Dynamic Reconfiguration of Software Systems using Temporal Planning

by

Naveed Arshad

B.S GIK Institute of Engineering Sciences and Technology, Topi, PAKISTAN

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirement for the degree of

Master of Science

Department of Computer Science

2003

This thesis entitled:

Dynamic Reconfiguration of Software Systems using Temporal Planning

written by Naveed Arshad

has been approved for the Department of Computer Science

Prof Alexander L. Wolf

Prof Dennis Heimbigner

Date

The final copy of the thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline

Arshad, Naveed (MS Computer Science)

"Dynamic Reconfiguration of Software Systems using Temporal Planning"

Thesis directed by Professor Alexander L. Wolf.

Dynamic Reconfiguration of Software Systems could be divided into three phases. These are sensing the need for reconfiguration, planning it and carrying it out. Most of the research in the research has focused on the sensing and carrying out of reconfiguration. However, not much attention has been paid into the planning part of reconfiguration. The planning of the reconfiguration process is considered to be difficult because there are not much automated and intelligent tools available that can carry out this process in an efficient way. This thesis describes a novel technique for carrying out the planning process using Temporal Planners. The Temporal Planners computes the plan for dynamic reconfiguration under tight time and resource constraints. In this thesis we have demonstrated two aspects of Dynamic Reconfiguration. The first is the optimum deployment of the Initial System and the second is the reconfiguration of the system while it is running. We have developed a tool called Planit that acts as mediator between the Temporal Planner and the actual system. This tool is capable of detecting any change in the system, planning for the reconfiguration and disseminating the new reconfiguration. We have tested the temporal planners on systems with large number of re-configurable artifacts with good results.

Dedication

This thesis is a modest attempt to contribute to the body of knowledge in the field of Software Engineering. It can be beneficial in many aspects but it may have many mistakes also. I wholeheartedly accept all the mistakes that I may have made in this thesis and attribute all the beneficial stuff in it to Allah almighty. I would like to thank Allah for helping me in every aspect of this thesis. I would also like to thank my wife for all the patience, help and support during this time.

Acknowledgements

I am very thankful to Professor Alexander L Wolf and Professor Dennis Heimbigner for providing me with their time, help and guidance at each and every aspect of this thesis.

Table of Contents

List of Tables

Table:

List of Figures

Figure:

Chapter 1

Introduction

This project examines a way to automate of the process of dynamic reconfiguration of Distributed Software Systems. Artificially Intelligent (AI) planners have been used to accomplish this task. Dynamic reconfiguration has been a major problem in the area of software configuration management; its aim is to change the configuration of the software system while the system is running. This problem tends to be a really complex problem. Not only this problem is very complex but the speed of change in technology adds more challenges. The techniques that have so far been proposed to attack this problem are mostly manual or they have a significant portion that is handled manually. Therefore, the dynamic reconfiguration techniques in the literature, although very innovative, can be rather difficult to implement in practice in large distributed systems. In this project we have attempted to reduce the complexity of the dynamic reconfiguration problem by automating it with Artificially Intelligent Temporal Planners.

Artificially Intelligent planners have been used to solve problems in Robot Motion Planning [21], Workflow Management [20], Intelligent Manufacturing, Online Scheduling [31]. Previously, because of the huge search space, the AI planners were limited to solving toy problems only. However, in the last few years enormous research effort has been put into the performance of the AI planners. Therefore, now techniques and heuristics are available in AI planners that can solve the real world problems in a predictable amount of time.

Optimized planning of tasks is a real problem in many areas. Plans are required to efficiently execute a set of steps to go from one state to another. However, this involves many inter-dependent variables that need to be accommodated while performing these steps. It is difficult for a human to actually play with these variables and find out the optimal set of steps even for little problems. The AI planners' use heuristics and search tree reducing techniques that allow them to find many solutions in a much smaller amount of time than a human being.

Different plans have different requirements. Some plans need to minimize time, while others require optimized use of resources. Artificial Intelligent planners are geared to find out many different plans. Therefore there is a whole range of planners available for searching plan with these different priorities. Some of the planners include Temporal Planners, Graph Planners, Hierarchal Task Network Planners, Case Based Planners, Resource Sensitive Planners and many more.

Optimizing time and resources is the most important property required in the Dynamic Reconfiguration of Software Systems. Therefore in this project we have used a Temporal Planner (LPG) that uses local search based heuristic techniques to find plans.
Following is an introduction of the description of a distributed system for the sake of this project.

1.1 Distributed Software System Architecture

For the purpose of this project the architecture consists of three classes of software artifacts that make up the Distributed Software System. This is the simplest abstraction of a Distributed Software System that presumably one can make in a 'proof of concept' project. This is, however, not the simplest abstraction in terms of the complexity of a software system. This simple abstraction adds a huge amount of complexity to the software system domain in terms of the possible number of states this system can take. Following is a brief description of the three classes of software artifacts:

Component: Components are the class of software artifact that provides any type of service in the software system. This type of service includes, but is not limited to, computation, communication, observation etc. Any software system could be divided into the components based on the definition of the software in some abstraction. There is no limitation on the number and size of the components that are part of the software system. Two components can be connected to each other with a connector.

Connector: A connector provides a path of bi-directional communication among components. It is also used to disseminate information to the components. A connector can be thought of as a special type of component also. However, in this project we are treating connectors as a separate piece in the software system because of certain limitation of the AI planners. There is no limitations on the number of connectors; however a connector can have resource constraints like number of components connecting to it.

Machine: A machine hosts the components and connectors. In this project realm a component or a connector is placed on a machine in full. Connectors are also placed on machines. There is no limitation on the number of machines. However a machine has resource constraints like the number of components and connectors that can be deployed on it.

The interplay of components, connectors and machines determine the state of the Distributed Software System.

1.2 Temporal Planning System

The Temporal planning system used in this project is LPG [3]. Any planner, including LPG, expects a domain of the problem to be specified. This domain has the semantics of that the planner uses to figure out the plan. In our case the domain is the Distributed Software System. An example of this domain is given in Appendix A. The Planner also requires a problem specification. This problem file is based on the semantics of the domain file.

The problem file has two major parts. The first part defines the initial or present state of the system. The second part defines the goal state in which the system should go. Both the domain file and problem file are written in a language called PDDL (Planning Domain Definition Language) [22]. The planner takes the problem file and the domain file and figure out the plan(s). Depending on the time available and the metric specified in the problem file, the planner can compute the optimum plan available. This plan enumerates the steps that need to be taken to transition the present state of the system to the goal state.

1.3 Our Dynamic Reconfiguration Approach

Our dynamic configuration approach involves the Temporal Planner as a part of the reconfiguration process. We have developed a Planit called Planit. Planit takes the takes the domain of the software system. This domain describes the semantics of the reconfiguration process like start-component, move-component, connect-component etc. Any thing that can be expressed in PDDL can be used in Planit. Another piece of information that Planit expects is the structure of the system. The pre-deployment structure of the system consists of an enumeration of the components, connectors and machines including their properties like the start time, connect time etc.

Planit allows two kinds of reconfiguration: Explicit Reconfiguration and Implicit Reconfiguration. In Explicit Reconfiguration the user of the system has to explicitly state the target configuration that the system must go into. For example to deploy connector2 at machine3 in an Explicit Configuration. In Implicit Configuration just the desired property of the artifacts are enumerated. It is the task of the planner to devise an optimum strategy to reconfigure the system to satisfy those properties. One can also do a mix of Explicit and Implicit Configurations. Both the implicit and explicit reconfigurations are described in a problem file. The problem file along with the domain file is the given as input to the LPG planner. The planner computes an optimum plan and returns the result.

Planit parses the plan. It analyzes the plan and trigger necessary commands to execute the deployment and initial starting of the system. It saves the configuration of the system and monitor system health continuously. The artifacts that are being monitored informs about a need to reconfigure a system. A need to reconfigure the system can be required in cases where a machine goes down or a connector lost all its connections or a component is attacked by an external intruder and needs to be started at another location.

Once a problem is detected Planit checks the contingency information of the affected artifact(s). The contingency information can come in many forms; it may include the connection to the component to its nearest connector or restarting of a component on the nearest machine. It also can check if a component needs to be connected to a specific connector in case its own connector goes down.

Planit then makes a new problem file in PDDL. It writes the initial state based on the affected artifacts of the system. It writes the goal state based on the contingency information of the artifacts. It triggers the LPG planner by giving the problem file and the domain file. The planner computes an optimum plan for the dynamic reconfiguration and gives it back to Planit. Planit trigger the new configuration across the network and saves the new configuration in a persistent storage.

The above process continues throughout the lifecycle of the system. In the scope of this project we have used only one domain file, however multiple domain files can be use to get appropriate plans for different scenarios of reconfiguration. The different scenarios can be movement of components, tweaking the variables inside the components and so on. Having different domain files could create race conditions, however having designed the right domain for the right scenario can avoid this problem.

Chapter 2

Related Work

The related work of this project can be seen from two perspectives. The first perspective is the techniques that have been developed for solving the of Dynamic Reconfiguration problem in Distributed Systems. The second perspective is research and usage of planning to solve other real world planning problems.

According to our knowledge there has not been a direct usage of AI planners in the solution of dynamic reconfiguration of distributed software systems. Some authors propose planning for dynamic reconfiguration [11, 12]. However they do not use Artificially Intelligent planners in their real sense. The planning in here can be regarded as configuration scripts that trigger various parts of the script depending on the state of the system. Other authors have proposed different other solutions for the solution of the dynamic reconfiguration problem. Cook and Dage [14] propose an upgrade of components based on multiple versions working at the same time. Magee and Kramer [8] propose a technique of self organizing architecture. A few authors presented techniques based on middleware and different architectures like CORBA, J2EE and DCOM [4,15,19]. Following is an overview of related work in the area of Dynamic Reconfiguration of Software Systems.

Dynamic Reconfiguration is the ability to modify and extend a system while it is running. This facility is a requirement in large distributed systems where it my not be possible or economic to stop the entire system to allow modification to part of its hardware or software [1]. Dynamic reconfiguration is required at many times during the lifetime of a deployed distributed software systems. There are many operations like upgrade, adapt, remove or replace that are required on the components of the system while it is in a running state.

The Internet has made the very nature of almost every software application distributed. While it provides an excellent set of services to the users, it is increasingly difficult to maintain these applications because of the performance requirements imposed on them. The notion of 24 x 7 availabitity of the systems makes the maintenance of distributed systems hard to achieve. Apart from the trivial operation on components, required for system maintenance, there are many other factors that require the dynamic reconfiguration of distributed systems.

Security plays a major role in the dynamic reconfiguration of distributed system. A system may need to reconfigure as a result of an external attack carried out for malicious purposes. On the other hand some parts of the systems may need to be completely started on another site because the infrastructure at one side of the network failed due to a “Denial of service” attack.

2.1 What makes Dynamic Reconfiguration hard?

The very nature of distributed application is heterogeneous. There are many systems running on a number of platforms. This make hard to achieve an over all global change in the application. Addition of new components and removal of previous one without breaking the functionality is very difficult because of the interdependence of the components involved. The size of these Distributed Software Systems is huge as compared to the standalone applications. Size can change over a period of time.

Dynamic reconfiguration employs two kinds of changes: programmed changes and evolutionary changes [23]. The programmed changes are the changes that are anticipated and foreseen by system designers. The evolutionary changes are unseen and become necessary over the lifetime of a system. Changes without a prior knowledge are hard because the system may not have been tested and developed with the changes that it has to accommodate. Therefore consistency while performing these changes is an important factor. Consistency needs to be preserved. The state of the system before the change and after the change must be consistent with respect to the set of services it provides.

Resource constraints, interconnection dependencies and temporal restrictions make the problem even harder. The interplay of all these factors make it impossible for a human to perform all these changes manually in an optimum way.

2.2 Approaches to Dynamic Reconfiguration of Distributed Systems

There are many approaches suggested to solve the problem of dynamic reconfiguration of distributed systems [1, 4, 8, 10, 11, 12, 13, 14, 15, 16, 17, 19]. In the following paragraphs we describe the approaches to dynamic reconfiguration of distributed systems available in the research literature.

One of the very first research efforts in the area of dynamic reconfiguration of software systems was presented Kramer and Magee [1]. They described several properties that a component requires in order to be reconfigured dynamically. Some of the properties include the support of a modular programming language, interconnection of components, well-defined interfaces of components and communication transparency amongst components. They propose a process of dynamic configuration where configurations are written in form of a specification. The configuration changes are validated across the configuration specification written. These changes are then transferred to a configuration manager. The configuration manager in turns uses the operating system commands to make the reconfiguration. This approach was realized in their example system CONIC.

Another approach to the dynamic reconfiguration problem was of configuration languages [11, 12]. Agnew et al [11] proposed a declarative approach. This approach makes the programmers responsible for writing the configuration changes in form of a script. The programmers write the configuration scripts for the anticipated reconfigurations. Varying conditions across the system trigger invocation of these scripts. In this approach the reconfiguration of the application is a dynamic method in which the application is mapped between two execution states. The mapping may involve change to the stricture of the application or it may involve altering how the structure is mapped onto the underlying host resources. The configuration scripts generate the implementation details required to move from one state to the other.

Another language Gerel was [12] developed that have a different perspective in the programming of reconfigurations. In this approach the language mechanism selects the configuration objects dynamically using their structural properties. All configurations keep certain structural properties. This language defines the set of pre conditions for programmed changes, which specify the structural properties of the current configuration required by a change.

Research efforts have also been made to view the dynamic reconfiguration problem as a workflow system [10]. In this approach the application composition and execution environment is designed as a transactional workflow system that enables sets of inter-related tasks to be carried out in a dependable manner. A task model is developed to express the temporal dependencies between the constituent tasks. The workflow system persist these tasks and then allow transactional operations for performing changes to it.

Agent-Based approach for configuration management of distributed applications has also been proposed [17]. In this approach the components are divided into two categories. These are application component and management components. The interface between application components and management components provides appropriate methods to change the state of an application. The management component keeps track of the application state, initiates management operations. The management components have agents that travel across the distributed system for performing these operations. The agents act on the application components and return to the management components after applying the management tasks.

Research efforts have also been focused on developing of such framework that inherently supports the dynamic reconfiguration [16]. In this framework the component consist of the following: service interfaces, service implementation, control interfaces and control implementation. The control interfaces and control implementation provides the ability to reconfigure the component at runtime. The brain of the reconfiguration process is called a CM. The CM primary task of the CM is to check whether a configuration is consistent. When the reconfiguration is required the CM cooperate with its agents in order to carry out the reconfiguration consistently. A CM agent is responsible for the managing the components located in the same runtime environment.

Another approach suggested by Cook and Dage [14] takes the road of making multiple versions of the component run at the same time. They argue that in order to not break the present functionality of the system multiple versions of the same components need to coexist together. They present a framework called HERCULES. In this framework the existing versions of the components running and only removing them when it is proved that the new component is fully satisfying the functionality of the old ones. The replacement of component in this framework also needs to be made one at a time rather the whole system being upgraded at once.

Research efforts have also been spent in providing reconfiguration mechanisms on platforms like CORBA and J2EE [4, 13,15,19]. Batista and Rodriguez [4] provide an approach that supports both programmed-based and ad-hoc based approaches for reconfiguration. They developed a programming language called Lua. Lua is a procedural interpreted language and it is part of an environment called LuaSpace. The applications are created using components and binding their interfaces appropriately at the configuration level. Gluing component in this environments allow incompatible components to be glued together.

Middleware has also been used to provide the facility of dynamic configuration [19]. This approach uses the facilities of flexible computing environment provided by object middleware like CORBA, Java RMI and DCOM. In this model reconfiguration design activates produce the specification of well defined changes and constraints to be preserved during reconfiguration. The change management functionality make the system evolve from its current configuration to a resulting configuration.

Dynamic Reconfiguration approaches are also applied to the J2EE platform and Java based software in general [13 , 15]. In these approaches the reconfiguration has been achieved by employing the power of java to work across multiple platforms.

Dynamic Reconfiguration has also been tried to achieve through the use of lightweight infrastructure performs the dynamic and automatic reconfiguration using remote interfaces [41].

2.3 Shortcoming of the Dynamic Reconfiguration Techniques

All the techniques that have been proposed so far in the literature have not considered one important factor.

This factor is the handling complexity and performing reconfiguration with minimum human intervention i.e. in a fully automated manner. There are many change variables that play their role when a system needs to be configured dynamically. Some of these variables are temporal limitations, resource constraints, interdependencies, communication links etc. Performing dynamic reconfiguration in the presence of all these variables is a very difficult task for the person who is writing the reconfigurations. Some of the techniques [11,12]] require programmers to plan these reconfigurations manually and then these scripts reconfigure the system from one state to another state. This technique is not practical when the number of variables and components increase in the system. Moreover, for evolutionary changes these techniques do not provide a sound foundation. Another technique [14] proposes to make multiple versions of the components manage at the same time and replacing components one by one. This technique is useful for the application where time factor is not a major problem but for mission critical and real time systems this technique might not be that useful. Moreover, this technique also does not account for the large number of reconfiguration variables and huge number of components in the system.

The present techniques for dynamic reconfiguration also do not automate the process of reconfiguration fully. The process is mainly manual. Therefore whenever the system increases in size the programmers or system administrators are required to give explicit instruction for the reconfiguration of the system. This scenario may work in some situations. However, large size of the Distributed Software System and external attacks can make it pretty impossible for the human to keep up with the reconfiguration complexity. Humans as a lost resort, turn the system down.

Moreover, it is not possible for a human to take into account all the variables and still come up with an optimum reconfiguration plan. Therefore, an automated system for reconfiguration that automates the whole system of reconfiguration with minimum human intervention may be more useful for practical applications. An automated system will also relive the programmers and system administrator to write the reconfiguration scripts and worry about the system being going into an undesirable state. Whenever the system needs a reconfiguration it specifies the initial state or the state it is in presently and the final state. The automated reconfiguration system plans for an optimal reconfiguration fro one state to another. The automated reconfiguration system can also plan for the evolutionary changes in the system and takes the system to a safe state by planning the optimum path.

An automated system can be used not only in the reconfiguration of the system but also for the initial deployment of the system. If the initial deployment of the system is optimized the later reconfigurations can be relatively easier because of the right placement of the components.

2.4 Related Work in AI Planning

The second perspective is the usage of AI planners and their usage in other fields. Planning can be viewed as a type of problem solving in which the agent uses beliefs about the actions and their consequences to search for a solution over the most abstract space of plans, rather than over a space of situations [24]**.**

Planning has been an area of significant research from the mid 1950s. Newell, Shaw and Simon developed the first system to solve planning problems. This system's name was General Problem

Solver (GPS). After that many people have performed research in many areas of planning and performing the research from many perspectives.

A full detail of planning research is out of scope of this thesis. However, this thesis will focus mainly on the planning problems that are related to the problem of dynamic reconfiguration of software systems.

Planners have been developed to solve a range of problems in many different areas. Some of these applications have been discussed in the Related Work section of this thesis. Here we are going to describe the planning systems that take into account time and resource constraints for solving planning problems. There have been many research efforts that deal with temporal and resource planning [25, 26, 27, 28, 29]. These and other approaches attack the temporal planning problem through various ways. Some of these approaches include Graphplan extensions, model checking techniques, hierarchical decomposition, heuristic strategies and reasoning about temporal networks, are capable of planning with durative actions, temporally extended goals, temporal windows and other features of time-critical planning domains [30].

In almost all of the research there has not been an agreement on the representation formats for representing the semantics of the planning problems. The AI planners have not been standardized on one representation language up till very recently. Almost every planner needs a different set of input with varying format. Therefore it is very difficult to compare the effectiveness of AI planners in solving real problems. However, AI planning competition IPC 2002 [22] provides a standard language PDDL 2.1 (Planning Domain Definition Language) that is used by many planners. This language can be used to define the semantics of the planning activity using a single syntax.

The usage of AI planning systems for solving real world problems has significantly increased in recent years. The European Network of Excellence in AI Planning “PLANET” [32] identifies key areas where planning can be applied. These areas range from Robot Planning to Intelligent Manufacturing. PLANET has identified the various strengths and shortcomings of the AI planners. They have proposed areas of improvement for further research in AI planning.

One of the key areas in the PLANET network is of knowledge engineering. Under this research effort the usage of AI planners have been explored in Web Applications and Semantic Web. This effort [33] also focuses on the use of AI planners in E-Commerce applications, Software Process and Business Process Re-engineering.

Developing of a domain model could be a tedious task for a person in this area. There has been work also on developing the domain model easier and error-free [34].

Chapter 3

An Introduction to Planning

This chapter briefly introduces the basic idea of an Artificially Intelligent planner. This chapter also introduces the language PDDL 2.1 (Planning Domain and Definition Language). We have used PDDL in this project for solving Dynamic Reconfiguration Planning problems.

3.1 Planning Basics

A planner itself is a very obtuse thing. All the semantic knowledge that is inside a planner is specified in the domain. Other then the domain the planner needs a problem description to know what exactly is required in a particular situation. There are two files that a planner requires form its users. The syntax and semantics of these files are described briefly here.

Domain File
Problem File

Following is a brief description of the domain file and problem file.

3.2 Domain File

The Domain file consists of a description of the objects that are going to be planned for, the functions, predicates and the actions (durative and non-durative). All the actions need a notion of time (in case of durative ones), preconditions and postconditions. An example of a small domain file is given below. This example is taken from the 2002 International Planning Competition [35].

```
(define (domain Jugs)
(:requirements :typing :fluents :conditional-effects)

(:types jug nonjug)


(:functions (capacity ?j - jug)

            (contents ?j - jug)

       )


(:action fill

 :parameters (?j - jug)

 :precondition (< (contents ?j) (capacity ?j))

 :effect (and (assign (contents ?j) (capacity ?j))))


(:action empty

 :parameters (?j - jug)

 :precondition (> (contents ?j) 0)

 :effect (and (assign (contents ?j) 0)))

?j2))

                                    (and (assign (contents ?j1) 0)

(:action pour

 :parameters (?j1 ?j2 - jug)

 :precondition (> (contents ?j1) 0)

 :effect (and (when (<= (+ (contents ?j1) (contents ?j2)) (capacity

                                 (increase (contents ?j2) (contents ?j1))))

         (when (> (+ (contents ?j1) (contents ?j2)) (capacity ?j2))

              (and (assign (contents ?j2) (capacity ?j2))
                    (decrease (contents ?j1) (- (capacity ?j2) (contents ?j2)))))))

)
```

Figure 1: Domain File for Jugs Domain

3.3 Problem File

A problem file has two major parts. The initial state and the goal state: The initial state describes the initial state of the system. The system supposedly is in this state at the time of triggering of planner. The goal state is the desired state of the system. Apart from the initial state and the goal state the problem file also contains the name of the domain and the artifacts involved in the planning domain. A metric to minimize or maximize certain criteria can also be given optionally. Following is an example of a problem file that is based on the domain described above.

```
(define (problem jugs1)
  (:domain Jugs)
  (:objects jug1 jug2 - jug)
  (:init (= (capacity jug1) 5)
         (= (capacity jug2) 3)
     (= (contents jug1) 0)
         (= (contents jug2) 0)
  )
    (:goal (and (= (contents jug1) 1)))
    (:metric minimize total-time)
)
```

Figure 2: Sample Problem File for Jugs Domain

After the planner is given the above mentioned files as input i.e. Domain File and Problem File, it gives out a plan for going from the initial state to the goal state. Following is an example of a small plan. The formatting of output could be different from different planners.

```
1:(fill jug2)
2:(pour jug2 jug1)
3:(fill jug2)
4:(pour jug2 jug1)
5:(empty jug1)
6:(pour jug2 jug1)
```

Figure 3: Sample Plan for the Above Problem

There are a number of planners that accept the PDDL format and plan for Temporal domains. Some of them include LPG[3], MIPS[36], TALPlanner[37] and TLPlan[38].

Chapter 4

Applying Planning to Dynamic Reconfiguration

In this chapter we are going to present a novel technique for dynamic reconfiguration of distributed software systems that uses Temporal Planning with resource constraints. The aim of this technique is to fully automate the complexities of performing a Dynamic Reconfiguration of Software Systems. The use of automated planning using Artificially Intelligent techniques has been discussed before in the related work section. The recent advances in the field of AI planning have led to its use in solving real world problems. We have discussed some of applications of AI planning in the Related Work chapter.

AI planners have been used to solve toy problems in the past. A significant reason for the use of AI planners in real application now is because they accept much powerful and expressive lingo then before. Some aspects of this lingo incorporate time and numeric constraints that are really powerful concepts when it comes to optimal temporal planning with strict resource constraints. Moreover, previously the planners did not support planning for parallel tasks in their plans; however, they are now able to make plans that have multiple tasks in parallel. This power of AI planners is the major motivation for their use in this project to achieve dynamic reconfiguration of software systems.

4.1 The Need to Plan Dynamic Configuration

Researchers have long been struggling with the problem of finding optimal ways for the “State Transition” problem in Dynamic Reconfiguration of Distributed Software Systems. A “State Transition” is a way to bring the system from one state to another state without bringing the system down. The state transition problem intensifies, when the size of the software system increases. The increase in size could be an increase in the physical size of the system e.g. adding new components and machines. I could also be an increase in the amount of interdependencies created during the lifetime of the software systems. Therefore, this increase in size could make the reconfiguration process hard. Moreover, because of the high availability notion, the reconfiguration has to be carried out in a certain specified amount of time. The state transition problem, if done without a plan could be very time consuming. Moreover, even if it performs it in a specified time the new state could be not optimal. Apart from time, resource constraints could also make the problem much harder. There are many reasons that make it hard for the reconfiguration process to remain difficult if there is no intelligent technique used for solving this problem.

The first reason is the search space that is involved in finding an optimal plan. The complexity and size could make the search space huge. Therefore using traditional search based techniques e.g. Depth First Search, could take a lot of time to find an optimal plan. Moreover, this is not a node search; this search is in the space of plans [24].

The second reason is that in some situations an Explicit Configuration is not given for the state transition. This makes the reconfiguration problem harder, because the Reconfiguration process not only needs to find the right plan but also the best state to go into. This king of state transition is also called Evolutionary Change.

The third reason is the state transition with optimal usage of resources and time. Here time can be divided into two categories: The time for the finding the plan and the time to carry out the execution of the plan by the system. In case of an automated planner one can specify plan find time. It's the job of the planner to find the best plan in the specified time. Indeed the more the time we will give to the planner the more the plan quality is. Moreover, there are also resource constraints along the way of finding an optimal plan, for example some machine can not accept more then a certain number of components.

The fourth reason is the unanticipated state problem. Sometimes the software system needs to go into a state where it has no way to come out because the programmers of it did not anticipate this state. Unanticipated state is a very common problem now because of the connectivity and use of distributed complex systems. Many factors play their role in going into unanticipated states. These factors include external attacks, internal inconsistencies, failure of a critical resource and many more. These factors sometimes lead to a partial failure of the system or reduction of services the software system is providing.

These reasons convinced us to use more intelligent techniques for automating Reconfiguration Process. Finding the right plan for the reconfiguration process is the first step that we have taken in this direction. AI planners reduce the search space of finding the optimal plan by using different heuristics. The heuristic that is used by LPG [3], the planner that we used in this project, is called “Local Search on Planning Graphs”. Different planners like [36], [37] and [38] use different search heuristics to find an optimal plan.

4.2 Temporal Planning: Our Approach to Dynamic Reconfiguration

In this project AI planners have been used to find the optimum way for the state transition problem. This project is a proof of concept of the strengths and limitations of AI planners in the dynamic software reconfiguration domain. It is stressed here that according to the knowledge of the author, not a single flavor of AI planning technique has been used in the field of Dynamic Software Reconfiguration Management.

In this approach we have described a process of dynamic reconfiguration that will make the process of dynamic reconfiguration much more intelligent and automated then the techniques proposed so far. This technique is not an alternate to the techniques described in the previous chapter(s) but a solution to the problem of reducing complexity and automating the process of Dynamic Reconfiguration Management. Our technique can be used in the dynamic reconfiguration of any distributed software system provided that the system can be expressed in PDDL. A well specified domain can improve the efficiency of the reconfiguration process.

Before we describe the technique, we will present an overview of the architecture of the system that we have been working with. This system is just an example system that is required to explain the process of dynamic reconfiguration in a more concrete way later in this chapter.

4.3 Architecture of the System being reconfigured:

It is very difficult to describe a system that is an abstraction for all the system available in the real world. Therefore, we have made certain assumption about the system that needs to be reconfigured. However, we have used the general terms of components, connectors and machines. The role of components, connectors and machines has been abstracted to reduced level of details. However the planners, in general, can take a much more comprehensive description of the semantics of a system then described here.

Our system consists of three artifacts. These artifacts are components, connectors and machines.

4.3.1 Component:

The components contain the logic of the system. A component is any entity that we can manage. A component can exist at one machine at one time. Components need to be connected to a connector in order to communicate with other components. Some of the operations that can be performed on the components are starting a component, stopping a component, connecting a component etc.

4.3.2 Connector:

The connectors provide a link of communication. One connector exists on one machine at one time. The connector can be linked to other connectors for communication. A connector can be thought of a weak form of connectors described by Mehta et al. [18]. The connector needs to be connected to another connector before it can accept connections from the component. The connector has almost the same operations as a component except it has a interconnect operation that links it with other connectors.

4.3.3 Machine:

A machine is a place where components and connectors are deployed. The machine has a resource constraint: it can not deploy more then a certain number of components and connectors on it. The operations that can be performed on the machine are starting and stopping. It can accept connections from local artifacts and artifacts deployed on other machines.

All of the operations on components, connectors and machines have a time constraint.

Having described the system, we will now take a look at the Planning aspects of the dynamic reconfiguration.

4.4 Temporal Planning Process for Dynamic Reconfiguration:

In any planning system there are three pieces of information that needs to be specified in order for the planner to work. These pieces are Domain Description, Initial State Description and Goal State Description. These three pieces of information have been described in detail with respect to the planning systems in general. Here these three are described with respect to the problem of Dynamic Reconfiguration.

4.4.1 Domain:

The domain consists of all the operation, predicates, functions and durative actions that describe the behavior of the system. An action consists of parameters, function to describe the time required for this action and predicates that describe the state of the artifact before and after the execution. These predicates are described in terms of pre condition and post condition. A set of precondition has to be met in order to operate this action on an artifact. When the action completes the artifact takes the post condition predicates. Post condition for some actions are usually pre condition for another set of actions. An example domain is given in Appendix A.

4.4.2 Initial State:

The second piece of information for planning is the description of the initial state. The initial state is to be described in terms of the predicates and functions (for time). The initial state contains information about the state of the artifacts, for example like on which machine the component is running, whether its connected with a connector or not. Initial state also states the time required to perform the actions. It can state the time required to start the component or move component from one location to another.

4.4.3 Goal State:

The goal state states the required configurations that the system must go into. This can be stated in two ways Explicit Configuration or Implicit Configuration. In the Explicit Configuration the exact configuration is stated for example component A has to be started on machine 7. While in Implicit Configuration just the predicate of the component is stated. For example the component B needs to be started and connected. Now depending on the time and other variables the planner should come up with an optimum place where component B could start.

4.4.4 Plan:

The planner takes the three pieces of information and gives out an optimum plan depending on the time we give it to compute the plan. The plan consists of the series of steps in time that needs to be performed for going from initial state to the goal state. These steps could be parallel also. A sample plan from LPG is given in Appendix C.

Having described the planning process, we will now describe Planit Reconfiguration Engine, that we have developed. In this chapter we will described the conceptual details of Planit. Technical and Implementation will be described in the nest chapter. A flow chart of Planit is described in Figure 4: Reconfiguration Process.

4.5 Planit Reconfiguration Process

Planit performs three tasks.

Initial Deployment of the System

Sensing System Health

Reconfiguring the System

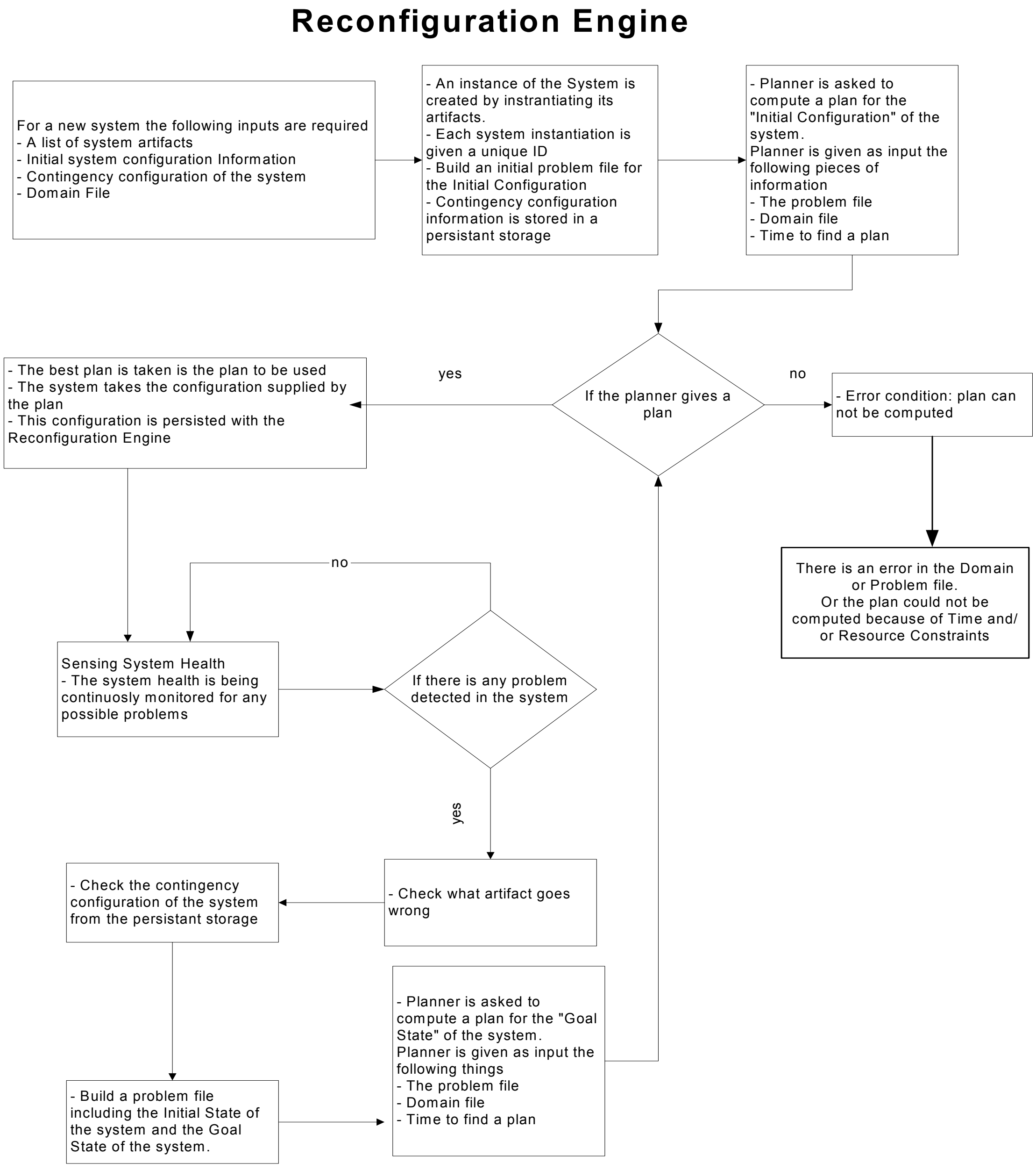
Reconfiguration Engine
For a new system the following inputs are required
- A list of system artifacts
- Initial system configuration Information
- Contingency configuration of the system
- Domain File
- An instance of the System is created by instrantiating its artifacts.
- Each system instantiation is given a unique ID
- Build an initial problem file for the Initial Configuration
- Contingency configuration information is stored in a persistant storage
- Planner is asked to compute a plan for the "Initial Configuration" of the system.
Planner is given as input the following pieces of information
- The problem file
- Domain file
- Time to find a plan
- The best plan is taken is the plan to be used
- The system takes the configuration supplied by the plan
- This configuration is persisted with the Reconfiguration Engine
yes
If the planner gives a plan
no
- Error condition: plan can not be computed
There is an error in the Domain or Problem file.
Or the plan could not be computed because of Time and/ or Resource Constraints
no
Sensing System Health
- The system health is being continuosly monitored for any possible problems
If there is any problem detected in the system
yes
- Check the contingency configuration of the system from the persistant storage
- Check what artifact goes wrong
- Planner is asked to compute a plan for the "Goal State" of the system.
Planner is given as input the following things
- The problem file
- Domain file
- Time to find a plan
- Build a problem file including the Initial State of the system and the Goal State of the system.

Figure 4: Reconfiguration Process

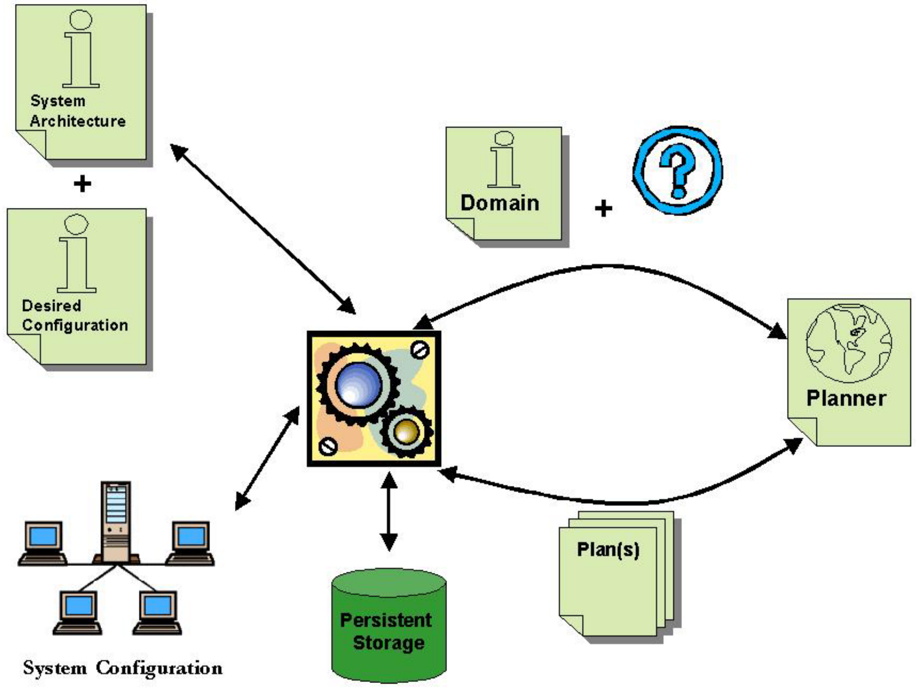

Figure 5: Flow of tasks in Planit

The above given figures Figure 8: Machine State Diagram and Figure 4: Reconfiguration Process gives an overview of the three tasks the system performs. Following is the description of these tasks.

4.5.1 Initial Deployment of the System:

The initial deployment is the process where the system is deployed across the network for the very first time. It has not been started anywhere on the network before. This process is carried out by supplying information about the structure of the system. This structural information consists of the components, connectors and machines involve in this configuration. All the components, connectors and machines are enumerated along with their properties. The goal state of the system is provided for example which component connects to which connector etc. The goal state can described in terms of Explicit Configuration or Implicit Configuration or a mix of both. Planit takes the information about the domain of the system and an enumeration of the artifacts. It asks LPG to compute the plan. LPG takes this information and devises the optimal placement of the components and connectors and their interconnections. Once the plan has been computed it is given out and an interpreter parses this plan. Planit deploys this configuration according to the steps given in the plan. Once the deployment process is complete Planit monitors the artifacts by receiving events from the artifacts.

4.5.2 Sensing System Health

Once the system is deployed and goes into a working state, its health is being continuously monitored. This monitoring can be carried out be explicitly getting heartbeat from the system artifacts or getting events from an external monitor that assess the system health. Once it is determined that the system is not keeping a workable state and it is going into an undesirable state then reconfiguration is necessary to restore system health.

4.5.3 Reconfiguration Process:

Planit trigger the reconfiguration processes when there is a need for performing dynamic reconfiguration. The reconfiguration may be necessary due to some external attack or explicit user request. The initial state and the target or goal state of the system are to be calculated by the Reconfiguration System. The initial state can be judged by the information about the health of the system. This information is written in terms of an initial state of the system in the problem file. This is the first part of the problem file. Writing this part is pretty straight forward.

The second part is writing the goal state of the system. There can be multiple approaches for specifying a goal state. The goal state could be defined through the two types of configuration supported by Planit i.e. Explicit and Implicit.

4.5.3.1 Explicit Reconfiguration

In case of Explicit Configuration, Planit reads the contingency configuration from a file or persistent storage and writes it as a goal state. In an Explicit Configuration the artifacts and their target configuration is explicitly described as a goal. Explicit Configuration information can be specified in a number of ways depending on the need the system. An example problem file for Explicit Configuration can be found in Appendix D.

4.5.3.2 Implicit Configuration

On the other hand if there is no Explicit Configuration available then the goal state is written using an Implicit Configuration. The Implicit Configuration only specifies a property of the system that needs to hold after the plan finishes. The planner has to find an optimal way to achieve the state where the property holds true. An example file for the Implicit Configuration is given in Appendix B.

The goal state can also contain a mix of both Implicit Configuration and Explicit Configuration.

The newly written problem file and the domain file are provided to the planner.

The planner finds an optimum path from the initial state to the goal state. The interpreter analyzes this plan and propagate the actual instructions to the system. Planit stores the newly developed configuration of the system and again started monitoring the system by watching the health of the system. .

4.6 Dynamic Reconfiguration Planning Domain

We have built a dynamic reconfiguration planning domain for our demonstration purposes. The full domain is given in Appendix A. This domain that we have developed consists of very simple operations on components, connectors and machines. It is built to test both temporal and resource constraints for the execution of the dynamic reconfiguration.

Following is a state diagram of the three artifacts. It specifies the states in which these artifacts can go. Some of the states of an artifact are dependent on the states of other artifacts.

4.6.1 Properties of Component

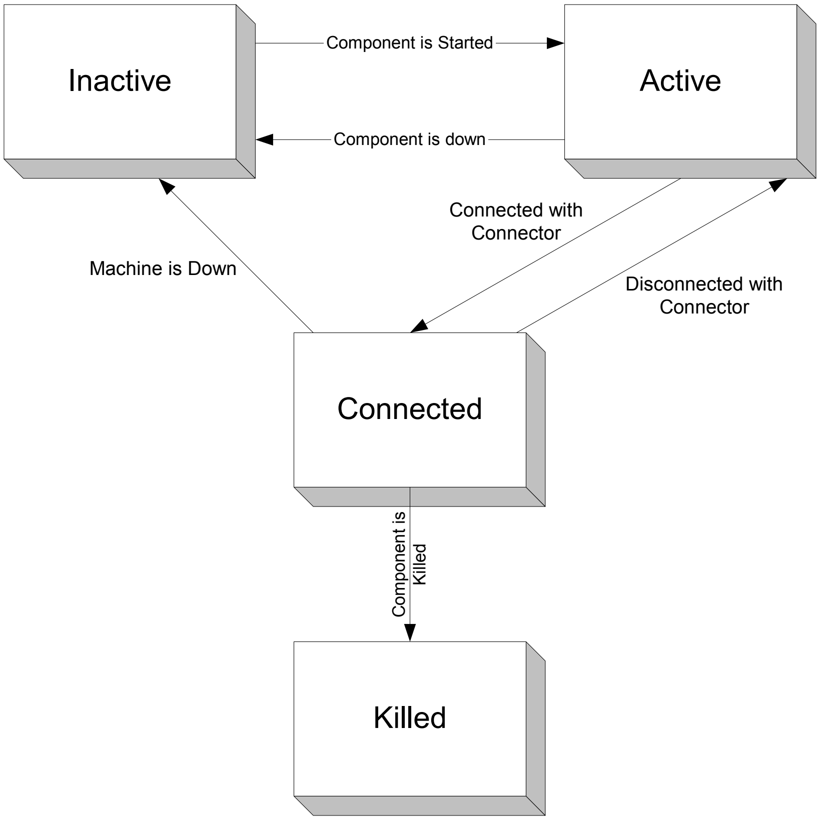

Figure 6: Comopnent State Diagram

In Figure 6: Comopnent State Diagram we have made the state diagram of the component. Following are properties that the component will hold in different states.

4.6.1.1 Inactive State

Component Not Started

Component is down

4.6.1.2 Active State

Component Started

Machine is up

Ready to be connected to connector

4.6.1.3 Connected State

- Component is connected to connector

4.6.1.4 Killed

- Component is killed

4.6.2 Properties of Connector

In Figure 7: Connector State Diagram we have made the state diagram of the component. Following are properties that the component will hold in different states.

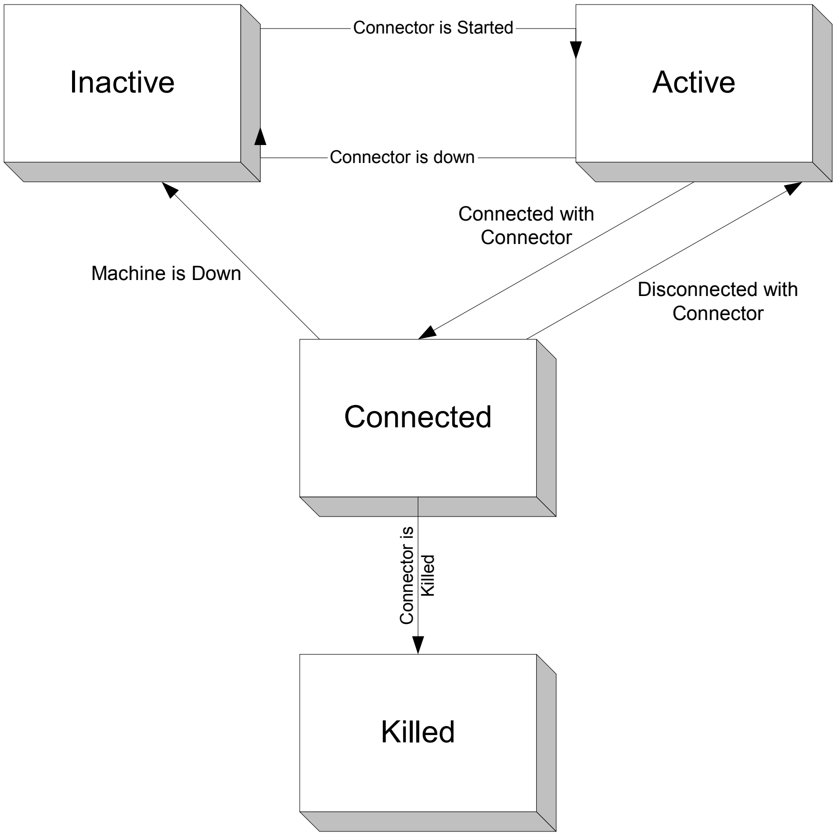

Figure 7: Connector State Diagram

4.6.2.1 Active State

Connector Not Started

- Connector is down

4.6.2.2 Inactive State

Connector Started

Machine is up

- Ready to be connected to another connector

4.6.2.3 Connected State

- Connector is connected to another connector

4.6.2.4 Killed State

- Connector is Killed

4.6.3 Properties of Machine

In Figure 8: Machine State Diagramwe have made the state diagram of the component. Following are properties that the machine will hold in different states.

4.6.3.1 Up State

- Machine is Down

4.6.3.2 Down State

Machine is up

Ready to start components/connectors

4.6.3.3 Killed State

- Machine is Killed

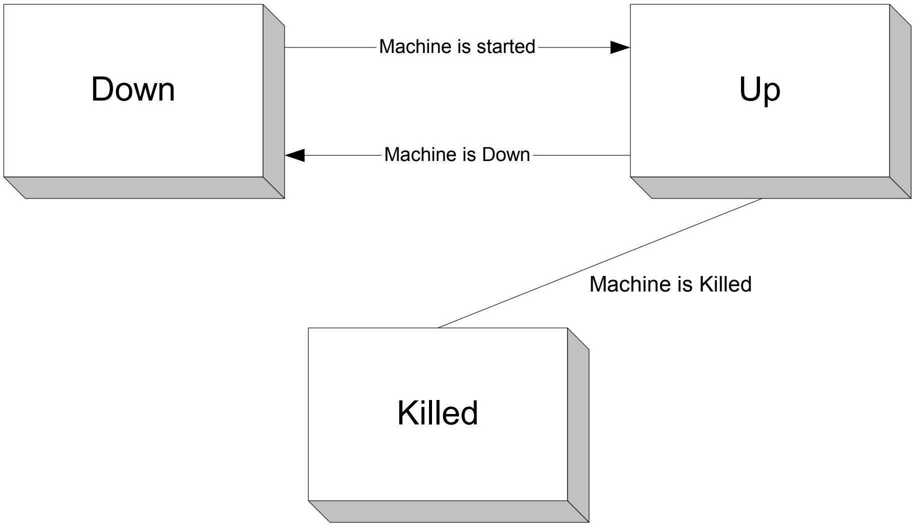

Figure 8: Machine State Diagram

Chapter 5

Planit: Technical Overview

In this chapter an overview of the implementation details of Planit is presented.

5.1 Planit Architecture

The architecture of Planit is given in Figure 9: Planit Architecture.

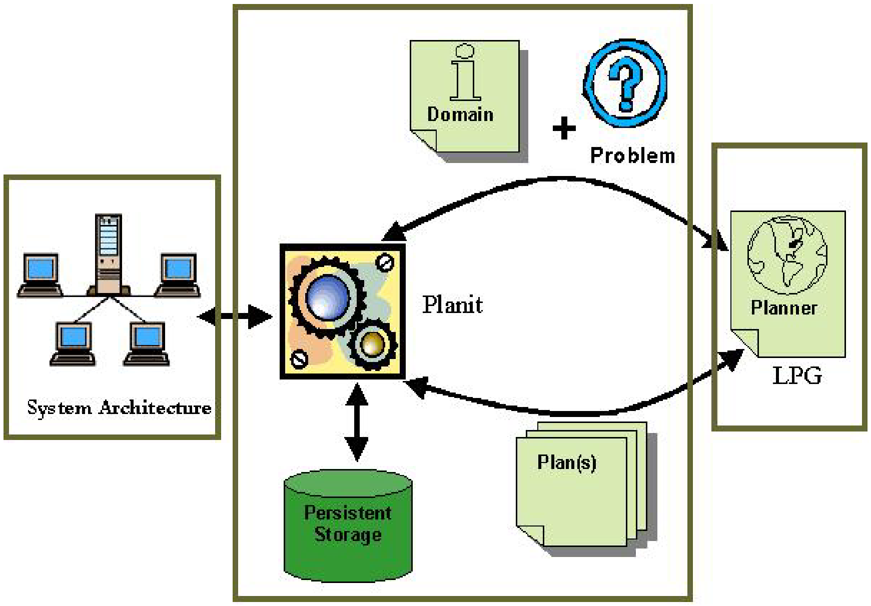

Figure 9: Planit Architecture

5.2 Implementation Overview

This Planit has been developed on Sun OS 5.8. The machine is an Ultra2/2200/512 with 2 x 200Mhz CPUs and 512 MB ram. The planner used is our experiment is LPG version 1.1 (Local search for Planning Graphs) [40]. This planner has been developed at Universit‘a degli Studi di Brescia, Brescia, Italy.

The main Planit has been written in Java 1.2. All the functionality has been tested for it compatibility with LPG. Planit can be used with other Temporal Planner with little modifications.

The class diagram of Planit is presented in the figure below.

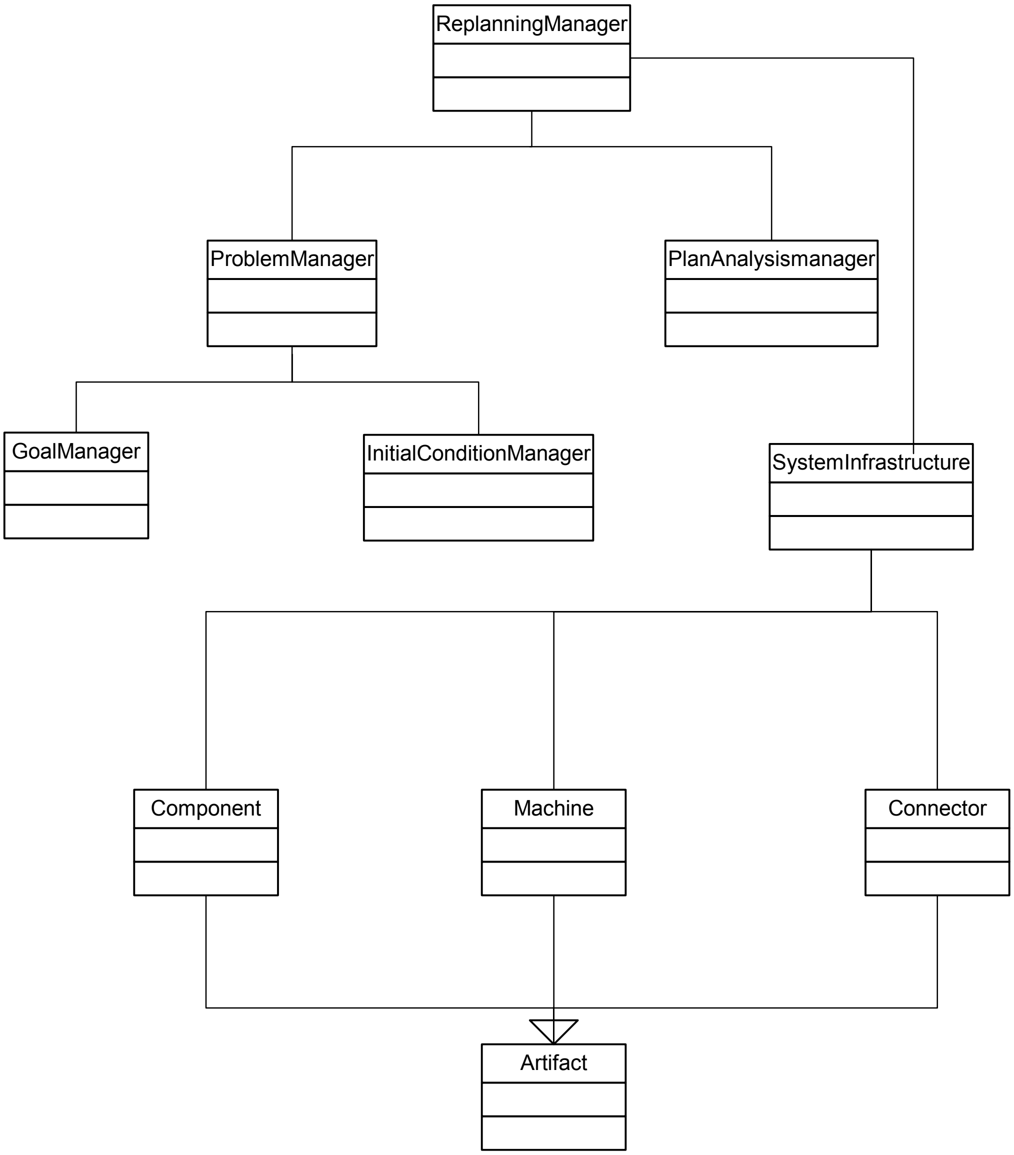

Figure 10: UML Class Diagram of Planit

5.3 Packages

The classes given in Figure 10: UML Class Diagram of Planit. Planit has four main packages.

replanningmanager

problemmanager

systemdatastructure

domaindatastructre

Following is a brief overview of these packages and their role in the big picture.

5.3.1 Package planit.replanningmanager

The replanningmanager is the brain of Planit. It has two classes. These classes are

ReplanningManager

PlanAnalysisManager

A brief description of the classes is given below.

5.3.1.1 ReplanningManager

This class is responsible for the initial deployment and handling subsequent reconfiguration requests of Planit. It has methods that support the persistence of the configuration or state of the system at any one instance. It also has the ability for the contingency configuration of the deployed system. The invoking of LPG and getting the best plan available is also the task of this class.

5.3.1.2 PlanAnalysisManager

The LPG planner returns the plan file for a particular problem instance. This Plan file has the information about the plan, schedule and artifacts involved. The role of this class is to parse, analyze and order the plan given out by the planner.

5.3.2 Package planit.problemmanager

This package has the following classes

ProblemManager

InitialConditionManager

GoalManager

A brief description of the classes is given below.

5.3.2.1 ProblemManager

The ProblemManager is invoked when there is a reconfiguration need from the system. The ProblemManager class is invoked by giving it the present state of the system. This state has the affected artifacts explicitly mentioned. The ProblemManager analyzes the present state of the system and perform three functions.

The planner expects the problem file having three pieces of information. The first one is the enumeration of the objects of the system. The ProblemManager writes this enumeration in the problem file and involves the two other classes for filing the rest of the information.

5.3.2.2 InitialConditionManager

The InitialConditionManager works as part of the ProblemManager. The task of the InitialConditionManager is to write the present state of the system into the problem file.
It checks the system state using the standard data structure for specifying system state in this Planit and performs the assigned task.

5.3.2.3 GoalManager

While the InitialConditionManager write the initial state of the system. The GoalManager does the task of writing the target or goal state of the system. The GoalManager checks the contingency configuration of the system and writes the Implicit, Explicit configuration or a mix of two.

5.3.3 Package planit.systemdatastructure

The systemdatastructure has the supporting data structure for representing the system in this Planit. It has five classes.

Artifact
Component
Connector
Machine
SystemInfrastructure

A brief description of the classes is given below.

5.3.3.1 Artifact

Artifact is the abstract class for specifying the common properties of the three artifacts used in Planit.

5.3.3.2 Component:

A Component is the representative class for describing a component and all the properties and operations on it. Some of the properties include “start-time” and “connect-time”. Some of the operations are “start-component” and “connect-component”.

5.3.3.3 Connector

A Connector is the representative class for describing a connector and all the properties and operations on it. Some of the properties include “start-time” and “connect-time”. Some of the operations are “start-connector” and “inter-connect connector”.

5.3.3.4 Machine

A Machine is the representative class for describing a machine. It has properties including resource limitations and operations for stating and stopping machine.

5.3.3.5 SystemInfrastructure

A SystemInfrastructure is the aggregation of system artifacts including components, connectors and machines. It has operations like adding and removing artifacts.

5.3.4 Package planit.domaindatastructre

This package captures the information contained in the Domain file that is supplied to the Planner for the domain of the system. It has the following classes.

Domain

Action

DurativeActions

NonDurativeActions

Function

Predicate

The Domain class is the aggregation of all the classes in this package. Following is a brief description of the classes in this package.

5.3.4.1 Domain

The Domain class is the list of aggregation of all the predicates, functions, actions etc.

5.3.4.2 Action

Action is the class that represents the "action" from the domain file. It does not have all the necessary information like the pre-condition or post-condition. However it only has the required information that is needed for writing a problem file. This class has further sub classes called Durative and Non Durative actions.

5.3.4.3 Durative Actions

The Durative actions are the actions that are timed. These actions need a specific time to execute. The time is specified by the function associated with each Durative actions.

5.3.4.4 NonDurativeActions

The class captures the non durative actions. The NonDurativeActions are the actions in which time is not associated as a property. These actions execute without any consideration of time.

5.3.4.5 Functions

The Function class has the information about the time of the Durative Actions. It specifies what time is required for the execution of each durative action.

5.3.4.6 Predicate

The Predicate class captures the predicates specifies in the system. A predicate is any specifies property of the system.

5.4 Running Planit

Planit can be run by invoking the ReplanningManager class inside the replanningmanager package. The command line arguments expect ways of supplying the information.

The first way is in which there is a demonstration of Planit. This run of Planit is for demonstration purpose only. The user has to supply the domain file, the number of components, number of connectors and the number of machines. Planit randomly generates these artifacts and assigns their variables random values. These random values have very realistic range that is able to demonstrate the functionality of Planit. An example command instruction is something like this.

```
home> java replanningmanager.ReplanningManager jtmc3 1 30 5 2 2 experiment1
```

The command line argument jtmc3 is the domain file in PDDL language. The other six arguments are the maximum time to find a plan, number of plans required, the number of components, number of connectors, number of machines and the name of the problem respectively.

When the user enters this command Planit makes a new system or if the system with that name is already present then it reads the present configuration of the system. . This system includes the components, connectors and machines along with their names and the values assigned to their variables. Using this information Planit writes the problem state of the system. The problem state at this time is nothing more then the enumeration of the artifacts of the system and their properties. Because for demonstration purposes it is hard to assign the goal state for the many number of artifacts, Planit assigns Explicit Configuration based on a fair criteria.

Planit after writing the problem file of the system gives this information to the LPG planner. LPG computes a plan and gives the result as the output. Planit parses the plan file given by LPG. The information from this file is analyzed and the system takes the necessary reconfiguration. This reconfiguration is actually a faked one because this tool is more of proof of concept then a real system.

The deployed system then sends Sienna events in case of problems. If there are problem events received by Planit, it checks the present configuration of the system and checks which artifacts are affected by the system. It writes the present state of the system in the initial condition part of the problem file. It then checks the contingency information about the affected artifacts. In case there is contingency information available the Explicit Configuration is written. In case there is no explicit configuration available the Implicit Configuration of the system is written.

This new problem file along with the domain file and other optional parameters is then given to the LPG planner. LPG computes the plan for the new configuration. Once the plan is calculated the process of executing that plan is again propagated in the system. The system takes the new reconfiguration and starts sending events about its health.

This process then continues continuously unless the user interrupts it.

Chapter 6

An Example: Joint Battlespace Infosphere

There could be many possibilities for a full-scale example of Planit. However, our criteria for selecting an example system consist of a practical and realistic system that is available out in the world being used for real applications. The example system should be able to demonstrate the strengths and weaknesses of the technique in context. Therefore we have chosen an example of a JBI (Joint Battlespace Infosphere). JBI is an evolutionary system for command and control information integration. It has various components, connectors (called disseminators in JBI) that are available to play with.

The JBI system was described in [39] as “The essence of the JBI is a globally interoperable information space that aggregates, integrates, fuses, and intelligently disseminates relevant battlespace information to support effective decision making. The JBI is part of a global combat information management system established to provide individual users with information tailored to their specific functional responsibilities. It integrates data from a wide variety of sources, aggregates this information, and distributes it in the appropriate form and level of detail required by users at all levels.”

There are many types of components in JBI, ranging from AWACS and Radars to Position Fuselets and Disseminator Nodes. All this different kinds of artifacts could be used to model a really practical usage of Planit we have been working on.

In JBI there are components like AWACS, Ground Radar, Disseminator Nodes (Equivalent to Connectors in our abstraction). These components are providing different functionality to the different parts of JBI. For example AWACS and Ground Radar are feeding the signals to the JBI for the location of the aircraft and ground troops. In case an AWACS is shot down the functionality of AWACS has to be restarted on another machine. If AWACS can not be started during a specified time the configuration of the system has to be changed in order to provide a full set of functionality.
If a disseminator node (or connector) is down, then all the components that are connected to the main communication link has to be linked to other disseminator nodes. This is also a problem of reconfiguration. In our experiments we have used the abstraction of components, connectors and machines. All three of them could be down and their downing could cause a major disruption in the system. We have to somehow remove the disruption by reconfiguring the system. Following is a subset of experiments that we have conducted on the JBI:

6.1 Experimental Results

We have conducted experiments for the evaluation of two aspects of Planit.

Explicit Initial Deployment of Systems

Implicit Initial Deployment of Systems

6.2 Experimental Setup

There are several system artifacts that we have made for our experiments. These artifacts can be broadly divided into components (AWACS, Ground Radar, Satellite, Position Fuselets etc), connectors and machines. The experiments have been conducted on only the initial deployment of the system because for the planner this is the toughest task. We have performed five experiments. The experimental setup for these experiments is given below.

Table 1: Experimental Setup

Experiment No / Artifact Name	Components	Connectors	Machines
1	10	4	4
2	20	6	6
3	30	8	8
4	40	10	10
5	60	10	10

The above given artifacts have certain properties like start time, resource constraints (in case of machines) associated with each of them. All the properties come under reasonable time and resource bounds.

6.3 Results of Explicit Configuration Deployment

Table 2: Results of Explicit Configuration for Deployment

Experiment	No of Plans Found in 30 Seconds	Time to Find the Best Plan(in sec)	Duration of the Best Plan(in sec)	Duration of the Worst Plan(in sec)
1	5	12.39	67	83
2	4	18.64	66	137
3	3	27.95	100	144
4	2	23.00	76	84
5	1	17.93	138	N/A

The above results show the plans that planner was able to find given a 30 sec period and a maximum of 5 plans. Time to find the best plan and the duration of the plan to go from the initial state to the goal state are given also. The time for the worst plan and the best plan are given for comparison.

One can see that in the case of Explicit Configuration the planner has performed quite well. It is able to calculate at least one plan for all the experiments and in some cases the best and worst duration have significant differences among multiple plans.

6.4 Results of Implicit Configuration Deployment

Table 3: Results of Implicit Configuration for Deployment

Experiment	No of Plans Found (in 60 sec)	Time to Find the Best Plan(in sec)	Duration of the Best Plan(in sec)	Duration of the Worst Plan(in sec)
1	3	4.92	62	70
2	5	56.71	65	81
3	2	36.99	108	124
4	0	N/A	N/A	N/A
5	0	N/A	N/A	N/A

The above results show the plans that planner was able to find given a 60 sec window. Time to find the best plan and the duration of the best plan and the worst plan to go from the initial state to the goal state are given also given.

The planner is able to calculate the results up till Experiment 3. In the case of Experiments 4 and 5 the search space is so huge that the planner is not able to calculate the plan in the specified time. In case of Experiment 4, we run this experiment for an unlimited time and the planner was able to find out a plan in 412 seconds as compared to the 60 seconds time limit for other experiment. Therefore, our conclusion is that the increase in the number of artifacts can decrease the ability of the planner to find out the explicit reconfigurations in a small amount of time. However if one gives ample amount of time it will eventually finds a solution, provided a solution exist for the problem.

6.5 Experimental Results Conclusion

The use of a Temporal Planner to compute plans for reconfiguration has resulted in very exciting and fruitful results with a few exceptions. In this section we will discuss the strengths and weaknesses of Temporal Planners with particular emphasis on the Dynamic Reconfiguration of Software Systems.

6.5.1 Strengths of using Temporal Planners

We have tested the planners using about 80 artifacts. It resulted in about 450 actions definitions and 350 facts. LPG is able to find out the first result in about 5 seconds. These results really show the power of using the Temporal Planners in large software systems.

The semantics of the planners can be divided into two or more categories. In our case we have used only one domain file both for deployment and reconfiguration. However, multiple domains could be constructed for separating the tasks deployment and reconfiguration. Use of multiple domains also allows the extensibility of the system. Therefore, the use of multiple domains can help increase use of planners to solve new problems in dynamic reconfiguration. There are other advantages that are possible by the use of multiple domains. These are the less search space a planner has to search because of the less semantic information and the expressiveness of the PDDL that can be applied to multiple domains.

6.5.2 Weaknesses of Temporal Planners

The semantic knowledge in the domain has to be thoroughly tested and verified before applying it to the real domain. The domain is the only semantic knowledge the planner has access to. Therefore, if there is a mistake in the domain knowledge it could create unanticipated problems while the system is running.

Sometimes in certain situations there is no solution for a certain problem. The planners are not able to find out always about this. Therefore, care has to be taken in cases where planners is taking too long to find a solution.

Initial conditions that are specified in the problem file have to be written very carefully. The initial condition can provide a lot of facts to the planner. If there are more initial conditions the planner will have to search a bigger search pace. If conditions are carefully specified the planner will have small search space and it can search for good plans in relatively less time.

Chapter 7

Future Work

There are many exciting dimensions to go from here in terms of future work. We have shown that this technique could very well be used for the Dynamic Reconfiguration of Distributed Software Systems. We have experimented with certain properties of the Distributed Software System. Now this work could be extended to see their other properties.

The first dimension is the scalability of the Distributed System. Planit can be extended to accommodate the new addition of new components. At this time Planit can deal with a fixed configuration of the system

The second dimension is the extent to which the AI planners can be used. In our case we have not experimented with more then 120 artifacts. This number can be increased to show if the use of planners worth it. Moreover, at this time we have only one domain file both for deployment and reconfiguration. Multiple domain files could be created that can capture the domain semantics in a better way and also reduce the search space of the planner.

The third dimension is the use of other properties of PDDL language. The PDDL is constantly being extended and now it has more powerful syntax then the one we have used. Therefore this powerful syntax can be used to capture more domain knowledge from the dynamic reconfiguration of software systems.

The fourth dimension is the use of metrics in PDDL. PDDL accept metrics to minimize or maximize total time and many more metrics are being added to make the search well targeted. The use of these metrics can be experimented with our domain and see if the results are better.

The fifth dimension is the experimentation with other planners. We have experimented with only one planner. However, there could be other planners that can be tested. It is quite possible that they give better results in our domain.

Chapter 8

Conclusion

This project aimed to view the Dynamic Configuration of Software Systems from another angle. This angle is the need for planning the dynamic reconfiguration before carrying it out. It uses intelligent techniques from Artificially Intelligent planning for computing the best plan. We hope that this will help the researchers to make their own technique more fruitful. It is stressed here that this technique is a complementary technique that can be used with any other dynamic reconfiguration management technique. It is quite possible that planner, due to a very complex problem, is not able to find even a single plan for the solution of a certain problem.

In this case the reconfiguration system that is using Planit must have an alternate plan to figure out what to do if there is no plan from Plait. Having a contingency plan that takes the system into a default state can solve this problem.

Bibliography

1. Kramer, J. and J. Magee. “Dynamic Configuration for Distributed Systems,” IEEE Transactions on Software Engineering, Vol. SE-11 No. 4, April 1985, pp. 424-436.
2. R. S. Hall, D. Heimbigner, and A. L. Wolf. "Evaluating Software Deployment Languages and Schema," Proc. of the 1998 Int'l Conf. on Software Maintenance, IEEE Computing Society, Nov. 1998.
3. A. Gerevini, I. Serina, "LPG: a Planner based on Planning Graphs with Action Costs", in *Proceedings of the Sixth Int. Conference on AI Planning and Scheduling (AIPS'02)*, AAAI Press, pp. 13-22, 2002.
4. T. Batista and N. Rodriguez. Dynamic Reconfiguration of Component-Based Applications. In Pro-ceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems, pages 32–39. IEEE Computer Society, June 2000.
5. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. ACM Transactions on Computer Systems, 19(3):332–383, August 2001.
6. D.M. Heimbigner and A.L. Wolf. Post-Deployment Configuration Management. In Proceedings of the Sixth International Workshop on Software Configuration Management, number 1167 in Lecture Notes in Computer Science, pages 272–276. Springer-Verlag, 1996.
7. J.C. Knight, D.M. Heimbigner, A.L. Wolf, A. Carzaniga, J. Hill, and P. Devanbu. The Willow Surviv-ability Architecture. In Proceedings of the Fourth International Survivability Workshop, March 2002.
8. J. Magee and J. Kramer. Self Organising Software Architectures. In Proceedings of the Second Inter-national Software Architecture Workshop, pages 35–38, October 1996.

9. Scientific Advisory Board. Building The Joint Battlespace Infosphere. Technical Report SAB-TR-99-02, U.S. Air Force, December 2000.

10. Shrivastava, S. and Wheater, S., "Architectural Support for Dynamic Reconfiguration of Large Scale Distributed Applications" The 4th International Conference on Configurable Distributed Systems (CDS'98), Annapolis, Maryland, USA, May 4-6 1998

11. B. Agnew, C. R. Hofmeister, J. Purtilo. Planning for change: A reconfiguration language for distributed systems. *Distributed Systems Engineering*, Sept. 1994, vol.1, (no.5):313-22.

12. M. Endler, J. Wei. Programming Generic Dynamic Reconfigurations for Distributed Applications, *Proc. of the International Workshop on Configurable Distributed Systems* London, pp. 68-79, IEE, March 92

13. Little M., S. Wheater, "Building Configurable Applications in Java," in Proc. 4th IEEE Int. Conferenceon Configurable Distributed Systems, May 1998

14. J. E. Cook and J. A. Dage. Highly Reliable Upgrading of Components. *21st International Conference on Software Engineering (ICSE99)*, Los Angeles, CA, May 1999

15. Rutherford, M.J., Anderson, K., Carzaniga, A., Heimbigner, D., and Wolf, A.L. "Reconfiguration in the Enterprise JavaBean Component Model" In *Proceedings of the IFIP/ACM Working Conference on Component Deployment,* Berlin, 2002, pp. 67-81

16. Chen, X. and Simons, M. (2002). A Component Framework for Dynamic Reconfiguration of Distributed Systems, accepted by the First IFIP/ACM Working Conference on Component Deployment (CD 2002), Berlin, Germany.

17. Jürgen Berghoff, Oswald Drobnik, Anselm Lingnau and Christian Mönch. Agent-based configuration management of distributed applications. In *Proceedings of the Third*

International Conference on Configurable Distributed Systems ICCDS '96, pages 52-59, Maryland, April 1996.

18. N. Mehta, N. Medvidovic and S. Phadke, Towards a Taxonomy of Software Connectors, Technical Report, Center for Software Engineering, University of Southern California,USC-CSE-99-529, 1999.
19. João Paulo Andrade Almeida, Maarten Wegdam, Luis Ferreira Pires, Marten van Sinderen. An approach to dynamic reconfiguration of distributed systems based on object-middleware. *Proceedings of the 19th Brazilian Symposium on Computer Networks (SBRC 2001)*, Santa Catarina, Brazil, May 2001
20. *"A Case Study: Using Workflow and AI Planners"* - Maria D. Rodriguez-Moreno, Paul Kearney and Daniel Meziat. PLANSIG 2000 UK Planning and Scheduling Special Interest Group Workshop.
21. "GOLEX - bridging the gap between logic (GOLOG) and a real robot"
    Dirk Hahnel, Wolfram Burgard and Gerhard Lakemeyer
    German AI conference, 1998
22. "The Third International Planning Competition: Temporal and Metric
    Planning" Maria Fox and Derek Long. University of Durham, UK
23. K. Moazami-Goudarzi. *Consistency preserving dynamic reconfiguration of distributed system*s. Ph.D. thesis, Imperial College, London, March 1999.
24. Stuart Russell and Peter Norvig, "Artificial Intelligence: A Modern Approach", Prentice Hall 1995.
25. Vere, S. 1983. Planning in time: Windows and durations for activities and goals. IEEE Trans. Pattern Anal. And Machine Intel. 5:246-267
26. Allen, J., and Koomen, J. 1983. Planning using a temporal world model. In Proc. 8th Intl Joint Conf. On Art. Intel., 741-747

27. Allen, J. 1991. Planning as temporal reasoning. In Proc. Conf. Knowledge Repr. And Reasoning, 3-14.

28. Penberthy, J. and Weld, D. 1994. Temporal planning with continious change. In Proc. 12th National Conference. Artificial Intelligence.

29. Muscettola, N. 1994. HSTS: integrating planning and scheduling. "Intelligent Scheduling". Morgan Kaufmann

30. Maria Fox and Alex Coddington "AIPS Workshop on Planning in Temporal Domains".

31. http://planet.dfki.de/service/Roadmap_on_AI_Planning_and_Scheduling/

32. http://planet.dfki.de/index.html

33. http://scom.hud.ac.uk/planet/ecp01_workshop/

34. http://scom.hud.ac.uk/planform/gipo/

35. http://www.dur.ac.uk/d.p.long/competition.html

36. Stefan Edelkamp, Malte Helmert The Model Checking Integrated Planning System AI-Magazine (AIMAG), Fall, 2001, pages 67-71

37. Doherty, P. and Kvarnström, J. (2001). TALplanner: A Temporal Logic Based Planner. Accepted for publication In *AI Magazine, Fall Issue*, 2001

38. Planning with Resources and Concurrency: A Forward Chaining Approach, F. Bacchus and M. Ady, *International Joint Conference on Artificial Intelligence (IJCAI-2001)*, pages 417-424, 2001.

39. AFRL Horizons, June 2001. http://www.afrlhorizons.com/Briefs/June01/IF0018.html

40. http://prometeo.ing.unibs.it/lpg/

41. Marco Castaldi, Antonio Carzaniga, Paola Inverardi, and Alexander L. Wolf: ”*A Lightweight Infrastructure for Reconfiguring Applications” Eleventh International Workshop on Software Configuration Management (SCM-11)*

Appendix A: JBI Domain

```
(define (domain jtmc3)
(:requirements :typing :durative-actions :fluents :conditional-effects)
(:types connector component machine - object)
(:predicates (connected-component ?c - component ?d - connector)
       (working-component ?c - component)
             (active ?d - connector)
             (dn-free ?d - connector)
             (interconnected ?d1 - connector ?d2 - connector)
             (ready ?d - connector)
             (killed ?d - connector)
             (up-machine ?m - machine)
             (at-machinec ?c - component ?m - machine)
             (at-machined ?d - connector ?m - machine)
             (component-is-connected ?c - component)
             (connector-started ?d - connector)
)

(:functions (start-time-component ?c - component)
                    (interconnect-time ?dn1 - connector ?dn2 - connector)
                    (connect-time-component ?c - component)
                    (start-dn-time ?d - connector)
                    (machine-up-time ?m - machine)
                    (machine-down-time ?m -machine)
                    (available-connection ?m - machine)
                    (local-connection-time ?m - machine)
                    (remote-connection-time ?m - machine)
                    (stop-time-component ?c - component)
                    (stop-time-connected-component ?c - component)
                    (stop-interconnected-dn-time ?d - connector)
                    (stop-dn-time ?d - connector)
)

(:durative-action connect-component
 :parameters (?c - component ?d - connector ?m - machine)
 :duration (= ?duration (remote-connection-time ?m))
 :condition (and
                    (at start (not (connected-component ?c ?d)))
                    (over all (working-component ?c))
                    (at start (not (killed ?d)))
                    (at start (active ?d))
                    (at start (ready ?d))
         )

 :effect (and       (at end (connected-component ?c ?d))
                    (at end (component-is-connected ?c))
                    )
)

(:durative-action local-connection-component
 :parameters (?c - component ?d - connector ?m - machine)
 :duration  (= ?duration (local-connection-time ?m))
 :condition (and
                    (at start (not (connected-component ?c ?d)))
                    (over all (working-component ?c))
                    (at start (not (killed ?d)))
                    (at start (active ?d))
                    (at start (ready ?d))
                    (over all (at-machinec ?c ?m))
          (over all (at-machined ?d ?m))
         )
 :effect (and           (at end (connected-component ?c ?d))
                        (at end (component-is-connected ?c))
                        )
)


(:durative-action start-component
 :parameters (?c - component ?m - machine)
 :duration (= ?duration (start-time-component ?c))
 :condition(
                        and
                        (at start (up-machine ?m))
                        (at start (not (working-component ?c)))
                        (over all (up-machine ?m))
                        (at start (> (available-connection ?m) 0))
                        )
 :effect (
                        and
                        (at end (working-component ?c))
                        (at end (decrease (available-connection ?m) 1))
                  (at end (at-machinec ?c  ?m))
            )
)

(:durative-action stop-component
 :parameters (?c - component ?m - machine)
 :duration (= ?duration (stop-time-component ?c))
 :condition(
                        and
                        (at start (at-machinec ?c ?m))
```

```
                (at start (working-component ?c))
                (over all (up-machine ?m))
                (over all (not (component-is-connected ?c)))
                )
 :effect (
                and
                (at end (not (working-component ?c)))
                (at end (increase (available-connection ?m) 1))
        )
)

(:durative-action stop-connected-component
 :parameters (?c - component ?m - machine ?d - connector)
 :duration (= ?duration (stop-time-connected-component ?c))
 :condition(
                and
                (at start (at-machinec ?c ?m))
                (at start (working-component ?c))
                (over all (up-machine ?m))
                (at start (connected-component ?c ?d))
                )
 :effect (
                and
                (at end (not (working-component ?c)))
                (at end (increase (available-connection ?m) 1))

                (at end (not (connected-component ?c ?d)))
        )
)

(:durative-action start-connector
 :parameters (?d - connector ?m - machine)
 :duration (= ?duration (start-dn-time ?d))
 :condition (
                and
                 (at start (not (active ?d)))
                 (at start (not (killed ?d)))
                 (over all (up-machine ?m))
                (at start (not (connector-started ?d)))
                )
 :effect     (
                and
                (at end (active ?d))
                (at end (decrease (available-connection ?m) 1))
                (at end (at-machined ?d  ?m))
                )
)

(:durative-action stop-connector
 :parameters (?d - connector ?m - machine)
 :duration (= ?duration (stop-dn-time ?d))
 :condition (
                and
                 (at start (active ?d))
                 (at start (not (killed ?d)))
                 (over all (up-machine ?m))
      (at start (at-machined ?d  ?m))
                )
 :effect     (
                and
                (at end (not (active ?d)))
                (at end (increase (available-connection ?m) 1))
```

```
                                )
)

(:durative-action start-machine
 :parameters (?m - machine)
 :duration (= ?duration (machine-up-time ?m))
 :condition (and
                                (at start (not (up-machine ?m)))
                        )
 :effect       (and
                                (at end (up-machine ?m))
                                )
)

(:durative-action stop-machine
 :parameters (?m - machine)
 :duration (= ?duration (machine-down-time ?m))
 :condition (and
                                (at start (up-machine ?m))
                        )
 :effect       (and
                                (at end (not (up-machine ?m)))
                                )
)

(:durative-action dn-interconnection
 :parameters (?d1 - connector ?d2 - connector)
 :duration(= ?duration (interconnect-time))
 :condition (
                                and
                                        (at start (active ?d1))
                                        (at start (not (killed ?d1)))
                                        (at start (not (killed ?d2)))
                                        (at start (active ?d2))
                                        (at start (not (ready ?d1)))
                                        (at start (not (ready ?d2)))



)

:effect (and
                                (at end (interconnected ?d1 ?d2))
                                (at end (interconnected ?d2 ?d1))
                          (at end(ready ?d1))
                          (at end(ready ?d2))
                        )
)
)
```

Appendix B: Implicit Configuration

```
(define ( problem TFGUH )
(:domain jtmc3 )
(:objects
        TWLYBJSEN0 - component
        HFGTBXO1 - component
        SKENYGM2 - component
        PTU0 - connector
        EGK1 - connector
        DOLOUE0 - machine
        LTXYRY1 - machine
)
(:init
        (= (start-time-component TWLYBJSEN0) 27.0)
        (= (connect-time-component TWLYBJSEN0) 19.0)
        (= (start-time-component HFGTBXO1) 27.0)
        (= (connect-time-component HFGTBXO1) 19.0)
        (= (start-time-component SKENYGM2) 21.0)
        (= (connect-time-component SKENYGM2) 14.0)
        (= (start-dn-time PTU0) 28.0)
        (= (start-dn-time EGK1) 28.0)
        (up-machine DOLOUE0)
        (= (available-connection DOLOUE0) 12)
        (= (local-connection-time DOLOUE0) 11.0)
        (= (remote-connection-time DOLOUE0) 64.0)
        (= (machine-up-time LTXYRY1) 14.0)
        (= (available-connection LTXYRY1) 9)
        (= (local-connection-time LTXYRY1) 14.0)
        (= (remote-connection-time LTXYRY1) 64.0)
)
(:goal
(and
        (component-is-connected TWLYBJSEN0)
        (component-is-connected HFGTBXO1)
        (component-is-connected SKENYGM2)
        (ready PTU0)
        (ready EGK1)
)
)
)
```

Appendix C: A Sample Plan

```
;; Version LPG-v1.1

;; Seed 65675280

;; Command line: ./lpg -o jtmc3.pddl -f TFGUH.pddl -n 5 -out TFGUH -cputime 20


;;Problem: TFGUH.pddl

;;Search time: 0.050    Parsing time: 0.050    Mutex time: 0.000

;;Actions: 10    Execution cost: 10.000    Duration: 154.000


        0.001:   (START-MACHINE LTXYRY1)[14.000] ;; cost 1.000

        14.002:   (START-CONNECTOR PTU0 LTXYRY1)[28.000] ;; cost 1.000

        14.003:   (START-CONNECTOR EGK1 LTXYRY1)[28.000] ;; cost 1.000

        42.004:   (START-COMPONENT TWLYBJSEN0 LTXYRY1)[27.000] ;; cost 1.000

        69.006:   (START-COMPONENT SKENYGM2 LTXYRY1)[21.000] ;; cost 1.000

        90.007:   (START-COMPONENT HFGTBXO1 LTXYRY1)[27.000] ;; cost 1.000

        69.008:   (CONNECT-COMPONENT TWLYBJSEN0 EGK1 LTXYRY1)[64.000] ;; cost 1.000

        117.009:   (LOCAL-CONNECTION-COMPONENT HFGTBXO1 PTU0 LTXYRY1)[14.000] ;; cost 1.000

        90.010:   (CONNECT-COMPONENT SKENYGM2 EGK1 LTXYRY1)[64.000] ;; cost 1.000


Time 200
```

Appendix D: Explicit Configuration

```
(define ( problem TFGUH )
(:domain jtmc3 )
(:objects
        TWLYBJSEN0 - component
        HFGTBXO1 - component
        SKENYGM2 - component
        PTU0 - connector
        EGK1 - connector
        DOLOUE0 - machine
        LTXYRY1 - machine
)
(:init
        (= (start-time-component TWLYBJSEN0) 27.0)
        (= (connect-time-component TWLYBJSEN0) 19.0)
        (= (start-time-component HFGTBXO1) 27.0)
        (= (connect-time-component HFGTBXO1) 19.0)
        (= (start-time-component SKENYGM2) 21.0)
        (= (connect-time-component SKENYGM2) 14.0)
        (= (start-dn-time PTU0) 28.0)
        (= (start-dn-time EGK1) 28.0)
        (up-machine DOLOUE0)
        (= (available-connection DOLOUE0) 12)
        (= (local-connection-time DOLOUE0) 11.0)
        (= (remote-connection-time DOLOUE0) 64.0)
        (= (machine-up-time LTXYRY1) 14.0)
        (= (available-connection LTXYRY1) 9)
        (= (local-connection-time LTXYRY1) 14.0)
        (= (remote-connection-time LTXYRY1) 64.0)
)
(:goal
(and
        (connected-component TWLYBJSEN0 PTU0)
        (connected-component HFGTBXO1 EGK1)
        (connected-component SKENYGM2 PTU0)


)
)
    )
```