepts", without altering their descriptions. This way, techniques like the Virtual Hand can be used together with other techniques without having to be coded again. Toolkits such as HsmTk [2], for example, provide mechanisms to graphically manipulate these concepts, allowing designers and programmers to identify different tasks associated with an interaction technique. However, we have not found an evaluation of these resources for immersive virtual environments.

Arnd Vitzthum [30] presents a visual design language that focuses on the formal pre-implementation specification. His solution, called SSIML (Scene Structure and Integration Modeling Language), is an extension of the UML that allows specifying the three-dimensional world structure and the integration of application logic, associating UML classes and three-dimensional scene elements. This approach also provides automatic code generation, parsing the models to programming languages such as Java, XML, and VRML97.

In tests with small Augmented Reality applications, the SSIML was compared with traditional development techniques and was considered an asset for task-focused domains, generating less code than the traditional implementation. According to the author, the model-driven implementation encourages a structured development, but previous experience with software engineering principles and visual modeling languages is essential. The use of a simple model-driven approach composed of a small number of graphical components, for example, could make the use of this kind of specification easier in projects in which non-expert users are involved.

Figueroa et al. [13] present a conceptual model of user interface components which allows to generalize user interface components and to port them to different hardware settings and application contexts. Based on two previous specification frameworks (InTML for interaction techniques [14], and Contigra for three-dimensional widgets [9]), this model focuses on the standardization of the three-dimensional user interface development, facilitating the process of reusing and documenting Three Dimensional Interface Components (3DICs) in a uniform and extensive way. For this purpose, the authors are developing an XML-based specification language called Interface Component Description Language (ICDL) able to describe these interface components. An online repository[4] already shows the first versions of the 3DICs specifications, but the authors comment that the proposed model and language are still being refined. However, this model requires a parsing tool integrated to the application, which could interfere in the quality of interaction if the hardware does not support the system requirements.

Using a different perspective from the previously presented works, Ying and Gračanin [34] aim to understand the interaction process of existing VR applications. Analyzing an existing code, the information related to the user interaction is extracted and organized in an XML file that serves as a base for building a PN model representing the target application. By simulating the PN, one can "view" the interaction process through the PN behavior pattern while the user is interacting with the VR application. Nevertheless, this approach is restricted to the test phase and does not contemplate previous steps of the software development process, because reverse code engineering is used to create description files. Moreover, this idea is limited to specifying non-immersive virtual environments written in VRML/X3D languages. More complex VEs written in C/C++ or Java languages, for example, cannot be described or projected.

Despite that, the use of a formalism like PNs before and during implementation allows each module of a system to be analyzed both separately and in conjunction. Components could be built and reused in new projects, avoiding rework and the occurrence of errors. The behavior of the system also could be visually displayed. The result of this process would be a coherent and structured application, with interconnected modules that keep the system organized and controlled.

3. Methodology

From the literature review, it can be noticed that the approaches presented have specific advantages and goals. However, in general, they do not address the entire development cycle of computer applications, especially concerning the final phases of debugging and code generation. Aiming at this goal, this project proposes a methodology for the hierarchical development of a VR interaction process using Petri Nets, beginning at the design stage, based upon Bowman's interaction taxonomy, up to the implementation phase, relying on the Object-Oriented Programming paradigm.

The main purpose of the proposed methodology is to design and implement modules that represent the steps of the interaction process. The use of formalism in conjunction with an interaction taxonomy allows for a detailed specification of the system, as well as facilitates the structuring and implementation process, encapsulating functionalities. These characteristics enable the generated modules to be tested in advance and reused later, simplifying and accelerating the development of VEs.

Our methodology uses PNs because they are a well-known formalism and have many variations. Petri Nets

[4] 3DICs is available at http://w3-mmt.inf.tu-dresden.de/3dic/.

are a graphical notion and at the same time a precise mathematical notion [12]. According to David and Alla [11], it is a graphical and formal modeling method used to describe and analyze systems with parallelism, concurrency, synchronization and resource sharing, among which we can include VR applications. Its graphical representation, composed of four basic elements (places, transitions, arcs, and marks or tokens), allows a person without advanced knowledge of VEs to easily understand the functioning of the system, and outlines the desired behavior patterns in a new environment. In other words, a PN represents the application states as places, and state changes as transitions [19]. Transitions have input and output places representing the pre- and post-conditions of the events. Arcs are used to connect places and transitions, directing the sequence of actions. Tokens indicate that data items or resources are available to satisfy conditions.

Moreover, the power of expression of the formalism combined with the existing tools for simulation and analysis allows PNs to be used to preview and test application behaviors before actually implementing them. Formal description techniques make it possible to describe systems in a complete and unambiguous way, thus allowing for an easier understanding of problems between the various subjects taking part in the development process. Analysis methods, such as place invariants and reduction techniques, can be used to verify PN properties, such as liveness, a desired property in VR systems because it can prove the absence of deadlocks. However, in this work we are focused on simulation tasks (program debugging and execution), as a first step to establish our methodology.

Compared with other representation forms, such as UML and FSM, PNs also support the modeling of conflicting actions, a common feature in computer games and Collaborative Virtual Environments, where two or more users may try to manipulate a specific object at the same time. In these kinds of applications, there is generally a priority rule that guarantees one of the users the exclusive access to the object.

Like FSM, PNs also allow hiding application details using hierarchical representations, which abstract low-level details. However, the use of PN graphical tools facilitates the visual understanding of the system, because the diagram can be represented by marks that indicate the execution flow. This visual feature helps different kinds of users understand the application's behavior patterns. PNs also allow expressing parallel activities or concurrency of an application [19], such as a simultaneous interaction over the same object or task manipulated by two or more users inside a VE. Moreover, FSMs can be represented by a subclass of PNs.

The graphical representation adopted here is based on Colored Petri Nets (CPNs) [15]. During the modeling process, we need a quick and easy way to differentiate the various types of data that are manipulated in a VR application. Moreover, during the development process, we need an intuitive way to simulate the interaction process in order to allow different software modules to interconnect and communicate, exposing, in example, the behavior of an interaction technique in the course of a specific interaction stage.

In fact, in the case of CPN models the data types can be represented by tokens with different colors, different patterns or even different icons. Likewise, the simulation provides the user with a tool to investigate the CPN models, testing different interaction technique solutions and checking whether the model works as expected. Our methodology offers a code generation solution that creates an application from a model, where the resulting program runs internally as a CPN, allowing different configurations of modules to be more easily created and simulated.

These extensions are also useful in the PN interpretation process, as long as the different abstraction levels represent specific data sets. Moreover, according to Jensen et al. [16], the behavior of a CPN model can be analyzed, either by means of simulation or by means of more formal analysis methods, which allows to investigate the system design and to verify its properties using an intuitive and complete modeling language. Tools like CPN-AMI[5] and CPN Tools[6] provide resources to simulate a CPN model and to test its syntax and semantics. For simplification, this work refers to CPNs as Petri Nets (PN).

In the proposed methodology, the use of the Object-Oriented paradigm in conjunction with PN helps specifying the structure of applications, allowing the code to be organized into classes. This allows for code reuse and, consequently, the reduction of time and effort during the development phase.

The methodology presented in this paper aims to approximate the designer and developer's conception of the application to the user's point of view, modeling the application under the perspective of tasks which the user has or wishes to perform inside the VE.

These tasks can be decomposed into elementary tasks that can be easily identified in most VR applications. These elementary tasks, according to Bowman and Hodges [6], split the interaction process into three phases: selection, manipulation and release. Our work adapts this taxonomy by subdividing the selection phase into selection and attachment. The former represents the indication of the object which the user wishes to manipulate, while the latter deals with the confirmation of this selection. Both provide feedback to the user in order to confirm their execution. The manipulation (positioning and orientation) and release tasks remain unmodified, following Bowman's original conception.

Based on these definitions, our methodology assigns to each element used in the PNs (places, transitions, arcs, and tokens) a specific role or function during the interaction process in a VR application.

Places define the current application state. They have communication channels to receive and to transmit the necessary information to execute the PN. They do not produce any data.

[5] CPN-AMI environment is available on http://move.lip6.fr/software/CPNAMI/.

[6] CPN Tools in available on http://wiki.daimi.au.dk/cpntools/cpntools.wiki.

In order to make, for example, the selection task accessible to the user, the place which represents the selection state must have at least the pointer and the objects that can be selected, and must ensure that the user is not already manipulating an object. Fig. 1 represents this situation using one place and one transition.

Transitions are the PN elements that perform actions, modifying the application's behavior pattern. They represent the tasks in the interaction process. Like places, transitions have communication channels to receive and transmit the necessary information to execute the PN, but they can also produce data and insert them in the network.

An example of this behavior pattern can be the action performed at the exact moment the user confirms the selection of an object. A transition responsible for this task is fired, executing the operation that attaches the object to the selection pointer. Fig. 2 models the *Attachment Task transition*. Once fired, it establishes the *Manipulation State*.

Arcs define the execution sequence of the PN. They are responsible for carrying data between places and transitions (and vice versa), setting pre- and post-conditions for firing a transition or for establishing a state. Consequently, the arcs define the order of execution among tasks in the VE. It is important to point out that a place or a transition can never be directly connected to another node of the same type.

In order to manipulate an object, first it is necessary to execute the selection and attachment tasks, because they provide the information about the object the user has chosen. Fig. 3 shows this sequence, labeling the arcs with the necessary data for each task.

In this example, the *Selection Task transition* needs the resources (tokens) *pointer object*, *objects available*, and *no selected object* to be placed in *Selection State place* to be fired. For this, an arc labeled with these data connects both elements. Finally, the result of the *Selection Task transition action* (*selected object*) is passed to the *Attachment State*

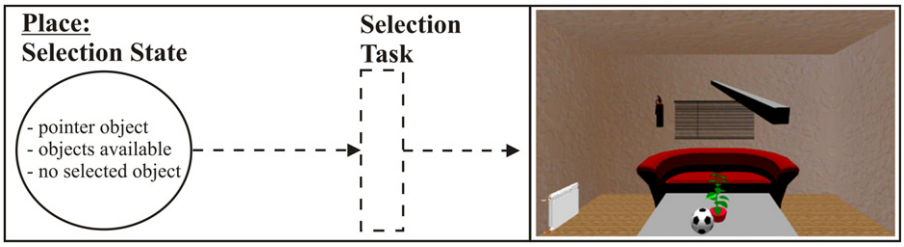

Fig. 1. The *Selection State place* holds the necessary information that makes the *Selection Task* accessible to the user.

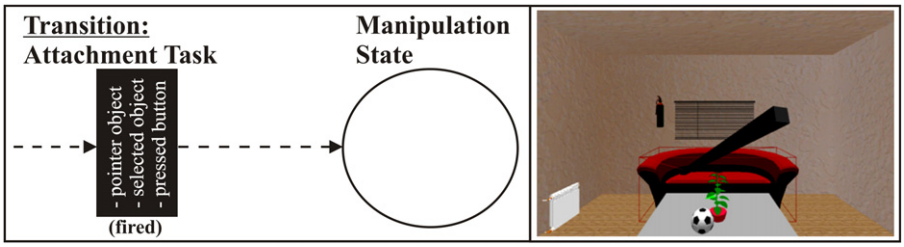

Fig. 2. Transition that attaches the selected object to the pointer. Afterwards the application passes to the *Manipulation State*.

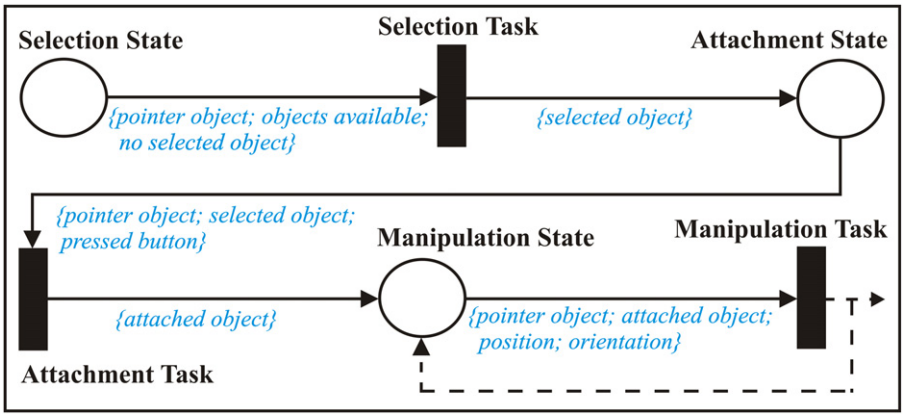

Fig. 3. Arcs between places and transitions defining the order and the resources for each phase of the interaction process.

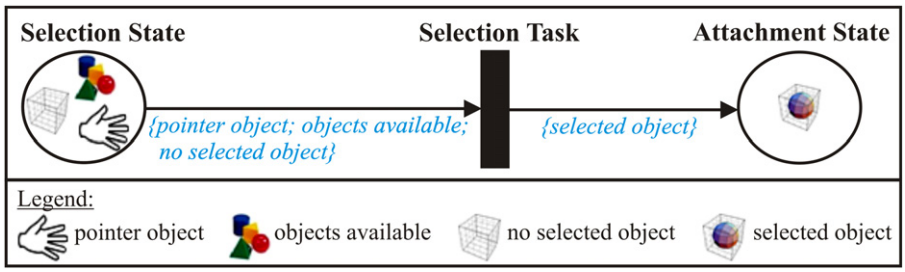

Fig. 4. Objects and their representation as PN tokens.

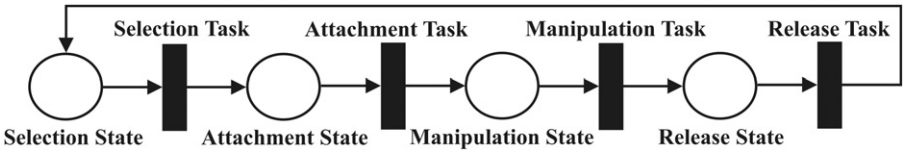

Fig. 5. A high-level PN for the application.

place, which also may have additional tokens (*pointer object* and *pressed button*) received from other sources. The other tasks follow in a similar way, resulting in the object manipulation action.

Tokens are the resources available for executing the application. Application data such as geometric objects, menus, tracking data, button clicks, and so on, are examples of possible tokens. These kinds of resources can come from the application or from devices like the mouse, keyboard or tracker. They can be stored inside the PN's places, and their values can be updated when they go through transitions. In our methodology, one token can be distinguished from another by the type of data they encapsulate. Fig. 4 shows a PN with tokens represented by icons. This graphical representation was adopted simply to facilitate the understanding of the model when it is based on CPNs.

Because the described model is based on PNs, we need to set up initial tokens. The starting state of the devices and the list of geometric objects used in the VE can be considered as initial tokens, representing the preliminary system configuration. Following the PN rules we will always need places to store these initial data.

4. Applying the methodology

4.1. Platform for methodology testing

In order to illustrate the use of our methodology in the specification process of a VE, we built a virtual living room application in which the user's primary goal is to organize the furniture. Initially, the objects appear distributed in the room inside the user's field of vision. There is no set order for the users to arrange the furniture. They use the ray casting technique [4] to interact with the objects.

This application, although simple, presents all the interaction process phases proposed by Bowman, besides allowing for the identification of each state of the interaction process. It was built using C++, OpenGL, GLUT, and the SmallVR[7] toolkit, which simplifies the development of VR applications by abstracting many implementation aspects such as device control and scene graph management, while maintaining the GLUT structure for the program.

The designer needs to follow three steps to apply the methodology:

1. Identify the VE tasks according to Bowman's taxonomy, as well as the main states reached by the application after executing each task.
2. Define a PN with the tasks and states identified in the previous step.
3. Implement the model, using a set of classes specially developed to build the PN and to control its execution.

Each of the above steps is detailed below.

4.2. Interaction identification step

The first step of our methodology identifies the application phases based on Bowman's taxonomy.

The application starts with the *Selection State* (see Fig. 5), in which the user can move the pointer, looking for an object to select. From this point the *Selection Task* tests whether there is a virtual object indicated by pointer. If so, the *Selection Task* transition is fired, and the *Attachment State* is established.

At this point, if the user presses and holds the selection button, the *Attachment Task* is fired attaching the selected object to the pointer and establishing the *Manipulation State*.

Once this state is established the PN fires the *Manipulation Task*, which allows the user to relocate the object using the pointer. In order to simplify the model, we omitted some feedback normally provided by the user.

If the user releases the selection button, the *Release State* place enables the firing of *Release Task*, separating the pointer from the previous selected object.

[7] SmallVR toolkit is available at http://grv.inf.pucrs.br.

4.3. PN model building step

After identifying the application tasks at a high abstraction level, it is necessary to perform a task subdivision process, splitting them into smaller parts (see Table 1) based on the operations each of them has to execute. Fig. 6 shows the new PN configuration.

Following the methodology, the identification of the necessary resources (data) – represented as tokens in the PN – for each interaction phase must be initiated. For this, PN arcs must be labeled with the token types, in this case graphically represented by icons.

The places *Selection State*, *Attachment State*, *Manipulation State* and *Release State* need to be updated with information about the devices and control variables of the application. Therefore, tokens with these data must be transferred to them, as can be seen in Fig. 7, which presents the complete PN model for the application. Using formalism, interaction devices and the application can be represented as source transitions in the PN model, as they do not have input places, being always enabled to fire and to produce tokens for the net (in this case, information about the user's physical interaction and the VE state). However, the token generation and the firing of these transitions are controlled by the application, according to the interaction process stage. This procedure prevents the storage of unnecessary tokens in places. In Fig. 7 the devices are represented by triangles, while the application is represented by hexagons. These shapes are merely illustrative and serve only to help understand the network.

The pace of the PN simulation is controlled by the application, which tests each transition every time the user's view needs to be modified, guiding the execution of the actions. In this step, the simulator fires the transitions that have their pre-conditions fulfilled. Firing a transition generates a call to a function previously assigned to the task. In other words, all the transitions are checked at every rendering cycle and either fired or not, depending on the existence of the pre-conditions (the required tokens). Therefore, it is possible to analyze the logic of the process combined with the system and device behavior patterns.

A complete cycle of the PN execution can be interpreted as follows: initially, the application controls the firing of the source transitions (devices and control variables), checking whether they are enabled to send tokens to the *Selection*, *Attachment*, *Manipulation*, or *Release states*, according to the task being performed. When the *Indication Subtask transition* receives all the necessary tokens from the *Selection state*, a function of the application is fired, unpacking its data (*list of objects*, *pointer*, and *control variable*, which indicates that there is no object being manipulated) and testing for a collision between the user's pointer and one of the objects inside the VE. If there is a collision, the transition creates a new token to represent this object and passes it to the *Indication state*. Once established, this state fires the *Indication Feedback Subtask transition*, which is responsible for calling a function that highlights the object in order to differentiate it from the other objects in the VE.

Immediately, the *Attachment state* receives the token that represents the selected object. This state receives the pointer token and may include the information that the button has been pressed by the user. When all these tokens are present at the *Attachment state*, the *Confirmation Subtask transition* is fired, calling a function that attaches the selected object to the pointer.

The token that represents the selected object is then sent to the *Confirmation state*, which then fires the *Confirmation Feedback Subtask transition*, which calls a function that produces a beep, informing the user of the success of the attachment process. Then, a token encapsulating the name of the selected object is sent to the *Manipulation state*, which defines the start of the manipulation process. This state can store the tokens that represent the pointer, the data that come from the tracker device (considering that the button is still pressed) and the selected object. As soon as the *Manipulation state* receives these tokens, the *Positioning Subtask transition* is fired. This will request a function to update the object's position based on the tracker data received from the *Manipulation state*. After this the transition generates a new token with the name of the selected object, and sends it to the *Release state* as well as back to the *Manipulation*

Table 1
High-level tasks detailing.

High-level tasks	Basic operations
Selection task	• Indication subtask • Indication feedback subtask
Attachment task	• Confirmation subtask • Confirmation feedback subtask
Manipulation task	• Positioning subtask • Repositioning subtask
Release task	• Detachment subtask • Detachment feedback subtask

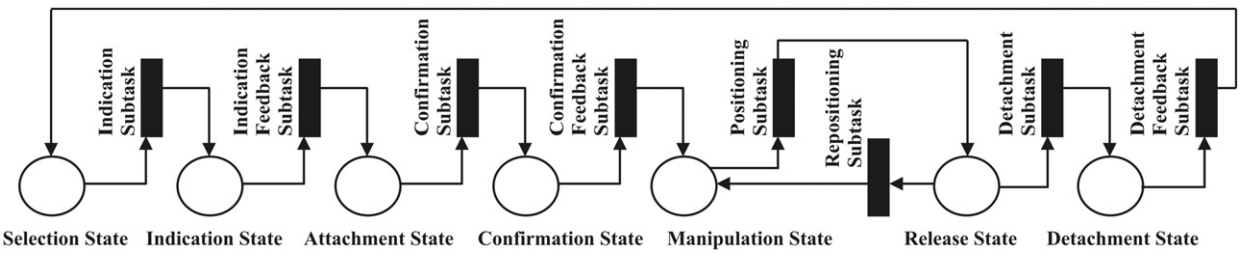

Fig. 6. PN model detailing the interaction process of the virtual living room application.

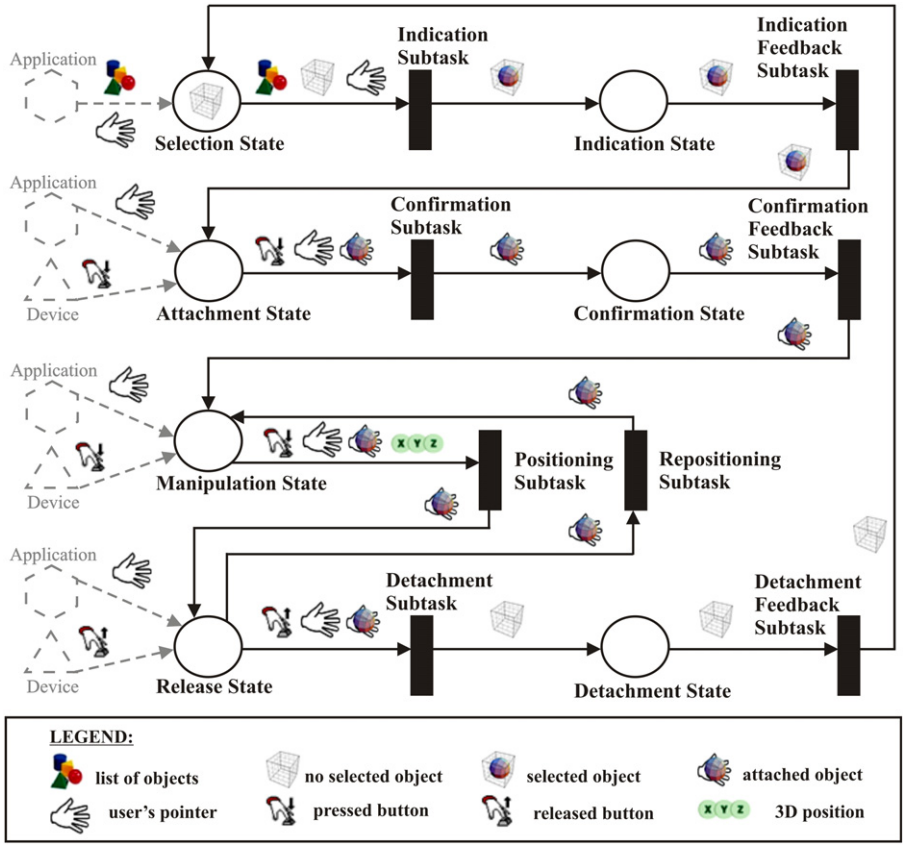

Fig. 7. PN model and data resources for the interaction process in our application.

state, through the *Repositioning Subtask transition*. While the selection button remains pressed, the *Positioning Subtask transition* is repeatedly fired, allowing the user to freely move the object.

If the button is released, the *Manipulation state* will no longer fire *Positioning Subtask transition*. Concurrently, the *Release state* receives a token informing that the user has just released the previously selected object, as well as a token that encapsulates the object's name. As the information about the pointer is available, the *Detachment Subtask transition* is fired. This transition requests a function to release the object in its new position inside the VE. Immediately, the *Detachment state* receives a token that fires the *Detachment Feedback Subtask transition*, which then calls a function that produces a beep informing the user of the success of the detachment process. A token is then sent to the *Selection state*, returning to the initial state and allowing for a new selection to be executed.

4.4. Implementation step

Once the modeling process is concluded, the next step is to generate the application code. In order to facilitate this process, we developed a set of C++ classes, described by the UML diagram presented in Fig. 8. These classes represent the PN nodes and use *signal and slot mechanisms* [3] as the communication tools between places and transitions.

The classes that define places, arcs and tokens can be directly used to instantiate objects. Nevertheless, transitions must be implemented with new classes derived from the abstract class that represents a generic transition. This approach forces the developer to implement the methods that are essential for the model's simulation.

In order to derive the implementation, we start with a graphical description of the PN, created in the Dia[8] editor. From this diagram, an XML specification is obtained which, in turn, originates a C++ code (Fig. 9).

The following subsections intend to detail these three stages illustrated in Fig. 9, presenting solutions and procedures used for each one.

4.4.1. PN graphical modeling

There are different tools to build graphs and other kinds of diagrams, such as Xfig[9], MS-Visio[10], allCLEAR[11], and Dia8.

In our project, we chose the Dia editor because it is an open-source and multi-platform tool, released under the

[8] Dia editor is available at http://live.gnome.org/Dia.
[9] Xfig editor is available at http://www.xfig.org.
[10] Microsoft Office Visio 2007 is available at http://office.microsoft.com/visio.
[11] allCLEAR flowchart is available at http://www.allclearonline.com.

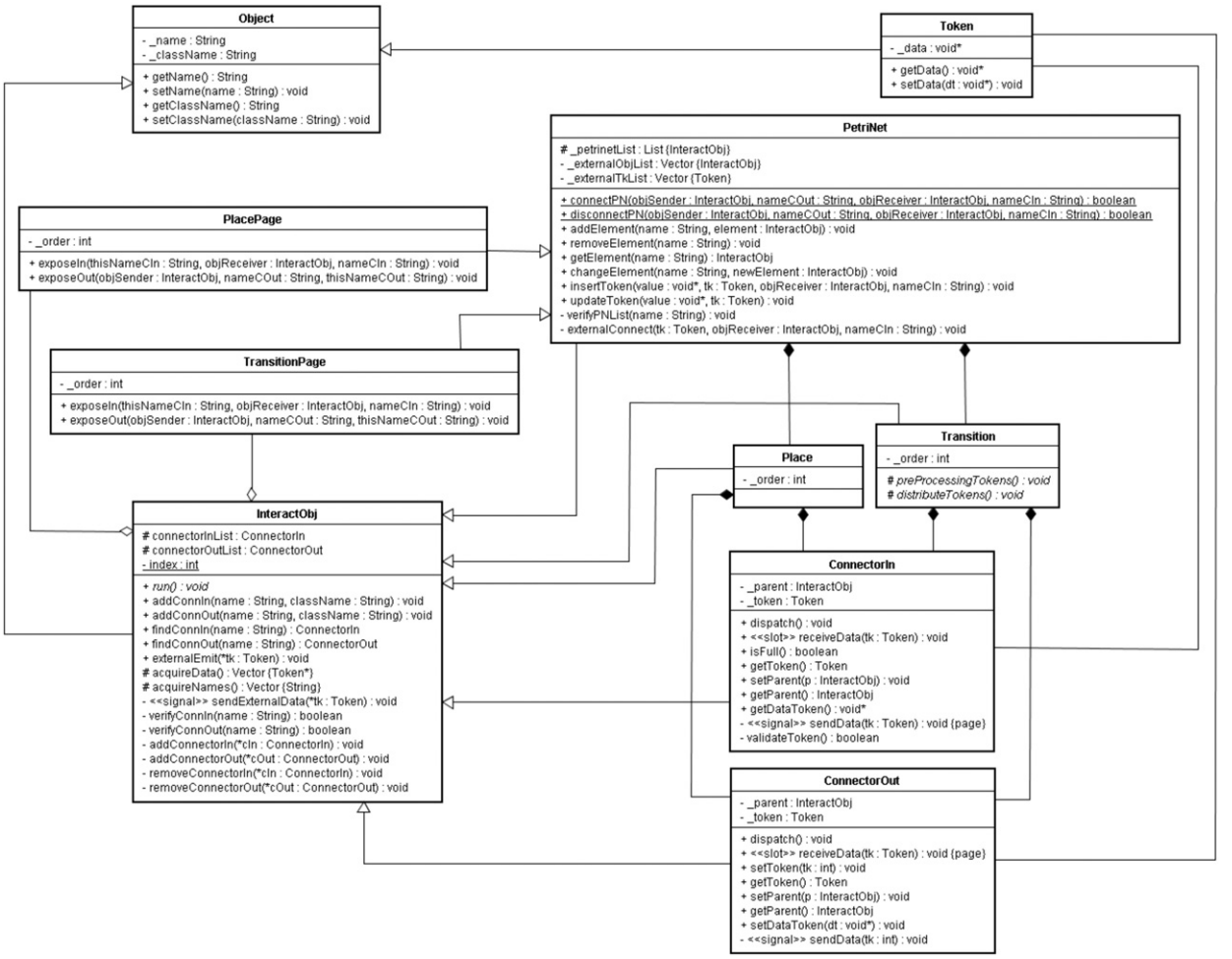

Fig. 8. Classes used to implement the PN model.

Fig. 9. Methodology stages during the implementation step.

GPL License and part of the GNOME Project.[12] Using Dia, it is possible to add support for new types of diagrams by writing simple XML files, thereby creating specific libraries with elementary objects called "shapes".

This feature also allows diagrams to be stored in XML files, facilitating the conversion of models to other codification forms, such as Java and C++ languages, or other markup languages, such as PNML (Petri Net Markup Language) [1]. The flexibility of the conversions allows designers to reuse models in new projects or different development platforms, and to use other tools able to validate the syntactical structure of PNs, since Dia has no native support to formal validation.

In order to support the PN modeling, new shapes have been created to represent the PN elements place, transition, arc and token, organized in a library called Petri Net Interaction Process Diagram. This library, besides facilitating diagram drawing, also enables the generation of C++ code from a PN model. Fig. 10 presents this library incorporated in the Dia environment.

Once the PN shapes have been defined, it is also necessary to establish permissions and restrictions for each PN element, according to our methodology. This guideline requires the designer to establish names for places and transitions, to use arcs to connect them, to label these arcs, and to annotate tokens with their respective data types. Therefore, the definition of this shape library also includes the specification rules established by the class diagram, as presented in Fig. 8. In doing so, the models will include a specification of the functioning of the system during the code generation process. Even so, the designer still needs to know basic PN concepts and rules to analyze and improve the models.

[12] GNOME project is available at http://www.gnome.org.

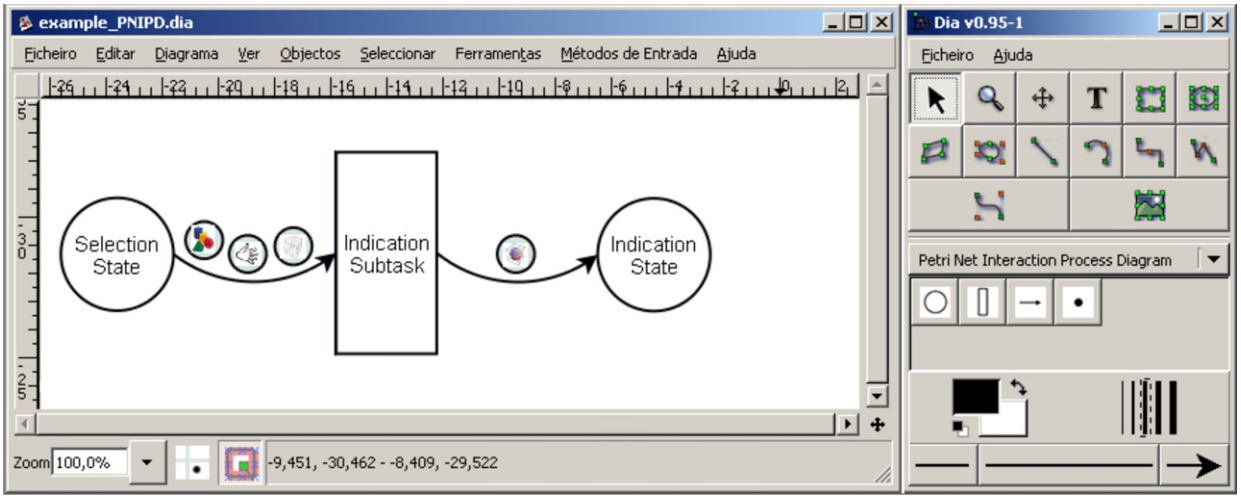

Fig. 10. Petri Net interaction process diagram sheet loaded inside the Dia environment.

4.4.2. XML specification

As said in the previous section, Dia allows loading and saving diagrams in XML format.

An important advantage of these XML documents is that they can be easily transformed into other text formats. Therefore, Dia uses an XSLT (eXtensible Stylesheet Language Transformation) file format, which allows converting XML tags to construct a programming language, such as C++.

We therefore created two XSLT files that facilitate this conversion, using an export option offered by Dia. The first file defines rules to generate an XML file from the graphical diagram, whereas the second contains rules to convert from the PN elements to C++ classes.

Fig. 11 shows briefly how the XML code is interpreted during the C++ code generation process. Based on the transformation rules, the XML file is parsed twice: for generating a PN code to control the application flow and for creating the C++ classes to represent actions and behavior patterns in the interaction process. Since our methodology uses the current XML file that describes the drawn structure (Dia has no native support to PNML code generation), the application flow refers to how the objects are interconnected (direction of arcs) while classes are defined by base classes and transition object properties (tokens to be sent and received). Therefore, it is assumed that the designer knows basic PN concepts and is able to draw consistent models. A conversion to PNML or to other input format for a PN validation tool could be necessary to check the PN consistency.

4.4.3. Code generation

To start the model implementation, the code generation process defines the main application file, creating a function to control the PN. Inside this function, an object from the *PetriNet class* is instantiated (see Fig. 12, line 1). This object represents the entire PN for the application, and is used to run the simulation and to store places and transitions. Moreover, it has methods to encapsulate and retrieve data from the tokens, and to connect the nodes of the PN.

After instantiating the object that represents the entire PN, places, transitions (see Fig. 12, lines 2 and 3) and tokens that will traverse the network (see Fig. 12, line 4) are created. Places are instantiated from the *Place class*, transitions from a class derived from the *Transition class* (*Indication class* in Fig. 12), and tokens from the *Token class*. Places and transitions are also added to the PN (lines 5 and 6).

As described in Section 3, places and transitions exchange tokens through communication channels. These channels are established by linking *connectors* that are added to places and transitions. They receive a name and a data type (lines 7 and 8). After this, to establish communication between two objects, the code generation process explicitly connects them, defining the sender object and its connector, and the receiver object and its connector (line 9). Fig. 13 shows a brief example of the use of connectors in PN elements.

For the transitions it is necessary to derive new classes from the *Transition class* to represent each transition defined in the model. This is the second part of the code generation process. These new classes are composed of three virtual methods, and their attributes are defined by tokens that arrive and depart from each transition.

From the example depicted in Fig. 6 we identify the following new classes and their roles in the model:

- *Indication*: detects the collision between VE objects and the user's pointer.
- *Indication_Feedback*: highlights the indicated object.
- *Confirmation*: attaches an object to the pointer.
- *Confirmation_Feedback*: notifies the attachment.
- *Positioning*: updates the object position.
- *Repositioning*: notifies the new object position.
- *Detachment*: releases the object from the pointer.
- *Detachment_Feedback*: notifies the releasing.

As was the case with the filters defined by Figueroa et al. [14], the abstract class, which originates the *Transition classes*, requires the designer to implement

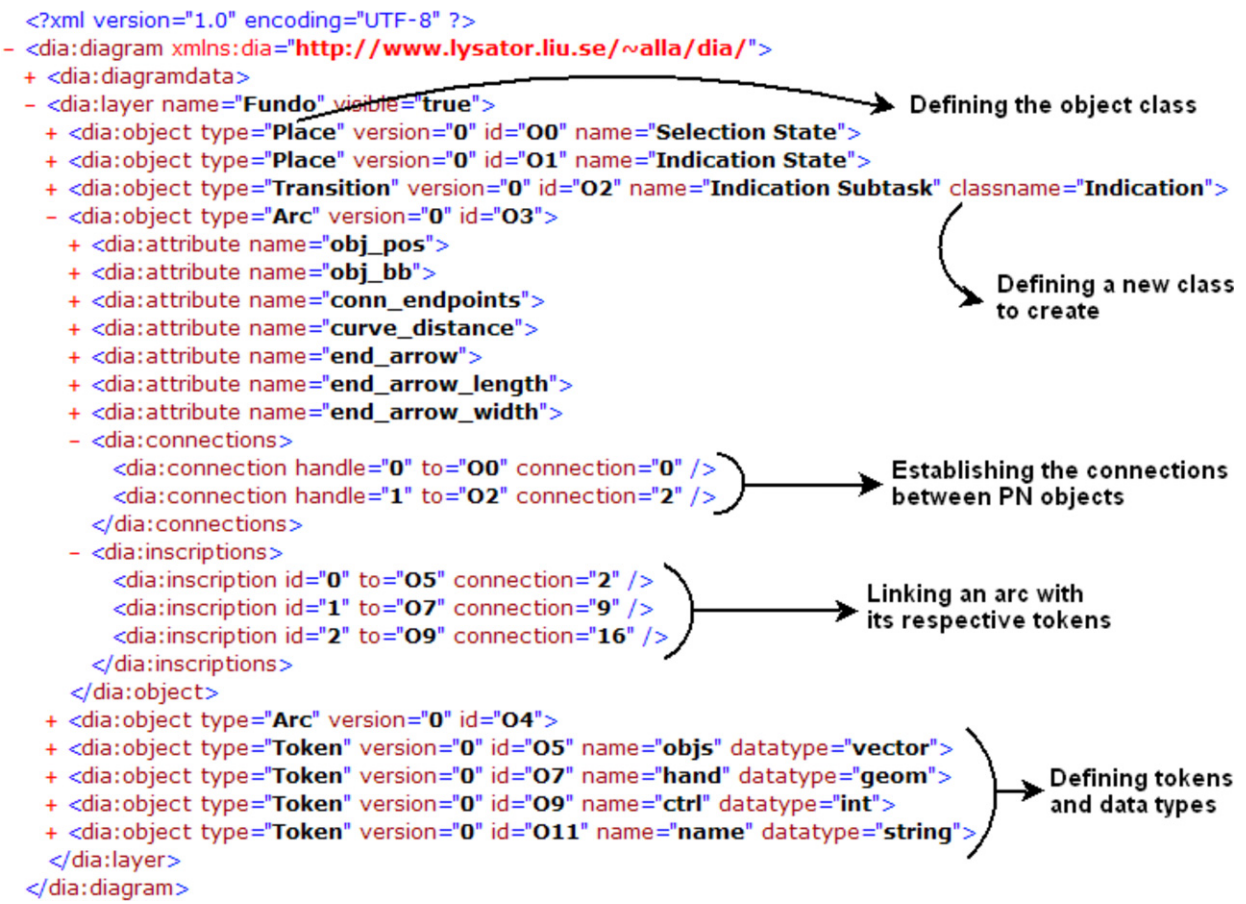

Fig. 11. XML file generated from the model. Its code defines classes to be implemented and determines control connections.

Line	Code
01	`PetriNet *pn = new PetriNet();`
02	`Place *pSel = new Place();`
03	`Indication *tInd = new Indication();`
04	`Token *tkObjs = new Token();`
05	`pn->addElement("Selection State", pSel);`
06	`pn->addElement("Indication Subtask", tInd);`
07	`pSel->addConnOut("objs", "vector");`
08	`tInd->addConnIn("objs", "vector");`
09	`pn->connectPN(pSel, "objs", tInd, "objs");`
10	`pn->insertToken(vecObjs, tkObjs, pSel, "objs");`
11	`pn->run();`

Fig. 12. A simple example of the code generated.

three methods responsible for collecting and distributing tokens and for processing data inside the transitions.

The *preProcessingTokens method* receives tokens, "opens" them and can access the application data. The *distributeTokens method* packs the application data inside a token and delivers them to the PN. Finally, the *run method* allows the designer to insert the application-specific code to do whatever is necessary for application's execution. The *run method* also defines the calls to *preProcessingTokens* and *distributeTokens*. As an example, Fig. 14 presents the *Indication class* codification, defining its methods and attributes (header file) and their use (source file).

The connection between the PN and external elements such as data from devices or application-specific data (like objects, pointers, menus, etc.) can be made with the *insertToken method*, available in the *Place class* (Fig. 12, line 10).

After all these steps, the PN execution can be started by calling the *run method* of the *PetriNet class* (Fig. 12, line 11). In order to guarantee that the application's rendering cycles and the PN simulation are synchronized, the designer needs to call the *run method* at the beginning of every rendering cycle.

Therefore, the code generation process in the Implementation Step can be summarized as follows:

- Derive new classes from the *Transition base class* for representing the VE tasks.
- Instantiate the PN object.
- Instantiate the objects for places, transitions and tokens.
- Add these objects to the PN object.
- Add connectors to places and transitions.

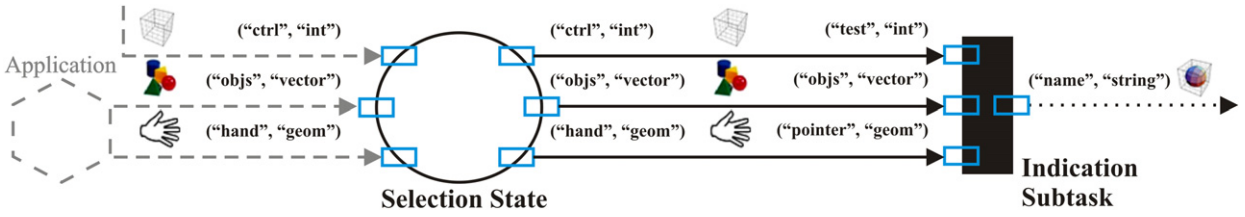

Fig. 13. Connectors of each PN element. The only restriction to link connectors is that the data type must be the same.

a

```
#include "Transition.h"
class ConnectorOut;

class Indication : public Transition
{
  Q_OBJECT

  public:
  Indication() { /* constructor */ };
  ~Indication() { /* destructor */ };
  virtual void run();

public slots:

signals:

protected:
  virtual void preProcessingTokens();
  virtual void distributeTokens();

private:
  vector<char*> _objectsAvailable;
  int _objectControl;
  geom *_pointerObject;  // Specify your geometric type here
  char *_objectSelected;

};
```

b

```
#include "Indication.h"
#include "Token.h"
#include "ConnectorOut.h"
// ============================================================================
// Method...: Indication::run()
// ============================================================================
void Indication::run()
{
  // Clean output attributes
  _objectSelected = NULL;
  preProcessingTokens();
  // ============================================================================
  // Insert your code here
  // ============================================================================
  distributeTokens();
  // Clean input attributes
  _objectsAvailable.clear();
  _objectControl = NULL;
  _pointerObject = NULL;
}
// ============================================================================
// Method...: Indication::preProcessingTokens()
// ============================================================================
void Indication::preProcessingTokens()
{
  string localListNames;
  vector<Token*> localListData;
  Token *temp = new Token();
  localListNames = this->acquireNames();
  localListData = this->acquireData();
  for(int i = 0; i < (int)localListNames.size(); i++)
  {
    temp = localListData[i];
    if(localListNames[i] == "objectsAvailable")
      _objectsAvailable = *(vector<char*>*)temp->getData();
    else if(localListNames[i] == "objectControl")
      _objectControl = *(int*)temp->getData();
    else if(localListNames[i] == "pointerObject")
      _pointerObject = *(geom**)temp->getData();
  }
}
// ============================================================================
// Method...: Indication::distributeTokens()
// ============================================================================
void Indication::distributeTokens()
{
  list<ConnectorOut*>::iterator i;
  Token *tempObjectSelected = new Token();
  tempObjectSelected->setData( _objectSelected );
  for(i = _connectorOutList.begin(); i != _connectorOutList.end(); ++i)
  {
    if( ((ConnectorOut*) *i)->getName() == "objectSelected" )
      ((ConnectorOut*) *i)->setToken(tempObjectSelected);
      // Send new or updated token
      ((ConnectorOut*) *i)->run();
  }
}
```

Fig. 14. Example of a C++ class generated from the graphical representation: (a) Header file (*.h) and (b) Source file (*.cpp).

- Link places and transitions.
- Set up the value for the initial tokens.
- Define where the application will update the PN.
- Execute the PN.

4.5. *Hierarchical modeling*

The hierarchical modeling of an application using a PN allows the system to be described in many different levels of abstraction, thus simplifying the representation and providing different views for the same system. This facilitates its understanding by people with different levels of expertise. Moreover, the possibility of representing some model parts (an interaction technique, for example) as a unitary module can also be a useful way to simplify the model.

As an example, we can have a situation in which it is interesting to group, in a single transition, the entire selection process. Thus, the transitions *Indication Subtask*,

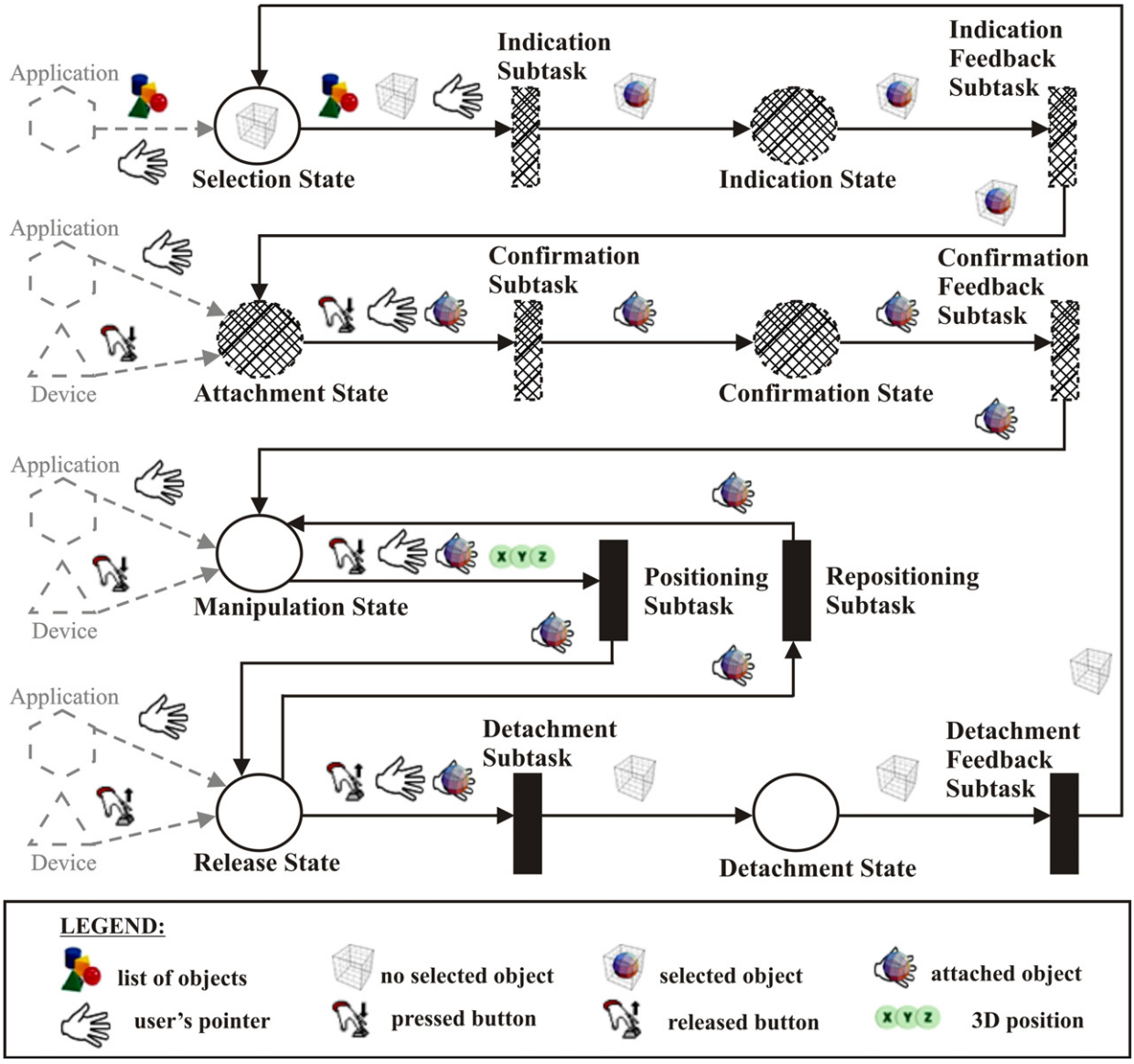

Fig. 15. PN model with the candidate nodes for grouping.

Indication Feedback Subtask, *Confirmation Subtask*, and *Confirmation Feedback Subtask*, and the places *Indication State*, *Confirmation State*, and *Attachment State* can be grouped in a single element. The transitions and places hatched in Fig. 15 show the nodes to be grouped. Fig. 16 depicts the new model with a hierarchical PN. In this case the model interpretation remains almost the same, because the *Selection Task transition* receives the tokens from the *Selection state* and passes the *Selected Object token* to the *Manipulation state*.

In order to represent the group of transitions and places, a new entity called page (or subnet) needs to be defined. In our methodology these entities can be instantiated from the special classes *PlacePage* and *TransitionPage*. The former abstracts a network that starts and ends with a place, while the latter encapsulates a network that starts and ends with a transition.

For this example we used a *TransitionPage* to represent the *Selection Task*. Afterwards, in a new code generation process (updating), this object was added to the PN as was done with other objects. The connections between places and transitions and the definitions of tokens remained the same. However, *PlacePage* and *TransitionPage* classes have two methods to reveal the page connectors and establish the communication between the PN hierarchies. The first of them, *exposeIn*, reveals the values stored in input page connectors to the inner page elements, whereas the second, *exposeOut*, reveals to the outer page elements the values stored in output page connectors.

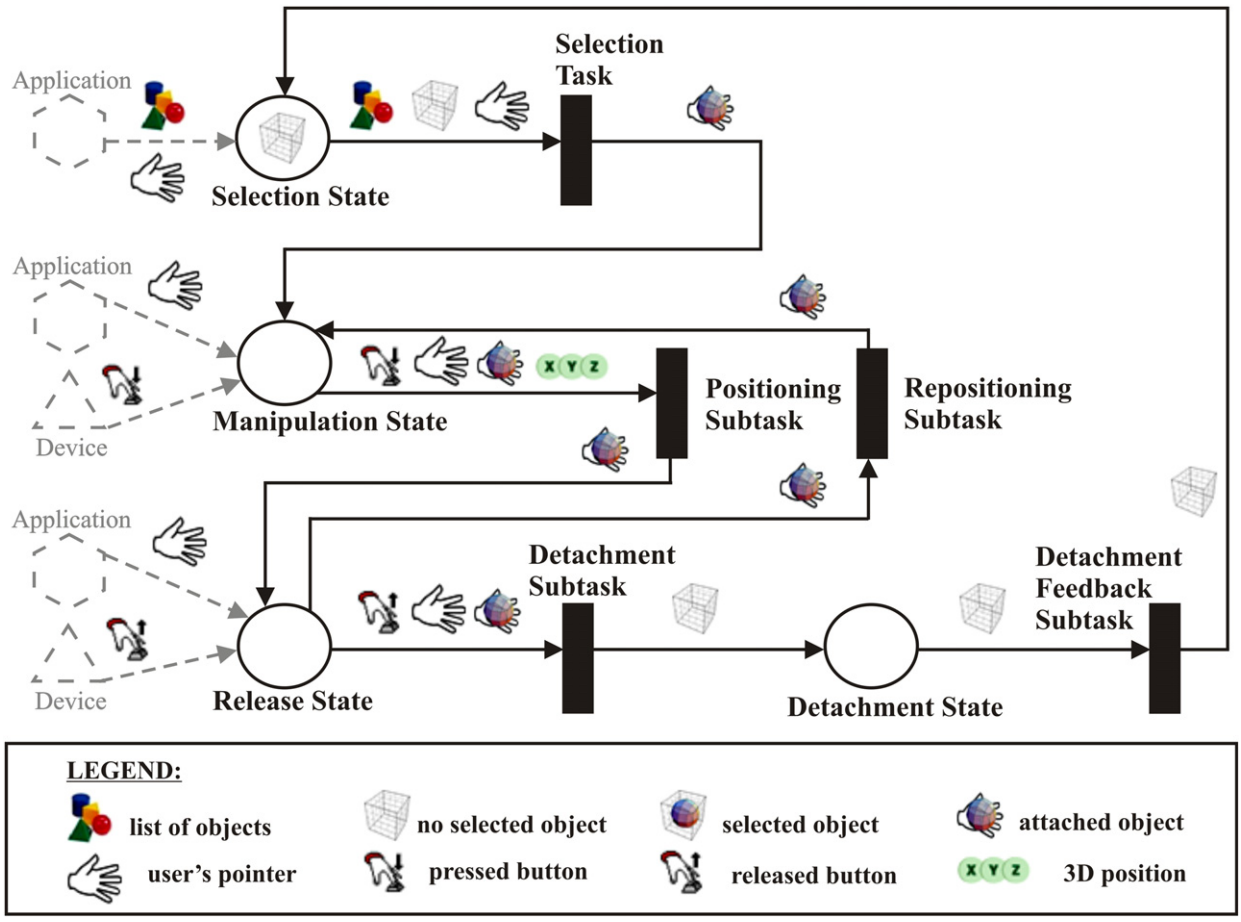

Fig. 16. The PN with *Selection task*, abstracting the model represented in Fig. 15.

5. Case studies

In order to validate our methodology with tasks and techniques applied in realistic settings, we present two case studies that illustrate our approach. In the first case, we apply our methodology to model cooperative tasks, and in the second to model an interaction technique used for selecting objects in sparse or dense environments.

5.1. Collaborative interaction

The work by Pinho et al. [24] presents the concept of "collaborative metaphor" for simultaneous interaction in VEs. This metaphor is composed by a set of rules that define how to combine each step of the interaction process, allowing interaction techniques normally used in individual interaction to be combined in order to compose a collaborative technique. The reason for choosing collaborative interaction as a case study is that the concept of collaborative metaphor requires the representation of parallel activities, being a good opportunity to illustrate the advantages of PN modeling.

The combination of the steps of interaction techniques is obtained by distributing the degrees of freedom to control the objects among the users. This way during the manipulation step, for instance, a user may control the positioning of an object in the X-Y plane, while another user may control its orientation in the *Y*-axis.

We adapted the model of the virtual living room application previously presented in order to support collaborative manipulation tasks. Fig. 17 presents the new model, including two new tokens responsible for the objects' translation and rotation tasks.

Figs. 18–20 present three interaction states during a collaborative interaction between two users. For these examples, we considered the arc label definitions presented in Fig. 17. Each user interacting in the application has his/her own tokens, since the PN represents the behavior of the system as a whole. In that image, tokens representing each user have a specific border color (blue or red), while the shared token representing the object has a different border color (magenta). In this example, the *blue user* is responsible for the object's translation in the 3D space, while the *red user* is responsible for the rotation of the same object.

In Fig. 18, place *Manipulation State* receives the shared object, the tasks that each user may execute, and updated information about the devices and the application. The existence of the tokens enables the net flow to be monitored,

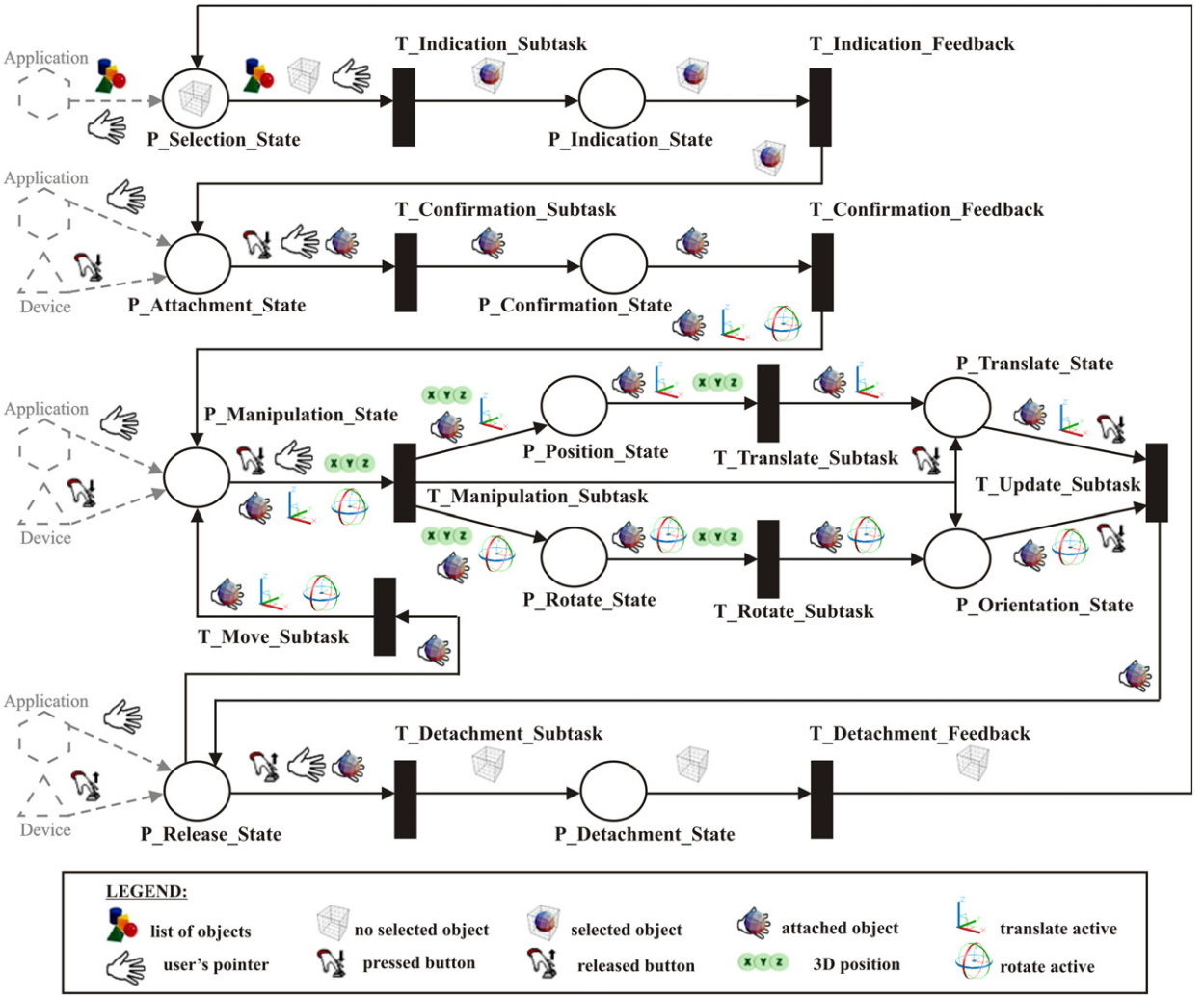

Fig. 17. PN model with added parallel activities: positioning and rotating. Now, "Rotation active" and "Translation active" tokens define the type of task to be executed.

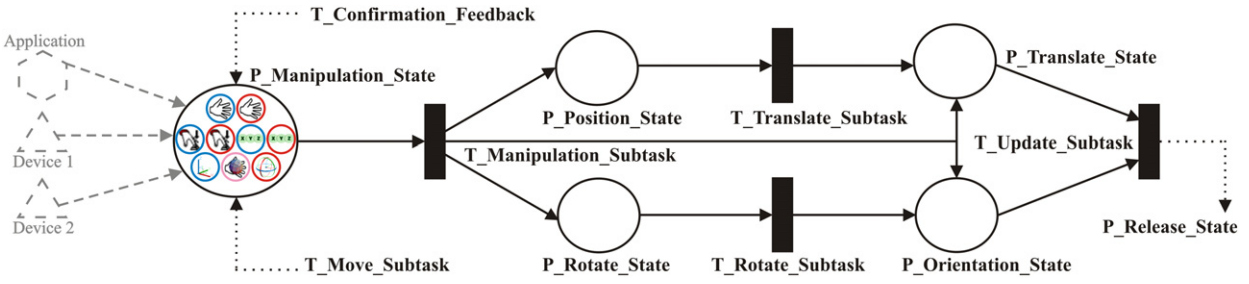

Fig. 18. Beginning of the cooperative manipulation state, where tokens can fire the next interaction tasks, distributing them for each user.

determining the beginning of the positioning and orientation tasks that will be executed in parallel (Fig. 19).

Places *Position State* and *Orientation State* illustrate the distribution of these activities between the users. Note that the tokens responsible for the user's decision to continue or not with the object manipulation (pressed button) are forwarded directly to places *Translate State* and *Rotate State*, as a means to wait for the conclusion of the translation and rotation tasks. Fig. 20 presents the situation in which the concurrent activities

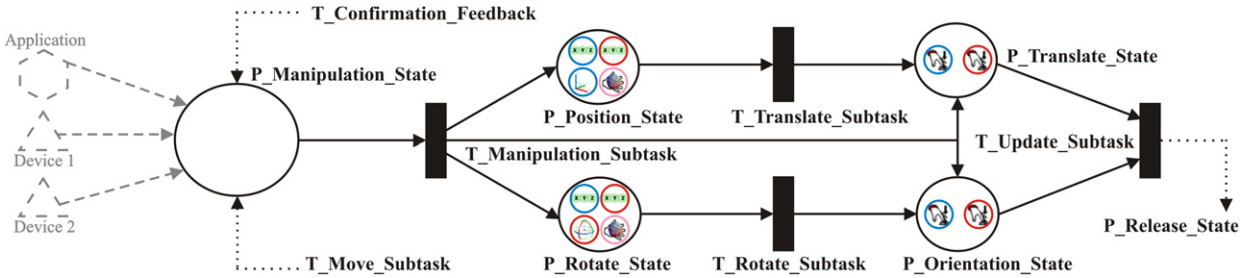

Fig. 19. Second PN state during the cooperative manipulation. The "blue" user may translate the cooperative object, whereas the "red" user may rotate the same object (for interpretation of the references to color in this figure legend, the reader is referred to the web version of this article).

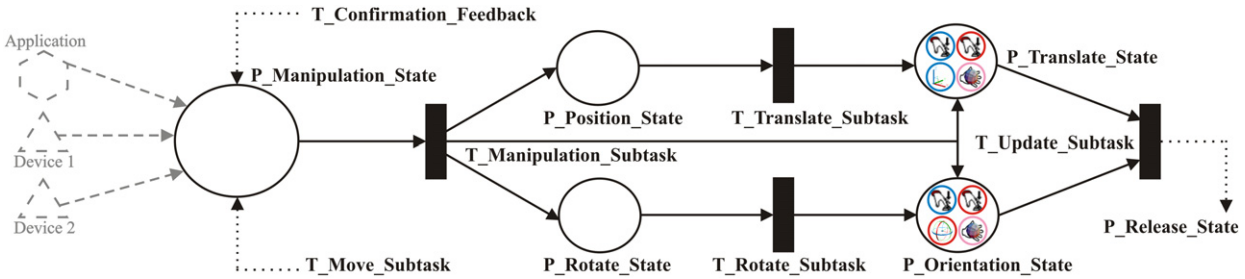

Fig. 20. Parallel activities completed. A new object manipulation can now be started or users can decide to end the cooperative manipulation.

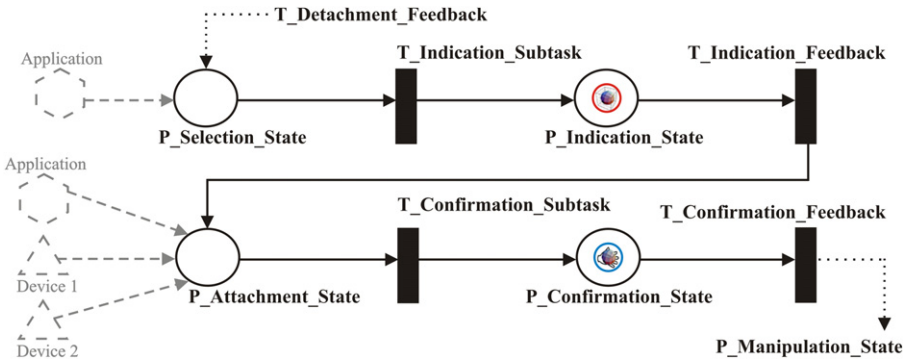

Fig. 21. The same PN model can be used to represent individual interaction in cooperative application. Individual tokens represent the interaction with two different objects, in different tasks (selection and attachment).

are concluded, enabling the continuation of the interaction process.

Still considering the arc labels of Fig. 17, it can be seen that the net also provides support for the representation of individual interaction. Fig. 21 depicts interactions of two users with different objects, in different steps of the interaction process (selection and attachment). Fig. 22 shows two interactions in the same step (release) for different objects.

5.2. 3D bubble cursor

As already mentioned, the subdivision of an interaction process allows for the detailing of the PN in different abstraction levels. A deeper level of detail, for example, enables different interaction techniques to be specified for the execution of each step of the interaction process.

As a second case study, we chose to use the proposed methodology to represent an interaction technique for a specific task. The 3D Bubble Cursor [29] interaction technique was selected. It is a selection tool that uses the hand extension metaphor, enabling the selection of objects in dense or sparse environments. This technique uses a cursor in the form of a semi-transparent bubble capable of capturing the target object within the cursor's volume.

This cursor's volume is dynamically resized to the dimensions of the target closest to its center (indicated by a cross). While the target object is not completely surrounded by the bubble, a semi-transparent spherical cover is drawn around the pointed object. As visual

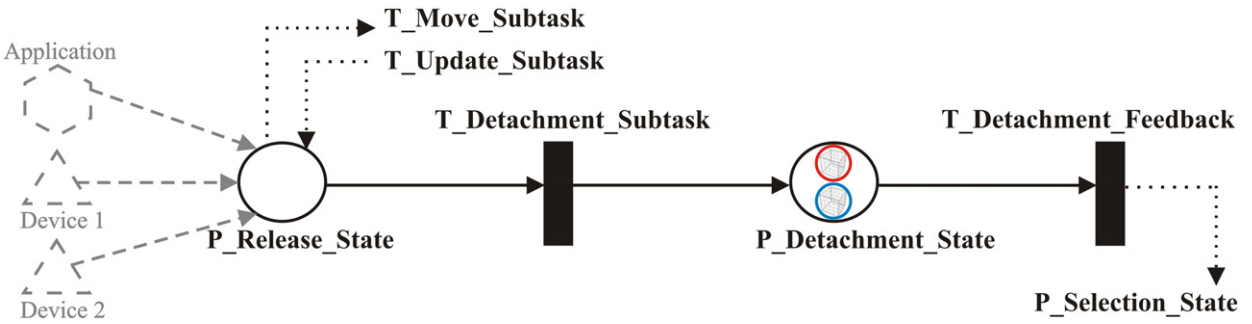

Fig. 22. Individual interaction in a cooperative application. In this case, individual tokens represent the interaction with two different objects, in the same task (release).

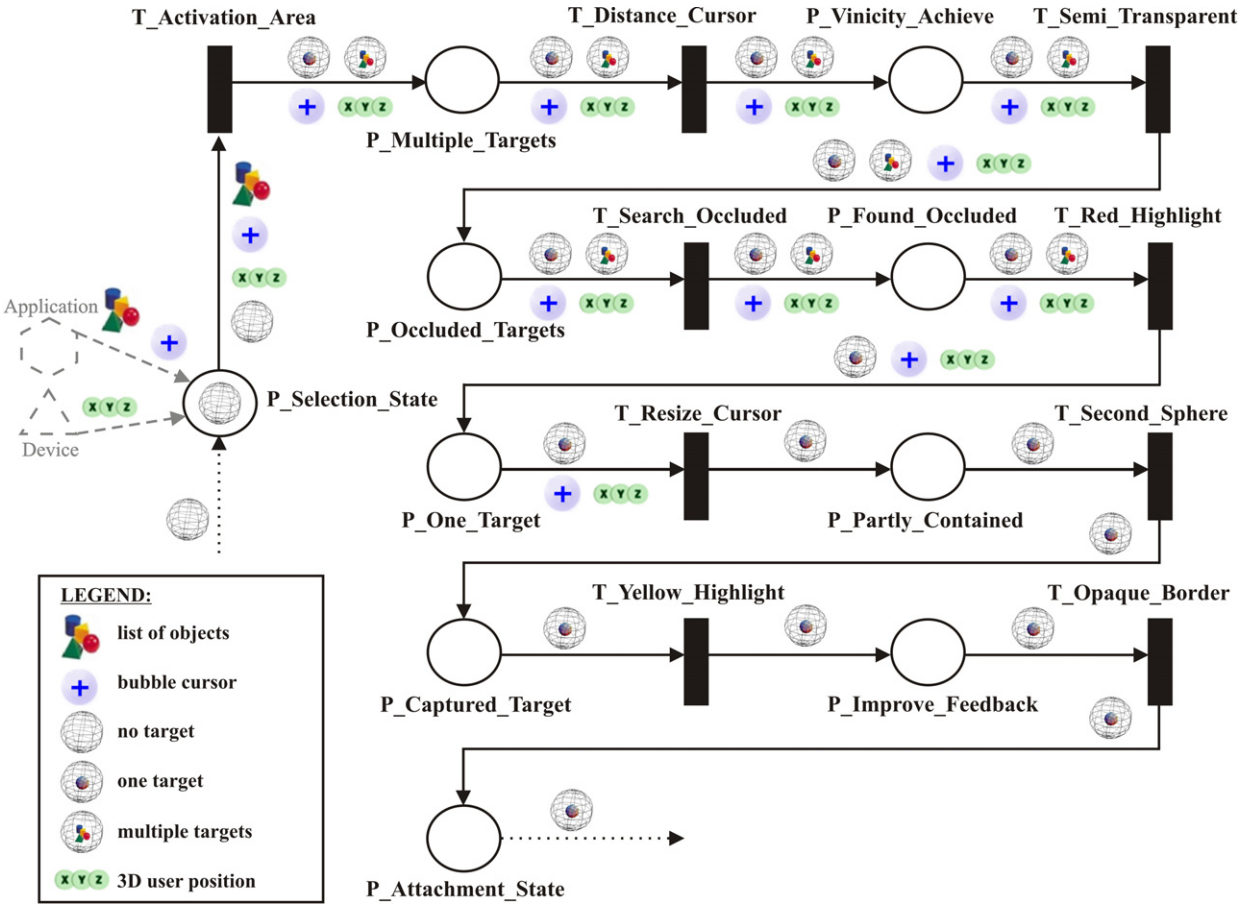

Fig. 23. 3D Bubble Cursor technique expressed in our PN methodology. From this point, it is possible to generate the skeleton code to implement this interaction technique.

feedback, when the object is captured, it is highlighted in yellow and a solid opaque border is applied to it, improving its visualization in the environment. However, the use of a volumetric cursor might cause the undesired selection of multiple objects, since they can be close to one another or occluded. To solve the first problem, the authors suggest that objects adjacent to the target become semi-transparent, facilitating the visualization. In the case of occluded objects, the red color is used together with transparency for visual enhancement.

Considering these characteristics, we modeled the 3D Bubble Cursor technique as an expansion of the tasks and states responsible for the selection process presented in the virtual room example (*Selection State*, *Indication Subtask*, *Indication State*, *Indication Feedback Subtask*, and *Attachment State*). Fig. 23 presents the details of this specification, which can be used for automatic

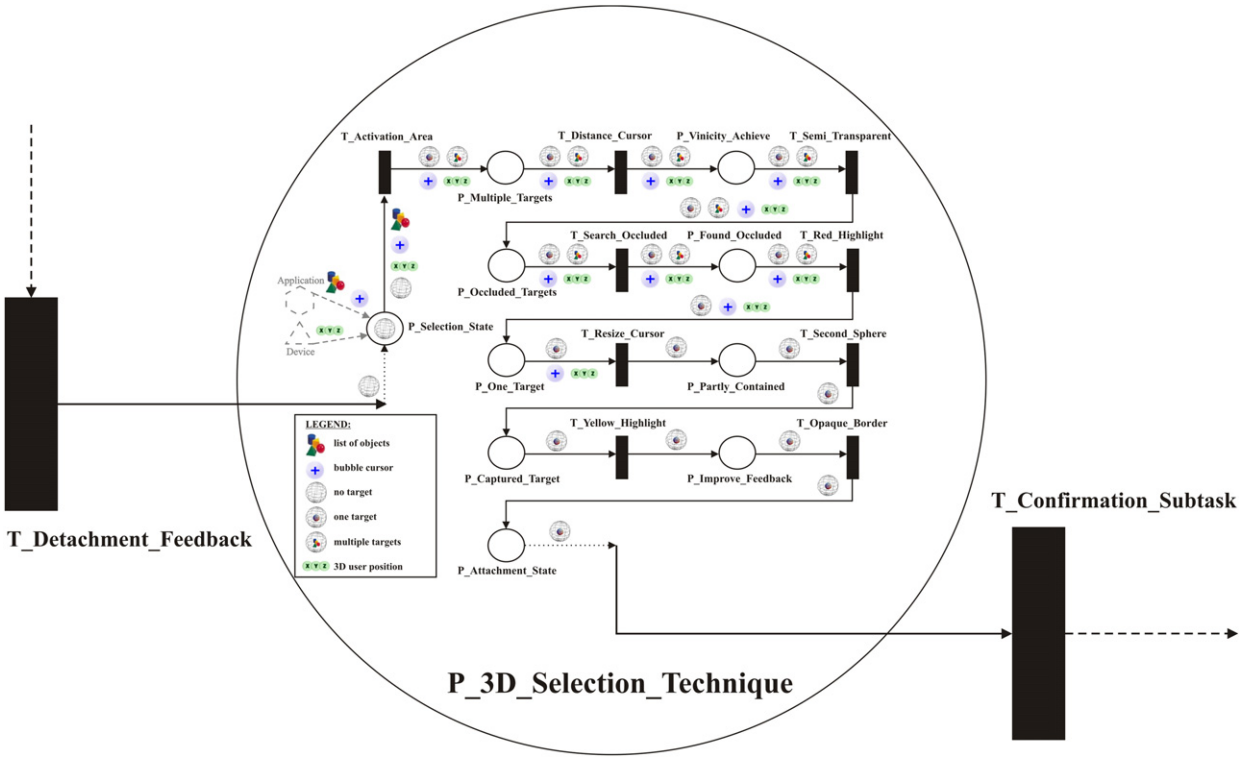

Fig. 24. 3D Bubble Cursor technique expressed in hierarchical model, representing the object selection process in a simplified way.

code generation using the definitions proposed by our methodology. Fig. 24 shows the hierarchical representation of the 3D Bubble Cursor technique represented by a place page.

6. Discussion

Even though we have derived the implementation using a particular VR toolkit as described in Section 4 (Applying the Methodology), there is no dependency between this toolkit and the methodology steps, thereby allowing the use of any other toolkit.

Similarly, there is no restriction on the use of another programming language as default for the code generation process. Our approach has adopted the C/C++ language because the majority of VR applications is developed in this language or uses resources from graphics/scene libraries written in it. However, if the designer needs to export PN models (stored in XML files) to another language, such as Java, he must create an XSLT file containing rules to convert the PN elements into Java classes.

With regard to the first case study, compared with the standard approach, our methodology approximates the application modeling to the user's point of view, allowing effective communication between designers, developers and end-users during the entire development cycle. The representation of the PN interaction process also facilitates the identification of the parts of the system that could be parallel or support similar situations, such as individual and cooperative interaction. This can result in simplification of the system and reduction of design time and development time.

Concerning the second case study, our methodology supports the interaction technique representation at different levels of abstraction, in design time, allowing basic elements to be used in new projects. The PN decomposition technique, combined with the code generation process, allows for the interaction technique features to be elucidated to the developers, who can dedicate their time and attention to improve them and optimize their codes. Moreover, the use of levels of abstraction in projects can also hide technical features from end-users, facilitating the understanding of the interaction technique during their interaction in VEs.

On the other hand, our methodology requires designers to know the basic PN concepts and rules in order to draw and revise their models, since Dia has no native support to PN projects. Depending on the system's complexity, or the development team's knowledge level, a training step may be required to prepare the professionals to design and review the PN model.

This methodology also considers that the software development process takes a top-down approach. The adoption of this strategy facilitates both the use of PNs and the application of our methodology during the interaction technique decomposition, since the top-down approach aims to break down a system to gain insight into compositional sub-systems.

7. Future work

Since the control of the PN simulation stages is autonomous, it would be interesting to show the application running process in a graphical animation superimposed over the PN graph itself, parallel with the application usage. Currently we only generate a textual output during the application's execution. Our intention is to use the XML specification file as input to a PN simulator, parsing the PNML language. By doing this, we intend to present another method to analyze and visualize the behavior pattern of each stage of the interaction process, mainly to solve problems in complex VEs.

However, this graphical animation is only possible if there is a mechanism able to verify the correctness of the PN model. The use of this resource would allow performing PN validation before the automatic code generation and the PNML specification, providing consistency and completeness to the created model.

We are analyzing ways to verify the correctness during the model export process, since Dia is a general graphical editor and does not let the structural analysis take place in design time. This approach could allow for error detection within the editor, avoiding redraws and recoding during run-time and simulation-time stages. As a result, an accurate model could be generated, allowing the specific tools to verify the PN properties, through formal techniques to certify the absence of undesired system behaviors, such as deadlocks. Next, it would be necessary an evaluation session to validate these models, using one of the PN tools to analyze formally the system.

Another interesting idea would be to incorporate our methodology to a VR framework, presenting a complete development platform. Resources for analysis, project, development and evaluation of VE prototypes could be integrated in a single tool, allowing for the detection of faults in project time. With this in mind frameworks, such as VR Juggler,[13] MORGAN,[14] and DIVERSE,[15] are being analyzed, as they already use an extensive set of software modules that abstract devices, avatars and VEs. Our methodology could be adapted to function as one interaction framework integrated to existing resources, becoming an important feature of these tools.

In particular, it would also be important to extend the evaluation of the use of our approach to different VR projects. Designers and developers could contribute with their experience, through the validation of models and skeleton codes, and the identification of situations in which our method offers advantages or difficulties to use. This way, our approach could be improved and adapted to support varying requirements of VR applications.

[13] VR Juggler is available at http://www.vrjuggler.org.

[14] MORGAN is available at http://www.fit.fraunhofer.de/products/morgan_en.html.

[15] DIVERSE is available at http://diverse-vr.org.

8. Conclusions

This work has presented a methodology to specify interaction tasks for VR applications using the Petri Net formalism as a base for the software design.

Furthermore, our methodology aims to facilitate the application development from the conception and design phases to the implementation, test and documentation processes. The option of code generation from the graphical model helps the communication between designers and developers, avoiding the breaking of paradigms, and speeding up the development process. Moreover, an application built with our methodology can offer optimized functions, allowing users to complete their interaction tasks in an intuitive way.

Acknowledgments

This work was partially funded by Tecgraf (Computer Graphics Technology Group). Tecgraf is one of the laboratories of the Computer Science Department at the Pontifical Catholic University of Rio de Janeiro (PUC-Rio) and is mainly supported by Petrobras.

Rafael Rieder and Márcio Pinho are also grateful for the fellowships granted by the Dell/PUCRS Agreement and CAPES – the Brazilian Ministry of Education Agency. Alberto Raposo was also funded by CNPq – National Council for Scientific and Technological Development, Process Number 472967/2007-0.

References

[1] J. Billington, S. Christensen, K. van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, M. Weber, The Petri Net Markup Language: concepts, technology, and tools, in: Proceedings of the 24th International Conference of the Applications and Theory of Petri Nets (ICATPN 2003), Lecture Notes in Computer Science, Springer-Verlag, 2003, pp. 483–505.

[2] R. Blanch, M. Beaudouin-Lafon, Programming rich interactions using the hierarchical state machine toolkit, in: Proceedings of the Working Conference on Advanced Visual Interfaces, 2006, pp. 51–58.

[3] J. Blanchette, M. Summerfield, C++ GUI Programming with Qt 4, Upper Saddle River, NJ, 2006.

[4] R.A. Bolt, "Put-that-there": voice and gesture at the graphics interface, ACM SIGGRAPH Computer Graphics 14 (3) (1980) 262–270.

[5] A. Bozzon, S. Comai, P. Fraternali, G.T. Carughi, Conceptual modeling and code generation for rich internet applications, in: Proceedings of the Sixth International Conference on Web Engineering (ICWE '06), 2006, pp. 353–360.

[6] D.A. Bowman, L.F. Hodges, Formalizing the design, evaluation, and application of interaction techniques for immersive virtual environments, Journal of Visual Languages and Computing 10 (1) (1999) 37–53.

[7] D.A. Bowman, E. Kruijff, J.J. LaViola Jr., I. Poupyrev, 3D User Interfaces: Theory and Practice, Addison-Wesley, Boston, MA, 2005.

[8] M. Csisinko, H. Kaufmann, Towards a universal implementation of 3D user interaction techniques, in: Mixed Reality User Interfaces: Specification, Authoring, Adaptation (MRUI'07), 2007, pp. 17–24.

[9] R. Dachselt, M. Hinz, K. Meissner, Contigra: an XML-based architecture for component-oriented 3D applications, in: Proceeding of the Seventh International Conference on 3D Web Technology (Web3D'02), 2002, pp. 155–163.

[10] R. Dachselt, A. Hübner, Three-dimensional menus: a survey and taxonomy, Computers and Graphics 31 (1) (2007) 53–65.

[11] R. David, H. Alla, Discrete, Continuous, and Hybrid Petri Nets, Springer-Verlag, 2004.

[12] H. Ehrig, G. Juhás, J. Padberg, G. Rozenber, Unifying Petri Nets: Advances in Petri Nets, Springer-Verlag, 2001.
[13] P. Figueroa, R. Dachselt, I. Lindt, A uniform specification of mixed reality interface components, in: Proceedings of the IEEE Virtual Reality, 2006, pp. 295–296.
[14] P. Figueroa, M. Green, H.J. Hoover, InTML: a description language for VR applications, in: Proceedings of the Seventh International Conference on 3D Web Technology (Web3D'02), 2002, pp. 53–58.
[15] K. Jensen, Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use, Springer, New York, NY, 1997.
[16] K. Jensen, L.M. Kristensen, L. Wells, Coloured Petri Nets and CPN tools for modelling and validation of concurrent systems, International Journal on Software Tools for Technology Transfer 9 (3) (2007) 213–254.
[17] S. Lepreux, J. Vanderdonckt, B. Michotte, Visual design of user interfaces by (de)composition, in: Proceedings of the 13th International Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS '2006), Lecture Notes in Computer Science, Springer-Verlag, 2006, pp. 157–170.
[18] R.W. Lindeman, Bimanual interaction, passive-haptic feedback, 3D widget representation, and simulated surface constraints for interaction in immersive virtual environments, Ph.D. Thesis, Faculty of the School of Engineering and Applied Science, George Washington University, 1999 (Director: James K. Hahn).
[19] T. Murata, Petri nets: properties, analysis and applications, Proceedings of the IEEE 77 (4) (1989) 541–580.
[20] D. Navarre, P.A. Palanque, R. Bastide, A. Schyn, M. Winckler, L.P. Nedel, C.M.D.S. Freitas, A formal description of multimodal interaction techniques for immersive virtual reality applications, in: Proceedings of the 10th IFIP TC13 International Conference on Human–Computer Interaction (INTERACT 2005), Lecture Notes in Computer Science, Springer-Verlag, 2005, pp. 170–183.
[21] D. Nunes, D. Schwabe, Rapid prototyping of web applications combining domain specific languages and model driven design, in: Proceedings of the Sixth International Conference on Web Engineering (ICWE '06), 2006, pp. 153–160.
[22] A. Olwal, S. Feiner. Unit: modular development of distributed interaction techniques for highly interactive user interfaces, in: Proceedings of the Second International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia (GRAPHITE '04), 2004, pp. 131–138.
[23] P.A. Palanque, R. Bastide, Synergistic modeling of tasks, users and systems using formal specification techniques, Interacting with Computers 9 (2) (1997) 129–153.
[24] M.S. Pinho, D. Bowman, C.M.D.S. Freitas, Cooperative object manipulation in immersive virtual environments: framework and techniques, in: Proceedings of the ACM Symposium on Virtual Reality Software and Technology, 2002, pp. 171–178.
[25] I. Poupyrev, S. Weghorst, M. Billinghurst, T. Ichikawa, A framework and testbed for studying manipulation techniques for immersive VR, in: Proceedings of the ACM Symposium on Virtual Reality Software and Technology, 1997, pp. 21–28.
[26] G. Reitmayr, D. Schmalstieg. An open software architecture for virtual reality interaction, in: Proceedings of the ACM Symposium on Virtual Reality Software and Technology VRST'01, 2001, pp. 47–54.
[27] S. Smith, D. Duke, Using CSP to specify interaction in virtual environments, in: Technical Report YCS 321, University of York, Department of Computer Science, 1999.
[28] S. Smith, D. Duke, Virtual environments as hybrid systems, in: Proceedings of the Eurographics UK 17th Annual Conference (EG-UK'99), 1999, pp. 113–128.
[29] L. Vanacken, T. Grossman, K. Coninx. Exploring the effects of environment density and target visibility on object selection in 3D virtual environments, in: Proceedings of the IEEE Symposium on 3D User Interfaces, 2007, pp. 115–122.
[30] A. Vitzthum, SSIML/components: a visual language for the abstract specification of 3D components, in: Proceedings of the 11th International Conference on 3D Web Technology (Web3D'06), 2006, pp. 143–151.
[31] R. Wieting, Hybrid high-level nets, in: Proceedings of the 28th Conference on Winter Simulation (WSC '96), 1996, pp. 848–855.
[32] S. Willans, M.D. Harrison, Prototyping pre-implementation designs of virtual environment behaviour, in: Proceedings of the Eighth IFIP International Conference on Engineering for Human–Computer Interaction (EHCI '01), Lecture Notes in Computer Science, Springer-Verlag, 2001, pp. 91–108.
[33] C.A. Wingrave, D.A. Bowman, CHASM: bridging description and implementation of 3D interfaces, in: Proceedings of the Workshop on New Directions in 3D User Interfaces, 2005, pp. 85–88.
[34] J. Ying, D. Gračanin, Petri net model for subjective views in collaborative virtual environments, Proceedings of the Fourth International Symposium on Smart Graphics, Lecture Notes in Lecture Notes in Computer Science, vol. 3031, Springer, 2001, pp. 128–134.
[35] D. Zhao, J.C. Grundy, J.G. Hosking, Generating mobile device user interfaces for diagram-based modelling tools, in: Proceedings of the Seventh Australasian User Interface Conference (AUIC 2006), 2006, pp. 101–108.