Software Architecture and Design of Task Deduction and Task Planning Components for a Multiple Robot Simulation System

John Karigiannis

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfilment of the Requirements
for the Degree of Master of Applied Science
Concordia University
Montreal, Quebec, Canada

July 2000

0-612-54318-8

Abstract

Software Architecture and Design of Task Deduction and Task Planning Components for a Multiple Robot Simulation System

JOHN KARIGIANNIS

Visual simulations of industrial processes (e.g. welding, assembling etc.) involving multiple robots, are part of the overall design process for such systems. A 3D simulation and visualization software tool has been developed named MRS, for Multiple Robot Simulation system. Experiences in this area of industrial robotic applications have shown that there is a great concern about operability and supervision efforts. The goal is to give user confidence in operating such a system in an intuitive manner. In order to reach that, a Virtual Reality (VR) interface is integrated in the existing MRS. In addition, user must be provided with convincing information concerning the conduct of the tasks to be performed. Moreover, the system must demonstrate a certain degree of autonomy regarding the planning of these tasks. In this thesis, we present a software architecture for two components that we believe satisfy these functional requirements that are posed by the specific robotic system. The Task Deduction and Task Planning components presented here are designed and implemented as an integral part of MRS. We analyze the design proposed for the Task Deduction component, which will allow communication between the user in the VR space and MRS. In addition, by employing Petri Net structures, we provide a formal model of representing the tasks the user can deduce in the VR environment. We describe the design of the Task Planning component we believe will increase the system's autonomy concerning the planning of the tasks that will undertake. In addition, in order to add to the system the capability to resolve automatically certain unsatisfied task constraints, a formal methodology of modeling each task in terms of its constraints is presented. The software architecture of both components is described using the Unified Modeling Language (UML) and a component-based-development approach.

Dedication

To my parents, Nikolao and Stamatia, with love and endless thanks.

Στους γονείς μου, Νικόλαο και Σταματία, με αγάπη και ένα μεγάλο ευχαριστώ.

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Dr. Rajnikant Patel, for his guidance, support, patience and encouragement during the course of this research. Dr. Rajnikant Patel, besides being a great teacher is also an excellent man. Working in association with him has been a pleasant experience for me and I thank him for that.

I consider myself fortunate to have as my co-supervisor Dr. Ferhat Khendek. I wish to express my deepest gratitude to Dr. Ferhat Khendek; his academic intuition was fundamental to the completion of this work. I would like to thank him for his technical and moral support, and the good advices he always gave me.

My good friends, Mr. Tadeus Obuchowicz and Mr. Nestor Caouras, deserve special acknowledgment for helping me to keep my focus and for being always there for me. I wish to single out Mr. Iain Bryson not only for being a great friend, but also for his great support both technical and moral, and for been my, barring my supervisors, closest colleague on this work.

It is with pride that I acknowledge the support and inspiration of my parents Nikolao and Stamatia, and my sister Katerina - back home. I would like to thank them for believing in me, and making me understand that nothing is unachievable.

Last but never least, I wish to thank Maria Sarlis for her love, sacrifices and moral support all these years that we have been together.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

In order to control and supervise a robotic system with multiple agents, the operator usually has to cope with a large amount of information. The conventional technologies for control and supervision often overwhelm the user with the displayed data. In addition, in order to ensure safe operation of the system, several operators with great experience in the specific area of robotics, are usually required to monitor the system. Therefore, working towards the development of intelligent, autonomous robotic systems, that will minimize the need of "expert" users and will allow operability and supervision to be done in a more intuitive manner, a twofold approach has to be followed. On one hand the capabilities of the robots have to be enhanced in a way that they can act and react more autonomously, and on the other hand, the development of a suitable man-machine interface has to be pushed further in order to be able to command and to keep control over the system.

The realization of a man-machine interface, based on modern virtual reality (VR) techniques, is a promising approach for a new command and supervision interface that is intuitively operable. The general aim of the development to be described here is to provide a general framework for a system which allows to map actions that are carried out by users in the virtual world into the real world with the help of the robots composing the robotic system. This framework relies on the task deduction and the automatic task planning components of the virtual reality system that we wish to develop. A large variety of components have to be taken into account in order to establish automatic task realization by the robotic system, and to make the system capable of making decisions on planning the execution of different tasks.

The creation of simulations of systems consisting of multiple cooperating robots interacting with each other and their environment, and which can be controlled and supervised via a smart man-machine interface is therefore a large, complex task. Such VR-systems, however, are an essential part of virtual prototyping, tuning and testing new robotic applications. Moreover, it is frequently the case that it is impossible to test a single part of such a system in isolation. For instance, if one wishes to evaluate a new task deduction strategy, one must implement not only the new algorithm, but at least some model for interaction with the virtual environment. In addition, one would also want to view the results as a 3-D visualization of the workcell. As we can see, it is not feasible to examine a module as a static entity, since it involves interactions with other components in order to function. Because of coupling between these modules, the scope of software that needs to be constructed increases rapidly. An observation that can be made by looking over many different systems, is that the majority of the modules cannot be reused because they were designed quickly and without having reusability and future changes as a primary design concern. The robotics engineer who is interested in testing a task deduction algorithm now faces a software engineering load that has to be carried, and which is increasingly disproportionate to the original goal.

The purpose of this thesis is to provide a software architecture, intended to provide a practical framework for the automatic task deduction and task planning components of a VR robotic simulation. By using advanced software engineering notations such as Unified Modeling Language (UML), we provide both a modular design and implementation that will satisfy the requirements of the system. Although we went through the entire software process in order to come up with the proposed architecture, in this thesis we will be presenting the final result of that process only. The architecture focuses on providing the users with a default suite of generic modules allowing the user to add new ones. The advantage of developing a series of easy-to-use generic modules is quite obvious - modules could be reused without having to be rewritten. By adapting a software architecture where new modules can be easily added, and where all components are well defined modules with well defined interfaces, designed based on principles like extendability, reusability and design for change, we expect to limit some of the software design and maintenance difficulties.The following is a list of the issues this thesis is tackling:

- Petri Net models of robot tasks
- Task deduction
- Task planning
- Modeling task constraints

One of the main themes, which we expanded throughout this thesis is our desire for the software, named MRS, for Multiple Robot Simulation system, to function in as many different ways as possible. For instance, since we want MRS to support a VR interface, it should be able to support an automatic task deduction mode of operation, where the system should be able to recognize a user's action in the virtual space and map them to task descriptions. In addition, it should support automatic task planning functionalities, in order to decompose complex tasks and generate a plan of execution, which will be composed of a set of primitive (also called elementary [22]) actions. Integrating these different goals into a coherent and cohesive whole is what this thesis aims at.

1.1 Related Work

Automatic task deduction and task planning are not new issues. Several existing robotic simulation systems that provide a VR interface, support these capabilities in order to enable users to function both in a more efficient as well as intuitive way. In the following subsection, we will review briefly some of the work that has been done in this area.

1.1.1 Virtual Reality Robotic Simulation Systems

There are several academic and commercial products that include in their list of features some of the capabilities that we are interested in. We mention them here for completeness and we do not claim to have duplicated the multi-year efforts by significantly large corporations. One of them is the Cell Oriented SIMulation of Industrial Robots (COSIMIR)[23]. COSIMIR operates in various domains, both industrial and academic. By using latest VR techniques it provides a new and intuitively comprehensible way for man-machine communication in different types of automation applications. The domain of applications covers from space laboratory services, industrial assembly, spray painting and laser ablation applications to VR-based training facilities. The features that this system demonstrates, and which are more important to us, are those of task deduction [21], and automatic action planning [22]. The architecture that was developed for these two capabilities of COSIMIR, demonstrates several key ideas, some of them were adopted and further expanded to serve the purposes of MRS. For instance, in the task deduction module, COSIMIR introduces the idea of using petri net structures to model the tasks that the robotic workcell can execute.

A second system, Distributed Interactive Virtual Environment (DIVE) [25] is a virtual reality tool developed by the Swedish Institute of Computer Science, used to simulate workcell for manufacturing assembly processes. The important characteristic that this system demonstrates is that all the objects in the workspace establish relationships between them and between the agents in the workspace. For example, an object can send a signal when it touches another entity in the workspace or could report its status with respect to certain agent. These relationships allow the agent to query at any instant the position of the object with respect to the environment or with respect to other objects and based on the answer that it receives it is able to generate a multi-stage task plan. We say multi-stage because in order to have a plan of execution several stages of analysis are required. The system generates a set of simple pre-conditions that must be satisfied, then based on these pre-conditions and the relationships between the object and the agent, responsibilities are assigned. Finally, the evaluation of agent responsibilities with respect to the objectives of the problem leads to the reorientation of the agent goals to better address the problem. The concept of establishing relationships between the entities in the workcell is something we will be using when we discuss the architecture of task planning modules of MRS.

1.1.2 Task Planning Systems

Since the work of this thesis involves also the development of a task planning architecture, it is instructive to examine various planning components that have been developed. We hope to gain insight into existing planning component architectures, even if they are not wholly within our domain.

The REusable Task Structure based Intelligent Network Agents (RETSINA) planner is a very interesting architecture [30]. The way that this planning system is formulated is as follows. Instead of having a single global task planning component that formulates the general action plan, we have several distributed planners that are internal to the agents that are participating in the task. Each agent, using its internal planner, formulates detailed plans and executes them to achieve local and global goals. Knowledge of the domain is distributed among agents, therefore each agent has only partial knowledge of the world. This forces agents to cooperate with the planners of other agents, and to monitor their plan. The RETSINA planner, allows agents that adapt that type of architecture to interleave planning, execution and replanning in a dynamically changing environment despite having only partial knowledge of the domain. Another important concept that is being introduced in the design of the RETSINA planner is that of task representation using the Hierarchical Task Network (HTN) formalism [37]. Basically, this is a structure that has been adapted to decompose a single task to subtasks and each subtask to sub-subtasks, generating a tree structure.

Yet local and distributed planning brings about other problems. For instance, once it is found that an agent's local plans conflict with other agents' local plans, the agent's plans must be revised and re-planned. Moreover in multi-agent systems, the computation of an agent's local plan partly relies on other agents performing parts of the plan. This implies that agent's i local planning may require information from other agents to complete their plans and provide results; until then, agent i will be waiting. This will cause serious delays for a system in which time is a critical constraint.

The AREAS [26] system addresses a new approach for planning and evaluating assembly sequences. This approach requires cooperation between the user and the automatic assembly planner. This planner takes a representation of the design of a certain product, whose assembly sequence has to be planned, and determines the best sequencing alternatives. The planner interacts with the user through "augmented reality", in order to identify the best solution. The interconnection between the parts that have to be assembled form a liaison graph. All the possible assembly transformations can be represented in terms of the assembly graph. The assembly graph basically represents all possible state transitions in the assembly of a product. The planner with input from the human operator traverses the assembly graph to determine a path that constructs the product in an efficient way.

1.1.3 Object-Oriented Techniques in Simulation

Both task deduction and task planning modules are software components of a larger system, which is the robotic simulation system. Simulation systems have been built using traditional structured approach and not in an object-oriented way. The application of software engineering principles and object-oriented analysis, design and programming techniques have been introduced recently into simulation systems. Harrell et al. [7] provide an overview of a component-based distributed simulation architecture for discrete-event simulations. McMillan et al. [8] addressed the common manipulator dynamics simulation problem [6] using object-oriented design. The RETSINA planner, mentioned above, introduced an object-oriented approach for the design of the planner. In all these cases, it is clear that the resulting code reads like a high-level scripting language that promotes fast prototyping, while maintaining the fast execution speed of the compiled code. This is a major advantage of using object-oriented programming.

1.1.4 MRS Versions

The software architecture of the task deduction and the task planning modules presented in this thesis is closely coupled with the architecture of MRS v2.0 [10]. There exist also a first version of MRS (v1.0), which is described in [11] and [12]. We should note here that all the references to MRS in this thesis refer to the second version.

1.2 Outline of the Thesis

Chapter 2 begins the thesis by reviewing background information. This information is important for the understanding of the design described in subsequent chapters. For instance, Petri Net structures are elements used in the specific domain of the application we are developing. A brief description of these structures as well as the benefits that we gain from their usage, are topics presented in this chapter.

Chapter 3 is dedicated to the description of the task deduction module. It starts with a brief discussion of the analysis of the problem domain. We describe the functional requirements of the task deduction module. Then we proceed with the decomposition of the system into its basic components. We describe the role of its individual units and its internal design. Subsequently, the dynamic behavior and the way of interacting with the rest of the system are described towards the end of this chapter.

The next chapter, Chapter 4, presents the task planning component of the system. It begins by analyzing the role of this module in the entire system. Then, the functional requirements with respect to the entire system and the modules that interact with it are presented. In addition, the basic software building blocks of the task planning component are described and the most important architectural decisions that were made are presented. In particular, we focus on the architectural features which allow us to enhance and extend the software.

Chapter 5 presents a case study. In this chapter we present the task deduction and task planning module through MRS (v2.0). A complete example is presented. It puts together both components that are developed, providing a study of their interactions and their behavior with respect to each other and with respect to MRS.

The thesis concludes with Chapter 6, which contains a recapitulation of the major work of the thesis and suggestions for future work.

Throughout this thesis, we will be using the UML notation to describe the architecture and our design. An introduction to the UML notation is given in Appendix A.

Chapter 2

Petri Nets

This chapter presents some background information regarding Petri Net structures as a modeling tool. Since these structures will be used in a subsequent chapter, where task deduction will be presented, it is useful to address here some basic definitions and terminology. In addition, this chapter presents Petri Nets as a modeling tool for a wide range of applications. Starting with Human-Machine Interactions (HMI), we move on to discuss the application of Petri Nets to control production systems. We conclude this chapter by introducing the software representation of Petri Net that has been adopted in MRS.

2.1 Introduction

A Petri Net (also named place/transition net in [16]) is defined as an oriented graph comprising two types of nodes: (1) places and (2) transitions. This graph is constituted in such a way that the arcs can only link places to transitions or transitions to places [17]. Places are represented by circles and transitions by bars (Figure 2-1). Peterson [14], defines the structure of a Petri Net as a composition of four parts: (1) place, (2) transitions, (3) input function and (4) output function. The input and the output functions relate transitions and places. The input function is a mapping from a transition to a collection of places, known as input places of the transition. The output function maps a transition to a collection of places, known as the output places of the transition.

Petri Nets represent not only one of the most formal, but also the best developed, models for representing multi-process systems. Several existing ways of representing processes (e.g. finite-state automata, register automata [9] etc.), are limited by the fact that they can represent only sequential processes. Thus, these limitations pose a great obstacle in the modeling of the inherent parallelism of Multi-Agent Systems (MAS) [9]. For this reason, when we want to model precisely the interactions between agents, we need more complex formalisms, able to describe both behaviors and organizations in which several processes execute concurrently. In addition, Petri Nets can easily model non-determinism and parallel computation, two essential features of concurrent systems, such as VR systems. Moreover, Petri Nets can model easily synchronization of asynchronous processes. An additional advantage is that Petri Nets can accommodate quite easily models at different abstraction levels [39].

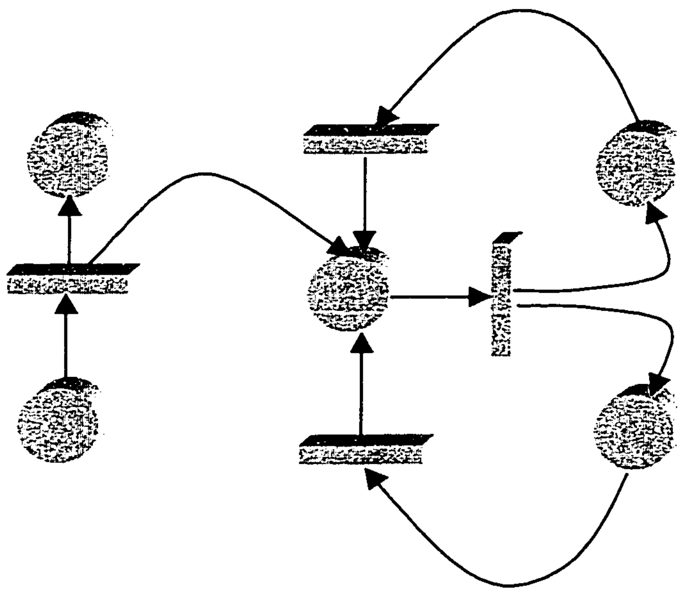


Figure 2-1. An example of a Petri Net consisting of five states and four transitions.

2.2 Modeling with Petri Nets

Petri Nets were designed and are used mainly for modeling. Petri Nets can be used to model a wide range of systems; computer hardware, computer software, physical systems, etc [15,17,18]. Petri Nets are used to model the occurrences of various events and activities in a system. In particular, Petri Nets may model the flow of information within a system as well as the interactions of the system with entities external to the system. The following sections present Petri Nets in the area of VR-based Human-Machine Interaction (HMI) systems, as well as in developing control schemes for manufacturing systems.

2.2.1 Modeling a VR-based Interaction Scheme

In [19], Freund and Rossmann present the application of Petri Net structures in modeling an automatic task deduction component for their system. The domain of their application is the control and supervision of an autonomous multi-robot-system for a space laboratory, by means of virtual reality techniques. By using the task deduction component, they allow the system to recognize the task the user is trying to perform. This was achieved with the use of Petri Nets; all the tasks that the system supported were modeled with these structures in a very formal way.

2.2.2 Modeling a VR Environment

In [39], the University of Illinois at Chicago present CAVE. A VR environment which consists of several walls that display computer-generated images for the benefit of a human viewer. These images are drawn in real-time on the basis of the viewer's perspective in the virtual world in such a way as to create the impression of real-life, three-dimensional view of a given scene. By employing a timed extension of Petri Nets they modeled the operational and timing behavior of all the different subsystems that compose CAVE.

2.2.3 Modeling Control of Production System

Valette [38] gives examples of many projects both commercial and academic, involving control of manufacturing and assembly systems, that utilize Petri Net structures, in order to solve concurrence and synchronization problems. Moreover, in [38], the Japanese company

Hitachi has developed a decentralized control system for factory automation where the programming of the station controllers (corresponding to the level of coordination), is done by Petri Nets. This approach has been applied in several applications including stations responsible for scheduling assembly sequences.

2.3 Petri Nets in MRS

Having in mind the typical structure of a Petri Net and the different domains of its application, this section will introduce the software representation of Petri Nets that was adapted in MRS. First, the role of Petri Nets in the context of MRS will be presented briefly, since Chapter 3 covers the topic completely.

MRS is expected to undertake several tasks (welding, assembling etc.). These tasks involve the different objects that are present in the robots' workspace (e.g. metal parts, wooden pallets etc.). For instance, the welding task involves the two or more objects that are going to be joined together. In order to accomplish a specific task, the objects involved will have to pass through certain states. For example, to move an object from one point to another, the object passes through the following states: (1) Grasped, (2) Moving and (3) Released. Thus, we employ Petri Nets to model this behavior of all the objects in the workspace.

In Figure 2-2, we see a Petri Net structure with three states and two transitions. We note here the existence of a new symbol in the overall Petri Net structure, a six-edged symbol [20]. The only reason this symbol is introduced is to provide a visual representation of the fact that a certain transition causes a certain effect in the system. We will call that effect *"Action"*, and we will represent it with the six-edged symbol throughout this thesis.

We begin our discussion by elaborating on how we model Petri Nets in MRS, since the decisions we make are quite important in order to provide an easy way to examine them and to store them. The formulation that we will choose will definitely affect the design, and as a result, the implementation of certain modules, which we will address in the following chapters.

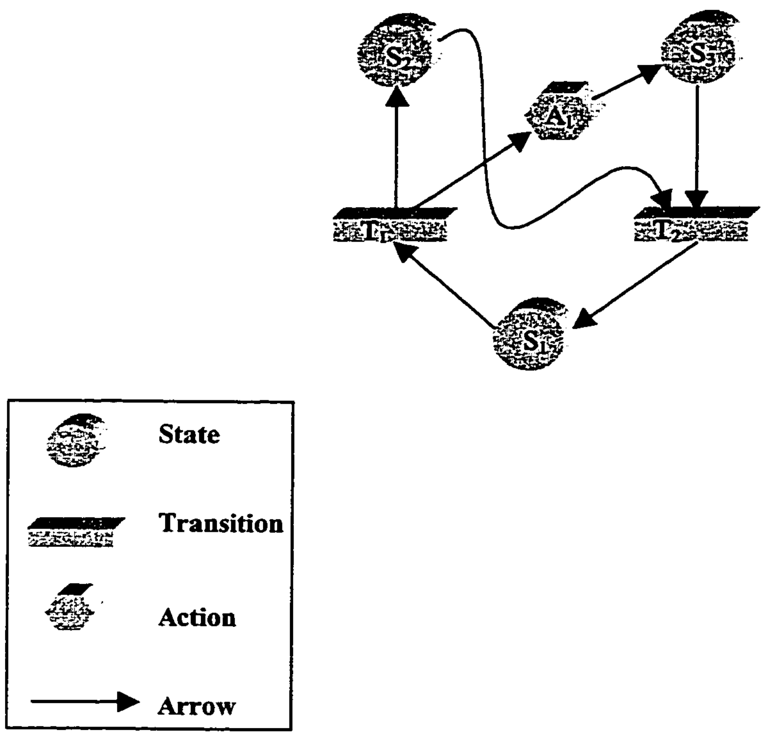


Figure 2-2. Petri Net graphical representation in MRS

The software representation that we adopt for our Petri Net structures is the formulation of the Petri Net graph in a vector representation. The entire structure is a set of quadruples of the following format:

$$<Starting\_State_i,\ Transition\_Condition_i,\ Action_i,\ Ending\_State_i>$$

or

$$<S_i,\ T_i,\ A_i,\ E_i>$$

where *i* is the index that identifies a quadruple in the entire set. If we would now like to represent the Petri Net in Figure 2-2, in this representation, we would have the following general description:

$$Petri\ Net[A_1] = \{<S_1, T_1, A_1, E_1>, <S_2, T_2, A_2, E_2>, <S_3, T_3, A_3, E_3>, <S_4, T_4, A_4, E_4>\}$$

The number of quadruples is equal to the number of unique *paths* that we can form. We define, going from starting state S_i to ending state E_i, through transition T_i and by generating action A_i as a *path*. Referring again to Figure 2-2, we have one possible path going from $\mathbf{S_1}$ to $\mathbf{S_2}$, via $\mathbf{T_1}$ and without producing any action. The fact that no action is generated in that path, simply means that for that specific quadruple, assume the i^{th}, A_i is equal to ***null***. A different possible path is, if we move from $\mathbf{S_1}$ to $\mathbf{S_3}$, via $\mathbf{T_1}$ with generating action $\mathbf{A_1}$. In order to understand better the utilization of the Petri Nets we proceed with the analysis of Figure 2-2. Considering the fact that the general description of our Petri Net has four different paths, we list the values that are assigned to each variable of every quadruple:

Quadruple #1, $<S_1 = \mathbf{s_1}$, $T_1 = \mathbf{T_1}$, $A_1 =$ ***null***, $E_1 = \mathbf{s_2}>$

Quadruple #2, $<S_2 = \mathbf{s_1}$, $T_2 = \mathbf{T_1}$, $A_2 = \mathbf{A_1}$, $E_2 = \mathbf{s_3}>$

Quadruple #3, $<S_3 = \mathbf{s_2}$, $T_3 = \mathbf{T_2}$, $A_3 =$ ***null***, $E_3 = \mathbf{s_1}>$

Quadruple #4, $<S_4 = \mathbf{s_3}$, $T_4 = \mathbf{T_2}$, $A_4 =$ ***null***, $E_4 = \mathbf{s_1}>$

We should note here that the index on a certain variable (state, transition condition or action), for instance S_3, does not necessary mean that the path S_3 to S_2, through T_2, is the third quadruple in our set. The quadruples could be in any order in the set, the Petri Net structure does not change. This is actually quite important, since the meaning or status of a process that is modeled with our software representation of Petri Nets does not depend on the order that the Petri Net quadruples are stored. Thus, all the following expressions are equivalent:

$$Petri\ Net[A_1] = \{<S_2, T_2, A_2, E_2>, <S_3, T_3, A_3, E_3>, <S_1, T_1, A_1, E_1>, <S_4, T_4, A_4, E_4>\}$$

$$Petri\ Net[A_1] = \{<S_1, T_1, A_1, E_1>, <S_4, T_4, A_4, E_4>, <S_2, T_2, A_2, E_2>, <S_3, T_3, A_3, E_3>\}$$

$$Petri\ Net[A_1] = \{<S_3, T_3, A_3, E_3>, <S_1, T_1, A_1, E_1>, <S_2, T_2, A_2, E_2>, <S_4, T_4, A_4, E_4>\}$$

$Petri\ Net[A_1] = \{<S_2, T_2, A_2, E_2>, <S_1, T_1, A_1, E_1>, <S_4, T_4, A_4, E_4>, <S_3, T_3, A_3, E_3>\}$

Moreover, for the specific context of MRS, we decided that each Petri Net is uniquely characterized and identified by the "*Action(s)*" that is (are) described. For that purpose, our notation specifies in the general description of the Petri Net, between the square brackets, the action that is generated by following a certain path. In the example presented before, only one action is generated. This of course does not represent a general case, in fact, in the case where more that one action is generated, all of them are listed.

Chapter 3

Task Deduction Subsystem of MRS

This chapter covers the first part of the work presented in this thesis which is the task deduction subsystem proposed for the VR environment of MRS. The fundamental functional requirements of this module are first presented, explaining the need for its existence and describing exactly what we want to accomplish by introducing it to the overall architecture of MRS. Next, the analysis of its internal architecture is presented. The decomposition of the entire subsystem to its basic components follows, addressing the functionalities of each individual building block. Subsequently, the object-oriented structure of each individual component is examined. The chapter ends by addressing the dynamic aspect of the subsystem, we describe the interactions and the communication between the different units comprising the task deduction module.

3.1 Introduction

One of the major subsystems that is vital for the interaction with MRS via the VR interface is the Task Deduction Component (TDC). The TDC is responsible for recognizing the actions the user performs in the VR environment and provides a task description to the next level which is assigned the responsibility of planning the specific task that the user is trying to execute. That next level corresponds to the action planning system and we will see more of its design and operations in the next chapter. The block diagram in Figure 3-1 shows the layout of the overall system.

The functionality of the task deduction component is based on the fact that the set of tasks that can be performed in MRS have been pre-defined and also have been modeled in a very specific way to serve the purposes of our application. At this point we should note that the task deduction mode of operation works only for a specific class of simple tasks, and does not provide solutions to complicated tasks. This subsystem is a composition of several units interacting together, in order to support the functionalities required. The main idea behind its operation is the modeling of each task in the form of a Petri Net structure [21, 28, 29]. With the help of these structures, the task deduction subsystem can filter, analyze and categorize the users' actions which can be further summarized into task descriptions for the action planning system.

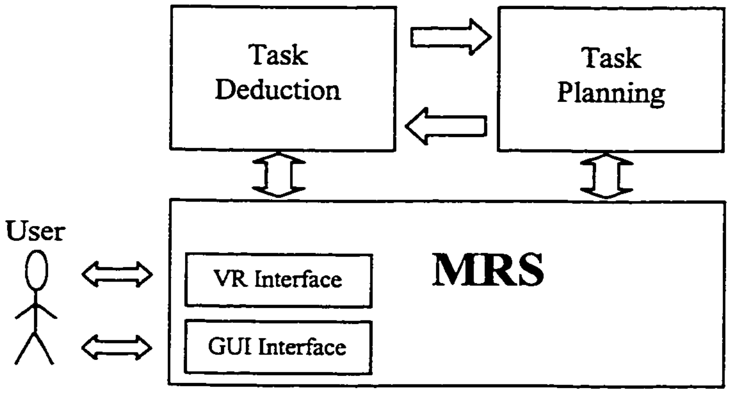


Figure 3-1. Placing Task Deduction subsystem in the overall architecture of MRS

3.2 Functional Requirements of the TDC

One of the most common techniques for reusing functionality in object-oriented systems is *Object Composition* [2],[5]. *Object Composition* requires that the objects being composed have well-defined interfaces. This style of reuse is called *Black Box* reuse [2]. This style is the one adopted for the task deduction component, since reusability is one of the primary goals of our system. Here we examine the task deduction component as a *Black Box*, for which the only information we have is the type of inputs provided and the corresponding outputs that we get (Figure 3-2). This approach is adopted through out the thesis. The following subsection presents the set of functional requirements which characterize the behavior of TDC in MRS.

3.2.1 General & Specific Requirements

REQUIREMENT 1: The task deduction component receives as input *Physical* events (Figure 3-2). Since the user acts in the VR environment of MRS, with the input device that is available, these physical events that are generated in reaction to his/her actions are received as inputs from the upper layers of the task deduction module.

REQUIREMENT 2: The task deduction component provides a mapping from *Physical* to *Logical* events. The task deduction is responsible to interface the user in the VR-environment with MRS. In order to accomplish this, the physical input that it receives gets translated into a different form, which will be used in the rest of the system.

REQUIREMENT 3: Implement structures and algorithms that will allow for storing and examining Petri Net structures which are used to represent the tasks the user can generate in the VR environment of MRS.

REQUIREMENT 4: The output that the task deduction should generate is a message to the next level, which is the task planning component, and that will contain a description of the task that has to be planned.

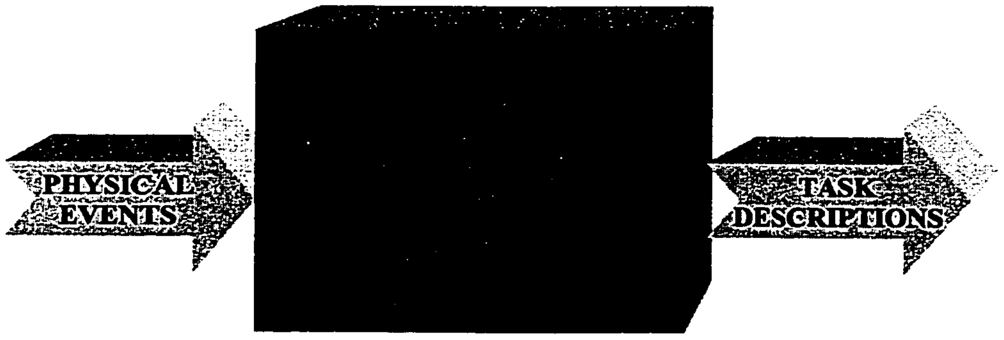


Figure 3-2. Black Box representation of the Task Deduction Component

3.3 Decomposition of the Task Deduction Component

Since in the previous section we dealt with our requirements for the task deduction component, we will present now the architectural foundation of this module, the software framework with which we hope to satisfy those requirements. MRS was developed using paradigms of object-oriented analysis (OOA) and object-oriented design (OOD) ([4],[5]), we follow the same principles in the development of the task deduction subsystem.

We will start with an analysis of the problem space. This will provide us with a good grounding for the presentation of the architecture following it. By showing the structure and the relationships between the entities that comprise this subsystem, the motivation for the specific architecture will become clearer.

3.3.1 Analysis

An object oriented analysis starts by identifying the objects in the problem space. In this specific domain, the problem space is the task deduction module. We decompose the entire problem space into a group of logically distinct units that provide the different functionalities the problem space demands. The units that we can identify are: the *User Action Detection* unit, the *Task Interpreter* unit, the *Messenger* unit, the *Task Manager* unit and *TaskDBase* unit (Figure 3-3). The *User Action Detection* and the *Task Interpreter* are those that form the task-recognition functionality of the task deduction module. The *User Action Detection* unit receives the physical events coming from the user in the VR environment. For instance, when the user in the VR environment "Selects" an object, the *User Action Detection* unit is responsible for assigning a tag to the object selected and mapping the user's action to a higher level input (e.g. GRASP or RELEASE that object), sending it afterwards to the *Messenger* unit. The *Task Interpreter* contains algorithms that are based on the Petri Net structures introduced in the previous chapter. Its major functionality is to trace these structures by using as inputs the messages stored in the *Messenger* unit. By analyzing these structures the *Task Interpreter* generates task descriptions for the next level which is the task planning subsystem. The *Task Manager* unit is responsible for loading a specific task structure from the *TaskDBase* and passing it to the *Task Interpreter*. A very important functionality that the *Task Manager* performs is to identify the type of task that should be retrieved from the *TaskDBase*. The static entity that keeps all the Petri Net structures stored, is the *TaskDBase*. It is very closely associated with the *Task Manager* in order to provide safe access to the task structures. The *Messenger* unit provides a communication pipeline between the different units.

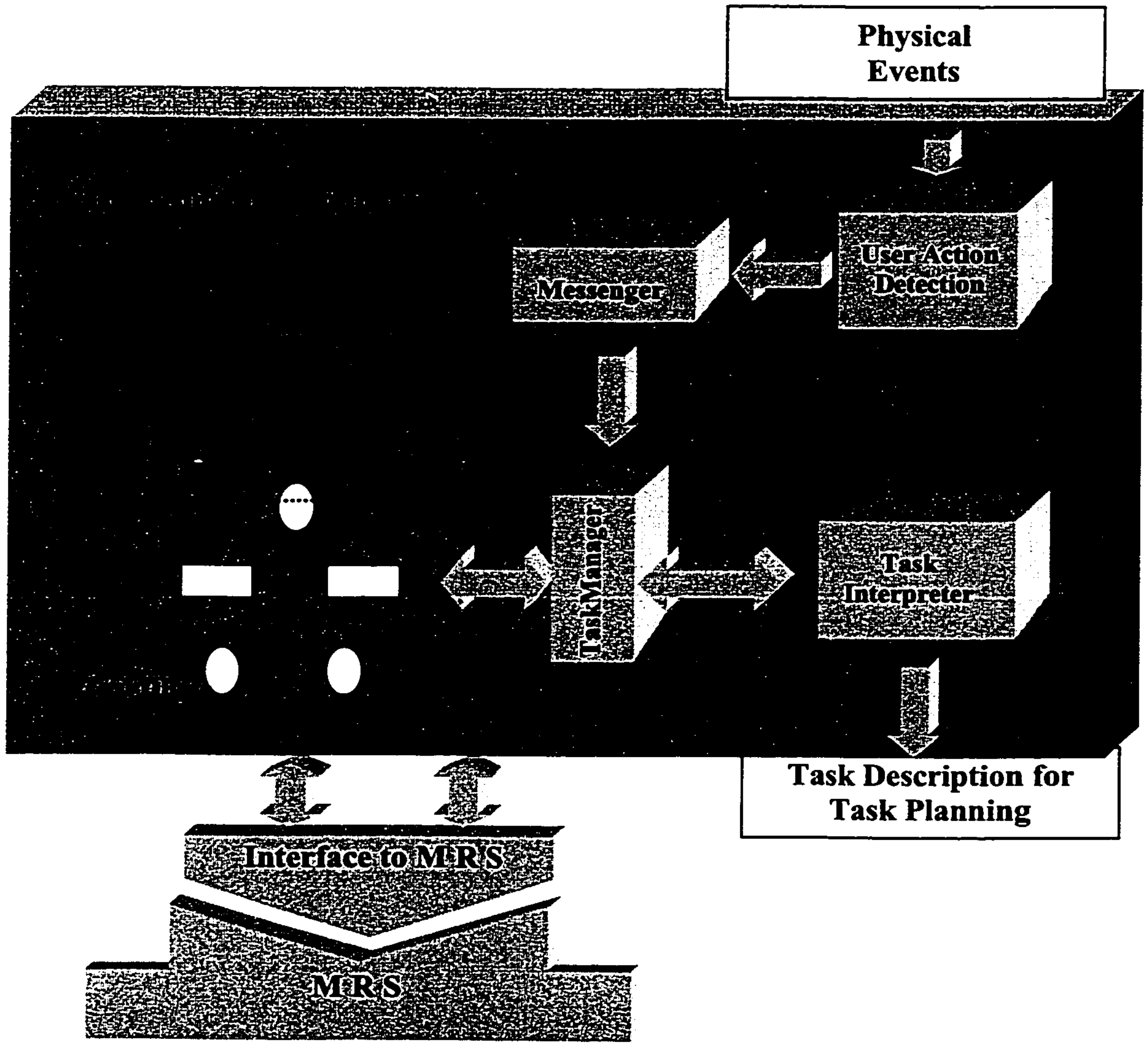


Figure 3-3. The architectural structure of the Task Deduction Module

Having the overall architecture of the TDC in mind we proceed to the analysis of each individual unit that is present. The following sections analyze the overall structure as well as the functionalities that these units provide.

3.3.1.1 User Action Detection Unit

The *User Action Detection* unit, as already mentioned, receives signals from the user, and is responsible for performing a very specific mapping of those low level user inputs to a higher more abstract type. This means, mapping from *Physical* to *Logical* events. For example, when the user points to an object inside the VR environment ([31,32,33,34]), that object gets selected. That low level signal which represents the selection of certain objects is sent to the *User Action Detection* unit that maps it to a "GRASP" or a "RELEASE" message. The *User Action Detection* unit is responsible for keeping track of the messages that are generated with respect to each individual object, in order to distinguish between different signals and perform a simple constraint checking on user's actions. For instance, when the user issues a "GRASP" signal for a certain object, that is recorded, then when the user issues another "GRASP" without a "RELEASE" being issued before, the *User Action Detection* unit, will detect that and act accordingly. In addition, the *User Action Detection* unit tags the object that is currently active, and is responsible for propagating that tag to the next unit which is the *Messenger*.

3.3.1.2 Messenger Unit

The *Messenger* unit receives inputs from the *User Action Detection* unit and stores them until the *Task Manager* sends a request to retrieve one. The role that the *Messenger* performs is very simple: it operates as a temporary storage where we have the scheme shown in Figure 3-4. The messages passed to the *Messenger* unit contain information regarding both the type of action performed by the user and also the tag that uniquely identifies the object in the entire MRS. The messages are stored in that buffer in the same order that they arrive, forming a queue. When the *Task Manager* requests a message, it retrieves the one that is in the front of the queue.

3.3.1.3 Task Manager Unit

The next unit of the task deduction module that we will address, is the *Task Manager* unit. That entity is responsible for handling the loading of the task structures from the *TaskD-Base* to the *Task Interpreter*, based on the outcome of the evaluation that it performs on the messages that are retrieved from the *Messenger*. We should note here that only through the *Task Manager* can the tasks (in the form of Petri Nets) in the *TaskDBase* be accessed.

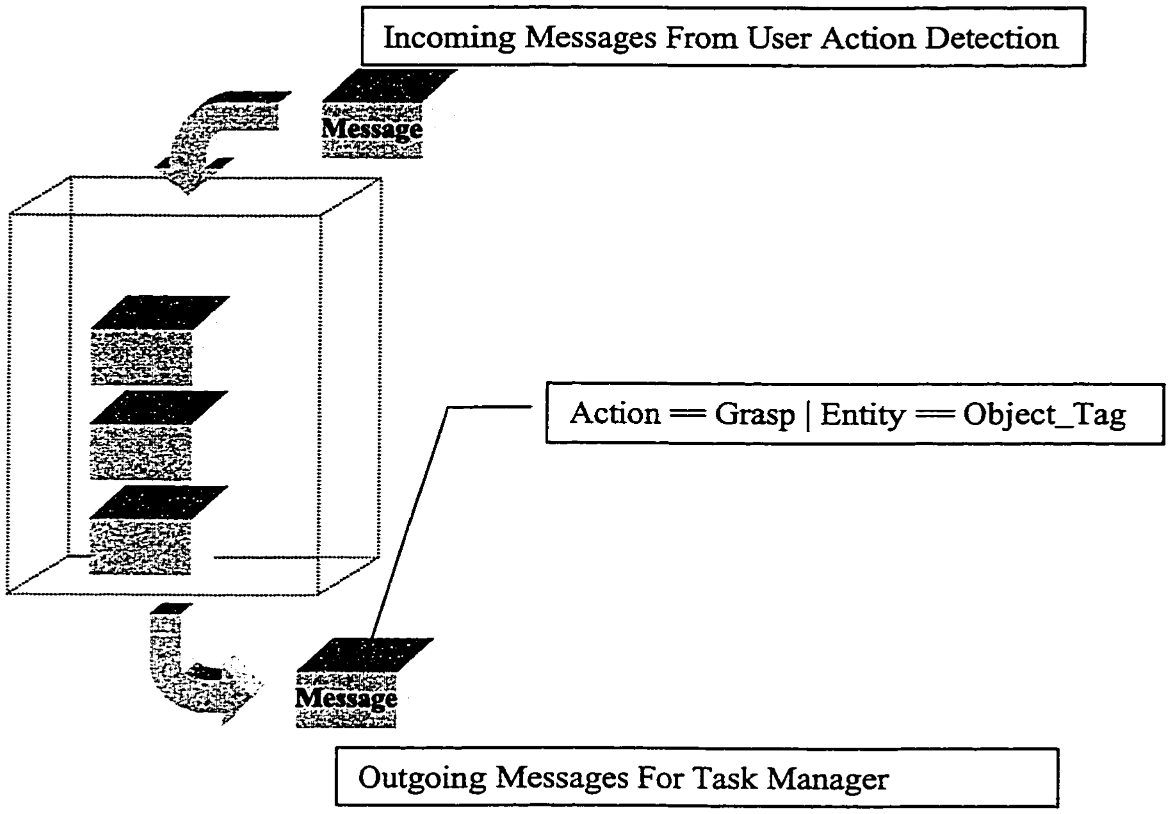

Incoming Messages From User Action Detection
Message
Action == Grasp | Entity == Object_Tag
Message
Outgoing Messages For Task Manager

Figure 3-4. The queuing of the messages and their internal structure.

In addition, the *Task Manager* is responsible for decomposing the message retrieved to the two components that comprise it: the user's action and the object's unique identifier. The object's tag (or identification number) is used as a key for retrieving the task structure (or Petri Net) that is associated with that specific object in the *TaskDBase*. The other part of the message that contains the action which the user performed is used in the *Task Inter-*

preter (when the Petri Net is loaded) as the input function which will cause a certain transition from a specific *Starting_State* to a certain *Ending_State*. Another functionality of the *Task Manager* is to provide an interface for handling the flow of messages going to the *Task Interpreter*. Since the *Task Interpreter* needs messages to traverse the Petri Net structures (we will see more about this in the following section), instead of going directly and accessing the *Messenger*, it goes through the *Task Manager*. This is quite important to our overall architecture of the task deduction component in order to ensure that no messages are lost or disregarded. The problem that will occur if we allow the *Task Interpreter* to directly access the buffer is the following: a message that is important for the *Task Manager* to retrieve a task is removed from the queue by the *Task Interpreter* while on the other side the *Task Manager* will be waiting for it and will get stalled. With the proposed architecture, only the *Task Manager* retrieves the messages, keeps a record of the information it needs and passes a copy to the *Task Interpreter*.

3.3.1.4 TaskDBase Unit

The next unit that will be addressed is the *TaskDBase* unit. It contains a collection of all the tasks that can be deduced from the user's actions in the VR-environment of MRS. We should note here that from the group of tasks that MRS supports, if operated via its GUI, only the tasks that can be deduced from the VR-environment are modeled in the *TaskDBase*.

In order to store and manipulate the tasks in a formal way that will provide efficiency and will resolve possible problems with concurrency (since we operate in a multi-agent environment), we adopted the Petri Net structures as our modeling tool. In Section 2.3, we saw the software representation of the Petri Net structure that we formed. We will see now how this representation fits into the context of MRS. The way task deduction works in the context of MRS is as follows. We combine the user's actions with predefined tasks descriptions, in order to a derive what the user is trying to accomplish. We have seen the units that provide the functionality of recording the user's actions; what the *TaskDBase* adds to that is the place to store the Petri Nets used to model the MRS tasks.

The next unit, the *Task Interpreter*, will show how the tracing of the Petri Net is performed, by using the task descriptions in the *TaskDBase*.

3.3.1.5 Task Interpreter Unit

The *Task Interpreter* is the unit where algorithms have been implemented to trace the Petri Net structure. When we say the *Task Manager* loads a Petri Net to the *Task Interpreter*, what we mean is that it passes a pointer to the software representation of the Petri Net. Then the *Task Interpreter*, because the process of tracing the Petri Net requires the type of action that the user performs, requests inputs from the *Task Manager* which has access to the *Messenger*. The *Task Manager* returns the type of action performed and the process of tracing continues.

3.3.1.6 Example of Task Deduction with Petri Net Tracing

In order to understand the method of task deduction with Petri Net structures an example of Petri Net analysis will be presented. We will describe a Petri Net that can generate two different types of actions (see Section 2.3 for actions), "Move_Metal_Part" and "Weld_Metal_Part". In Figure 3-5, the Petri Net structure used to model these actions is presented. Recalling from Chapter 2, in this structure we have five states, six transitions and two possible actions.

3.3.1.6.1 List Of States

STATE (1): Welding Position

This is the state we reach after performing the welding operation between two metal parts. At this state the two metal parts are considered not as two distinct parts but as one single object.

STATE(2): Free Position

This is the state we can reach if two possible situations occur. One possibility is when the user in the VR-environment moves the object anywhere within the robots' *workspace* [6] except the location that is characterized as the location where the welding operation will take place. The second possibility is the case where the user moves the virtual object to a location assigned to the welding operation and it is the only object that is located in the welding position at the moment; then again the state we reach is the same, but no action is generated. The reason why we don't generate an action when only one object is in the welding location is because we want the user to bring both virtual objects to the specific location before the process of welding gets initiated, so the task planning component will be able to plan the operation accordingly, having the information of both objects that take place in the process.

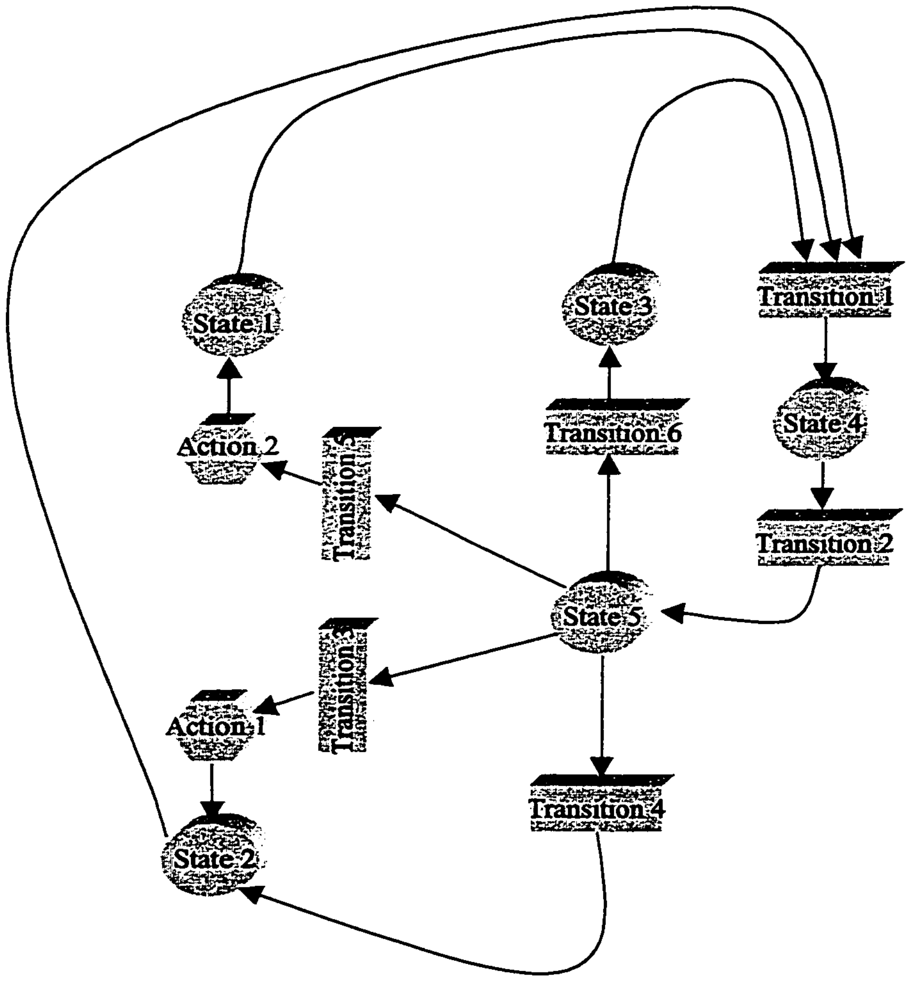


Figure 3-5. Petri Net for deducing actions "Move_Metal_Part" and "Weld_Metal_Part"

STATE (3): Undefined Position

This is the state that the task reaches when the user in the VR-environment moves the object in a location outside the robots' workspace. Since the virtual space in which the user operates does not have any physical limitations, the possibility of moving the object to a location that cannot be mapped to an actual location in the real workspace is always present.

STATE (4): Metal_Part In Motion

This is the state that the task reaches when the user "GRASP"s the object in the virtual environment. We note, that it does not mean the object is actually moving, it could be just placed anywhere in the workspace without being involved in any type of motion and still be in that state, provided that a "GRASP" signal has been issued.

STATE (5): Metal_Part Released

After a "RELEASE" signal is generated from the user, we reach to this state.

3.3.1.6.2 List of Transitions

TRANSITION (1): Grasp

This is the transition condition that is enabled, when the user selects an object in the work-cell.

TRANSITION (2): Release

This is the transition condition that is enabled when the user issues a message that is mapped to a logical event "RELEASE".

TRANSITION (3): Inside the Workspace

In order for this condition to be evaluated, the task deduction component communicates with MRS via the interface available to check if the entity examined is inside or outside the workspace.

TRANSITION (4): One Object In Welding Position

This transition is triggered when one object is positioned at the predefined welding location.

TRANSITION (5): Two Objects In Welding Position

This transition is enabled when after communicating with MRS, both entities that will be welded are in proper position so the process can be initiated.

TRANSITION (6): Any Other Position

This transition is enabled when the result of the evaluation of the position of an object in the virtual environment is that it is outside the workspace of any robot available in the workcell.

3.3.1.6.3 List of Actions

ACTION (1): Move_Metal_Part

This action initiates the process of transporting the object that the user manipulates in the VR workspace to the location also specified by the user.

ACTION (2): Weld_Metal_Part

This action generates a task description that will trigger the planning of a process that welds the two objects that are placed by the user at the locations predefined as the welding locations.

When the user in the VR environment selects an object, the associated physical event is sent to the *User Action Detection* unit, which maps it to the corresponding logical event. We assume for the purpose of our example that the signal is a "GRASP", that is sent to the *Messenger* unit from where the *Task Manager* retrieves it, analyzes it, and based on the object's tag that the message contains, the *Task Manager* accesses the *TaskDBase* and loads the proper Petri Net into the *Task Interpreter* (Figure 3-6). Suppose that the initial state of the object, before the "GRASP" was inside the workspace but not at any welding configuration. Then the state from where we start tracing the Petri Net is **State(2)**. Since a "GRASP" was signaled, **Transition(1)** is fired and without causing any action the task moves to **State (4)**. We could have many different paths (see Chapter 2). In the specific example, the one described is a simple one, considering the fact that no action was generated.

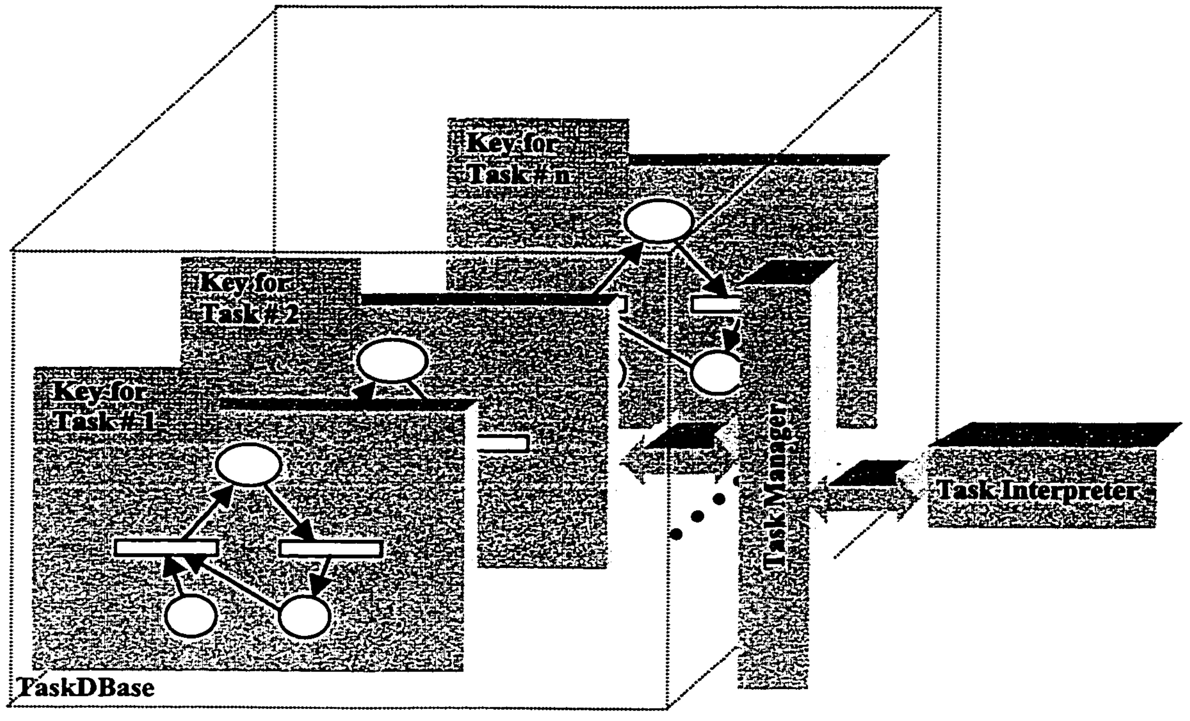


Figure 3-6. The Task Manager is accessing the TaskDBase to load a Petri Net into the Task Interpreter

Having addressed the basic units of the task deduction component of MRS, we will proceed to the software structure of this module. By following an OOD process, we provide a modular design that can be easily adapted, and that will allow future modifications to be made.

3.4 Design of Internal Units

The task deduction component can be seen as two subsystems working together. The first is responsible for working as an interface with the user and the other for doing the processing. First, we present the software structure of each individual subsystem separately, and then we join the two together in order to study the overall software architecture.

3.4.1 User Action Handler

This is the first subsystem of the TDC and it handles the events coming from the user. It operates as a link that joins together the low-level user interaction subsystem, with the more abstract and higher level subsystem responsible for interpreting these actions and understanding their meaning. In fact, it is responsible for mapping every valid action of the user to a certain type of message. In addition, it should allow both the dynamic creation of messages when a user's action occurs, and the dynamic adjustment of the buffer size that stores the messages. The last requirement demands the existence of a dynamic structure that will expand and shrink according to the situation. Moreover, the type of information contained in the messages passed as well as the order in which they are generated should be recorded. For these purposes the software model we adopt for the system is the one shown in the class diagram in Figure 3-7.

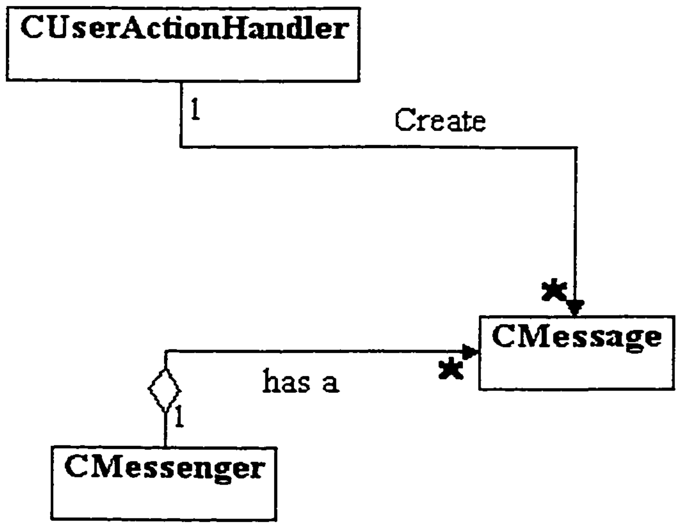


Figure 3-7. The class diagram in UML notation for the User Action Handler subsystem

We note that in order to provide background information for readers not familiar with object-oriented modeling languages, a complete description for the Unified Modeling Language (UML) notation ([1],[3]) is included in Appendix A.

CUSERACTIONHANDLER class:

The CUSERACTIONHANDLER class is the one that provides the implementation for distinguishing different events coming from the user. It keeps a record of the previous events that arrived, and based on a set of predefined constraints, it maps the events to messages and sends them to the rest of the subsystem. This class has a relationship "create" with class CMESSAGE of type *Association* [3]. To be more precise, the relationship between the CUSERACTIONHANDLER and CMESSAGE is *one-to-many* [1], meaning, one instance of CUSERACTIONHANDLER could be related to many instances of CMESSENGER. The important point we should note is that this relationship simply represents the fact that the CUSERACTIONHANDLER instantiates messages, and does not correspond to the actual flow of data.

CMESSAGE class:

The CMESSAGE class is responsible for providing the implementation for the CMESSAGE objects, that will be used as transportation units for the information in the subsystem. It is used by the CUSERACTIONHANDLER as a factory for creating objects of type CMESSAGE.

CMESSENGER class:

This class is the one that implements dynamic structures for storing objects of type CMESSAGE. It is responsible for allocating memory to the incoming messages and free memory space used by messages that have been processed. It is also responsible for keeping track of both start and end points of the buffer so that messages can be removed and added respectively. This class establishes a relationship "has a" with class CMESSAGE of type *Aggregation* [3]. *Aggregation* here expresses a strong coupling between these two class. In fact, the relationship between the specific classes CMESSENGER and CMESSAGE is *one-to-many*, denoting that one instance of CMESSENGER could be related to many instances of the class CMESSAGE.

3.4.2 Petri Net Handler

The subsystem already described interacts with a second subsystem that is responsible for handling the tracing of the Petri Net structures. In addition, this subsystem is managing the tasks' storing in the structure we have created (the *TaskDBase*). Finally, this subsystem is also responsible for controlling access to the buffer where the messages are store until they are processed by some unit of the system. The general behavior of this subsystem is shown in the following block diagram (Figure 3-8).

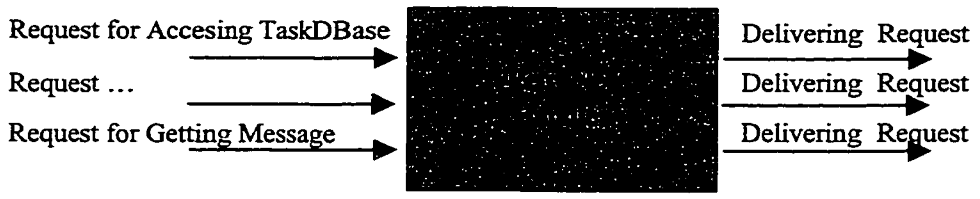


Figure 3-8. Petri Net Handler subsystem

The *Petri Net Handler* accepts as inputs different types of request, which it has to deliver. For instance, a request could be to retrieve a Petri Net structure based on a message that was consumed. The class diagram in Figure 3-9, shows the software components that compose this subsystem.

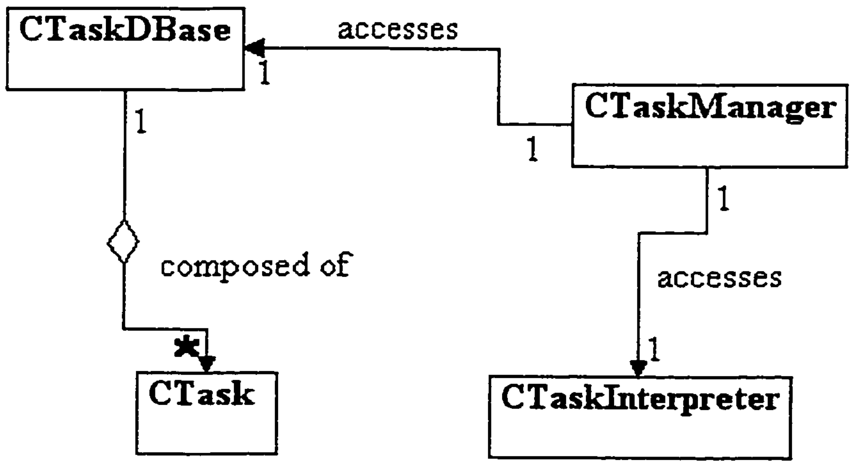


Figure 3-9. Class diagram for the Petri Net Handler subsystem.

CTASKMANAGER class:

This class implements the algorithms responsible for retrieving Petri Net structures from the CTASKDBASE and loading them into the CTASKINTERPRETER. This class has a relationship "accesses" of type *Association* with both the CTASKDBASE and the CTASKINTERPRETER classes. Since only one instance of the CTASKMANAGER is related to only one instance of the CTASKDBASE class, we set the cardinality of both classes to one, and the relationship between them is *one-to-one*. In addition, the relationship of the CTASKMANAGER class with the CTASKINTERPRETER is also *one-to-one*, since we allow only one instance of the class CTASKMANAGER to be associated with one only instance of the class CTASKINTERPRETER.

CTASKINTERPRETER class:

The CTASKINTERPRETER class contains algorithms to traverse the Petri Net structure in order to detect the type of action the user is trying to perform in the workcell. The CTASKINTERPRETER interacts with the CTASKMANAGER to get information about the transition conditions necessary to trace the Petri Net. The outcome of this class is very important since it generates the task description that we propagate to the next level which is the task planning subsystem.

CTASKDBASE class:

This class provides a software structure which consists of instances of the class CTASK. This class is static since the number of the predefined tasks is known, and it does not change during run time. The CTASKDBASE class has a relationship "composed of", of type *Aggregation*, with the class CTASK. The cardinality of the class CTASKDBASE is one while the cardinality of the CTASK class is many. This implies that one instance of the CTASKDBASE could be related to many instance of the class CTASK.

CTASK class:

This class is responsible for constructing the Petri Net structures (as presented in Chapter 2), and is used by the CTASKINTERPRETER to deduce the desired task description.

Having described the software model of the two individual subsystems, we conclude this section by joining the two subsystems together in one software architecture (Figure 3-10).

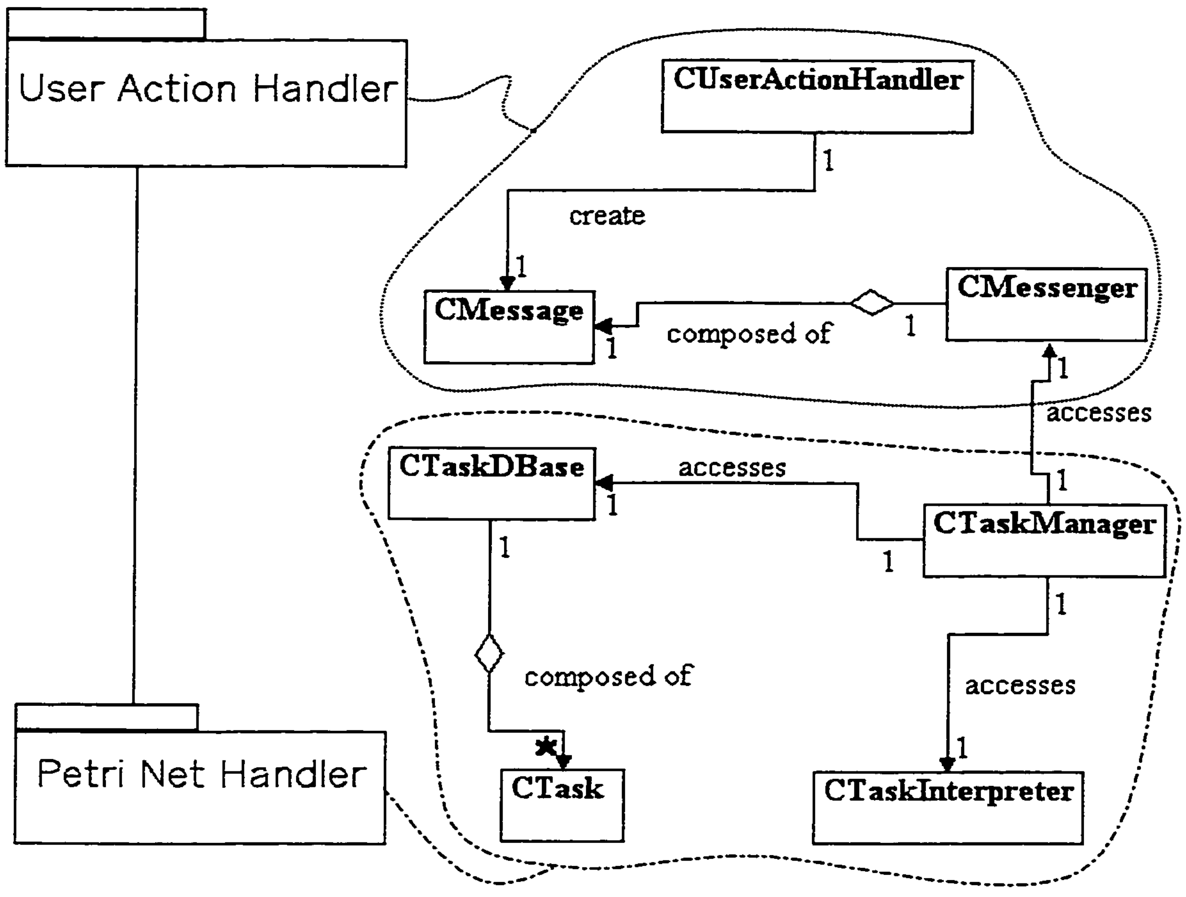


Figure 3-10. The class diagram after joining the two subsystems

In order to represent the interactions between different objects of the software model, we introduce a set of *Sequence Diagrams* [1]. This representation, presented in the following section, focuses on expressing the dynamic behavior of the objects.

3.5 Behavior and Interaction

It would be impossible to present all the possible interactions that take place among the set of objects existing in our software model in sequence diagrams. In this section we simply present two examples that will give a picture of the dynamic behavior of our software model. The following Sequence Diagrams will provide a graphical representation of the message broadcast chronology, among the objects of our model.

3.5.1 Sequence Diagram for Generating - Storing a Message

The process of creating and sending a message involves several interactions between the different objects of both the TDC and the MRS. In order to form a message, as we described in Section 3.4.1, we need several items of information regarding the entities involved in the workspace. These items are stored in the *CWorkcellModel* subsystem of MRS [10],[14], that forces messages to be passed from the objects of *CWorkcellModel* to objects of the TDC. In Figure 3-11, the object CUSER issues a signal to the CWORKCELLMODEL of MRS via the VR interface. This signal identifies the entity involved, and passes the information (for example, object identification, etc.) to the object CUSERACTIONHANDLER. The CUSERACTIONHANDLER creates a CMESSAGE and loads it with the information described in Section 3.3.1.3. Subsequently, the object CMESSAGE calls the object CMESSENGER to store the message and subsequently, the CUSERACTIONHANDLER calls the CTASKMANAGER to notify the existence of a new message in the CMESSENGER.

3.5.2 Sequence Diagram for Loading a Task to the Task Interpreter

The Sequence Diagram in Figure 3-12, shows the process of loading a task from the CTASKDBASE to the CTASKINTERPRETER via the CTASKMANAGER. Having described the functionality of the *Petri Net Handler* subsystem in Section 3.4.2, we proceed with an example of a Sequence Diagram that will put a finer point on that static presentation. The objects involved are communicating with a synchronous broadcasting of messages. That means that the object that sends the message waits until the called object finishes the processing of the message. In the case of loading a task, the object CTASKMANAGER requests a message from the object CMESSENGER. The CTASKMANAGER gets the message and retrieves from the CTASKDBASE the corresponding task. A pointer to the CTASK is passed to the CTASKINTERPRETER, which initiates the process of examining the Petri Net. At a certain instance, the CTASKINTERPRETER requires further input from the CTASKMANAGER. As soon as the CTASKMANAGER obtains the information necessary, it returns control to the

CTASKINTERPRETER. When the CTASKINTERPRETER completes the processing, it returns the resulting task description to the CTASKMANAGER.

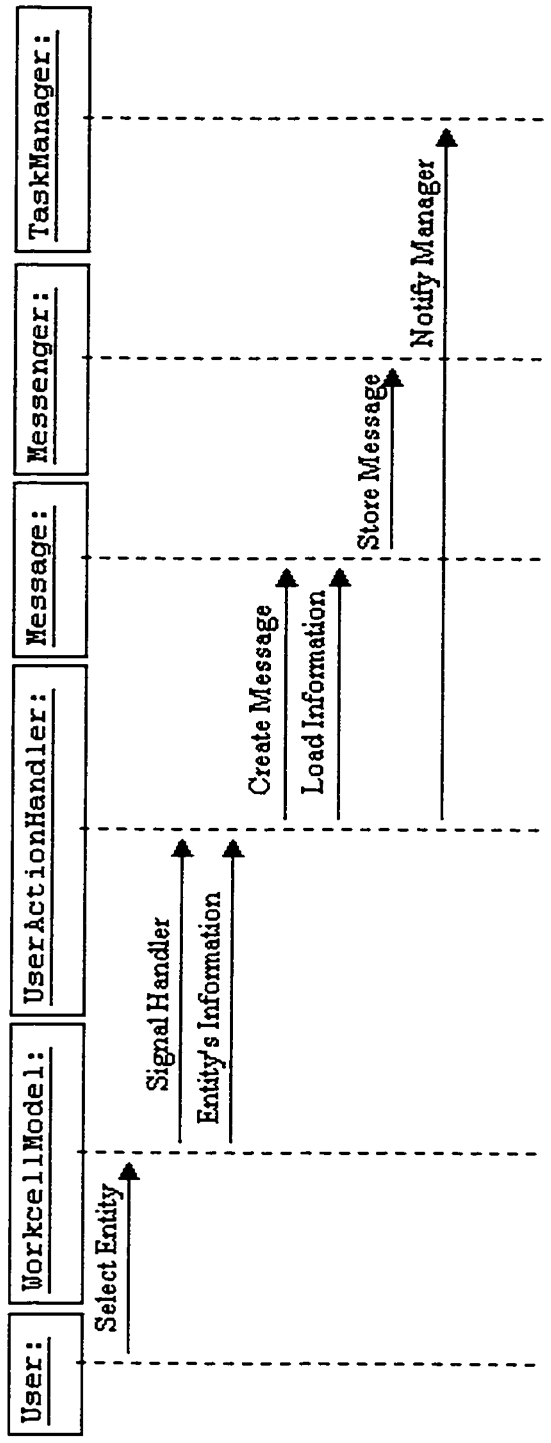


Figure 3-11. Sequence Diagram for Creating & Storing a Message

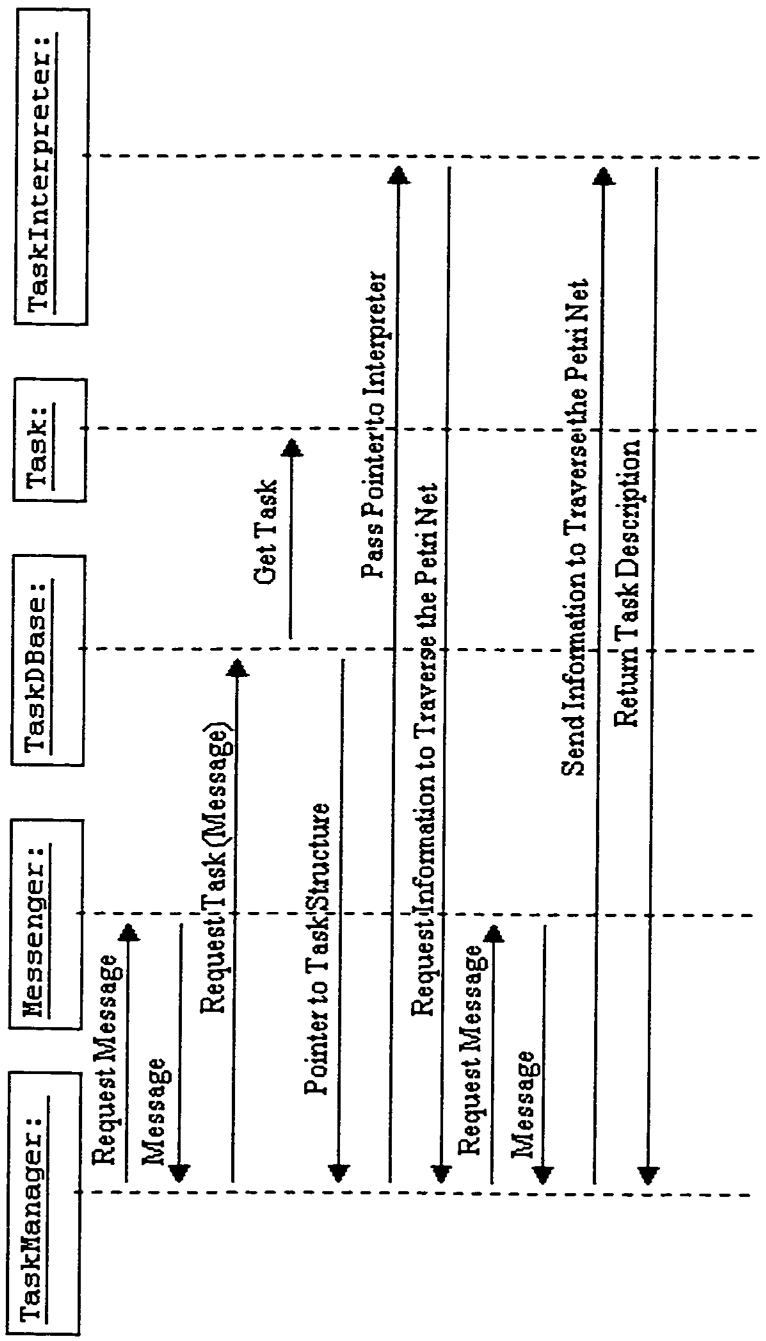


Figure 3-12. Sequence Diagram for loading a Task to the TaskInterpreter

3.6 Summary of Task Deduction Subsystem

The Task Deduction Component (TDC) is the software module responsible for monitoring the actions the user performs in the VR space. It is required to handle the *physical* events that it receives from the VR operator and map them to the corresponding *logical* events. In addition, it implements software structures and algorithms that are responsible for storing and examining Petri Net structures. The Task Deduction Component is composed of a collection of basic software building blocks, each one responsible for performing certain functionalities. These are: the *User Action Detection*, the *Task Interpreter*, the *Messenger*, the *Task Manager* and the *TaskDBase.* Proceeding to the description of the software architecture, TDC is analyzed into two major subsystems: the User Action Handler and the Petri Net Handler. These two subsystems after performing certain processing generate as output a *logical* event. This *logical* event is a task description that is propagated to a subsequent component which is called Task Planning Component of MRS and which will be discussed in the following chapter.

Chapter 4

Task Planning Subsystem of MRS

The purpose of this chapter is to elaborate on the architecture of the task planning subsystem of MRS. To start with, the functional requirements of this module will be addressed in order to understand the role of this subsystem in the overall MRS. Explaining what the system is expected to do is very important in order to realize the purpose of its existence as a component of our simulation system. We proceed by presenting the decomposition of our subsystem into a group of basic units which are closely coupled in order to provide the required functionalities. Next, an object-oriented analysis is performed to address the software structure of each individual unit of our subsystem and to give a formal description of their internal design. The software dependencies and relationships among the basic building blocks of the task planning subsystem are discussed and we raise issues of dynamic behavior, interaction and communication between them

4.1 Introduction

After the analysis and design of the TDC, we come to the point of introducing the next subsystem which is the Task Planning Component (TPC). This is the second major subsystem that will be covered in this thesis. TPC is the key determinant of how well MRS will support the VR-mode of operation, since it is responsible for "understanding" and handling the task descriptions coming from the TDC.

As the user carries out actions in the virtual environment, different task descriptions are deduced by the TDC through the process that has been described in the previous chapter. These abstract task descriptions are forwarded to the TPC which receives them and initiates an entire process of analysis. This process contains the decomposition of the task description received into a set of primitive actions that will be scheduled and executed by MRS. The entire operation of the TPC is based on the fact that the set of primitive actions have not only been pre-defined but also modeled based on a very specific methodology that serves the purpose of our application. Keeping in mind the modularity of our design we form the TPC as a composition of several modules interacting together to support the services required. The main idea behind the TPC's operation is to model as many as possible of the actions that the robots are responsible for performing in the workcell into a "*Task Pattern*", also called an "*Action Pattern*" [22]. In addition, we establish a set of rules that link one Action Pattern to another, providing in that way solutions to more complicated tasks. Finally, by analyzing these patterns, we determine a set of actions that the agents have to follow in order to accomplish the desired task.

4.2 Functional Requirements of the TPC

Following the same principles of *Object Composition* and *Object Inheritance* [5], as in the design of the TDC for reusability, we will proceed with the analysis of the TPC. The *Black Box* reuse style is adopted, since the TPC will definitely be evolving as the needs of MRS grow and cover a wider domain. In Figure 4-1, the Black Box representation of our sub-system shows the inputs provided and the corresponding outputs expected, dictating a specific behavior which will be presented by a set of functional requirements listed in the following sections.

4.2.1 General & Specific Requirements

REQUIREMENT 1: The TPC receives as input logical events from the TDC task descriptions. These are logical events which the TPC is responsible for detecting and categorizing.

REQUIREMENT 2: The TPC is responsible for "understanding" the incoming task descriptions and decomposing them into well defined sets of primitive actions.

REQUIREMENT 3: TPC should be able to provide a mapping between a task description and certain software structures called Action Patterns, which will be further discussed in the sections that follow.

REQUIREMENT 4: The TPC is responsible for analyzing the Task Pattern invoked and generate a software model that will describe the plan that has to be adopted for proper execution of the desired task. This software model is called a Constraint Net [22],[35], and its description in the context of MRS is extensively covered in the following sections.

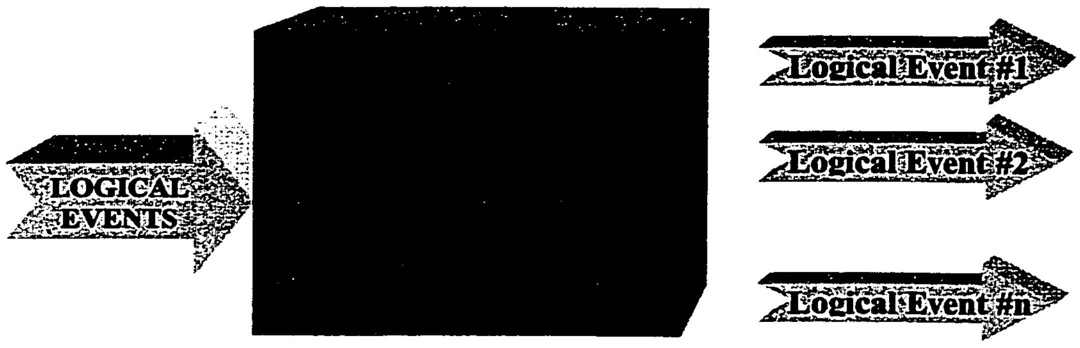


Figure 4-1. Black Box representation of Task Planning Component

REQUIREMENTS 5: TPC should provide mechanisms to analyze the Constraints Nets. This analysis consists of identifying the basic components of a Constraint Net which are the Variables and the Constraints [22],[36].

REQUIREMENTS 6: The TPC should support mechanisms to link different Action Patterns providing solutions to more complex task descriptions.

REQUIREMENT 7: The TPC should assign a specific order of execution to the set of primitive actions that are derived from the analysis of a certain Action Pattern.

<u>REQUIREMENT 8:</u> The TPC should output an ordered set of actions that the robot(s) will follow to accomplish the task

4.3 Decomposing the Task Planning Component (TPC)

Having addressed the functional requirements of the Task Planning Component (TPC), we proceed by presenting the architecture of this subsystem. The software framework of the TPC is created by using paradigms of OOA and OOD. These principles provide a solid foundation which will support a flexible architecture and that will make further expansion of the system as easy as possible.

The following subsection will present the analysis of the problem space. The system's decomposition will provide us with all the software units that comprise the base of the TPC. By elaborating on the dependencies and the interactions generated among these units, as well as their role with respect to each other and to the overall function of MRS, we will justify the decisions taken for the specific architecture.

4.3.1 System Decomposition

The problem space in our domain is the TPC. We need to identify the objects in the problem space that are architecturally significant [4]. In order to achieve this, we need to decompose the entire problem space into a set of software units, that will be tightly coupled and that will be characterized by the services they provide. The entities that we identify are: *Action Controller* unit, the *Action Dispatcher* unit, the *Collection of Action Patterns* unit, the *Constraint Solver* and the *Utility* unit (Figure 4-2). The *Action Controller* unit and the *Action Dispatcher* unit are those that form the link between the TDC and TPC. In fact, the *Action Controller* unit provides an interface that allows abstract task descriptions to be submitted, while the *Action Dispatcher* unit is responsible for categorizing them and supporting their mapping to the corresponding action pattern. All the action patterns are associated with a very specific task description. When the proper input arrives the related action pattern is signaled. The structure of every action pattern will be analyzed in great detail in the subsequent sections. For now, we will only mention that action patterns are composed of two

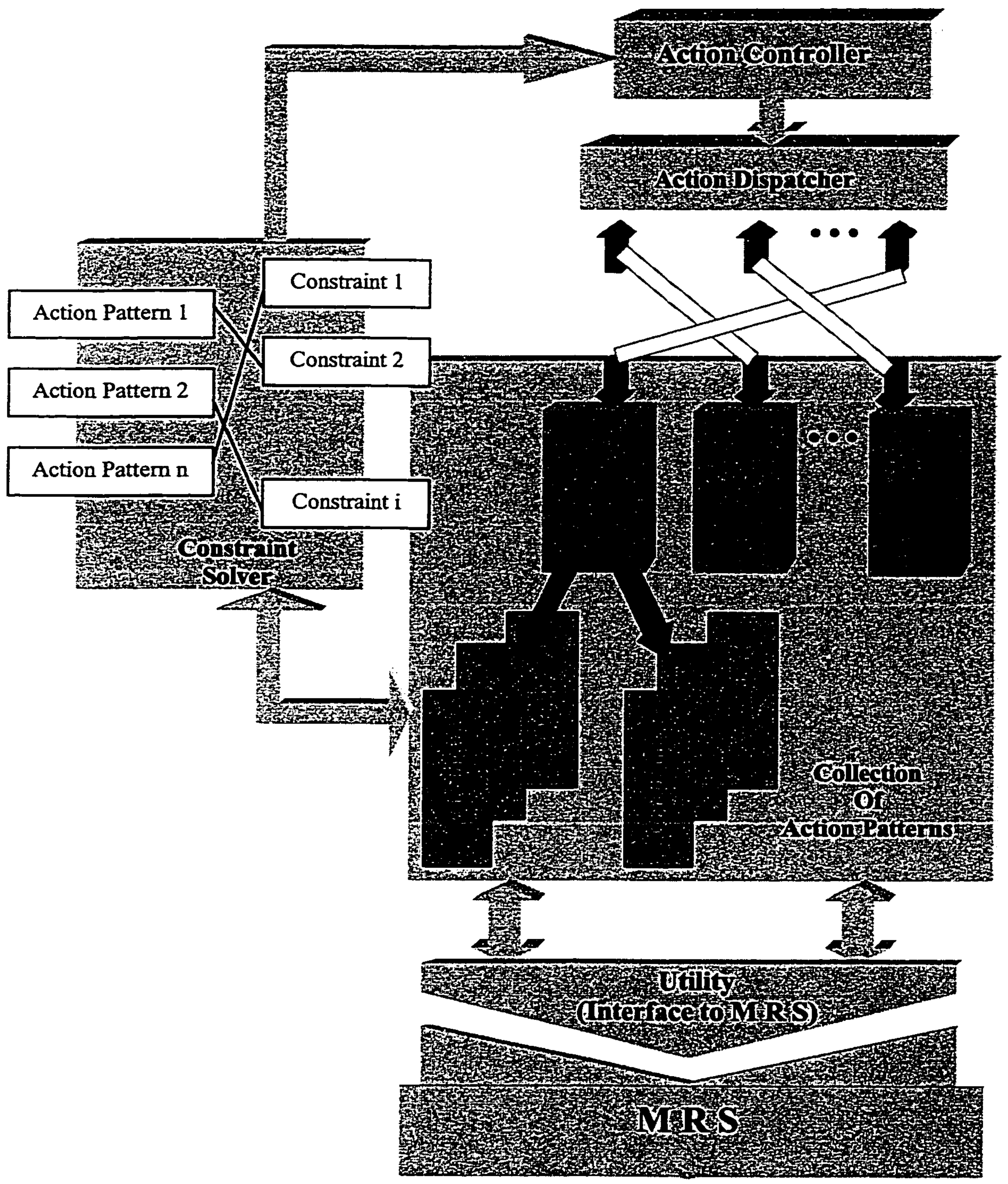


Figure 4-2. The overall architecture of the Task Planning Component

software entities which are called *Variables* and *Constraints*. These entities are essential elements that dictate the handling and the behavior of a certain action that the robots will be assigned to perform. The *Constraint Solver* is the unit that provides some sort of feed-back to the *Action Controller*. We will be able to understand better its role later when we discuss in greater detail the character of its operation. The *Utility* unit is the one responsible for connecting TPC to MRS.

We will continue our analysis by moving to the functional description of each individual unit present in the overall architecture of TPC. The following sections elaborate both on their structure, and the services that they are expected to provide to TPC and to MRS in general.

4.3.1.1 Action Controller Unit

The *Action Controller* unit, is the first layer between the TDC and the TPC. This module provides an interface to access the inner layers of the planning component via the *Action Dispatcher* unit. In addition, the *Action Controller* unit is also responsible for performing a first level of validity check on the incoming task descriptions. By doing so, we get a fast rejection for some of the task descriptions. Moreover, the *Action Controller*, is responsible for controlling the access to the *Action Dispatcher,* in this way providing the capability for keeping track of who is accessing a certain action pattern, and when.

4.3.1.2 Action Dispatcher Unit

The next module is the *Action Dispatcher* unit which is responsible for forwarding the inputs coming from the upper layer and triggering the corresponding action pattern from the *Collection of Action Pattern* unit.

4.3.1.3 Collection of Action Patterns Unit

We create a group of action patterns that form the *Collection of Action Patterns* unit. The size of this group is variable, meaning that new action patterns can be added or existing ones can be modified. Moreover, these patterns can be associated in order to solve more complex problems. These associations among the different action patterns are built upon a very specific set of rules that are established to fit the domain of MRS. In the following sub-sections, we will analyze the logic behind the process of creating an action pattern, as well as the elements that compose a pattern.

Action Pattern. In the context of MRS, an action pattern is a logic structure that is used as a modeling tool to model the tasks the robotic simulation system will support. It is a composition of two major software components: the *Constraint Net* and the *Set of Rules* [22]. The first, is the software structure itself that describes the model of the task, and the second, the set of rules that associates one action pattern with another. Each component will be analyzed further.

Constraint Net. A *Constraint Net* is a software structure that involves two types of parameters: (1) variables and (2) constraints, in order to model each action pattern that MRS needs. In Figure 4-3, we can see an example of a constraint net. The cylindrical shapes correspond to the variables and the arcs to the constraints. The constraints are differentiated to unary and binary. A unary constraint is one that begins and ends at the same variable. A binary constraint is the one that associates two distinct variables. For the specific domain of MRS we can think of the variables in a constraint net as the entities that are involved in a task, for example robots, end-effectors etc. Also, we think of the constraints as the conditions that must be satisfied in order to establish proper associations between these entities.

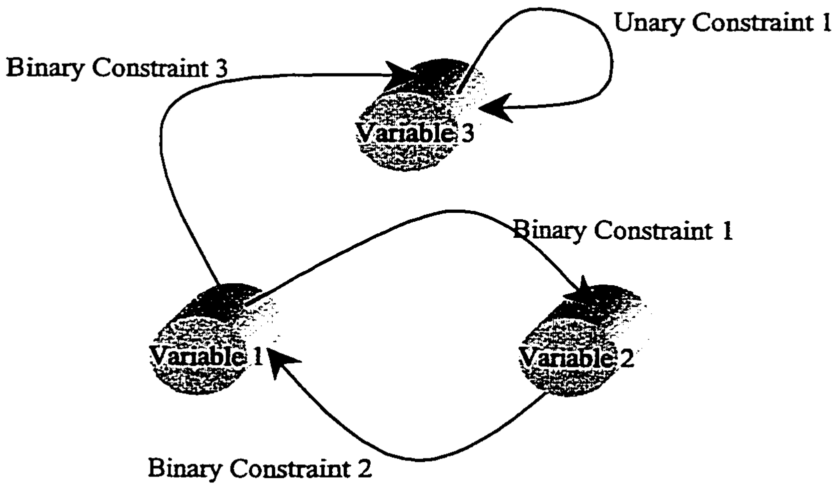


Figure 4-3. The graph represents a typical Constraint Net.

The TPC is able to generate a plan of execution for a certain task, if and only if, the constraint net that models the specific action pattern has a solution. In order for a constraint net to have a solution, all the variables must be assigned with values and all the constraints must hold. If the above conditions are satisfied we have successfully defined a set of entities (or variables) that have been assigned with some specific values, and which behave according to the constraints that are established between them. If there exists at least one constraint that does not hold, then the constraint net cannot be solved and we employ the set of rules that we defined in order to call a new action pattern. The process repeats itself until either all constraints get satisfied or until the set of rules that we have cannot provide us with a solution. Let us consider the logic synthesis of a constraint net. We know that the components of a constraint net are variables and constraints. The question that arises is the following: "How do we define the dependencies between certain variables and the constraints associated with that variable?". In order to answer this question, let us look at Figure 4-4. We can see that we have a variable a_i, and a collection of constraints, some of them are generated from that variable and are moving outwards and some of them end at that variable. We define a constraint to be a function of a certain variable a_i if and only if the constraint is generated from that specific variable. If this is the case then the constraint depends on that variable.

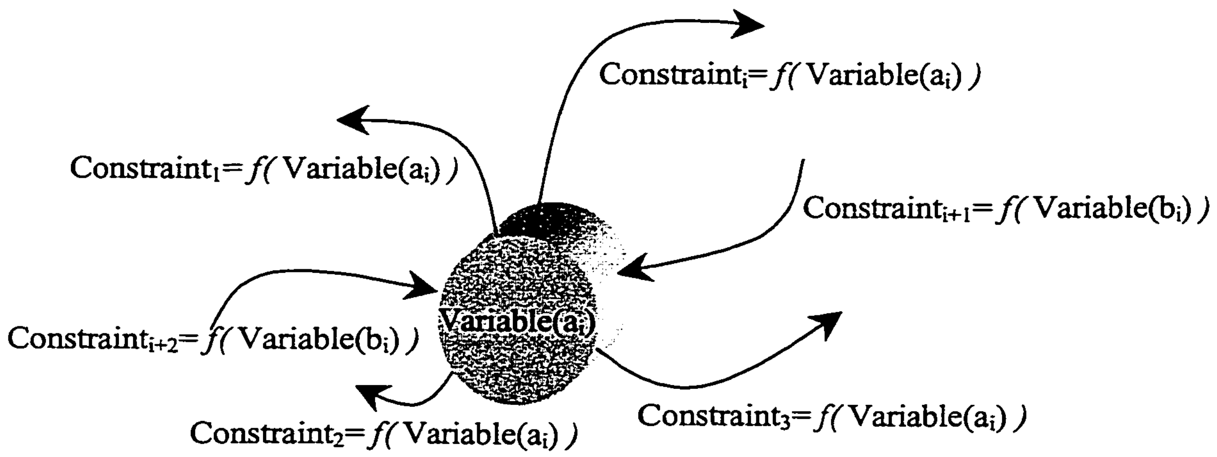

Constraint$_i$=$f($Variable(a_i)$)$
Constraint$_1$=$f($Variable(a_i)$)$
Constraint$_{i+1}$=$f($Variable(b_i)$)$
Constraint$_{i+2}$=$f($Variable(b_i)$)$
Variable(a_i)
Constraint$_3$=$f($Variable(a_i)$)$
Constraint$_2$=$f($Variable(a_i)$)$

Figure 4-4. Dependencies between Constraints and Variables

In the same figure, a second group of constraints, *f(Variable(b_i)*, has as an ending point the variable a_i. This set of constraints depends on the hypothetical variable b_i, and that is noted by putting b_i as the argument of the function. By categorizing the constraints with respect to the variables that they depend on, we establish a formal description for both the constraints and the variables that compose a constraint net structure. This formalism promotes not only a more concrete design, but also enables us to model the implementation of the algorithm to analyze a constraint net in such a way as to follow a certain methodology.

Rules to Associate one Action Pattern to Another. The second part of an action pattern is the set of rules that are used to associate every unsatisfied constraint to either an action pattern or to a specific solution other than generating an action pattern. By doing so, we allow the system to solve by itself the unsolved constraints, and proceed further to the planning process of a task.

4.3.1.4 Constraint Solver Unit

The *Constraint Solver* is the part of the overall system that maps the constraints that are not satisfied to the corresponding pre-assigned solutions. A solution to an unsatisfied constraint could be a simple process or it could be a more complicated one. In the case where the process is simple, the solution could simply involve the assignment of a different value to the variable that the unsatisfied constraint is dependent on. On the other hand, when the process is complicated, the solution could involve the selection of a different action pattern that will satisfy the unsolved constraint. We should note that not all unsolved constraints can be solved. Situations exist where a constraint cannot be solved and the planner is unable to provide a solution by itself. In that case the control is passed to the user, who is informed about the deadlock situation that has occurred and the user is now responsible for performing an action (if the user is capable) to solve the unsolved constraint. In the case where the *Constraint Solver* is able to match the unsolved constraint to a certain solution, two possible situations exist. First, the control is passed to the *Action Controller* which will trigger the action pattern that the mapping rule in the *Constraint Solver* dictates to solve the unsatisfied constraint. Second, the control is passed back to the *Collection of Action Patterns*. We can see these possibilities in Figure 4-2, where a bidirectional communication can be established between the *Constraint Solver* and the *Collection of Patterns* while one way communication is feasible between the *Constraint Solver* and the *Action Controller*. In Figure 4-5, we can see an example of the operation described. A constraint net that describes a hypothetical action pattern with four variables and five constraints is employed. In addition, a set of rules is formed that is associated with every constraint of the action pattern. Moreover, the *Constraint Solver* provides a table that maps the rules to a set of solutions.

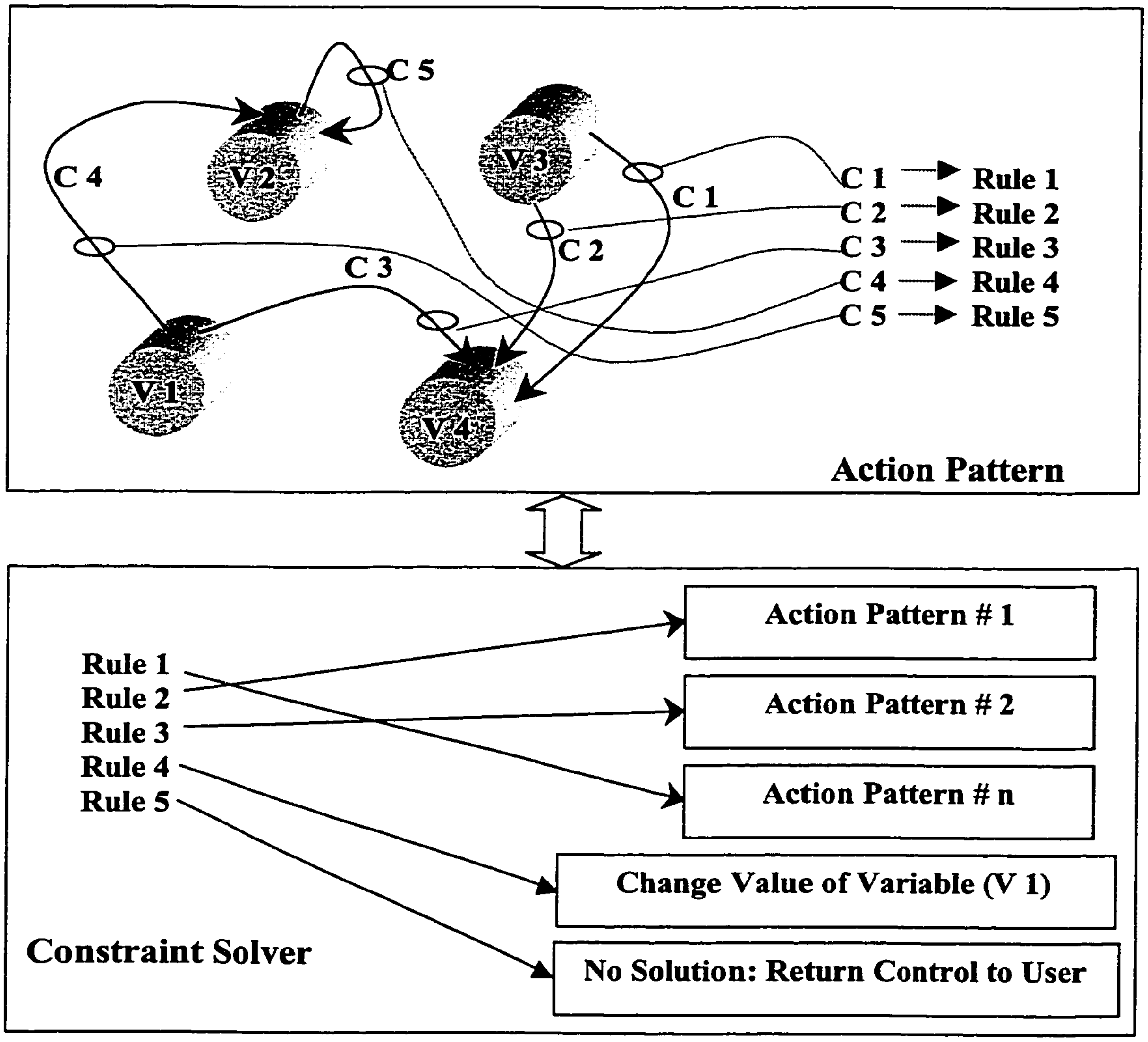


Figure 4-5. Interaction between an Action Pattern and a Constraint Solver

We have mentioned that certain situations exist where the solution requires a different value to be assigned to the variable in order for the function of the constraint to be satisfied. When that situation occurs then the TPC has to communicate with MRS to obtain the values for the specific variable. Recalling the fact that the variables of a constraint net represent entities of the workcell, in order to obtain values for these variables, we need to request them from MRS. For this purpose the existence of an interface that will provide communication between TPC and MRS is absolutely necessary.

4.3.1.5 Utility Unit

The *Utility* unit is the interface that groups a collection of static function calls that will allow the TPC to retrieve information from MRS. The *Utility* unit allows the TPC to adopt an independent architecture that will be easy to interface with MRS. Some typical information requested could be the unique identification number of the object involved or the robot assigned to execute the task. According to the description of the architecture of MRS developed by Bryson [10], different layers of the architecture have to be accessed to obtain the necessary information. Figure 4-6, presents an example of the functions grouped in the *Utility* unit.

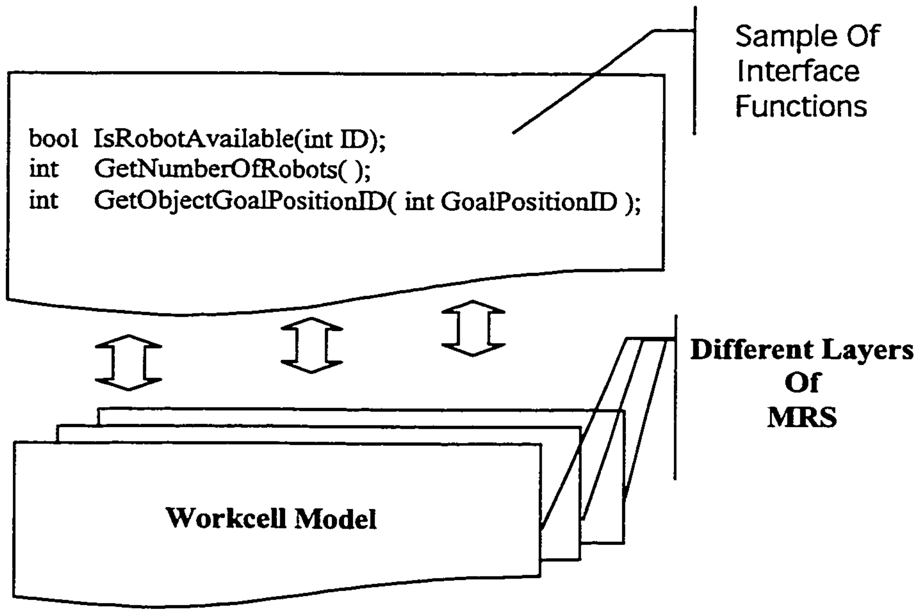

Sample Of Interface Functions
bool IsRobotAvailable(int ID);
int GetNumberOfRobots( );
int GetObjectGoalPositionID(int GoalPositionID);
Different Layers Of MRS
Workcell Model

Figure 4-6. Example of the Utility unit.

Having described the overall architecture of the TPC and the operation of every unit composing that module, we proceed to the analysis of the software structure. Object-oriented principles dictate a modular design which will be presented in the following sections.

4.4 Design of Internal Units of the TPC

In the following sections we will divide the problem space into two major software subsystems that will be analyzed separately. These subsystems represent the framework which is expected to handle the requirements that we have for the TPC. By addressing the software synthesis of the components involved individually, we present the expandability and the adaptability demonstrated by each of these modules via the proposed software design. We begin with the description of the first subsystem.

4.4.1 Controller

Controller is the subsystem for achieving the activation of an action pattern. It is the software layer that connects the TDC and the TPC. In fact, it provides a way to distinguish between the different task descriptions coming from the TDC, and to provide a mapping to the corresponding action pattern. So the input to the subsystem is a task description and the output is the corresponding action pattern. In order to isolate the algorithmic details of these functionalities from the main component, the *Controller* uses the collection of classes associated as shown in the class diagram in Figure 4-7.

<u>CACTIONCONTROLLER class:</u>

The CACTIONCONTROLLER is an abstract class that inherits its member functions to the CMYACTIONCONTROLLER class. It provides a simple interface to operations while the implementation of the actual functions is given in the CMYACTIONCONTROLLER class. By forming this set of virtual functions in the CACTIONCONTROLLER class, we will be able in the future to replace, if necessary, the existing implementation in the CMYACTIONCONTROLLER class while keeping the same interface in the CACTIONCONTROLLER class. In addition, it has a relationship "accesses" with the abstract class CACTIONPATTERN of type

Association. Moreover, the association is *one-to-many*, since one instance of the class CACTIONCONTROLLER can be associated with many instances of the abstract class CACTIONPATTERN.

CACTIONPATTERN class:

This class is also an abstract one, and represents a common interface to all the action patterns that are implemented in the TPC. It provides a collection of definitions of virtual functions that will permit, for the same function, different implementations to be provided from the different action patterns. For instance, we have several action patterns, all of them have the *BinaryConstraints()* method but each one implements it differently. This way of organizing our software structure creates the flexible design we need.

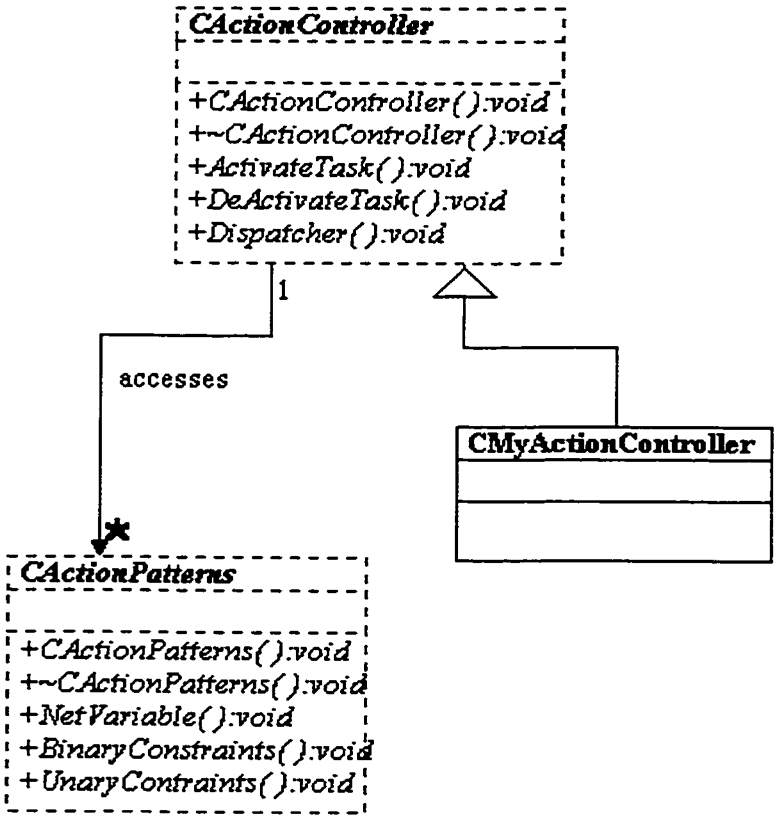


Figure 4-7. The class diagram for the Controller subsystem

CMYACTIONCONTROLLER class:

The CMYACTIONCONTROLLER class simply provides the algorithmic details for the abstract class CACTIONCONTROLLER.

4.4.2 The Action Pattern Handler

The second major subsystem of the our problem space is the *Action Pattern Handler*. This is the part responsible for forming and examining the action pattern required for a certain task description. In fact, it has the responsibility for performing the analysis of every action pattern, communicating with MRS to retrieve information about the task that has to be accomplished and also, providing solutions in the cases where the constraints are not satisfied in the constraint net.

The *Action Pattern Handler* subsystem receives as input from the *Controller* the type of pattern that has to be used. According to the type of pattern a constraint net gets generated. We note here that each pattern has been pre-assigned with a certain constraint net structure. That structure gets instantiated when the corresponding action pattern gets called. By instantiating a certain constraint net structure, we automatically generate the group of variables that are included in the structure and we assign to them values. These values come from MRS that has to be accessed (e.g. the identification number of a robot). After the variables have been created and a set of values has been assigned to them, the *Action Pattern Handler* proceeds to the next step which is the analysis of the constraints that are established among those variables. The *Action Pattern Handler* subsystem does not provide a general solution to all the unsatisfied constraints that are detected during the tracing of a constraint net structure. The process of examining a certain constraint net could very easily result in a situation where a solution cannot be provided automatically by the system itself, and the user has to intercede in order to address the difficulty that has occurred. The class diagram in Figure 4-8, represents the software components of the *Action Pattern Handler*. In addition, some definitions of the functions in these classes are also included in the class description in order to give a broad picture of the services that every class provides. During the analysis of every class

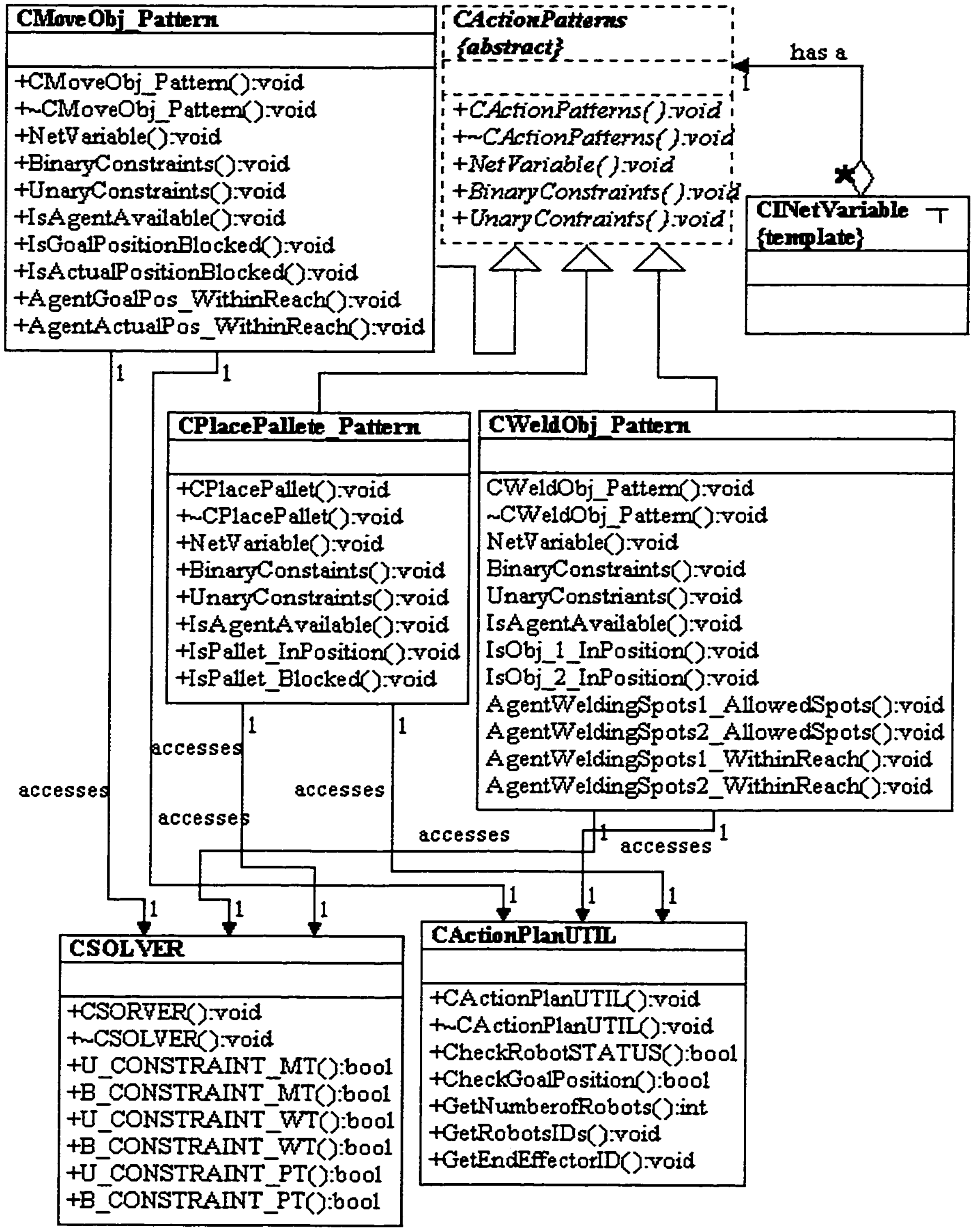


Figure 4-8. The class diagram for the Action Handler System

individually, we will be able to acquire a better understanding of the dependencies that we establish among them and their contribution to the overall software architecture of TPC.

CMOVEOBJ_PATTERN class:

This class represents the action pattern responsible for modeling the task of moving an object in the workcell. Basically, it contains all the variables and all the constraints that must be satisfied in order for the entire action pattern to be executable. CMOVEOBJ_PATTERN class is responsible to assign values to the variables of its constraint net. This can be achieved by accessing the CACTIONPLANUTIL class, which contains all the definitions of functions for retrieving information from MRS. Then the CMOVEOBJ_PATTERN has to examine every unary and binary constraint in the pattern. The evaluation of both types of constraints will definitely require accessing MRS. For instance, a typical constraint could be whether or not, the position of a certain object is within reach for a specific robot. Since MRS contains the algorithmic tool to calculate inverse kinematics [6],[10], we pass the information required (e.g. robot ID, object ID etc.) and we let MRS to do the processing. In case where the specified robot cannot reach the target, the CMOVEOBJECT_PATTERN will have to try to solve the problem, by calling the CSOLVER. Since we are working on a multi-robot environment, the CSOLVER will automatically assign a different robot as active and will return control to the CMOVEOBJ_PATTERN class which will try to solve the pattern with the new robot as active. As we can see we try to make CMOVEOBJ_PATTERN a self-defined and self-contained structure. What we mean is that it defines itself all the variables and the constraints it needs. By doing that, we force the class to a minimum number of external dependencies. To be more precise, the relationships between CMOVEOBJ_PATTERN and CSOLVER, CACTIONPLANUTIL, are "accesses" of type *Association*, and moreover, all of them are *one-to-one* relationships.

CWELDOBJ_PATTERN class:

This class follows the same design principles as those described above. It is responsible for modeling the task of welding two objects together. The objects could be single objects or composites. Composite objects are those that have been produced by the welding of two or more single objects. This class is also associated with the CSOLVER and the CACTIONPLANUTIL with *Association* type of relationship.

CPLACEPALLET PATTERN class:

The CPLACEPALLET_PATTERN is similar to the CMOVEOBJ_PATTERN class not only on the design principles that it follows but also on the type of action pattern that it models. This class is responsible for modeling the task of placing the welded object in the pallet which is placed at a certain position. The purpose of distinguishing this task to a separate model is that it involves a set of constraints that cannot be generalized and therefore we cannot use the class CMOVEOBJECT_PATTERN to model that pattern. In addition, we do not want to make any class very complex just because we want it to model several actions. By using CPLACEPALLET_PATTERN class, we create a more precise model of the task that the robots have to execute. Once again, this class is associated with CACTIONPLANUTIL and CSOLVER classes.

CSOLVER class:

This class undertakes the responsibility for solving the unsatisfied constraints that are detected during the analysis of each action pattern by the corresponding class. The class groups the solutions that it offers based on the action pattern that requests a solution. For instance, we can see in Figure 4-8, function definitions of the type: *U_CONSTRAINT_MT()* and a *U_CONSTRAINT_WT()*. The first function definition is responsible for handling the requests for solutions to unsolved unary constraints coming from the class responsible for modeling move-object tasks. The second is responsible for handling requests for solutions also to unary constraints but coming from a different class. In fact, for the second case the requests are coming from the class responsible for modeling weld-object tasks. Moreover, the CSOLVER class "decides" on the type of solution that a certain constraint requires. We should note that its decisions are based on a set of predefined rules that exist. In the case where the solution requires a new action pattern to be involved, the CSOLVER communicates with the *Controller* subsystem according the procedure described in Section 4.3.2.1.

CACTIONPLANUTIL class:

The CACTIONPLANUTIL class hosts the interface of the TPC with the rest of MRS. It provides a collection of static function definitions that provide a way to send and receive information to and from different layers of MRS. Typical items of information are robot identification numbers, number of robots available in the workcell, object positions and orientations, etc.

CINETVARIABLE class:

The CINETVARIABLE class is a *template* [1] class that allocates space to construct the software structures that will represent the variables in the constraint nets. Basically, this class creates arrays of different types for storing the entities that the variables represent on a constraint net. Elaborating a bit more on this point, we will give an example. Let us consider a variable on a hypothetical constraint net, that has to keep a record of all the robots that are present at a certain point in time in the workcell. That variable will be an object of the class CINETVARIABLE. The requirements demand that the CINETVARIABLE allocate space to its object, let us call it *Agent,* in order to store all the identification numbers of the robots in the workcell. Again referring to Figure 4-8, we see the relationship between the CINETVARIABLE and CACTIONPATTERN to be an *Aggregation* since every instance of the class CACTIONPATTERN "has a" collection of CINETVARIABLEs.

To recapitulate, the two subsystems that we have to join together are the *Controller* and the *Action Pattern Handler*. When a task description is passed from the TDC to the CACTIONCONTROLLER, the type of action pattern required is identified. According to the type of action pattern that gets selected, the corresponding objects of CINETVARIABLE class are instantiated and the examination of the constraints involved gets initiated. During this process we have continuous access to different layers of MRS via the CACTIONPLANUTIL, in order to get the values for the variables and to check the constraints. When we come across an unsolved constraint the CSOLVER gets signaled to handle that constraint. The solution could be either a new value for the variable that the unsolved constraint depends on, or it could be a new action pattern that has to be called in order to solve the problem. In either case, all the constraints must be satisfied in order for the action that the action pattern models to be executable. The following class diagram shows the entire software model that joins the two subsystems together (Figure 4-9).

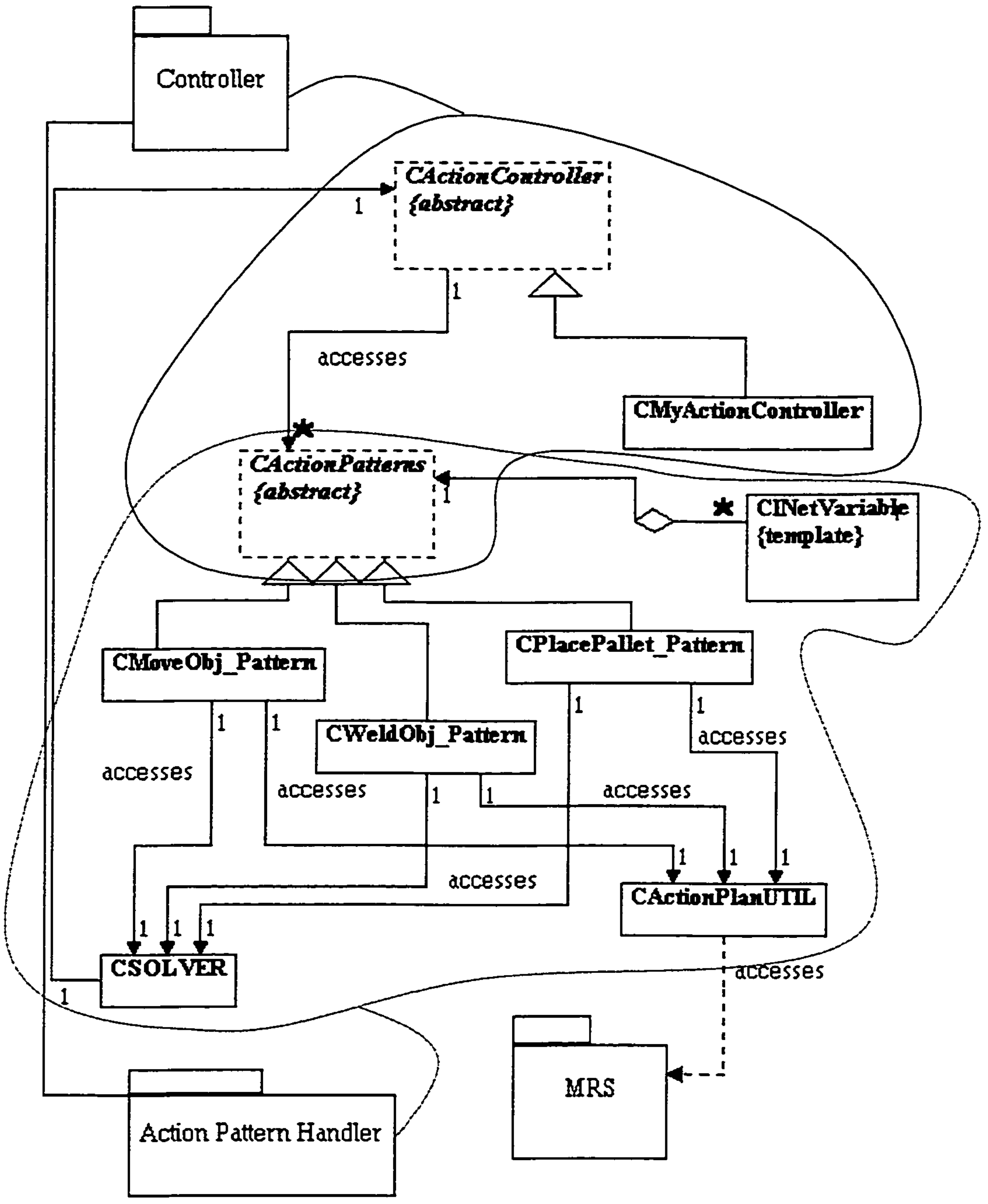


Figure 4-9. The class diagram that joins two subsystems together

4.5 Behavior and Interaction in the TPC

In the following section we will present an example of the dynamic aspect of the subsystems. The process we have selected to present with a sequence diagram demonstrates the interactions that occur in order to form a constraint net with five variables. We focus on the construction of five objects of type CINETVARIABLE and also to load the corresponding values from MRS.

4.5.1 Sequence Diagram for Instantiating Variables

In Figure 4-10, we see the object CACTIONCONTROLLER to send a message, MOVE_OBJ, to the object CACTIONPATTERNS containing the type of action pattern that has to be triggered. The last one receives the message and creates an object CMOVEOBJ_PATTERN. As soon the pattern is created, all its variables have to be generated and also to be assigned with the appropriate values from MRS. We can see this process by looking at the messages that are passed between the objects CMOVEOBJ_PATTERN and CINETVARIABLE. In addition, we have calls being issued from the CMOVEOBJ_PATTERN to the CACTIONPLANUTIL. These messages are requests for the values of the variables that the constraint net requires for that pattern. We point out that the object CACTIONPLANUTIL accesses different MRS layers to retrieve the information. This is not shown since these layers are external to our subsystem. Moreover, to realize the interactions between the object CSOLVER and the rest of the subsystem, we assume that a certain constraint of the pattern is validated. In fact, the constraint is an unsatisfied one, which means that the CMOVEOBJ_PATTERN has to call the CSOLVER to report the situation. Another assumption that we make is the following: in order to solve the problem, a new action pattern has to be triggered. That can only be achieved through the CACTIONCONTROLLER which is called by the CSOLVER. This small sequence of messages gives us a general idea of the flow of messages between the different objects. Obviously many other possible sequences exist but it is impossible to present all of them in the thesis.

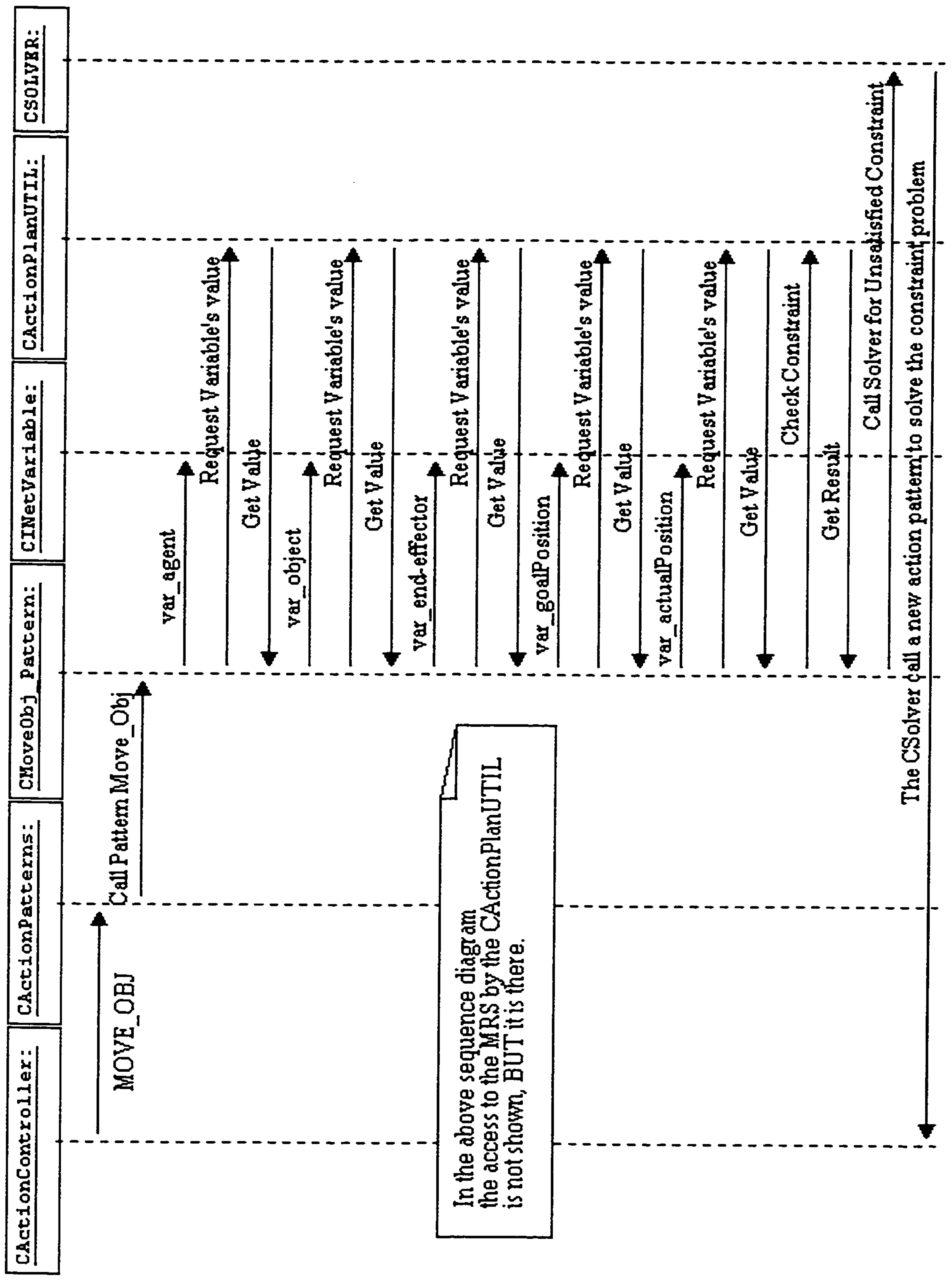


Figure 4-10. Sequence diagram for instantiating Variables

4.6 Summary of Task Planning Subsystem

The Task Planning Component (TPC) is the software module that is assigned with the responsibility of analyzing the task descriptions generated from the Task Deduction Component. It receives as input the task descriptions that TDC produces and by performing certain analysis decomposes them into well defined sets of primitive actions. This decomposition is done by employing certain predefined structures called Action Patterns. As a result of this process generates a software model of the plan that has to be adopted in order to execute the specific task. This software model is called a Constraint Net. TPC supports mechanisms of analyzing both the Action Patterns that are involved and the Constraint Nets that comprise each Action Pattern. The Task Planning Component is composed of a number of basic building blocks, each one assigned with specific responsibilities. These are: the *Controller*, the *Action Dispatcher*, the *Collection of Action Patterns*, the *Constraint Solver* and the *Utility*. Moving further to the description of the software architecture, TPC can be seen as two subsystems: the Controller and the Action Pattern Handler. These two subsystems analyze the task description that is provided as input and generate as output an ordered set of actions that the robot(s) will have to follow in order to accomplish the task. In the following chapter we will study a complete example that addresses an abstract description of a welding operation of two objects and in which both TDC and TPC are involved.

Chapter 5

Case Study

The purpose of this chapter is to show a complete example that will demonstrate the functionality of the two components, TDC and TPC, that were analyzed in detail in the previous chapters.The operation that will be used as an example is the welding of two metal pieces together. The purpose of this example is to focus on the operations that the two modules perform in order to produce a set of primitive steps that the execution has to follow in order to plan successfully the specific action. In the first part, we start with the activities that the user performs in the VR environment and what effect these actions have on the TDC. In the second part, we describe how the corresponding action (welding operation) that gets generated from the TDC is passed to the TPC and is analyzed in terms of the corresponding action pattern.

5.1 Analysis of the Scenario

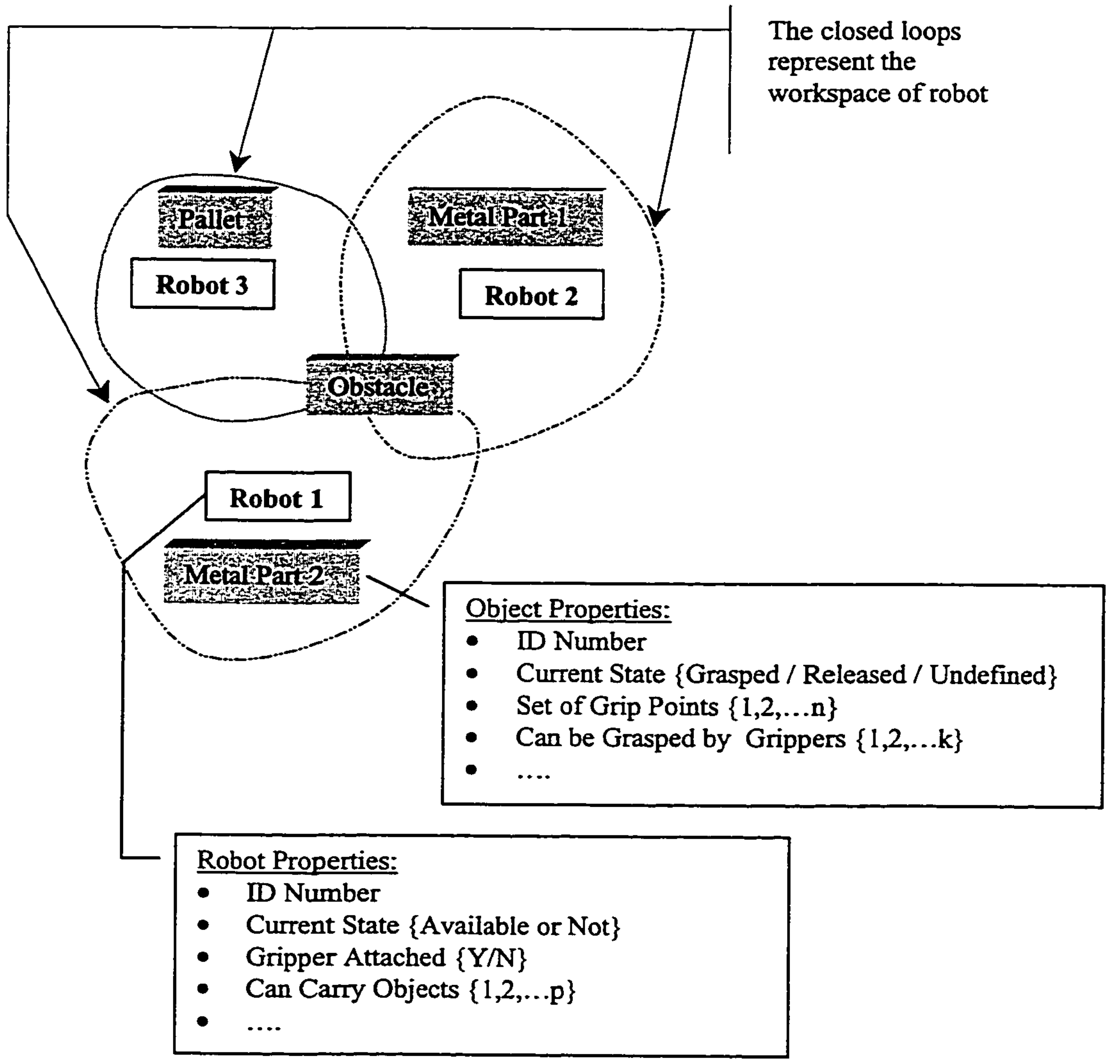
The process that we address in the following sections consists of multiple stages. At the beginning, the user in the VR environment has to "select" the parts that will be welded and bring them, one by one, to the location in the 3-D environment that is assigned to be the welding position. The system recognizes the actions performed and helps to plan the process of welding. Although the process might appear to be simple, several points have to be taken into account. The user does not have to move the parts to the goal position directly. The system allows the user to move them around in the workspace in order to organize its workspace in a more natural way, before it proceeds to the welding process. In order for a task description to be generated, certain steps have to be followed. In fact, the user has to bring first one part (any one) in the welding location then the other, and only after that the task description or action as described in Section 3.3.1.6.3, *Weld_Metal_Part*, gets generated. Since the task that the user is trying to accomplish is recognized by the system, what has to be done next is the analysis of the task description by the TPC. At this stage, the system processes the task description and generates a set of primitive actions.

5.2 Describing the Workcell

Before proceeding further, we should describe briefly the environment and the objects that are present. This is quite important since several of the properties that the objects and the agents (robots) demonstrate are used by the TDC and TPC. We assume that MRS, at a very specific instance, contains three robots and four objects. The four objects correspond to the pallet where the welded parts will be placed, the two metal parts that have to be welded and a third object that we introduce purposefully, to demonstrate how the system will behave in terms of dealing with obstacles.

Every robot in MRS has several properties that govern its operation, e.g., its status that identifies if it is available to perform a certain task, has a unique identification number, can carry certain objects, etc. In addition, every object in MRS also has certain principles that dictate a certain behavior, e.g., has a set of pre-defined points that can be grasped from, can only be grasped by certain grippers, etc. In addition, all the objects in MRS have dual 3D representations. The first one, is a solid 3D rendered object, and the second is a wireframe representation. When the user manipulates an object in the workcell, he manipulates its wireframe representation while the solid one remains at the position and orientation that the physical object is located. The solid 3D model of an object is moved only when the actual manipulator acts on it, giving in that way a visual effect on the motion of the physical model. This is very useful since it provides the operator with the capability to move the virtual object in the workcell, while having at the same time, knowledge of where the physical object is located.

All these properties play a significant role in order for the TDC and TPC to function properly. In Figure 5-1, we present the situation we have in a graphical way, showing some of the properties that the entities in MRS demonstrate just before we start. We can see the circular loops that represent the workspace of each robot and the objects that are placed within the robots' workspace. For demonstrating certain issues, we assume that the welding location is at the same as where the obstacle is located.



Object Properties:

- ID Number
- Current State {Grasped / Released / Undefined}
- Set of Grip Points {1,2,...n}
- Can be Grasped by Grippers {1,2,...k}
- ....

Robot Properties:

- ID Number
- Current State {Available or Not}
- Gripper Attached {Y/N}
- Can Carry Objects {1,2,...p}
- ....

Figure 5-1. The hypothetical setup of the workcell

5.3 Joining the Two Metal Parts Together

Having the overall picture of the workcell, we proceed to the first step which is the deduction of the task that the user wants to perform. The user through the VR interface selects the 3D model of **metal part 1**. This is a physical event that gets detected by the *User Action Detection* unit of the TDC (see Section 3.3.1.1), the identification number of the object is then retrieved and by following the procedure we described in Section 3.3, the corresponding Petri Net structure is selected to be examined. We have seen in Section 3.3.1.6, the description of the Petri Net structure responsible for generating two different actions "Move_Metal_Part" and "Weld_Metal_Part". In the description that follows we will be referring to Figure 3-5 that shows this structure.

We assume that the state of the **metal part 1**, at the beginning is at the state **Free Position** (see Section 3.3.1.6.1). So, when we load the Petri Net we start from that state, referring to Figure 3-5, **State(2)**. When the user "GRASPs" the **metal part 1**, **Transition(1)** gets satisfied and the state of the object changes to **State(4)** which is **Metal_Part In Motion**. At this state, the user drags the wireframe representation of the object in the 3D virtual environment. Assume that the user issues a "RELEASE" signal to the **Metal_Part** when it reaches the predefined welding location. The TDC detects the action the user has performed and by continuing the tracing of the Petri Net, it moves from **State(4)** to **State(5)** which represents the object's state, **Metal_Part Released**.

This state initiates a sequence of requests to MRS to check whether or not the location where the wireframe was released is inside the workspace of any of the robots or not. In addition, it checks whether or not the position released is a welding location. If it is, checks if any other object that **metal part 1** can be welded with, is at the welding location too. Since the user released the wireframe at a welding location, **metal part 1** is the first object that arrives at that location and thus **Transition(4)** is fired and the state where the object moves to is **Free Position**. We can see that no action is generated although the user moved one of the two objects to the welding location. This is a design decision that we made in order to make sure that before we initiate the action "Weld_Metal_Part", both objects have to be moved to the welding location by the user. Subsequently, the user "Selects" the **metal part 2**, as the next object to manipulate. The steps that are followed are exactly the same up to the point where it reaches **State(5)**. Since now we have already one object placed at the welding location, **Transition(5)** gets fired and the state of the object is **State(1)** while the action generated is **Action(2)**. The resulting object after the welding will be a new object that will be a composition of **metal part 1** and **metal part 2** and will be assigned a new identification number. Its initial state will be **State(1)**, meaning that it is already in **Welding Position**.

In the case where the user releases **metal part 2** at a location different from the welding position, then assuming that the location is within the workspace of at least one of the robots in workcell, **Transition(3)** is fired and the state of the object goes to **State(2)**, generating in the meanwhile **Action(1)**, which is **Move_Metal_Part**. What this means, is that one of the robots will have to move the metal part to the location specified by the user, while the state of the object is **Free Position**. After that, the user can select again the same object (**metal part 2**), from the location previously released and this time move it to the welding location where the first metal part is already located.

Note here that, although the welding location is occupied by a different object-obstacle, the TDC does not block the user from continuing in describing the task that has to be achieved. What actually happens, is that it lets the TPC to handle that problem at a later stage by introducing proper planning. This is quite useful since the user does not have to worry about moving the obstacle to one side and then proceed, but can simply go ahead and start describing the task that is needed.

As soon as the action **Weld_Metal_Obj** is generated by the TDC, that signal is sent to the *Action Controller* unit of the TPC. The entire process in the TPC is presented in the following sections, where the decomposition of the **Weld_Metal_Obj** task description takes place.

5.3.1 Tracing the Welding Pattern

The task description for welding **metal part 1** and **metal part 2** involves the analysis of the related action pattern. In Figure 5-2, we can see the action pattern that corresponds to the task description **Weld_Metal_Obj**. When we load this structure, the variables that are contained in the action pattern are assigned with the entities that are involved in the task.

The assignment is done for some of the variables directly and for the others indirectly. When we say directly, we mean that they are read directly from the MRS workcell setup to the constraint net variables. Table 5-1 and Table 5-2, show the assignment of values that we make to the variables that the constraint net has. In the first table we can see the variables **?object_1**, **?object_2** and **?agent**. The first two are directly assigned from the MRS workcell with the values of **metal part 1** and **metal part 2**. The third one does not get assigned with a single value, but gets assigned a range of values. The value of the robot that it receives changes according to the robot that is active at a certain instant. This is the reason we characterize this assignment as not a direct assignment. In a similar way, we assign the values for the variables presented in the second table.

As soon as all the variables are assigned with values, the constraints start to be evaluated. First we examine the unary constraints then the binary. The unary constraints we have are: (1)**?IsAgentAvailable,** (2)**?IsObj_1_InPosition** and (3)**?IsObj_2_InPosition**. The first one evaluates whether or not there is an agent (robot) which is available to undertake the task. If the agent selected is not available, that means the constraint is false. The *Constraint Solver* is passed the unsolved constraint, and automatically assigns a different robot to the variable **?agent**. After that, the same unary constraint is re-evaluated. If that constraint or any other cannot be solved, the process terminates and the user is informed about the situation. Usually, in a welding operation, there is a dedicated welding robot. That means, another robot cannot take over the welding operation in the case where the dedicated robot has its status marked not available. What happens then is that since the *Constraint Solver* cannot solve the problem, it returns control to the user in order to solve the unsatisfied constraint. In the case where the unsatisfied constraint comes from the fact that the agent is

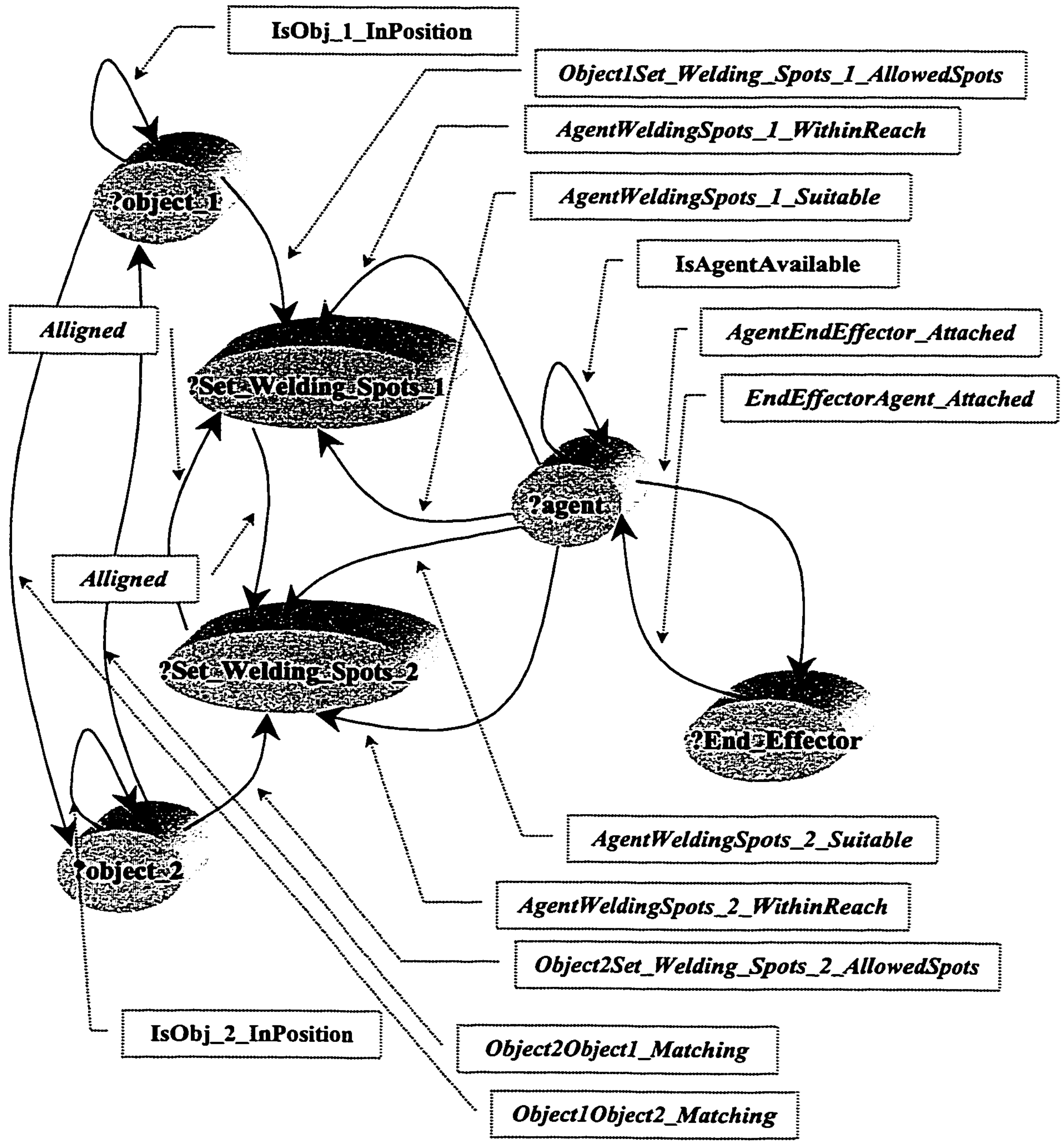

IsObj_1_InPosition
Object1Set_Welding_Spots_1_AllowedSpots
AgentWeldingSpots_1_WithinReach
AgentWeldingSpots_1_Suitable
IsAgentAvailable
AgentEndEffector_Attached
EndEffectorAgent_Attached
?object_1
Alligned
?Set_Welding_Spots_1
?agent
Alligned
?Set_Welding_Spots_2
?End_Effector
?object_2
AgentWeldingSpots_2_Suitable
AgentWeldingSpots_2_WithinReach
Object2Set_Welding_Spots_2_AllowedSpots
IsObj_2_InPosition
Object2Object1_Matching
Object1Object2_Matching

Figure 5-2. Action Pattern for welding operation

involved to another welding process and that is why its marked not available, the user can either interrupt that process or wait until it is completed. In the case where the agent cannot reach a specific point then the user moves the dedicated welding robot to a new position. With this example we saw how the user can try to solve the problem when the system cannot solve it by itself.

Let us assume that for this example **Robot 3** is available to perform the welding operation, so the variable **?agent** will be assigned that value and we proceed to the next constraint. The next unary constraint is **?IsObj_1_InPosition.** We should point out here that the constraint refers to the physical object - whether it is at the welding position or not.

Table 5-1. Part a) List of variables for welding operation - Action Pattern

Variable	**?object_1**	**?object_2**	**?agent**
Value	metal part 1	metal part 2	set of robots
Direct	yes	yes	no

Table 5-2. Part b) List of variables for welding operation - Action Pattern

Variable	**?set_of_welding_spots 1**	**?set_of_welding_spots 2**	**?end_effector**
Value	set of welding spots in metal part 1	set of welding spots in metal part 2	according to the robot assigned
Direct	no	no	no

The answer to that is no, since only the wireframe representation is placed at the desired location. Again the *Constraint Solver* is called and the solution that it provides is to call the *Action Controller* to generate an action pattern which will move the actual object to its welding position. We observe that the tracing of the action pattern associated with the welding operation is paused. The validation of the remaining unary and binary constraints will continue later. The control will then return from the *Constraint Solver* with a solution for the unsatisfied constraint. We should note here that in the case where no solution can be provided for the unsatisfied constraint, no other constraints will be tested for validity. The action pattern that is assigned with the responsibility to plan the process of moving the actual object to its welding location will be presented in the following subsection.

5.3.2 Solve an Unsolved Constraint - Call Move-Object Pattern

The unsolved constraint is that the **metal_part_1** is not in welding position, that means the move-object action pattern has to be called in order to resolve the unsolved constraint. The structure that represents the associated action pattern is shown in the following diagram (Figure 5-3).

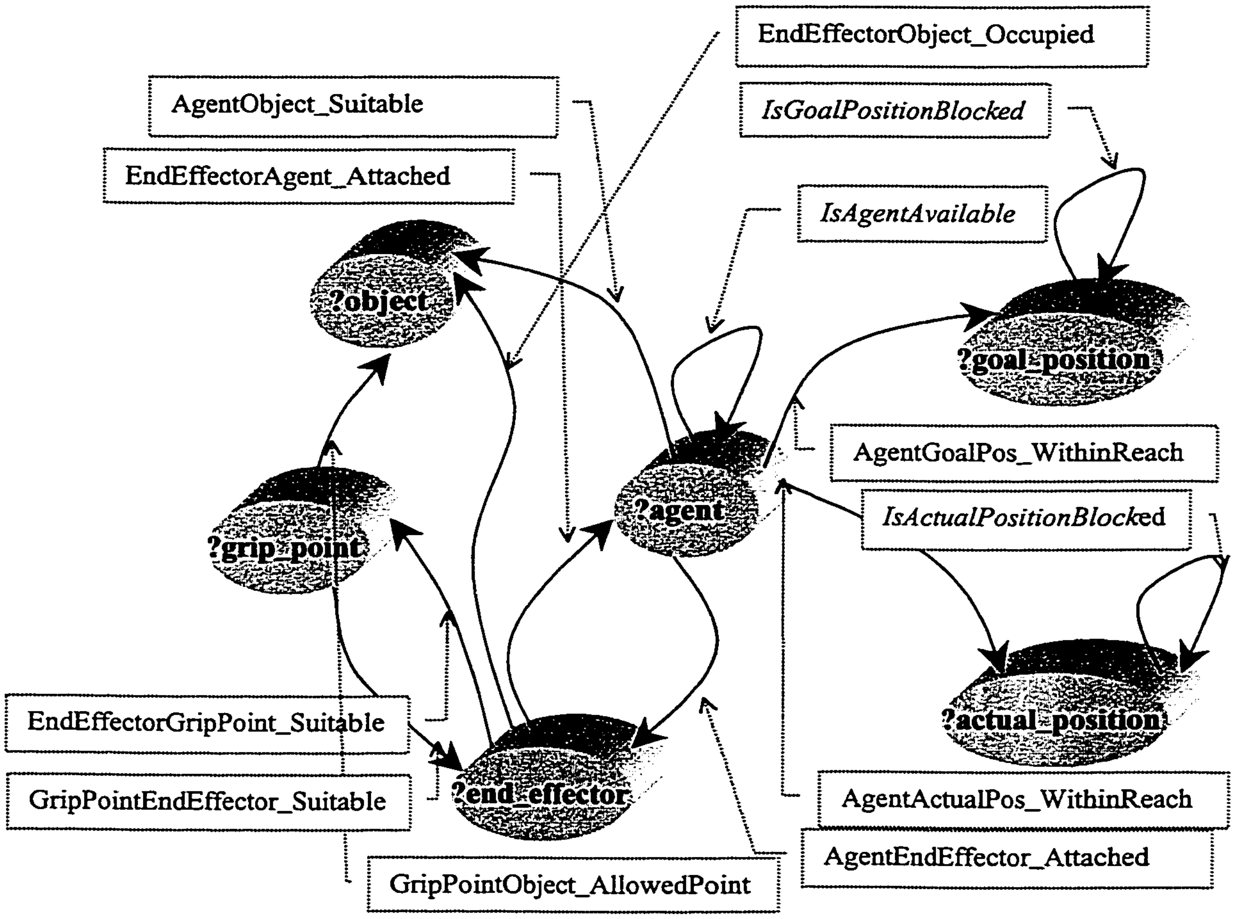

EndEffectorObject_Occupied
AgentObject_Suitable
IsGoalPositionBlocked
EndEffectorAgent_Attached
IsAgentAvailable
?object
?goal_position
AgentGoalPos_WithinReach
?agent
?grip_point
IsActualPositionBlocked
?actual_position
EndEffectorGripPoint_Suitable
GripPointEndEffector_Suitable
?end_effector
AgentActualPos_WithinReach
AgentEndEffector_Attached
GripPointObject_AllowedPoint

Figure 5-3. Action Pattern for transporting an object

When the structure is loaded, the same procedure as before gets initiated. All the variables that are contained in the structure get assigned with the appropriate values. For instance, the variable **?agent** can get assigned with any of the values from the set of robots except **Robot 3** which has been assigned for the welding task, and so is marked as unavailable. The **?object** variable will be assigned with the value **metal part 1**, the **?end_effector** will get the value **gripper** etc. The complete list of the variables and the corresponding values that they get are listed on Table 5-3 and Table 5-4.

Table 5-3. Part a) List of variables for transporting an object - Action Pattern

Variable	**?object**	**?goal_position**	**?actual_position**
Value	metal part 1	wireframe's position	solid's model position
Direct	yes	yes	yes

Table 5-4. Part b) List of variables for transporting an object - Action Pattern

Variable	**?agent**	**?end_effector**	**?grip_point**
Value	set of robots (Not Including Robot 3)	gripper	set of grippoints the object has
Direct	no	yes	no

After assigning the appropriate values to the variables, we proceed with the validation of the unary constraints. The unary constraints in the specific action pattern are: (1) **?IsAgen-tAvailable**, (2) **?IsActualPositionBlocked**, and (3) **?IsGoalPositionBlocked**. The first one evaluates whether or not a robot is available to be assigned with the specific action, same as before. The second checks whether or not the position where the **metal part 1** is currently located, appears to be blocked or not by another object. The third constraint checks whether or not the position where the wireframe is located is blocked. In our case we have the unary constraint (3) to be false, since we placed at the beginning of our scenario, the object-obstacle at the same location where the welding will take place.

The unsolved constraint gets detected and the *Constraint Solver* unit gets notified about the unsolved constraint. The solution that it proposes is that a new action pattern will have to be called to move the object-obstacle away from the welding location. Thus, the process of validating further the constraints of the action pattern responsible to bring **metal part 1** to welding location is paused. The rest of the constraints are validated at later stage. Assuming that the process of moving the object-obstacle away from the welding location is completed successfully by **Robot 1**, we return to the evaluation of the rest of the constraints in the action pattern responsible for moving the **metal part 1** to the welding location. The remaining constraints are binary, so they involve two variables. Their evaluation is required to be done in a systematic manner. We group all the constraints according to the dependencies they have to a certain variable (see Section 4.3.1.3), and we examine them sequentially. Table 5-5 shows the groups that are formed.

Table 5-5. Binary constraint of transporting an object - Action Pattern

Group / Binary Constraint	Description
Group 1) AgentActualPos_WithinReach	Is the agent within reach of the position where the physical object is located
Group 1) AgentGoalPos_WithinReach	Is the agent within reach of the position where the wireframe is located
Group 1) AgentObject_Suitable	Is the agent suitable for carrying the specific object (e.g. can carry the load etc.)
Group 1) AgentEndEffector_Attached	Is the specific agent equipped with the required end effector
Group 2) EndEffectorObject_Occupied	Is the required end effector associated with a different object.
Group 2) EndEffectorAgent_Attached	Is the end effector required available for the robot selected.
Group 2) EndEffectorGripPoint_Suitable	Is the end effector appropriate for the grip point available
Group 3) GripPointEndEffector_AllowedPoint	Is the grip point appropriate for the end effector available
Group 3) GripPointObject_AllowedPoint	Is the grip point that the object can be grasped from available.

We assume that all the binary constraints that are listed above are satisfied, and **Robot 2** gets assigned with the task of moving **metal part 1** to the location specified. After completing this, we return to the action pattern that generated the unsolved constraint. In our case, we return to the pattern responsible for the welding operation. From Section 5.4, we recall that the unary constraint **?IsObj_1_InPosition**, was false and this forced the *Constraint Solver* unit to invoke the move-object action pattern. Continuing the process of examining the unary constraints of the welding pattern, we see that **?IsObj_2_InPosition** is also false. This again through the same process calls a new move-object pattern in order for **metal part 2** to be placed at the desired location. We assume that this is done, omitting the description since it follows exactly the same steps as before. After that the control returns to the welding pattern which now continues with the binary constraints. The next section elaborates on this.

5.3.3 Binary Constraints of the Welding Pattern

Since the unary constraints of the welding pattern have been examined, we proceed to the validation of the binary constraints. Table 5-6, shows the groups of binary constraints that are formed.

Table 5-6. Binary constraints of welding two objects - Action Pattern

Group / Binary Constraint	**Description**
Group 1) AgentWeldingSpots_1_WithinReach	Is the agent within reach of the spots on object 1 that have to be welded
Group 1) AgentWeldingSpots_1_Suitable	Is the agent appropriate for the type of welding that object 1 requires
Group 1) AgentWeldingSpots_2_WithinReach	Is the agent within reach of the spots on object 2 that have to be welded
Group 1) AgentWeldingSpots_2_Suitable	Is the agent appropriate for the type of welding that object 2 requires
Group 1) AgentEndEffector_Attached	Is the agent selected equipped with the proper end effector
Group 2) EndEffectoAgent_Attached	Is the end effector required assembled at the selected agent
Group 3) Aligned	Is the set of spots 1 aligned with the set of spots 2
Group 4) Aligned	Is the set of spots 2 aligned with the set of spots 1
Group 5) Object2Set_Welding_Spots_2_Allowed	Does object 2 have the allowed welding points
Group 6) Object1Object2_Matching	Does object 1 match with object 2
Group 6) Object1Set_Welding_Spots_1_Allowed	Does object 1have the allowed welding points

After all the binary constraints have been checked and validated, **Robot 3** undertakes the task of welding **metal part 1** which is manipulated by **Robot 2** and **metal part 2** manipulated by **Robot 1** which in the meanwhile has removed the obstacle from the welding location before moving **metal part 2** there. So we can see very briefly how the distribution of responsibilities was done automatically, without input from the user interfering. Let us now go one step further in order to complete our scenario and let the user in the VR space "GRASP" the welded object and place it on the pallet. The next section describes this last part of our case study.

5.4 Moving Welded Object to the Pallet

After the object has been welded, the user "GRASPs" the new object that has been formed and moves its wireframe representation to the place where the 3D model of the pallet is supposed to be. The TDC detects the user's actions and according to the type of object, the corresponding Petri Net structure is retrieved. The trace of the Petri Net described below refers to Figure 3-5. The state of the composite object is **State(1)**, which is **Welding Position**. When the user selects it, **Transition(1)**, is triggered and the state of the object moves to **State(4)**. Then the user drags the object in the virtual space and places it on the 3D model of the pallet. **Transition(2)** is fired and the object is moved to **State(5)**. Since the place where the object was released is the pallet (that means inside the workspace), **Transition(3)** is fired and **Action(1)** is generated while the state of the objects moves to **State(2)**.

As soon as the **Move_Metal_Part** task description is detected from the TPC, the appropriate action pattern is selected. The *Action Controller* triggers the pattern dedicated to handle the positioning of the composite object on the pallet.

5.5 Tracing the Placement-to-Pallet Pattern

The corresponding action pattern that is responsible to place the composite object on the pallet is given in Figure 5-4. The process is similar to the one before. The variables are assigned with certain values and the constraints are examined, first the unary then the binary. The Table 5-7 and Table 5-8 give a list of the variables that are involved in this pattern description. The **?agent** that will undertake the responsibility of placing the composite

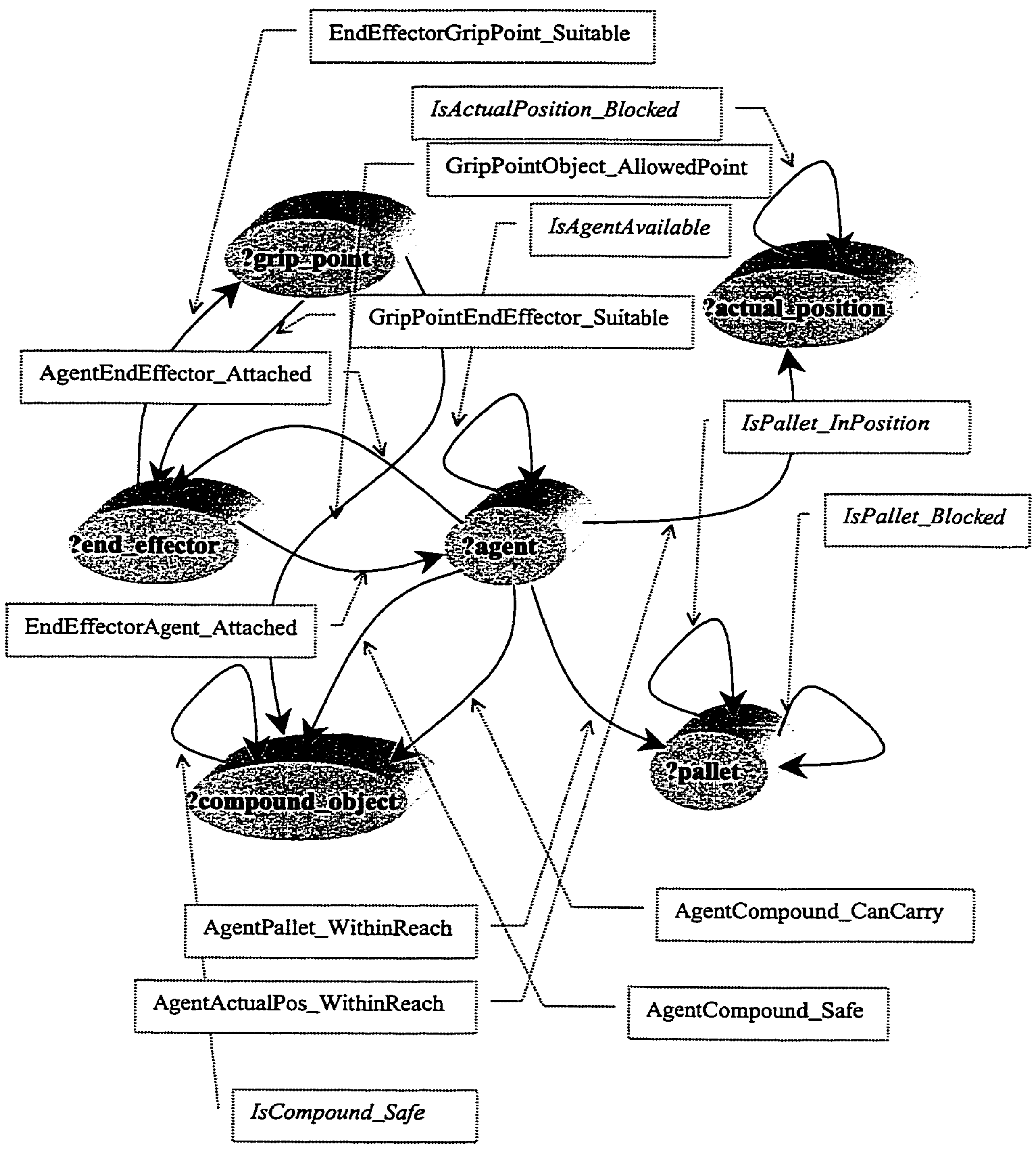


Figure 5-4. Action Pattern for placing a compound object on a pallet

object on the pallet could be one of the robots actually holding the composite object already. If this is the case, it is required that the other robot would release the object in time. For the rest of the variables the values that are assigned are quite obvious. The **?compound_object** takes as a value the identification number of the new object, the **?end_effector** is assigned with the gripper and so on. The unary constraints that can be observed are: (1) **IsAgentAvailable,** (2) **IsActualPositionBlocked,** (3) **IsCompound_Safe,** (4) **IsPallet_InPosition** and (5) **IsPalletBlocked**. The first and the second have already been discussed, the third one investigates if the welding has been performed properly and the compound object is properly welded.

Table 5-7. Part a) List of variables for transporting compound object to pallet - Action Pattern

Variable	**?compound_object**	**?pallet**	**?actual_position**
Value	metal part 1 + metal part 2	location where 3D model of pallet is located	welding position
Direct	yes	yes	yes

Table 5-8. Part b) List of variables for transporting compound object to pallet - Action Pattern

Variable	**?agent**	**?grip_point**	**?end_effector**
Value	set of robots	gripper	set of grippoints the object has
Direct	no	yes	no

The fourth and the fifth unary constraints validate if the pallet is within reach of the agent assigned to move the compound object and whether the pallet is blocked by a different object which the robots will have to move before the next can transport the compound object. After the unary constraints are examined, the binary constraints have to be validated. The list of the binary constraints for the action pattern are listed in Table 5-9.

Table 5-9. Binary constraints for placing compound object on the pallet - Action Pattern

Group / Binary Constraint	Description
Group 1) AgentActualPos_WithinReach	Is the agent within reach of the position where the physical object is located
Group 1) AgentPallet_WithinReach	Is the agent within reach of the position where the pallet is located
Group 1) AgentCompound_Safe	Is it safe for the agent to carry the specific object
Group 1) AgentCompound_CanCarry	Is the agent capable of carrying the weight of two objects
Group 2) AgentEndEffector_Attached	Is the agent equipped with the proper end effector
Group 2) EndEffectorAgent_Attached	Is the end effector required available for the robot selected.
Group 2) EndEffectorGripPoint_Suitable	Is the end effector appropriate for the grip point available
Group 3) GripPointEndEffector_Suitable	Is the grip point appropriate for the end effector available
Group 3) GripPointObject_AllowedPoint	Is the grip point that the object can be grasped from available.

In order to provide a 3D visualization of the actual robotic workcell we present here a screenshot of the workcell that we have simulated in MRS v2.0 (Figure 5-5). Present are three different robots, and several metal parts that will take place in the process of welding. Robot three will be assigned with the responsibility to perform the welding operation while robot two and one are assigned to bring together the appropriate parts together to the specified welding location.

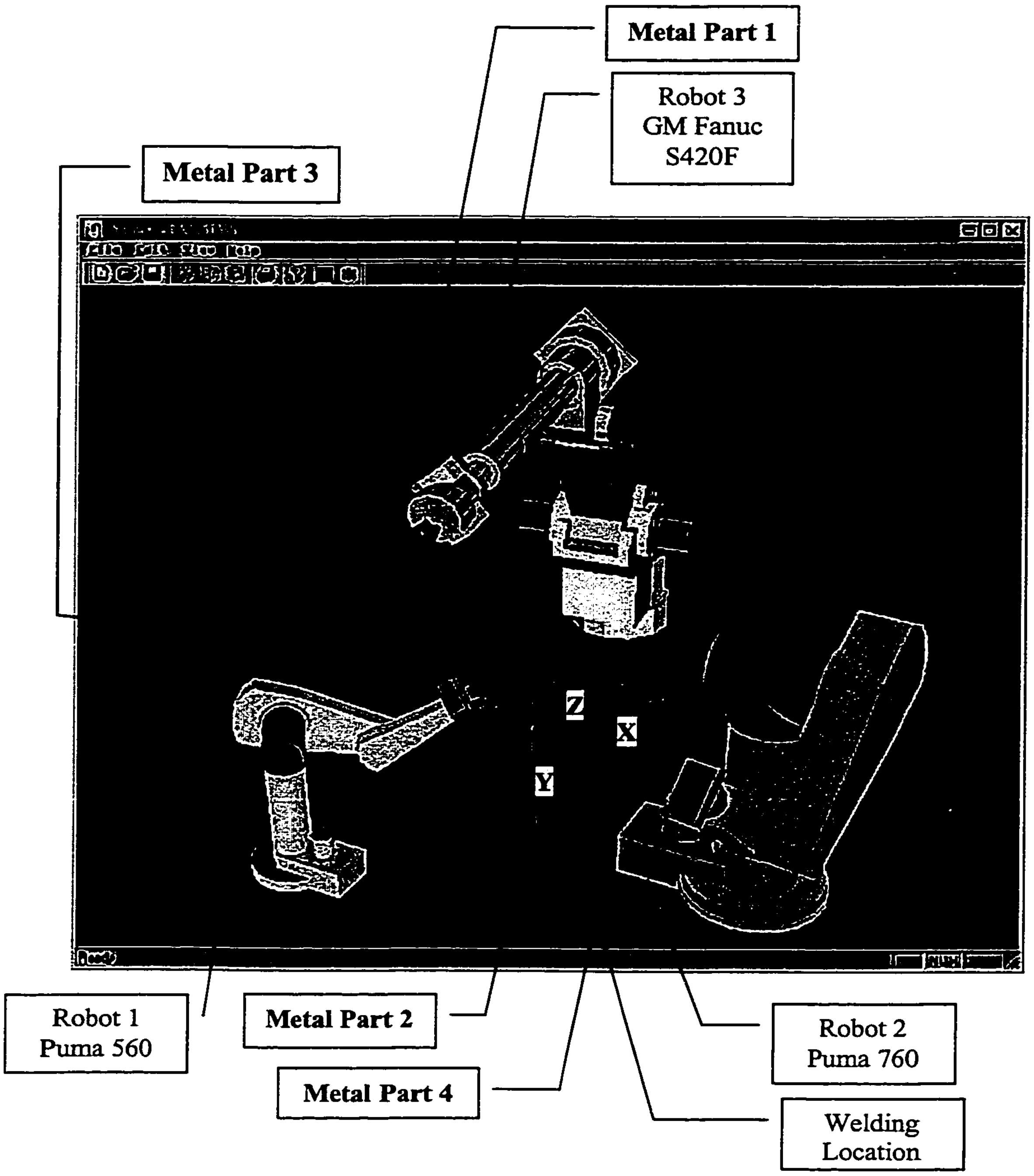


Figure 5-5. MRS version 2.0, showing three different simulated robots

5.6 Summary of the Scenario

As we mentioned at the beginning of this chapter, the process of planning consists of multiple stages. Each stage contributes certain amount to the overall plan. We saw that the initial action pattern that was triggered (welding operation action pattern), called other action patterns to solve certain unsolved constraints, forcing the planning to move to a second level. For instance, the welding operation action pattern, invoked two other action patterns. The first, responsible to move **metal part 1** at the welding location, and the second to move **metal part 2** at the same location. Subsequently, the action pattern that was assigned with the responsibility to move **metal part 1** at the specific location called another action pattern to solve a certain unsatisfied constraint. In fact, the action pattern responsible to move the object-obstacle is called, causing planning process to move to a third level. The following diagram (Figure 5-5) shows all these different layers.

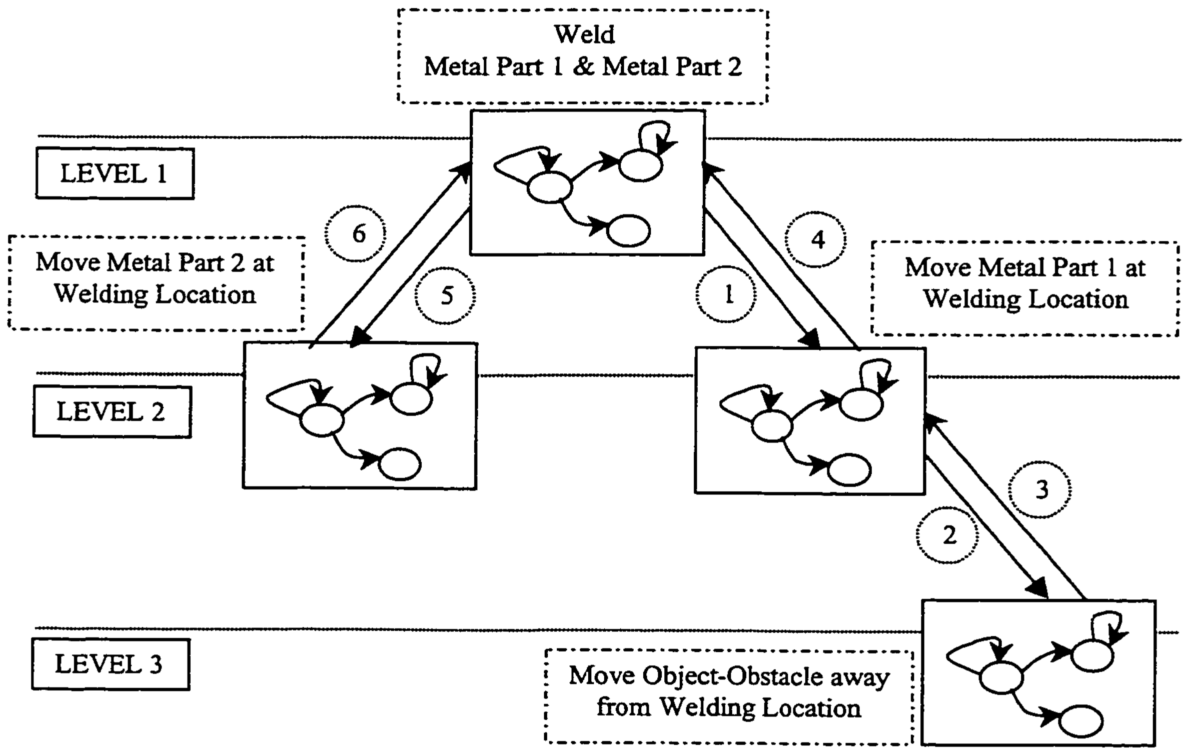

Weld
Metal Part 1 & Metal Part 2
LEVEL 1
LEVEL 2
LEVEL 3
Move Metal Part 2 at Welding Location
Move Metal Part 1 at Welding Location
Move Object-Obstacle away from Welding Location
1
2
3
4
5
6

Figure 5-6. The multi-stage planning of welding operation

The numbers presented inside the circles indicate the order in which every action pattern is invoked. For instance, the welding action pattern triggers the move metal part one action pattern and has to wait until that second action pattern returns the control. Subsequently, a third action pattern is invoked, which we assume is executed with success and so control is returned to level two of planning. The arrow with number three represents that the action pattern for moving the metal part can now continue. Assuming that there are no unsatisfied constraints the action pattern finishes and the control returns to the welding pattern (arrow four). The welding action pattern continues with the validation of the remaining constraints until it is realized that the second metal part has to be placed at the welding location. Once that occurs, the action pattern to move metal part two at the welding location is called. After completing the planning of that process, the control returns to level one so that the welding operation action pattern can finish with the validation of the constraints that remain.

Chapter 6

Conclusions

In this thesis, we have presented the software architecture for the Task Deduction and Task Planning Components for a Multiple Robot Simulation system called MRS. We began by providing some background information regarding Petri Nets, the software structure that we used in our process of development. We have described the utilization of Petri Nets both in MRS and in other similar applications.

Proceeding from this background, we performed an analysis of the problem space of the first component that we had to design and implement, namely the Task Deduction Component (TDC). We enumerated the basic components that compose its structure and we described the functions each one is responsible for providing in order to gain sounder understanding of the underlying software architecture. We then proceeded to describe the software architecture by providing a complete description of the classes that are designed and implemented and what responsibilities each one has been assigned. In addition, we elaborated on the interactions that are established between the different units that the Task Deduction Component is composed of, giving in this way not only a static picture of the component but also a dynamic aspect.

Subsequently, we tackled the description of the second component of our project that is the Task Planning Component (TPC) of MRS. We began with the definition of its functional requirements, as a black box architecture. We simply identified its input and output functions and we placed it in the overall architecture of MRS. We then went on to describe its decomposition to the basic building blocks that comprise it. This began by an examination of the generic modules that the architecture of the Task Planning Component introduces. The services that each module undertakes were addressed next, and the architectural decisions that were made during their development are presented. Moreover, the description of the software architecture of the task planning system covered the entire collection of classes that are implemented, the dependencies and the associations that are established among them, and their interactions both with MRS and with the Task Deduction Component.

Finally, we looked at a complete example which employed the functionalities that both Task Deduction and Task Planning Components provide. The example addressed an abstract description of a welding operation of two objects performed under very specific domain constraints. The description covers two major parts: (1) the recognition of the actions that the user performs in the VR space and (2) the proper planning of these actions by using the functionalities that Task Planning Component provides in order to accomplish the required task.

In the process of designing the Task Deduction and Task Planning Components, the following architectural decisions were made which are also contributions made by this thesis:

- To encapsulate task descriptions in self-contained Petri Net models.
- To distribute the services required by the Task Deduction and Task Planning Components among a collection of self-contained units.
- To make the classes that form the action patterns to share a common interface rather than provide multiple interfaces, one for each action pattern class.
- To use a simple and generic methodology for evaluating the constraints that are contained within each action pattern.
- To force both Task Deduction and Task Planning Components to communicate with MRS only through well defined interfaces.

Since both Task Deduction and Task Planning are still under development several limitations exist. The Task Deduction Component can process only tasks that have been predefined. In addition, the steps that the user has to follow, in order for the Task Deduction Component to realize the task one is trying to perform, are very specific and have to be executed in certain order. Moreover, the automated Task Planning Component provides solutions only to those tasks that can be modeled with Petri Net structures. The Task Planning Component provides in all cases a single solution, although there exists situations where more than one solution exist. Due to all these limitations both Task Deduction and Task Planning require further research.

6.1 Suggestion for Future Work

It is in the nature of research that the solution of one problem often gives rise to many new questions or problems. In this section we will introduce suggestions for future work. These suggestions mainly concern the Task Deduction and Task Planning Components.

We have mentioned in our discussion on TDC that the task descriptions are predefined and the user cannot modify them during execution. It would be useful to have a graphical tool component running within MRS where the user could have a visual representation of all the task descriptions (in the form of Petri Nets) from where one would have the ability to add, remove or modify task descriptions during execution.

When we talked about TPC we saw that the system evaluates the problem by itself and comes up with a single solution. Allowing the user to interact with the Task Planning Component and let him also participate to the selection of a proper plan, increases the flexibility of the system. That implies that the Task Planning Component must be able to produce more than one solution (if the problem has multiple solutions), demonstrate these to the user, and ask the user to participate (if he or she desires) to the selection of the plan.

In addition, the action patterns that are included in the TPC cannot be altered during the execution of a certain process. This means that the conditions that govern the execution of an operation (welding, assembling, etc.) cannot be modified dynamically during the process. It would be useful to allow the user in the VR space to provide direct input to the TPC in order to bypass temporarily certain constraints of a specific action pattern. Provided that the constraints that the user wants to be able to bypass are not critical for the stability of the system, one could cancel them and proceed with the execution of the specific task.

References

[1] Pierre-Alain Muller, *The Instant UML*, Wrox-Press, Birmingham, UK, 1997.

[2] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Longman, Menlo Park, CA, 1995.

[3] Ivar Jacobson, Graby Booch, James Rumbaugh, *The Unified Modeling Language User Guide*, Addison-Wesley, Reading, MA, 1999.

[4] Ivar Jacobson, Graby Booch, James Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, Reading, MA, 1999.

[5] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[6] John Craig, *Introduction to Robotics: Mechanics and Control*, Second Edition, Addison-Wesley, Menlo Park, CA, 1989.

[7] Charles R. Harrell and Donald A. Hicks, "Simulation Software Component Architecture for Simulation-Based Enterprise Applications", *Proceedings of the 1998 Winter Simulation Conference*, Vol. 2, pp. 1717-1721, 1998.

[8] Scott McMillan, David E. Orin, and Robert B. McGhee, "Object-Oriented Design of a Dynamic Simulation for Underwater Robotic Vehicles", *IEEE International Conference on Robotics and Automation*, Vol.2, pp. 1886-1893, 1995.

[9] Jacques Ferber, *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*, Addison-Wesley, Reading, MA, 1999.

[10] I.J. Bryson, *Software Architecture and Associated Design and Implementation Issues for Multiple-Robot Simulation and Visualization*, M.E.Sc. Thesis, University of Western Ontario, London, Ontario, 2000.

[11] I.J. Bryson, S.M. Noorhosseini and R.V. Patel, *MRS v1.0 User's Guide*, Concordia University, Montreal, Canada, 1996.

[12] F. Shadpey, S.M. Noorhosseini, I.J. Bryson and R.V. Patel, "An Integrated Robotic Development Environment for Task Planning and Collision Avoidance", *Third Biennial ASME European Joint Conference on Engineering Systems Design and Analysis*, Montpellier, France, 1996.

[13] I.J. Bryson and R.V. Patel, "A Modular Software Architecture for Robot Simulation and Visualization", *31st International Symposium on Robotics* (ISR 2000), Montreal, Canada, 2000.

[14] James L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, N.J, 1981.

[15] Rajesh Mascarenhas, Dinkar Karumuri, Ugo Buy, Robert Kenyon, "Modeling and Analysis of a Virtual Reality System with Time Petri Nets", *IEEE Transactions on Robotics and Automation*, pp. 33-42. 1998.

[16] Wolfgang Reisig, "Place/Transition Systems", *Lecture Notes in Computer Science-Petri-Nets: Central Models and their Properties*, Part I, pp. 119-141, 1986.

[17] Wolfgang Reisig, "Petri Nets in Software Engineering", *Lecture Notes in Computer Science-Petri-Nets: Central Models and their Properties*, Part II, pp. 63-96, 1986.

[18] Horst Oberquelle, "Human-Machine Interaction and Role/Function/Action-Nets", *Lecture Notes in Computer Science-Petri-Nets: Central Models and their Properties*, Part II, pp. 171-190, 1986.

[19] Eckhard Freund and Juergen Rossmann, "Projective Virtual Reality: Bridging the Gap Between Virtual Reality and Robotics", *IEEE Transactions on Robotics and Automation*, Vol.15, No.3, pp.411-422, 1999.

[20] Juergen Rossmann, "Virtual Reality as a Control and Supervision Tool for Autonomous Systems", *Intelligent Autonomous Systems*, IOS Press, pp.344-351, 1995.

[21] E.Freund and J.Rossmann, "Virtual Reality as a Novel Man-Machine Interface for Intelligent Robotic Systems", *Proceedings of 3rd European Control Conference* Rome, Italy, 1995.

[22] K.Hoffmann, E.Freund, J.Rossmann, "Resourse-Based Action Planning for Multi-Agent-Systems", *SPIE Conference on Sensor Fusion and Decentralized Control in Robotic Systems*, Boston, Massachusetts,1998.

[23] E.Freund, J.Rossmann, "Intelligent Autonomous Robots for Industrial and Space Application", *Proceedings of the IEEE/RSJ/GI Intelligent Robots and Systems* IROS'94, Vol.2, Munich, Germany, 1994.

[24] J.Uthoff, U.var der Valk, E.Freund, J.Rossmann, "Towards Realistic Simulation of Robotic Workcells", *Proceedings of the IEEE/RSJ/GI Intelligent Robots and Systems* IROS'94, Vol.1, Munich, Germany,1994

[25] Yi Yan, S.Ramaswamy, "Agent Based, Modeling and Simulation of Virtual Manufacturing Assemblies", *Communications of the ACM,* No.8, pp. 78-87 August 1998.

[26] Vijaimukund Raghavan, Jose Molineros and Rajeev Sharma, "Interactive Evaluation of Assembly Sequences Using Augmented Reality", *IEEE Transactions on Robotics and Automation*, Vol.15, pp. 335-349, 1999.

[27] Jan D. Wolter, "On the Automatic Generation of Assembly Plans", *IEEE Transactions on Robotics and Automation*, pp. 62-68, 1989.

[28] L.De Floriani, "A Graph Model for Face-to-Face Assembly", *IEEE Transactions on Robotics and Automation*, pp. 75-78, 1989.

[29] L.S. Homem de Mello, A.C.Sanderson, "A Correct and Complete Algorithm for the Generation of Mechanical Assembly Sequences", *IEEE Journal of Robotics and Automation*, pp. 56-61, 1989.

[30] M. Paolucci, D. Kalp, A. Pannu, O. Shehory, K. Sycara, "A Planning Component for RETSINA Agents", *IEEE Expert Systems and their Applications*, pp. 36-45, 1998.

[31] Richard W. Bukowski, Carlo H. Sequin, "Object Associations: A Simple and Practical Approach to Virtual 3D Manipulation", *Symposium on Interactive 3D Graphics*, Monterey CA, USA, pp. 131-138, 1995

[32] Mark R. Mine, Frederick P. Brooks Jr. Carlo H. Sequin, "Moving Objects in Space: Exploiting Proprioception In Virtual-Environment Interaction", *Communications of the ACM,* No.7, pp. 19-26, July 1997.

[33] Doug A. Bowman and Larry F. Hodges, "An Evaluation of Techniques of Grabbing and Manipulating Remote Objects in Immersive Virtual Environments", *Symposium on Interactive 3D Graphics*, pp. 35-38. 1997.

[34] Jeffrey S. Pierce, Andrew Forsberg, Matthew J. Conway, Seung Hong, Robert Zeleznik, Mark R. Mine, *Image Plane Interaction Techniques In 3D Immersive Environments*, *Technical Report*, University of Virginia, Brown University, University of North Carolina, 1995.

[35] Rafael Ramirez, Nets, Logic and Concurrent Object-Oriented Programming, *Technical Report*, University of Bristol, UK, Dept. Computer Science, 1995.

[36] Pascal Van Hentenryck Vijay Saraswat et Al, "Strategic Directions in Constraint Programming", *Communications of ACM* Vol. 28, No.4, December 1996.

[37] Kutluhan Erol, James Hendler, and Dana S. Nau. "HTN planning: Complexity and Expressivity", *Proceedings 12th National Conference on Artificial Intelligence (AAAI-94)*, Seattle, WA, 1994.

[38] R.Valette, "Nets In Production Systems", *Lecture Notes in Computer Science-Petri-Nets: Central Models and their Properties*, Part II, pp. 191-217, 1986.

[39] Rajesh Mascarenhas, Dinkar Karumuri, Ugo Buy, Robert Kenyon, "Modeling and Analysis of a Virtual Reality System with Time Petri Nets", *Proceedings 19th. International Conference on Soft. Engineering.* Kyoto, Japan, pp.19-25, April 1998.

Appendix A : Introduction to the UML Notation

The Unified Modeling Language (UML) is a standard notation for modeling object-oriented applications. It focuses on the description of software development artifacts, rather than on the formalization of the development process itself. It can therefore be used to describe software entities obtained through the application of various development processes. UML is a very flexible notation, it is generic, extensible, and can be tailored to the needs of the user. Here we will give an overview of the semantics of UML's model elements, and we will introduce the main concepts of modeling, articulating them in terms of the UML notation.

UML defines nine types of diagrams to represent the various modeling viewpoints [1]. The order in which we will present the various diagrams does not reflect the order of implementation in a real project. It only attempts to minimize the prerequisites and cross-references.

TYPES OF UML DIAGRAMS

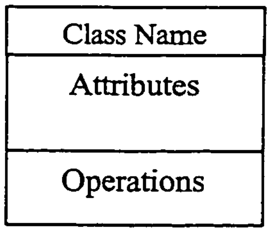
- Class Diagrams
- Use Case Diagrams
- Object Diagrams
- Collaboration Diagrams
- Sequence Diagrams

- Statechart Diagrams
- Activity Diagrams
- Component Diagrams
- Deployment Diagrams

CLASS DIAGRAMS

The class diagram is the structure that expresses, in a general way, the static structure of a system, in terms of classes and the relationships between those classes [3]. A class describes a set of objects, and an association describes a set of links. Objects are class instances, and links are association instances. Note here that a class diagram does not express anything specific about the links of a given object, but it describes, in an abstract way, the potential links from one object to another.

CLASSES: Classes are represented by rectangles that are divided into three compartments. The first compartment contains the class name. The class is not a function; the class is an abstract description of a set of objects from the application domain [1]. The other two compartments contain respectively the class's attributes and its operations.

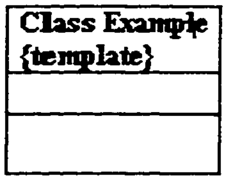


UML defines three visibility levels for attributes and operations:

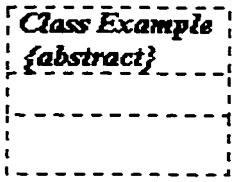
- **public** : the element is visible by all the clients of the class
- **protected** : the element is visible to subclasses of the class
- **private** : the element is visible only to the class

The visibility levels in UML are represented symbolically by the characters +, # and - for **public, protected** and **private**.

<u>TEMPLATE CLASSES:</u> Template classes are model classes. A template class cannot be used as is. It is first necessary to instantiate it in order to obtain a real class that must in turn be instantiated to produce objects. Template classes facilitate the construction of universal collections, typed by parameters.

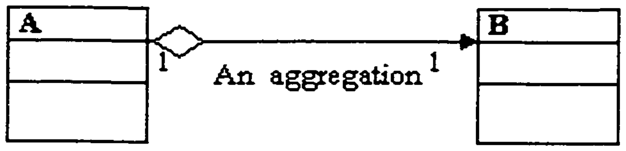


<u>ABSTRACT CLASSES:</u> Abstract classes cannot be instantiated directly. They do not give birth to objects, but may be used as a more general specification in order to manipulate objects that are instances of one of their subclasses. Abstract classes provide a general basis for extensible software applications. The set of general mechanisms is described according to the specifications of the abstract classes, without taking into account specific features gathered within concrete classes. The important feature of an abstract class is that new requirements, extensions and improvements are gathered into new subclasses which generate objects that can be manipulated transparently by mechanisms that are already in place.



RELATIONSHIPS:

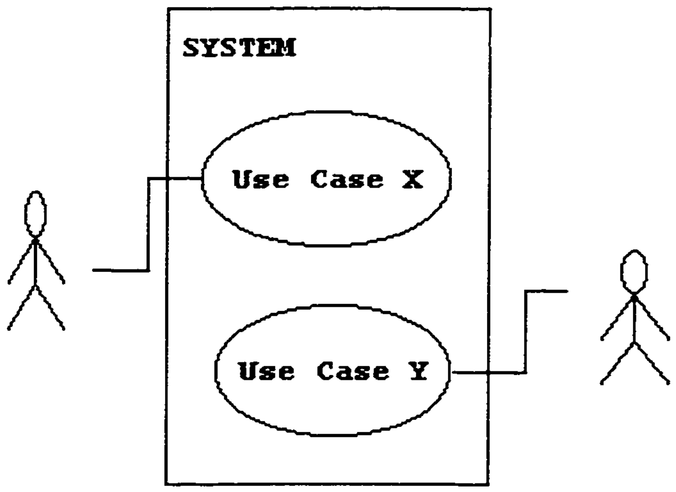
- **Association:** Represents a structural relationship between classes of objects. Associations may be named. Without being a systematic rule, experience recommends naming associations using either active, like "*accesses*" or passive "*is employed by*".
- **Aggregation:** Aggregation is an asymmetric association, in which one of the ends plays a more important role than the other. An aggregation is represented by adding a small diamond next to the aggregate. Aggregation is implied when a class is part of another class. Also, when the attribute values of one class propagate to the attribute values of another class. In addition, when an action on one class implies an action to another class.



- **Composition:**This association is a particular case of aggregation, where we have physical containment.
- **Navigation:** Navigation is an association with an arrow at one of the ends of the association. When there is no arrow, that simply means that the association may be navigated in both directions. The object instances of that class from from the navigation starts can see the object instances of the class where the navigation ends.
- **Generalization:** UML uses the term generalization to specify the classification relationship between a general element and a more specific element. In fact, the term 'generalization' specifies a viewpoint focused on a classification hierarchy.

USE CASE DIAGRAMS:

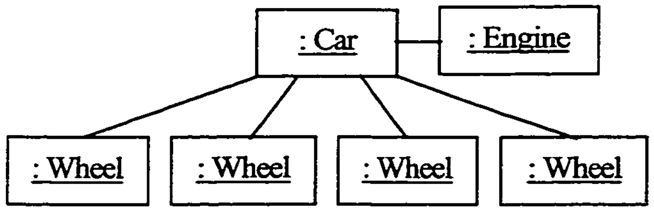
A typical use case diagram is composed of the *actors*, the *system* and the *use cases* themselves. With this model we present the functionalities of a system, as interactions between the actor and the several use cases. Actors are represented by little stick people who trigger the use cases which exist within the system.



An actor represents a role played by a person or a thing that interacts with the system. An actor could either be a direct user of the system or it could be another system interacting with the system that we are studing. We should point out here the fact that, the same physical person could play the role of several different actors (client). Moreover, several people may all play the same role, and for that reason we can group them into a single actor.

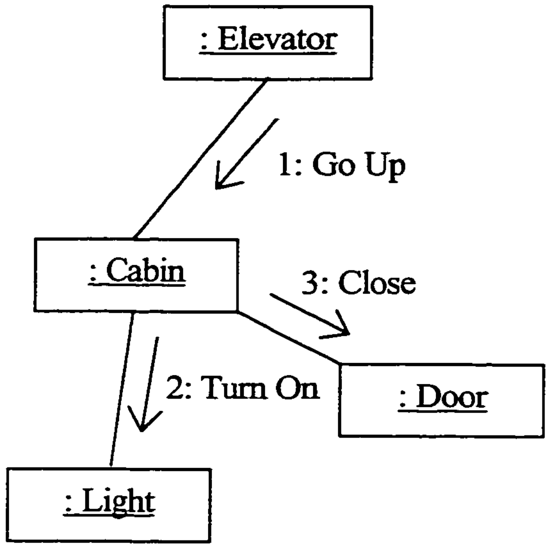
OBJECT DIAGRAMS

Objects diagrams, also called instance diagrams, illustrate objects and links. Similar to class diagrams, object diagrams represent the static structure of the system. The notation that is employed for the object diagram is derived from that of class diagrams; elements that are instances are underlined. Object diagrams are primarily used to show a context, before or after an interaction. However, they are also used to aid in the understanding of complex data structures, such as recursive structures.



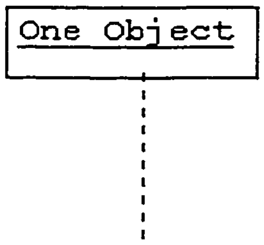
COLLABORATION DIAGRAMS

Collaboration diagrams are used to show the dynamic behavior of the system. They illustrate the interactions between different objects, using a static spatial structure that facilitates the illustration of the collaboration of a group of objects. With these diagrams we demonstrate both the context of a group of objects and also the interaction between these objects by providing representation of message broadcasts. The context of an interaction comprises the arguments, the local variables created during execution, and the links between the objects that participate in the interaction. The interactions are implemented by a group of objects that collaborate by exchanging messages. These messages are represented along the links that connect the objects, using arrows pointed towards the recipient of the message.

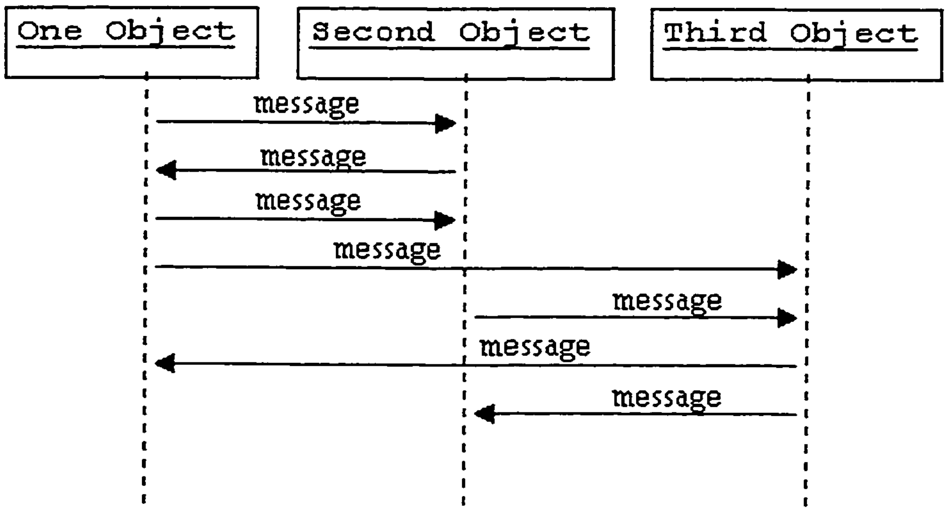


SEQUENCE DIAGRAMS

Sequence diagrams also display the dynamic structure of the system. The difference from a collaboration diagram is that the context of the object is not represented explicitly. The representation focuses on expressing interactions. A sequence diagram represents the interaction between objects and focuses on the message broadcast chronology. An object is represented by a rectangle and a vertical bar called the object's lifeline.

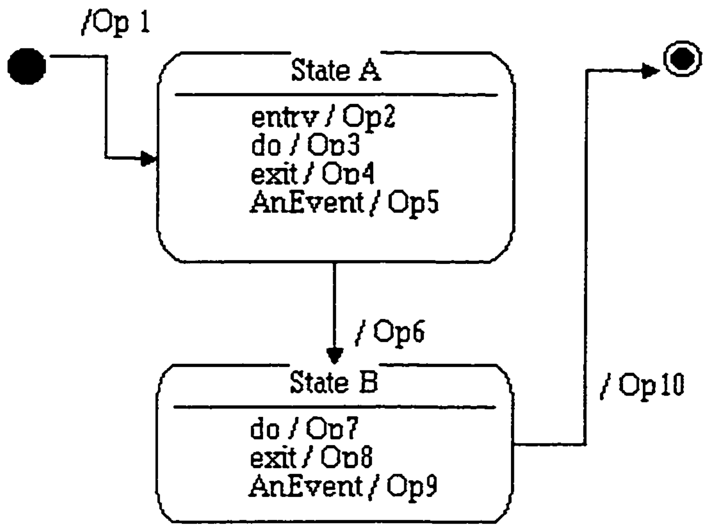


The communication between the objects is established by exchanging messages, which are represented by horizontal arrows drawn from the message sender to the message recipient. The sending order of the messages is indicated by the position of the message on the vertical axis. Sequence diagrams can be used in object-oriented modeling in two different ways according to the phase of the life cycle and the desired detail level. The first one focuses on the description of the interaction without getting into the details of synchronization. In this case, the information carried by the arrows corresponds to the events that occur within the application domain. The second use allows the precise representation of interactions between objects. The concept of a message unites all the different types of communication between objects, in particular, procedure calls, discrete events, signals between flows of execution and hardware interrupts.



STATECHART DIAGRAMS

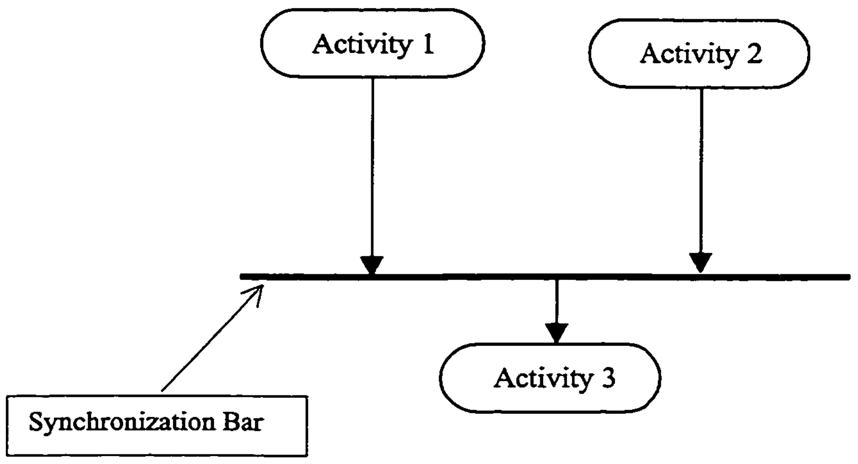
Statechart diagrams represent state machines from the perspective of states and transitions. A state machine is an abstraction of all possible behaviors, similar to the way class diagrams are abstractions of the static structure. Each object follows the behavior described in the state machine associated with its class and is, at a given moment, in a state that characterizes its dynamic behavior.



ACTIVITY DIAGRAMS

An activity diagram is a variant of the statechart diagram organized according to actions, and mainly targeted towards representing the internal behavior of a method. The activity diagram represents the execution state of a mechanism as a sequence of steps grouped sequentially as parallel control flow branches. The activities are represented by rounded rectangles.

Activity diagrams show synchronizations between control flows by using synchronization bars. A synchronization bar makes it possible to open and close parallel branches within the flow of execution of a method.



COMPONENT DIAGRAMS

Component diagrams describe the software components and their relationships within the implementation environment; they indicate the decisions that are made at implementation period. Components represent all kinds of elements that pertain to the piecing together of software applications. Among other things, they may be simple files or libraries loaded dynamically.

DEPLOYMENT DIAGRAMS

Deployment diagrams show the physical layout of the various hardware components that compose a system, as well as the distribution of executable programs on this hardware. Each hardware resource is represented by a cube, evoking the physical presence of the equipment within the system.