Supporting IT Service Fault Recovery with an Automated Planning Method

Dissertation

an der

Fakultät für Mathematik, Informatik und Statistik der Ludwig-Maximilians-Universität München

vorgelegt von

Feng Liu

Tag der Einreichung: 21.04.2011

Supporting IT Service Fault Recovery with an Automated Planning Method

Dissertation

an der

Fakultät für Mathematik, Informatik und Statistik der Ludwig-Maximilians-Universität München

vorgelegt von

Feng Liu

Tag der Einreichung:	21.04.2011
Tag der mündlichen Prüfung:	06.06.2011
1. Berichterstatter:	**Prof. Dr. Heinz-Gerd Hegering** Ludwig-Maximilians-Universitiät München
2. Berichterstatter:	**Prof. Dr. Gabi Dreo Rodosek** Universität der Bundeswehr München

Acknowledgement

This dissertation not only presents the research results that I have achieved while working as a researcher at the chair of Prof. Dr. Dieter Kranzlmüller and Prof. Dr. Heinz-Gerd Hegering, it is also an evidence of friendship and selfless help I got from the MNM Team during the completion of this work.

First and foremost, I'd like to express my heartfelt appreciation to my thesis advisor Prof. Dr. Heinz-Gerd Hegering for his committed guidance and kind advice during all phases of my work. As his doctoral student, I have deeply benefited from his knowledge, deep-insight and decades-long experience in the research area of network management. Also I would like to thank Prof. Dr. Gabi Dreo for the helpful discussions and important advice during the preparation of this work. I'd like to thank Prof. Dr. Kranzlmüller for his support during the final phase of my research.

As a member of the MNM Team, I have been privileged to get countless help, suggestions and support from all team members. Especially I want to thank Dr. Vitalian Danciu and Dr. Michael Schiffers, whose suggestions and advice helped me greatly in my research. I am indebted to you all, my friends! Thank you for the experience and memory that I will cherish for always.

I am grateful to my parents for their encouragement and their confidence in me. I am also indebted to my wife and my daughter for their patience and understanding during all phases of my work. Compared to what you have done for me, a simple "thank you" seems to be pale to express my gratitude.

Feng Liu
Munich, July 2011

Abstract

Despite advances in software and hardware technologies, faults are still inevitable in a highly-dependent, human-engineered and administrated IT environment. Given the critical role of IT services today, it is imperative that faults, having once occurred, have to be dealt with efficiently and effectively to avoid or reduce the actual losses. Nevertheless, the complexities of current IT services, e.g., with regard to their scales, heterogeneity and highly dynamic infrastructures, make the recovery operation a challenging task for operators. Such complexities will eventually outgrow the human capability to manage them. Such difficulty is augmented by the fact that there are few well-devised methods available to support fault recovery.

To tackle this issue, this thesis aims at providing a computer-aided approach to assist operators with fault recovery planning and, consequently, to increase the efficiency of recovery activities. We propose a generic framework based on the automated planning theory to generate plans for recoveries of IT services. At the heart of the framework is a planning component. Assisted by the other participants in the framework, the planning component aggregates the relevant information and computes recovery steps accordingly. The main idea behind the planning component is to sustain the planning operations with automated planning techniques, which is one of the research fields of artificial intelligence. Provided with a general planning model, we show theoretically that the service fault recovery problem can be indeed solved by automated planning techniques. The relationship between a planning problem and a fault recovery problem is shown by means of reduction between these problems. After an extensive investigation, we choose a planning paradigm that based on Hierarchical Task Networks (HTN) as the guideline for the design of our main planning algorithm called H^2MAP.

To sustain the operation of the planner, a set of components revolving around the planning component is provided. These components are responsible for tasks such as translation between different knowledge formats, persistent storage of planning knowledge and communication with external systems. To ensure flexibility in our design, we apply different design patterns for the components. We sketch and discuss the technical aspects of implementations of the core components. Finally, as proof of the concept, the framework is instantiated to two distinguishing application scenarios.

Zusammenfassung

Trotz zahlreicher Fortschritte in Software- und Hardware-Technologien, bilden Fehlersituationen nach wie vor einen festen Bestandteil des IT-Betriebs. Angesichts der Wichtigkeit von IT-Diensten, ist es unerlässlich, die durch Fehler verursachten Störungen unverzöglich zu beheben, damit die betroffenen Dienste die erwarteten Funktionalitäten ohne erhebliche Unterbrechungen weiter liefern können. Angesichts der Komplexität, Dynamik und Heterogenität moderner IT Dienst-Landschaften, stellt die Planung der Wiederherstellung fehlerhafter Dienste eine große Management-Herausforderungen dar. Zudem gibt es derzeit kaum wissenschaftlich fundierte Methoden oder Werkzeuge, Administratoren bei der Wiederherstellung von Diensten zu unterstützen.

Diese Arbeit liefert einen Beitrag zum Schließen dieser Lücke durch den Vorschlag und die prototypische Implementierung eines generischen Frameworks für das automatische Planen von Wiederherstellungsprozessen für IT Dienste. Das Herzstück des Frameworks bildet eine Komponente, die in der Lage ist, einen Wiederherstellungsplan anhand der verfügbaren Informationen (*sog. Recovery Knowledge*) automatisch zusammenzustellen. Dazu werden Techniken aus dem *Automated Planning*, einer Forschungsrichtung der künstlichen Intelligenz, verwendet. In einem allgemeinen Planungsmodell wird gezeigt, dass das Dienst-Wiederherstellungsproblem in ein Planungsproblem transformiert werden kann. Damit können Techniken, die für das Lösen von allgemeinen Planungsproblemen adquat sind, auf die Planung von Dienst-Wiederherstellungen anwendet werden. Wir entwerfen einen Planungsalgorithmus, der auf dem *Hierarchical Task Networks (HTN)* Planungsparadigma basiert. Im Vergleich zu anderen Paradigmen, ist das HTN-basierte Modell flexibler und effizienter.

Um den Einsatz der Planungskomponente zu unterstützen, besitzt das Framework eine Reihe weiterer Komponenten, die unter anderem dafür verantwortlich sind, Planungswissen abzuspeichern, aktuelle Informationen bzgl. der Infrastrukturen und Dienste zu akquirieren oder Informationsformate zu transformieren. Die Planungskomponente kooperiert mit diesen Komponenten, um die Planungsoperationen durchzuführen bzw. die Lösungspläne auf den Zielsystemen auszuführen.

Um die Flexibilität zu gewährleisten, verwenden wir im Frameworkdesign spezifische Entwurfsmuster, damit Änderungen der Planungsumgebung nur geringe Modifikationen nach sich ziehen. Zum Abschluss zeigen wir die Tragfähigkeit und Einsetzbarkeit des Frameworks in zwei unterschiedlichen Anwendungsbeispielen.

Contents

1 Introduction

Contents

Recent advances in the research and development of information technology (IT) have made deep social impacts. Society increasingly relies on IT services to exchange information, foster research and education as well as support commercial activities. From the perspective of IT service providers, this trend is something of a double-edged sword: on one hand, it opens up new business opportunities by offering novel services; on the other hand, to sustain the increasing number of offered services, operators have to struggle with intricate management problems in the highly complex IT environment.

Meanwhile, the lucrative IT service market has attracted many IT service providers competing for market share. This furious competitions compels providers to offer better service quality at a competitive cost in order to gain an edge over competition. They must find ways to reduce management costs without seriously impairing service quality. Finding balanced solutions is difficult. Such difficulty is furthermore augmented by the following aspects:

1. *Varieties of Services.* IT services today span a wide range of areas. Those services require rather complex infrastructure to sustain their operations. Consequently managing such diversities in highly heterogeneous IT environments poses challenges for IT personnels. Moreover, the requirements on "classical" IT services have been evolving over time, for example, a customer, who was once satisfied with simple web-hosting services to provide static content, may now need extra application servers to provide business logic and database servers as back-end to provide dynamic data services.

2. *Management Complexity.* Considering the rapid growth of complexities of IT services and the associated infrastructure, even the most-skilled IT staffs find it difficult to cope with the growing complexities [KC03, Kep05] and those complexities will eventually outgrow human capabilities to manage them. Additionally, the knowledge required to manage IT services and underlying infrastructure are very demanding and becoming highly specialized; this poses an extra barrier for IT personnel of large data centres.

3. *Consolidation of Infrastructure.* To cut the total cost of ownership (TCO), the current trend of consolidation of infrastructure urges IT providers to take their infrastructure under the central organisation and co-locate them in fewer facilities. Emerging technologies such as virtualisation and Grid computing pose new problems to all aspects of management. Consolidation of infrastructure leads to a high-density of services, which have to be managed with limited resources.

4. *Critical Role of IT.* IT is becoming a critical component of business models. It is the common belief now that the success of a business largely relies on how good its IT operates. This claim has two implications: first, the non-functioning of IT could cause a business catastrophic losses, both financially and in terms of reputation. For example, a 22-hour outage by Ebay^{TM}caused an estimated $3-million to $5-million loss in revenue and its stock dropped 8.63 on the news of outages on that day and a 1-hour outage of a brokerage system may cause around $6.5 million [Dav02]; second, if faults inevitably happen, a quick reaction is essential to keep the losses to a minimum.

Despite the advances of technology, service failures are still inevitable in a highly inter-dependent, human-engineered and administrated IT environment [FP02, Her02]. Human errors, false configurations, software/hardware defects or power outages [Bre01], to name just a few, are all possible causes of service failures. Given the critical role of IT today, it is imperative that faults, having once occurred, have to be dealt with both efficiently and effectively.

Fault management encompasses detection, isolation, diagnosis and recovery [HAN99]. The main objectives of fault management are to detect and recover faults quickly and to reduce the impact of a fault. Recovery, as the last phase of fault management, means eliminations of diagnosed faults using technical means and reinstatement of the impacted service, so that it can continue to deliver the expected functionalities.

Issues which make management of today's IT services difficult obviously have impacts on the fault recovery process. The challenges of fault recovery in a modern, large-scale IT environment are specified as follows:

- High fault rate. Significant number of services and their associated components can lead to a high fault rate. Components can be impacted by faults at anytime. The fact that some service providers (such as Google™) base their services mostly on off-the-shelf hardware makes this problem obvious. Furthermore, in today's data centres, a large number of services are run by a limited number of operators (the operator to component proportion at Google™1:500). To recover from multiple faults efficiently, operators need to plan as well as schedule their recovery operations accordingly, and automate the recovery process where possible.

- Intricate dependencies exacerbate the difficulty to perform fault recovery. Constantly evolving infrastructure is common and introduces new dependencies at different levels. Adoption of new hardware, software and constant updates to services are the main reasons for an evolving service infrastructure. Whereas knowledge on dependency is crucial, the fault recovery operations have to consider the changes in dependencies between services and components.

- High demand on the efficiency of recovery. The critical role of IT service for businesses requires that faults, once they have occurred, be efficiently recovered to avoid or to reduce the actual losses. This requires, besides the efficient fault detection and diagnosis, quick determinations of viable recovery solutions which could effectively reinstate the impacted services.

- Different service classes require diverse recovery prioritisations. The prioritisation of recovery operations is especially important in cases of multiple service faults, which is not unusual as discussed previously (refer to the discussion on *high fault rate*); in such cases, recovery actions must be duly planned and scheduled.

- Volatile infrastructure state during the recovery process. In a large IT environment where multiple services are hosted, the state of infrastructure is changing dynamically, even during the recovery. For

example, additional faults could appear and invalidate the original recovery strategy; in such cases, the recovery process need to be adapted dynamically to the changing states, so that recovery can be carried out consistently.

- Human errors during the recovery become obvious. A study [BP01] showed that operator-related errors accounted for about 75% of total observed faults in IT and operators tend to make more errors in stressful situations such as recovery services with time constrain.

- High requirement on the management knowledge. Fault recovery is a knowledge-intensive operation; it requires the integration of various management knowledge in order to produce a viable plan. Unfortunately such knowledge is currently available in a distributed and unorganized manner. In case of fault, considerable time is needed to collect, process and integrate this informations, which unnecessarily prolongs the recovery period, thus negatively affecting system availability. With this fact recognised, it is clear that one of the major challenges regarding fault recovery is the large number of non-uniform information or knowledge sources contributing to the recovery process. The Information Technology Infrastructure Library (ITIL) [Com07] explicitly addresses the importance of management knowledge and proposes an integrated Service Knowledge Management System with a layered architecture for the storage, integration, processing and presentation of management information.

In conclusion, the complexity of current IT services makes the recovery operation a challenging task for operators. Despite its importance, approaches for service fault recovery have not yet been thoroughly addressed, compared to other management disciplines. To fill this gap, this thesis addresses the challenges regarding automated fault recovery. Based on the automated planning theory [NGT04, LaV07], we investigate using automated planning approaches to solve service fault recovery problems. In the following section, the problem statement and specific research questions are presented more detail.

1.1 Research Problem and Vision of Solution

Motivated by the aforementioned management challenges, in this section we present and discuss the main research problem and our vision on the solution. To tackle the presented problem, we divide it into several specific research questions. We illustrate and sketch our visionary solution to provide an overview.

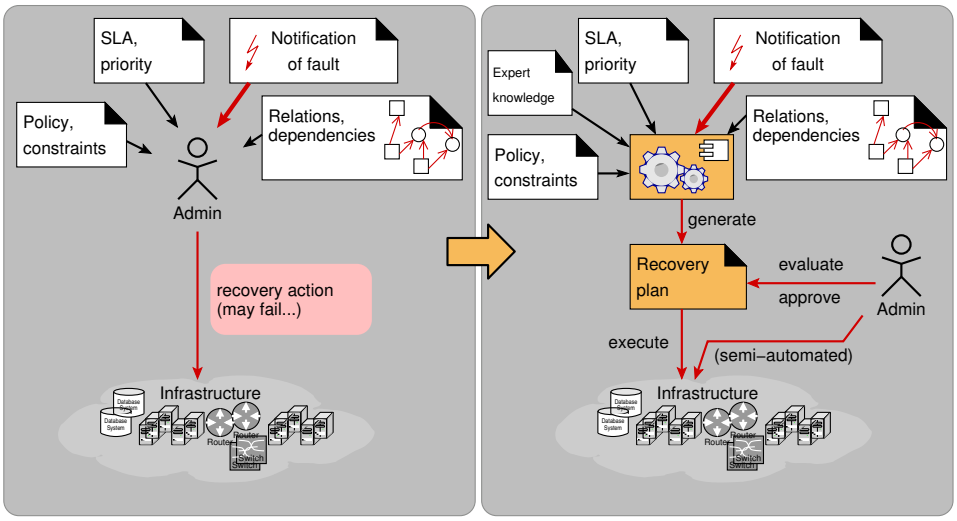


Figure 1.1: The Vision of the Proposed Approach

1.1.1 Problem Statement

The main problem statement of this thesis is articulated and summarised as follows:

> Having recognised that faults are inevitable during the operation of IT services, growing complexities and highly dynamic infrastructure make the planning for a service fault recovery an intricate task for operators [BP01]. Such difficulty is augmented by the fact that there are few well-devised methods available to support such activities. Additionally, studies [Bre01, DO03, Kuh97, DO02, BP01] showed that human error occuring during the fault recovery is one of the major factors contributing to an increased Mean-Time-to-Repair (MTTR) value. They identified that the majority of such faults are caused by unsystematic and unplanned recovery activities. *The main research issue concerning this work is therefore to investigate how to provide a computer-aided approach to assist operators with fault recovery planning and scheduling and, consequently, to increase the efficiency of fault recovery activities.*

1.1.2 Vision

Figure 1.1 illustrates an envisioned solution of this work. The left half of the figure shows a traditional approach where the administrative overhead regarding fault recovery completely falls back on human operators, which

make them at the centre of recovery processes. In order to find a feasible plan, he has to aggregate and process the relevant recovery knowledge. This time-consuming manual approach is inefficient and error-prone in a highly complex IT environment.

The right half of the figure depicts the main idea of this research, which is *to develop a framework based on the automated planning theory to generate recovery plans for of IT services failures.* This framework supports service operators regarding the fault recovery process by addressing the following issues:

- Modelling and representation of recovery knowledge.
- Integration of recovery knowledge into plan reasoning processes.
- Providing an automated approach to process recovery knowledge (i.e. translation between different formats, storage and retrieval of knowledge etc.).
- Facilitating reuse of management knowledge.

Instead of completely relying on the human operators, a planning framework provides a computer-aided mechanism which aggregates the relevant information and computes recovery plans. Based on the computed solutions, the operators could verify and selects suitable solutions as needed. With this approach, the overhead of planning the recovery solution is greatly reduced. Additionally, the finding of plan solutions can be performed much faster than by a purely manual approach.

Additionally, this research explores the strengths and weaknesses of the proposed method in the area of service recovery planning. Whereas fault recovery is an integral part of the fault management process, this framework also maintains interfaces to other fault management processes such as fault diagnosis.

Figure 1.2 provides an overview of the envisioned recovery process. As soon as the recovery process is initiated, a plan generation component is triggered to compute recovery plans for the reported fault. To achieve this, it needs to correlate information from different sources, for example, the dependency and current state of the impacted service, constraints on time and cost of recovery as well as available recovery options. However, it could not be guaranteed that a plan can always be found by the planning component. In such cases, recovery planning should be either retried or fall back on operators to make decisions. If a plan is successfully found, it needs to be evaluated and checked before execution, for example, by means of reviewing or editing by operators. The successfully evaluated plan will be then executed to reinstate the impacted service into operational state. Finally the result from recovery needs to be evaluated to see if the recovery goal is met.

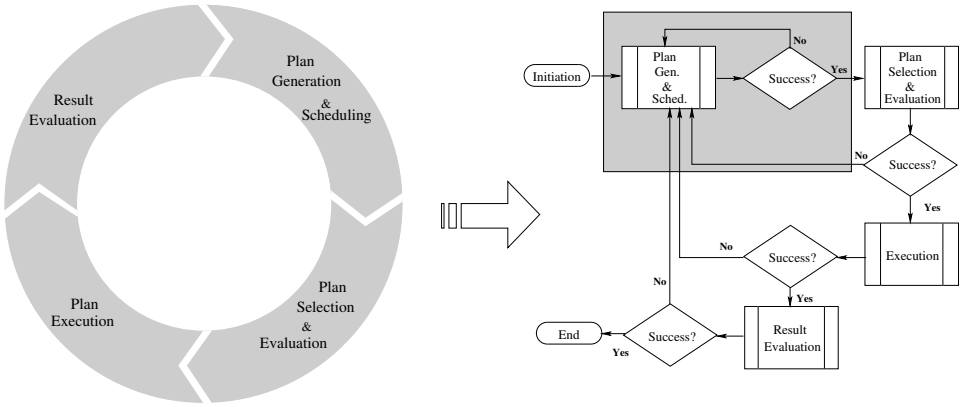


Figure 1.2: Overview of the recovery process

Nevertheless, during the plan evaluation, execution and result evaluation phases, exceptions also need to be considered and handled, an example of such an exception could be a new fault during fault recovery, which transits the state of the impacted service to a new fault state that cannot be treated by the original recovery plan.

Based on the illustrated vision, the main research focus would be on the plan generation and scheduling component with interfaces to other components. It shall not only possess the capability to automatically generate recovery plans and schedules, but also handle exceptions during the different phases and be responsible to dynamically adapt the recovery plan.

The benefits of this research are manifold: on one hand, it supports the service operator in dealing with complex recovery cases by automating the composition of recovery plans and schedules; on the other hand, using a systematic recovery planning method can increase the overall efficiency of the recovery process and help the operator to carry out the recovery operations in a more systematic way. Finally, the focus of this research provides a novel way to service recovery planning based on formal and well-established methods.

1.1.3 Research Questions

The envisioned solution to the main research problem can be broken down into the following specific research questions that need to be addressed.

Q1: *How can a recovery plan, recovery actions and applicable recovery constraints be modelled?*

Recovery actions are fundamental elements that a recovery plan consists of. In order to compose a viable recovery plan, proper actions

need to be selected based on their properties such as the time requirement, costs and pre-conditions etc. These properties need to be identified and considered in the model of recovery action.

A recovery plan consists of a sequence of recovery actions and it is characterised through its attributes such as the aggregated time of the plan. The identification of these attributes will directly contribute to the plan selection if, for example, multiple recovery plans with the same recovery objective but different time requirements exist. It is, however, necessary to point out that elicitation of the knowledge on recovery actions is not within the scope of this dissertation; however, there is wealth of work that has been done in the area on knowledge engineering research.

Recovery constraints are conditions that must be considered by the plan compositions, for example, the maximum recovery time for a particular service or the maximum costs of the recovery activity for that service. This information needs to be properly identified and modelled. Meanwhile, it is important to investigate where such constraints could be derived from.

Q2: *How could actions and constraints be properly stored so that the planning and scheduling mechanism can query that informations when necessary?*

In addition to the plan knowledge modelling, recovery actions need to be properly stored as well, so that planning and scheduling methods could query those actions when needed. A proper repository needs to be provided to facilitate querying and storing operations; it should act as a knowledge base for the automated planning operation.

Q3: *How could plans be generated automatically? Is there any existing approach from other research fields we could learn from?*

Automated plan generation is one of the main research issues this thesis investigates; there are some existing approaches from research areas such as automated planning, which are concerned with automated plan compositions. Automated planning is a branch of Artificial Intelligence (AI) research. Unlike other strong AI approaches, which try to give systems "intelligence", automated planning methods are based on solid algorithms, which have been successfully applied in different areas in industry and research. The purpose of automated planning methods is to provide knowledge-integration and decision support to complex planning processes. In the context of IT management, several contributions exist which investigate into applying automated planning in change management [BS05] and configuration management [NA07], but hardly any work exists on fault recovery planning.

Nevertheless automated planning methods in their original forms could not be directly applied to the service fault recovery; modifications and adaptations are necessary. Moreover, different plan generation methods are associated with different complexities and have their own constraints and limitations; choosing a proper approach requires an in-depth analysis of existing algorithms. However, automated planning methods has been successfully applied to application areas such as space exploration program, military operation planning and industrial production planning, to name just a few; the experience gained from these applications is helpful and indispensable for this research.

Q4: *How could plans dynamically be adapted to a new service state?*

In a real-world scenario, successful executions of a particular recovery action cannot always be assumed, which means it is possible that executions of certain steps in a plan failed (for example, because of instant changes of infrastructure). In such cases, the planning mechanism has to provide the possibility to adapt the original plan dynamically to the new service state. To achieve this goal, the results of the plan execution need to be fed back to the planning component in a real-time manner.

Q5: *How can the correctness of a generated recovery plan be automatically verified?*

A recovery plan needs to be verified to ensure its correctness before it can be executed by a human administrator or by an underlying system. Since a manual verification of a complex recovery plan is an arduous and error-prone process, an automated support is needed to check the correctness of the generated plan. In this context, the correctness could mean whether the generated plan fulfils the desired management objectives or whether the costs (either time or money) of the computed plan results are in alignment with corresponding constraints.

Q6: *How can the relevant management knowledge be integrated into planning process?*

Management knowledge is indispensable for the recovery plan composition. Such information exists in different sources, examples are policy repositories and monitoring databases. The planning process needs to integrate such knowledge during the planning process. Additionally, such knowledge must be converted into a format which the automated planning process can understand.

1.2 Scope and Alignment

This work touches upon three different research domains: automated computing, IT service management and system dependability. This section provides an overview of how this thesis is related to these research areas.

1.2.1 Autonomic Computing

The main objective of autonomic computing [KC03, Kep05, GC03] is to equip systems with self-managing capabilities to cope with complex management tasks that operators are currently facing.

The IBM-proposed K-MAPE model of autonomic management consists of four key functional phases: *monitoring, analyzing, planning and executing*. An additional knowledge base is presented in this model to support those key functions. The monitoring function is designed to supervise the current system and collect data regarding operational states as well as performance indices of the system. These data are subsequently analysed by an analysis function based on the existing knowledge. The planning function composes a plan according to the results of previous analysis and plans actions to achieve the desired management objectives. The planning function relies on knowledge that is available in a knowledge base. Such knowledge could encompass actions that are required in order to reach the determined management goals. Additionally, the autonomic computing includes four aspects of self-management. They are briefly explained below:

Self-Configuration Automated configuration of systems follows high-level policies representing management goals.

Self-Optimization Systems are continually seeking opportunities to improve their own performance and efficiency.

Self-Healing System automatically detects, diagnoses and repairs localized hardware and software problems.

Self-Protection System automatically defends against malicious attacks or cascading failures.

This research is related to the planning phase and knowledge bases of the autonomic computing model. It also addresses the research issues regarding the self-healing aspect of autonomic management. This work is applicable to the self-healing aspect in a way that it investigates issues regarding how to automatically compose a recovery plan based on the existing knowledge such as a dependency model, available recovery actions etc. Additionally, this thesis proposes using the automated planning methods to facilitate the planning function suggested by the aforementioned model.

1.2.2 IT Service Management

Figure 1.3 shows the scope of this thesis from an IT service management perspective. As mentioned above, this work falls into the area of fault management, which consists of fault detection, fault diagnosis, fault isolation and fault recovery. Within the fault management dimension, the focus of this work lies on recovery, particularly on the planning of recovery solutions. The suggested approach intends to facilitate recovery planning by means of automated correlation and reasoning on fault repair actions, configuration information of resources etc.[Rod02].

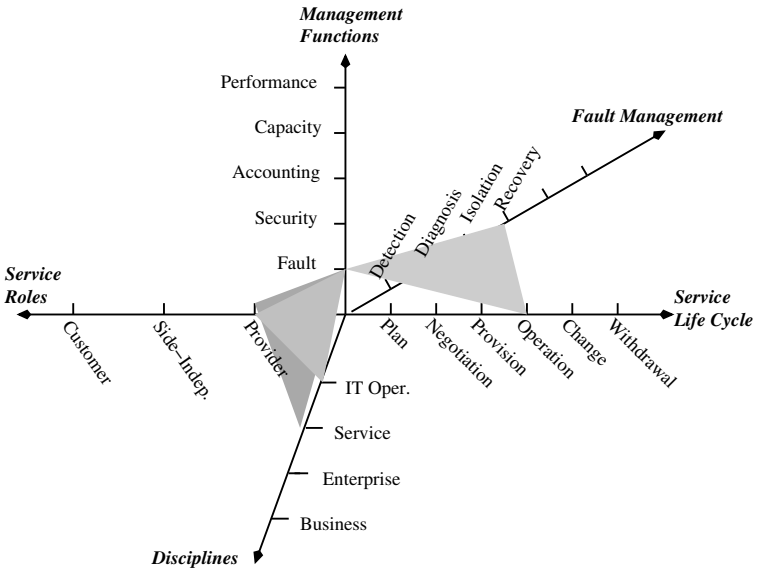


Figure 1.3: Scope regarding IT Service Management

An IT service has its lifecycle, which starts with planning of a service and ends with the withdrawal of that service. This work concentrates on the operational phase of an IT service, during which the correctness of a running service is monitored and controlled. As fault is an inevitable fact, an efficient fault recovery process during the operational phase is especially important for a service provider because during this phase, customers are actually using that service, and once impacted by fault, the efficiency of the fault recovery process will directly influence the customers' perceived service availability as well as swiftness of the provider's reaction to the fault.

Disciplines of IT management are classified into several levels. As shown in Figure 1.3, the business management deals with the management of a company as a whole, which involves tasks such as financial management and strategic planning, whereas enterprise management mainly deals with concreting and specifying abstract business goals. The management on the

operation level handles management tasks regarding systems, network and applications, i.e. it deals with management of networks and end systems. The objective of service management is to fill the gap between managing resources (also sub-services) with respect to the business objectives of the organisation. The research focus of this thesis with respect to management disciplines lies on the recovery planning from a service perspective. In the meantime, the recovery of a service will finally boil down to the recovery actions on the operation level, where concrete recovery actions are being performed.

According to the MNM service model [GHH+02], a service consists of customer domain and provider domain. The user and customer access a particular service through the side independent part of the model. Details of the MNM service model are provided in Chapter 2 of this thesis. This work concentrates on the recovery of those faults, whose root causes are in the provider's domain, i.e. user-perceived service faults not related to the service provider are not within the scope of this work, for example, a configuration error of a user's email client may give that user the impression that email service is not available, such a fault is confined to the customer domain of a service.

1.2.3 Dependability Research

Figure 1.4 shows how this work is related to the dependability research. Definitions on dependability in this work are rely on concepts and taxonomy suggested by Laprie et. al. [LR04]. Dependability is characterised by five attributes: *availability, reliability, safety, confidentiality, integrity and maintainability*. This work addresses availability and maintainability aspects. Availability means the readiness of a correct service and it is determined by Mean-Time-to-Failure(MTTF) and Mean-Time-to-Repair(MTTR). An efficient recovery process directly contributes to minimising the MTTR value therefore increases the availability of a service.

There are different means to achieve dependability. Fault prevention and fault forecasting are counted as proactive means to prevent faults from happening and using statistical methods to predicate faults that might happen in future. This work, however, focuses on the reactive means to achieve dependability through fault tolerance and fault removal. Fault tolerance is carried out through error detection and system recovery. The system recovery transforms an erroneous system state into an operational state. Fault removal means to eliminate current faults or reduce the severity of faults, which is directly related to the efficient recovery process.

The maintenance attribute includes repair and modification of the system that take place during the operational phase. It encompasses various forms: corrective maintenance, preventive maintenance, adaptive maintenance and augmentative maintenance. This work is related to corrective

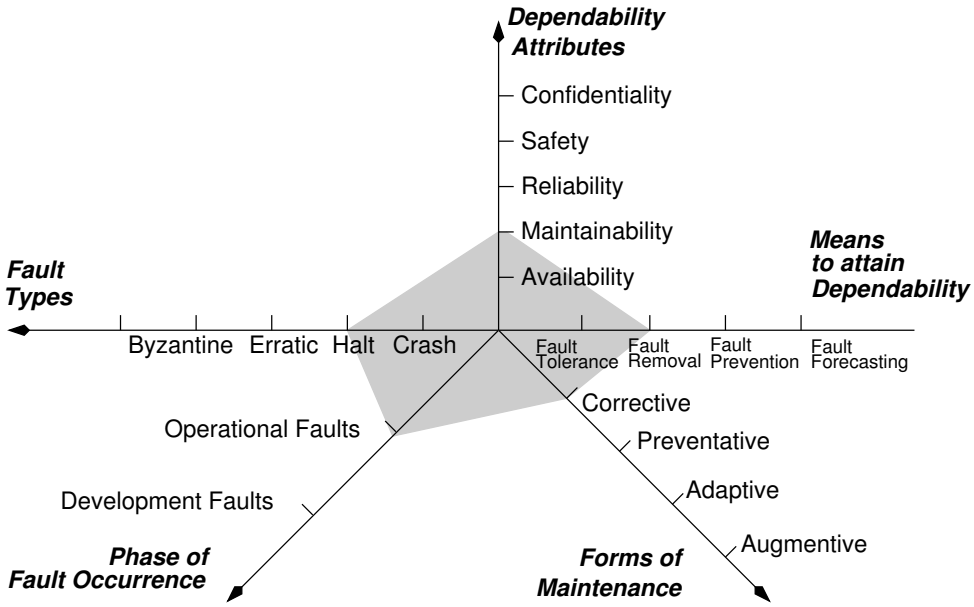


Figure 1.4: Scope regarding Dependability Research

maintenance, which means to remove reported faults.

Regarding the phases of fault occurrence, this work is related to operational faults, which occur during the delivery of a service. The development faults during the development phase are not covered in this work.

1.3 Delimitations to Other Dissertations

The theses of Sailer [Sai07], Hanemann [Han07] and Schmitz [Sch08] are closely related to this work. Sailer addressed the construction of a management information base that includes all information needed regarding technical service management. Information such as dependencies of services is crucial for service recovery planning and could be directly used by the approaches suggested in this thesis. The work of Hanemann mainly addressed the issue of fault diagnosis which is an important part of fault management. His approach is based on service-oriented event correlations which are based on case-based reasoning and rule-based reasoning. The list of identified root causes of faults could be used as inputs for the approach suggested in this thesis. The thesis from Schmitz mainly addressed the issues on the analysis of actual or assumed resource failures and determine their impacts on the services regarding the agreed SLA. In his suggested framework, he also proposed a decision aid to determine the appropriate recovery action as well as a recovery tracking mechanism. Nevertheless the framework does not

provide a proper plan composition mechanism. Despite open questions, his work could be regarded as a foundation and a reference on which to build a more concrete recovery planning approach.

In her post-doctoral thesis, Dreo [Rod02] provided a generic framework for IT service management based on the FCAPS management functional areas. The present thesis is related to the fault management part, specifically the fault recovery planning and formalisation as well as deployment of knowledge base for fault recovery, of her post-doctoral work.

1.4 Thesis Outline

Figure 1.5 illustrates the structure of this thesis. The dashed-line in Figure 1.5 show the relationships between topics *between* chapters and the solid-line connections describe relationships *within* the specific chapter.

Chapter 1 introduces the current IT management challenges and consequent research problems regarding the service fault recovery. A research scope is provided to offer an overview of the research areas, to which this dissertation is related.

Chapter 2 starts with specific definitions and explanations of the terminology used throughout this work in order to avoid ambiguities. Then scenarios regarding fault recovery are investigated with the intention to motivate and structure the requirements for the solution. Finally a complete list of requirements for our solution is derived and discussed.

Chapter 3 gives an overview of the related work. It starts with a general introduction of automated planning research with its theory and applications. We then present and analyse the existing competitive approaches on fault management with a focus on the fault recovery aspect. The investigated approaches are evaluated against the requirements derived from the previous chapter.

Chapter 4 provides our design of a service fault recovery planning framework. To justify the feasibility of leveraging the automated planning theory to solve the recovery problem, we build a model for fault recovery and draw the equivalence between the two problem models. We show that with a proper transformation, a fault recovery problem can be transformed into a general planning model, thus to be solved with automated planning method. With this theoretical background in place, we investigate different planning paradigms and propose a design of the core planning algorithm based on the HTN planning paradigm. We also specify a language to encode planning knowledge, so that it could be used by the planner to compute solutions. In the final part of this chapter, we propose a flexible framework design with components revolving around the planning subsystems.

Chapter 5 addresses implementation aspects of the core components in the framework. As the proof of concept, we demonstrate the applicability

of our approach to selected applications scenarios. Finally we evaluate the framework against the derived requirements to show the degree of fulfillment.

Chapter 6 summarises and concludes the work that has been done in this thesis. As an outlook, we discuss potential research directions for the future work.

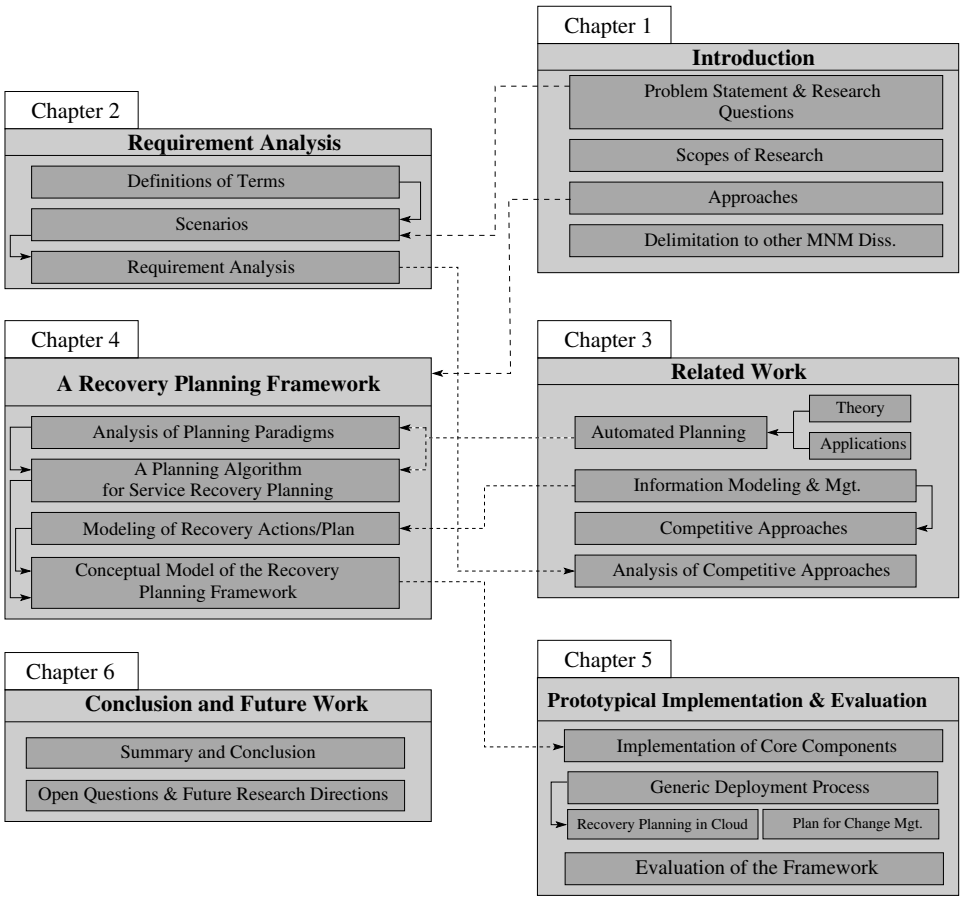


Figure 1.5: Approach and Layout of this Dissertation

1.5 Summary

This chapter provides an overview of novel aspects of the modern IT service landscape and, consequently, discusses the new management challenges. These challenges are explained in detail with focus on fault management, which is one of the key subjects among all functional areas of IT management. To address these challenges, the manual-based approaches are obviously no longer sufficient, especially by planning for fault recovery processes.

We therefore envision an automated planning framework as a solution to assist human administrators by planning for the recovery activities. The research focus of this dissertation is thus set on investigating and designing the recovery planning framework to assist human administrators by composing recovery plans in case of service failure. To achieve this objective, specific research questions are derived and discussed. This chapter is concluded with an overview of the structure of this dissertation.

2

Terminology and Requirements Analysis

Contents

We begin this chapter by introducing the definitions of terminologies that are used in this work. Since our work spans several research disciplines, thus the provided definitions are related to IT service management, system dependability and automated planning. Clear definitions of terminology used in this work help to avoid ambiguities in future discussions.

Following the definition of terminology, two fault management scenarios are presented to reflect the real-world management cases. The depicted scenarios range from a simple web-hosting service to the recovery of services in Grid. For each scenario, the challenges with regard to fault recovery are discussed in detail, which motivate the development of an automated recovery planning framework as the main focus of this research. Consequently, we propose a framework design with the main idea of applying the planning approach to automatically compose recovery procedures based on the fault

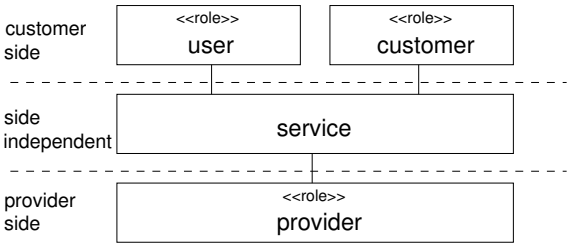


Figure 2.1: Basic MNM Service Model

situations. An overview of the framework is sketched with brief discussions of its sub-components.

Finally, requirement analysis of the framework is conducted. Requirements are categorised according to the individual component comprising this framework. A brief discussion is given to each requirement to explains its incentive. The purpose of the requirement catalogue is to shape and guide the development of the targeted framework.

2.1 Definition of Terms

2.1.1 Terms on IT Services

Terms concerning IT service management throughout this work rely on the MNM service model [GHH+02], which is designed as a generic model to define a service. As shown in Figure 2.1, the MNM Service model consists of provider side, customer side and side-independent part.

Provider Side: A provider is responsible for offering a customer the subscribed services. This includes implementations of the services together with their management functionalities. The services are made accessible to the customer through an interface called *Service Access Point (SAP)*. To properly manage that service, an interface to Customer Service Management (CSM) is implemented which allows the customer to access the management functionalities, for example, in the case of a service disruption perceived by the customer.

Side-Independent Part: This part of the model connects the customer and provider domains, but does not belong to either side. A *service* that a customer perceives is defined abstractly within this part which consists of its functionalities and associated QoS parameters.

Customer Side: An offered service is subscribed by a customer, who consequently enable a service users to access and use the service. The customer manages the offered service through the CSM interface.

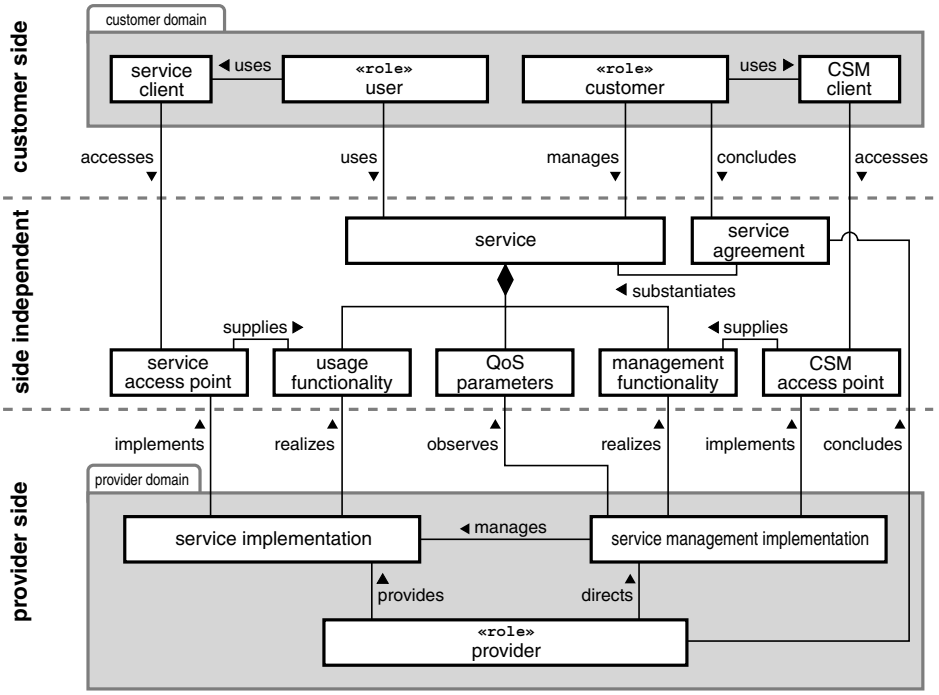


Figure 2.2: A Detailed View on MNM Service Model [GHH+02]

A detailed view on the MNM service model is given in Figure 2.2 [GHH+02]. A recursive application of the MNM service model is possible, which enables the definition of a chained service, for which diverse sub-providers and sub-services are combined to form a complex service.

Whereas the main focus of this work is on the service recovery and it is frequently conducted by a provider, therefore the further discussion is focused on the provider's side. Nevertheless, it is not always trivial to completely differentiate the roles of provider and customer. For example, in a chained service where a provider of a sub-service could be a customer of another sub-service at the same time. Therefore, it is necessary to give clear definitions on the terms related to the customer domain. Some definitions concerning the concept of an IT service are summarised as followings:

Service: A service in a generic sense is a set of functionalities, which a service provider offers to a customer. As defined in the MNM service model, a service is composed of usage and management functionalities. A customer or a user accesses the offered service through the CSM and SAP interfaces. Additionally, as part of the service definition, a set of Quality-of-Service (QoS) parameters is used to note the minimum service quality required by the customer. Those parameters are documented in the Service Level Agreement (SLA).

Sub-service: A complex service may consist of several sub-services, which implement partial functionalities to that service. A sub-service is either provisioned by a provider internally or located externally across the organisational domains. In short, a sub-service is a service, on which another service depends.

Resources: Resources are hardware or software components which realise specific functionalities required by a service or a sub-service. Physical server, server applications, network links, databases etc. can be categorised as resources.

Administrator: An administrator is a member of technical personnel whose tasks are mainly concerned with resource management, e.g. keeping and maintaining the resource in the operational state.

Service Manager: A service manager has an administrative role; he is responsible for not only keeping the targeted service operational but also for guaranteeing that the QoS of the delivered service is in accordance with the SLA. A service manager ought to have an overview of the service he is in charge of. Often, the service manager needs to work with the administrator cooperatively to solve the service problem.

customer and user: A customer is usually a legal entity or an organisation which has a contractual relationship with a service provider. A user is an individual who consumes the offered service.

2.1.2 Fault and Fault Recovery

As terms regarding faults, failures and errors are often used in a perplexing manner, it is necessary to give each of these terms a proper definition to avoid confusion and to allow a common understanding. The diversity and dependency of IT components which interact with each other to provide expected services requires that a clear distinction needs to be made.

According to Laprie et. al. [LR04] a *service* delivered by a system is characterised by its *behaviors* that are perceived by users. The *user* is another system or person, who receives the service from the provider. If the user is another system, it gets access to the service through the *service interface*, and a person as the user gets the service delivered from the *use interface*. A provider has internal states as well as external states. The states that are perceivable by the user at the service interface are said to be *external states* and the remaining of system states are denoted as *internal states*. A sequence of external states comprises the service.

Despite some minor differences, it is easy to recognise that Laprie's definition of service is in alignment with the MNM service model, thus their definitions of error, fault and failure can be used in the context of IT service

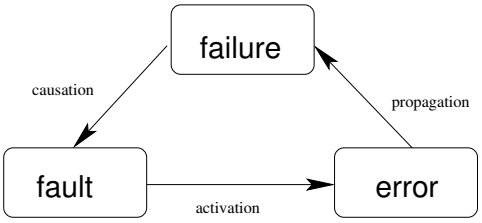


Figure 2.3: Transitions between Fault, Failure and Error

as well. Given the definitions of service, the terms failure, error and fault are defined as follows:

Failure. A failure is an event that occurs when the delivered service deviates from its correct behaviour. A service is said to have failed if it could not comply with the promised functional specifications.

Error. An error is the part of the system's state that *may* lead to a failure. For example, a damaged data packet transmitted over the network is an error, which may cause failure but not necessarily. It is worth noting that many errors do not propagate to the service interface; therefore, not all errors are causes of service failure, e.g. a corrupted data packet is unlikely to crash a service.

Fault. A fault is the adjudged or hypothesised cause of an error, e.g., a noisy network connection is often the cause for the corruption of data packets transmitted over that network. A fault is an unwanted but possible constellation of system states.

These three terms are closely related to each other. The relationships and transitions between error, fault and failure are shown in Figure 2.3

A fault is *active* when it starts to produce errors, otherwise it is dormant. A *dormant* fault is the fault which is located inside a system but not yet activated, e.g., a logic fault in a function not yet called or faults in hardware components not yet used by the system are said to be in a dormant state. Once the component is called by the system, it starts to produce errors and switches into active state. An error could cause a chain of errors which results in failure; this process is called *propagation of errors*. A service failure results when errors propagate to the service interface and cause the delivered service to deviate from the correct service. A fault may be caused by failure, e.g., an error in the configuration file of an application server leads to server failure and renders the web application, which depends on that application server, unusable. This is a *fault* (from the system's aspect) and the fault is induced by the *failure* of the application server.

The focus of this work is on the recovery of *faults*, which are, by the definition, adjudged or hypothesized causes of errors and failures of services. Those faults are determined by the fault detection and diagnosis, which are integral parts of fault management process precedent to recovery. This work assumes that the root causes of a service failure have already been determined by the fault detection and diagnosis. Related approaches concerning fault detection and fault diagnosis are extensively treated by [Han07, Sch08] in details.

Fault Types

Despite the rich diversities, faults in IT systems can be generally categorised into the following types [TS07]:

Transient Fault. This class of faults occurs once and then disappears. For example, electromagnetic disturbance near the network cable could cause the loss or corruption of data packets sent over that network and therefore disrupts the service. The network communication is normal again when such disturbance has gone.

Intermittent Fault. This class of faults resembles transient faults, but they reappear when several contributing factors occur simultaneously. An exceedingly overloaded web server could cause transient faults when the user cannot access the desired pages; accessibility resumes when the unexpected large number of access requests of the server becomes normalised.

Permanent Fault. This class of faults continues to exist until the faulty components are replaced. A software bug, defect hardware etc. are typical examples of permanent fault.

Failure Semantics

Failure semantics describe the behaviours that faults are likely to exhibit. It is crucial to distinguish various failure semantics, since organising proper recovery actions depends on failure behaviours of the impacted service. To classify different types of failure semantics, several schemata have been developed [Gär99, Cri91]. Table 2.1 gives an overview of those types and their corresponding interpretations.

Fault Recovery

Fault recovery is a process that utilises technical means to transform a system state, which contains a fault or multiple faults into a state in which an agreed service could be expected by the users of that system. In other words, fault recovery is an activity which reinstates a defect service to its normal

Failure Semantic	Interpretation
Crash Failure	A service halts prematurely and does not respond to an external query. Once a service halts, nothing is heard from it anymore. A service is in an undefined state.
Fail-stop	A service fails to respond to service requests and to deliver the correct service. It goes into a defined stop state. In some cases, a service may also announce its unavailability.
Timing Failure	A service's response lies outside the specific time interval.
Omission Failure	A service fails to send or receive messages. With a *send omission failure*, for example, a service may have correctly received a request and produced correct response, but then failed to send it to the requester. Vice versa, a service may omit the service request with a *receive omission failure*.
Byzantine Failure	A service fails by producing an arbitrary response in arbitrary time. This is the most difficult failure type to determine because the impacted service exhibits a random combination of failures discussed above. In the worst case, it may produce false-positive results, which make such failures even more difficult to be determined.

Table 2.1: A Classification of Failure Semantics

operational state. The fault recovery process is mostly done reactively after a fault has already impacted a service. An effective recovery process should bring the system back to its normal operational state as soon as possible by minimising the disruption period.

There are several ways to deal with faults: fault prediction and prevention, fault isolation and masking, fault recovery and service re-induction. Fault treatment in the context of this work refers to the *corrective maintenance* or the *systematic usage of compensation for a faulty service*. Other related terms such as fault detection and diagnosis are also important to the recovery, since a correctly detected and diagnosed root cause of a fault is fundamental to a successful recovery. However, as the main topic of this dissertation is recovery, an in-depth discussion of those terms is beyond the focus of this dissertation.

Clear distinctions and delicate relationships should be identified between them, even though the boundaries between those terms are not always clear.

Fault prediction and prevention methods are designed to prevent ser-

vice failure in both a predicative and preventive way. An early recognition of system anomalies, which could lead to a service failure if left unattended, is one of the keys to a successful fault prevention. Approaches such as statistical fault predictions [CKR05, LL06] and machine-learning-based fault-proneness analysis [Gon08] are all valid examples for the prediction of faults before they affect the system. Fault prevention reduces the rate of actual fault occurrences.

Fault masking and isolation approaches are designed to hide and isolate fault events from users, so that the users' perceived service is still in an operational state, even if it has been already impacted by faults. Fault isolation could be done through the replacement of faulty components with approaches such as job migration, redundant infrastructures [Sch06] etc. which provide opportunities for fault masking. To some extent, fault masking could be considered as a temporal approach for the fault mitigation.

Fault recovery and service re-induction are usually done after service failures, i.e. fault recovery does not prevent service failures, nor does it masks faults for users[1]. The main focus of recovery is finding and implementing proper strategies to mitigate as well as to repair the fault. Finally the repaired system should be announced and re-inducted as well. Although an efficient fault recovery process does not necessarily increase the reliability of a service, however, it does increase the availability of that service by reducing the Mean-Time-To-Repair (MTTR) value in the service availability calculation.

The main difference between those methods lies in the time point, relative to the fault occurrence, by which proper fault treatment methods could be invoked. Fault prevention and prediction approaches recognise the tendency of potential faults before they happen, whereas a fault isolation and masking approach is applied during or after faults take place. Fault recovery approach is usually done after faults and their resulting damages have been impacted and reported.

Despite their differences, a rigorous separation of those fault handling methods is inappropriate. Compared with others, fault recovery is a more border term, for instance in order to temporally mitigate the impact of a fault during a recovery, fault masking methods could be used to reduce the user-perceived service outage whilst the defected service component is being repaired. The fault prevention and prediction can lead to recovery as well, for example, an early recognition of an overloaded application server could trigger the activation of a load-balance mechanism to avoid a potential timing fault regarding the response time of that server.

[1] Although under some circumstances, fault masking could be a part of the fault recovery process as well

2.1.3 Definitions of Plan, Planning and Planning Algorithm

The term *planning* could be interpreted differently in different application domains; some examples of planning in general are financial planning, robot motion planning and military operation planning. However, despite these differences, planning does share some commonalities:

- Every planning procedure is intended to reach a goal or an objective.
- Every planning procedure is a part of a decision-making process.
- Every planning procedure usually involves multiple steps or strategies.
- Every planning procedure has to deal with limited resources.

Thus, *planning in general* can be formally defined as *the reasoning side of acting* [NGT04]. It is an elaboration process to select and to organise actions by anticipating their expected outcomes. The planning process targets achieving pre-determined goals by executing the selected actions.

The investigation into planning originates from the research of multi-agent systems (MAS), where software agents have to either adapt themselves to environmental changes or affect the environment according to the pre-defined goal. To achieve that goal, an agent must be capable of finding actions in particular sequences or causal order, upon which it could act. The process of coming up with a sequence of actions is called *planning* in MAS. Planning is also frequently referred to as problem-solving in artificial intelligence research, where solutions in the form of action sequences or strategies are searched for according to the given objectives and other additional conditions. Chapter 3 covers planning in more detail.

It is worth to note that, depending on how the states of the environment are represented and how time is modelled by solving the particular planning problem, planning could be categorised as *discrete-space planning* and *continuous-space planning*. The term *space* in this context refers to the state space involved in the planning that captures possible situations that could arise during the planning operations. It could, for example, represent the continuous movement of robotic arms or the velocity and speed of a helicopter, whose state spaces are continuous, e.g., they are uncountably infinite. Planning under such cases is called continuous-space planning, for example, to plan an optimal path for the arm movement of a robot with constraints on the length of the path and time of the movement. Furthermore the time for planning such activities is critical and therefore is explicitly modeled in continuous-space planning. To solve such planning problems, geometric models or differential equations are needed to characterise the continuous-space planning problem and the methods that solve planning problems in continuous-space are frequently referred to as *motion planning* in the control theory.

In contrast to continuous-space planning, discrete-space planning models its state space in a discrete manner and generally less time critical, for example, the state space of a board game such as chess can be easily discretized because these are countable finite states involved. Compared to its counterpart, time is less critical and hence it is implicitly modelled in terms of sequence order or succession of the actions in this kinds of planning problem. The planning process discussed throughout this dissertation refers to planning in a *discrete space*.

Having introduced the term *planning* we now provide several definitions on terms, which are related to planning; more detailed discussions on planning and planning algorithms are given in Chapter 3:

- A *plan* is the result of a planning process. It imposes a specific strategy or behaviour on a decision maker [LaV07]. A plan consists of a sequence of actions with causal relationships to each other. Those actions transform the current state of a underlying system into a desired state.
- A *planning algorithm* consists of procedures which are intended to compute and compose plans. A software entity that instantiates planning algorithms is frequently called *planner*. The research on planning algorithms is part of the research realm of *automated planning*.
- A *planning system* is a system which not only has a planner as a core component for plan syntheses, but also possesses the capabilities to exchange information and interact with other relevant external components through its interfaces.

2.2 Scenarios

To show the necessities of a computer-aided planning framework to assist the fault recovery process of modern IT services, two scenarios are discussed in this section. The range of selected scenarios intends to reflect IT service at different scales and complexities. For each of those scenarios, a general description of the setting is presented first and then challenges and difficulties of current fault recovery approaches in the scenario are discussed individually. Finally a comprehensive catalog of the requirements of an automated planning system is derived from the discussion.

2.2.1 Scenario 1: Recovery of Web Hosting Services

This scenario depicts a generic web-hosting service which customers use to provide dynamic and static web content. A data centre usually hosts a large quantity of such services for different customers with various requirements. To ensure the quality of the provided services, a service level agreement

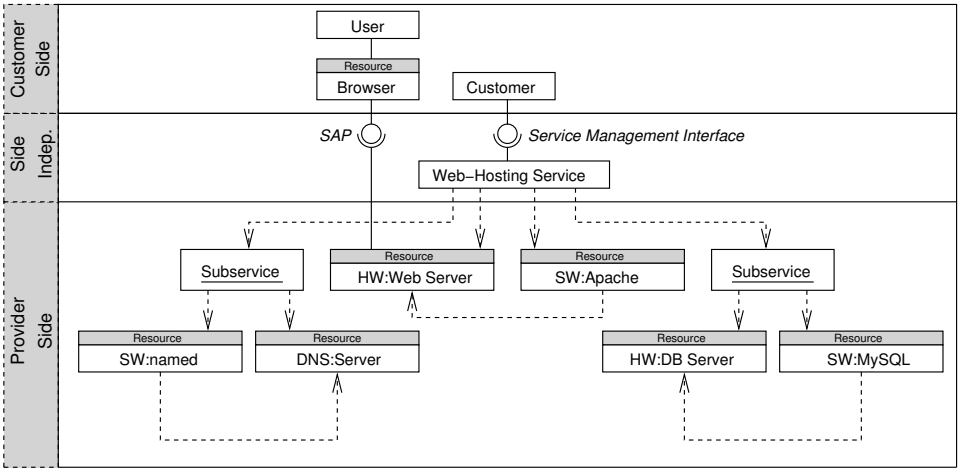


Figure 2.4: A Generic Web-Hosting Scenario Model

(SLA) is typically signed between customer and service provider to determine issues such as availability of services, service recovery time and maximum time for the service outage, to name just a few. Those terms must be observed and honoured by the service provider during the service operation.

Scenario

To simplify the discussion, the details of an instance of a hosting setting are given and discussed. Figure 2.4 illustrates a typical setting of such a service.

In this scenario, the customer operates a typical 3-tier web application service. The service is provisioned by the service provider, who maintains an amount of resources to sustain the required services. Those resources include an Apache server as front-end to deliver the content over the hypertext transfer protocol (HTTP) and a database system which provides the dynamic content according to the inputs from the user. In a typical 3-tier web application architecture, there is usually an application server required to deal with the business logic. To make the scenario readily comprehensible, it is assumed that the web application is integrated into the front-end. Additionally, a Domain Name System (DNS) is also required to do the name resolutions. The DNS and the database servers are also considered as the sub-services which are required by the web application. In this scenario, it is assumed that the sub-services are provided by the provider in the same organisational domain, although it is also possible for a sub-service to be offered by another provider across the organisational boundary.

The service provider supplies two different interfaces to the customer and user, respectively. The SAP is the interface where a user submits his

requests to and receives the results from the web application, i.e. it is where a user perceives the service. The user accesses the SAP by means of using a client; in this case, the client is simply a computer with a web-browser installed. Besides the SAP, a service interface is offered by the provider to the customer, who requires access to the service functionalities. A service interface could be as simple as a hotline telephone number to help desk or as complicated as a multiple-page web form.

In order to get the current operational states of the service, a monitoring system is usually required to fulfill this task. Meanwhile, the monitoring system also provides information on the measurement of the QoS parameters.

In the provider's domain, a service manager has an overview of the service and is responsible for the operation of the service offered. He has contact with the service management interface and possesses the responsibilities to coordinate the fulfillment of the service requests. In contrast to the service manager role, an administrator is a member of technical personnel, who is responsible for the operation and maintenance of resources such as a Apache server, DNS server, etc. Those two roles need to work cooperatively to maintain the services.

In the case of a service failure, for example, the desired web page could not be delivered or the queries to database through the web front-end returns *database not accessible* message. Those failures could be perceived by a user and he usually contacts the operator of this service for further support. If the perceived service failure is caused by the customer-written code, then the customer is responsible for recovering the impacted service. On the other hand, if the customer ascertains that he is not the cause of the failure, then a service request is dispatched to the provider to determine the causes and to recover the service. In the latter case, the fault management process is activated if the registered fault is confirmed.

Typical Fault and Recovery Strategies

Table 2.2 shows potential faults that could impact the web-hosting scenario. In addition to fault types, the table also shows the corresponding failure semantics and possible recovery strategies that could be applied. Note that all faults that are observed and considered are on the provider side. Faults that are liable to have be made by customers or users are not the focus of this discussion and therefore are not considered.

As examples, Table 2.2 lists some of the common faults to be seen in a web-hosting scenario. In this table, the symptoms describe the failure appearances a customer or a user usually experience when service is disturbed by faults. Faults could either be triggered by customer or user or be caused by the service provider. After service failure is reported and registered, the service provider needs to first determine if the cause of the reported failure

Symptoms	**Fault Cause**	**Failure Semantic**	**Possible Recovery Strategies**
Web server stops responding to service requests	Server crashes	Crash failure	Using failover site for the temporal operation, switching back to primary site once repaired
The response time to get results exceeds the agreed time frame during peak time	Web and database server are overloaded	Timing failure	Redirect partial requests to load balancing site
User's updates to database are ignored	Database caching overflowed	Omission	Redo/undo log shipping
Connections between web front-end and database is broken	software bugs in application server, which provides functionalities such as connection pooling and session management	Fail-stop	Update application server to new version
Customer could not install new applications	application container needs to be updated	Omission	Update application container on the server

Table 2.2: Typical Faults and Recovery Strategies for Web-Hosting Scenario

is caused by faults on his side. In our scenario, we currently consider the faults that are in provider's domain.

After the responsibility has been determined, root causes of the service failure (i.e. faults) need to be hunted down by technical personnel; Table 2.2 shows possible root causes of the enlisted service failures and their corresponding failure semantics. Root cause analysis is one of the essential procedures involved in the fault management; the correctness of root cause analysis directly influence the efficiency and success of the fault recovery. Although root cause analysis is not the focus of this work, a wealth of work complementary to this research is available on the research of root cause analysis in IT service management [Han07, Sch08].

Fault recovery strategies provide potential treatments for particular faults. There are admittedly many different recovery strategies available to treat those faults other than the ones listed; we only list a subset of several common practices which are usually employed in daily operation.

Challenges

In order to efficiently and correctly recover the faults, there are a number of challenges regarding fault recovery that need to be addressed in this scenario:

- *Scale of recovery operations.* In the scenario above, we merely described a simplified instance of a typical web-hosting service. In the operation of a real-world data centre, hundreds of such services are hosted on interdependent resources in a large-scale manner. For example, the Leibniz Supercomputing Centre (LRZ) is currently hosting approximately 350 such services for diverse research institutions and customers. Such services are run by under-numbered technical personnel such as administrators. Moreover, in such a complex IT environment, faults could impact any service instance in a random manner, therefore multiple faults are common scenarios. Administrators are constantly facing the challenges of dealing multiple faults.

- *Efficiency of the discovery of recovery solutions.* Faults, once they have occurred, must be treated efficiently , this requires that recovery solutions and recovery plans must be found in a timely manner. Furthermore, recovery solutions and plans should be made in accordance with policies and constraints on the resources.

- *Prioritisation of recovery operations.* Customers with different service classes obviously should be treated with corresponding priority during the recovery. This fact should be reflected and is especially important during the planning of recovery operations, when multiple faults happened concurrently.

- *Coordination of recovery operations.* The complexity of today's networked services requires that fault recovery operations involves different roles (e.g., service manager, database administrator, network operators, etc.); therefore the operation must be coordinated. The coordination of operations should be considered during the recovery planning.

- *Utilisation of recovery knowledge.* Knowledge such as applicable recovery strategies for the particular fault situations and management policies etc. are essential factors for successful recovery operations and must be considered during the recovery planning phase. Unfortunately such knowledge is scattered throughout the organisation in an often dis-organized manner.

- *Dependencies between underlying resources.* Dependencies are generally difficult to capture and they are usually not well-documented. A successful recovery operation requires a good understanding of underlying dependencies between components which, in most cases, are missing during the fault recovery. Non-understandings or even misunderstandings of the dependencies aggravate the recovery process or even unintentionally introduce new faults with the recovery.

2.2.2 Scenario 2: Fault Recovery in Grid Computing

Grid computing has been the focus of distributed and high-performance computing research for many years. The primary goal of Grid computing services is to provide an easy-to-access, on-demand computing platform for scientists around the globe to solve large-scale, computation and data-intensive scientific problems; these include the grand challenges in high-energy physics, genomic and biological research, computation and simulation of fluid mechanics etc., to name just a few. The increasing computational and storage requirements for solving such problems are pushing the traditional high-performance computing platforms further to their limits, on the other hand, the research communities require that more computing power and more storage should be provided in both efficient and cost-effective ways to facilitate computer-assisted problem-solving.

The idea of grid computing is to share computing power (such as CPU cycles) and storages over geographically distributed, multi-institutional computing infrastructures so as to provide a collaborative, shared problem-solving platform for scientists. Meanwhile, the complexity of underlying distributed infrastructures must be transparent to the users of such a platform. The grid middleware conceals heterogeneities and geographical diversities of infrastructures.

The collaborations of the different tasks are achieved by Virtual Organisations (VOs), in which different communities of researchers (grid users)

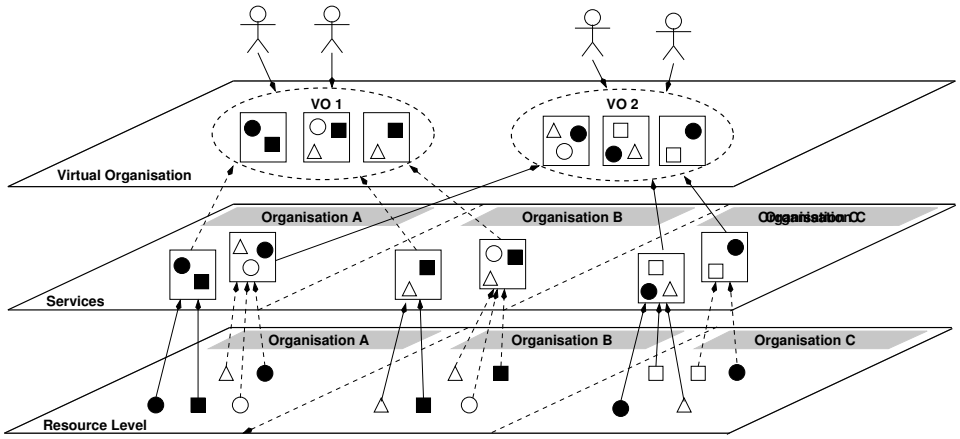


Figure 2.5: Overview on Grid Computing

across various research institutions with the same research purposes are assembled into virtual groups. Researchers utilise grid computing services by means of scientific workflows where different service components are composed accordingly to achieve predefined goals. The details on the implementation of services are transparent to the users. The users apply the submitting interfaces locally to submit their jobs to be processed.

Figure 2.5 shows an overview of Grid computing. At the bottom level, computational resources are offered by real organisations such as data centres. The types of offered resources include hardware, such as computing nodes, mass storage facilities, network components, and software, e.g. simulation software or customised software components for special purposes (e.g., protein fold predications).

At service level, resources are aggregated to provide desired service entities, e.g., simulation software needs to be installed on required servers, corresponding computing nodes need to be added and network components need to be provisioned in order to provide inter-connections. To facilitate easy access, each service has a corresponding interface where it can be called with parameters. The users of a service could be human or other services if it is a part of a compounded service. Currently services are made available by grid middleware, which also takes care of issues such as security, accounting, remote access and job scheduling, etc.

At VO level, institutions with same research interests are formulated dynamically into virtual organisations. The lifecycle of a VO consists of initialisation, operation, adaptation and dissolution. In a VO, members agree to share of Grid services with each other.

Managing grid services is a daunting task. All aspects of classical management, including management of fault, configuration, accounting, perfor-

mance and security (FCAPS), need to be considered in order to keep the services operational. Multiple levels of abstraction and the cross-institutional infrastructure augment the difficulties of the effective management of grid computing services. In this scenario we analyse the challenges and what is necessary to be done for the fault recovery of grid services.

Scenario

Fault management is one of the most important research issues regarding management of Grid services. Due to the hierarchical abstraction of services and the distributiveness of underlying resources, Grid services are more prone to failures than traditional computing services [ARSS05]. Although Mean-Time-between-Failure values of individual components have been constantly improved, a large number of participating components connected through unreliable networks make it impossible to completely rule out faults in a Grid [SG06]. Therefore faults in Grid should be accepted as *facts* rather than exceptions.

Scientists as primary users of Grid systems rely on them to meet their massive computational and data requirements. They need a reliable computing environment to finish their complex scientific computation workflows. It is crucial for them to get their computing jobs done reliably in a Grid environment. Once a fault unavoidably happens, it should be solved as quickly as possible and, in the meantime, losses of results and data must be reduced to minimum.

Table 2.3 shows several fault scenarios which are commonly observed by operators and users. In this table, *symptoms* describe the appearances of diverse faults that could be perceived by users. In most cases, they realise that their jobs are in trouble by a significant delay in retrieving the computing results or when their jobs just hang. *Fault cause* shows possible root causes of the failure symptoms. Since Grid is such a complex system, same fault symptoms could be caused by different root causes. It is up to fault detection and root cause analysis processes to pinpoint the right causes. *Failure semantic* maps the fault to the fault classes. *Possible recovery strategies* describe general steps that could be taken to remedy the fault so that the computation could be either resumed or started anew. Note that these recommendations are given in very general and high-level forms; these high-level descriptions still need to be resolved into sequences of executable actions. Furthermore, different computing sites are guided by management policies; these policies must be considered during the search for specific recovery solutions.

Symptoms	Fault Cause	Failure Semantic	Possible Recovery Strategies
Significant delay for user to retrieve job output	Malfunctioning of Resource Broker	Timing Failure	Replace the Resource Broker and resume the job.
Computing elements permanently waiting for input	Storage element is holding data necessary for the computation	Omission failure	Use replicas of data if available.
Job completion is delayed or fails because of mismatching between job requirements and allocated resources	Information index is failed to be updated on time	Timing failure	Update the information index and migrate or resubmit jobs.
Certain computing sites cannot exchange information	Network connection is broken between those two sites	Crash failure	Switch to another connection or use alternative site.
During the job processing, certain site is no longer responding	Hardware failure renders site unusable	Omission failure	Migrate part of jobs to other site, roll back to the most recent checkpoints if possible to reduce the losses.

Table 2.3: Typical Fault and Recovery for Grid Scenario

Challenges

To make Grid services more reliable while accepting the fact that faults are unavoidable, researchers have to confront with many challenges, this section discusses several important ones regarding fault recovery in Grid systems.

- Levels of abstraction augment the difficulties to recovery operations. As shown in Figure 2.5 the operation of a Grid system is built on different abstraction levels; the vertical dependencies between hardware resources and services need to be resolved during recovery operations. In the meantime, horizontal dependencies in terms of workflows of computational jobs and their underlying service components pose extra problems on recovery operations.

- Users of Grid services are mostly not computer experts; however, once a fault happens in their job workflows, they are commonly exposed to the gory details of failure information, which they can neither correctly interpret nor derive recovery solutions [MCBS03]. Users have to communicate with operators or administrators in order to work out correct solutions. Those processes unnecessarily prolong the recovery time and postpone the completion of computation. On the other hand, operators do not have special knowledge of the scientific workflows and their corresponding requirements for components; therefore the communication and coordination between users and operators during fault recovery are arduous processes. Finding a way to support the coordination of different roles and bridge the knowledge gap between those roles at the different abstraction levels during fault recovery amounts to another challenge for researchers.

- Management policies/constraints of participating computing sites must be considered during the recovery planning. Grid involves resources expanded over different computing sites, which belong to various institutions; these participating institutions usually have policies which guide the operation of resources, for example, a computing cluster or storage facilities could be used as a part of Grid resources only during a certain time of the day or the bandwidth of a network connection may vary according to the management policies. Since fault recovery in Grid context frequently requires job rescheduling and reallocation of computing resources between different sites, those management policies must be observed during the planning of the recovery operation. However, designing an applicable recovery solution plan while considering all applicable management policies would become a challenging task as the number of applicable management policies across different computing sites increases. These policies impose extra conditions on the planning of fault recovery processes. While human are inefficient

at constraint-solving, it would be beneficial to have an integrated automated mechanism deal with these constraints during the fault recovery planning.

- Recovery tasks need to be resolved into implementable actions. Even if solutions to fault recovery in Grid are found, for example, using a replica of the data or migrating computing jobs to other similar site, they are usually high-level descriptions of the recovery strategies or solution sketches. These solution sketches have to be further resolved into executable recovery actions, which could be carried out either manually or automatically. The concretisation processes demand high expert knowledge and a deep understanding of the underlying systems as well as middleware itself. To correctly derive an implementable recovery plan is a great challenge faced by Grid operators as well as users; a recent survey [MCBS03] on Grid dependability shows that 48% of survey participants have great difficulties implementing the application-independent fault recovery actions. They simply do not have a clue what steps to take to recover from faults, even if they are provided with solution sketches.

- Heterogeneity of underlying resources poses a challenge on the recovery knowledge. Grid infrastructure involves a large quantity of heterogeneous applications, middleware, network components etc. To find a recovery solution in such a complex system, it is not reasonable to expect a single operator or user of a Grid system to possess all of the necessary in-depth knowledge on the underlying systems, let alone the knowledge on the fault recovery. On the other hand, management knowledge is distributed among different resources in different forms, such as CMDB databases, management policy repositories and other proprietary formats. To collect the desired knowledge, one needs to not only query those resources manually but also deal with different formats. The necessity of finding ways to unify the search and application of management knowledge must be one of the primary objectives of improving Grid management.

- Faults during the recovery need to be considered. The original recovery plan must be adapted to the new state of the underlying system. A Grid is a highly dynamic system; underlying resources constantly change their states. A previously composed recovery plan may quickly become outdated; therefore the recovery plan must be dynamically adapted to the recent state of the underlying system. However, the manual process of composing the recovery plan and reacting effectively to the changing states is not adequate and, therefore, this process must be automated.

Recent research on Grid is much more focused on the development of middleware to provide basic functionalities rather than on providing means to facilitate management of Grid. The scenario above is intended to show the challenges regarding fault recovery, which is an essential facet of management of a Grid system. Having introduced problems and challenges regarding fault recovery, this dissertation endeavors to design an automated approach to compose recovery plans in a Grid system. This approach is expected to assist users and operators to compose feasible recovery plans across different abstraction levels of a Grid system based on formalised recovery knowledge.

2.2.3 Visionary Solution and Requirement Analysis

In this section, a framework of the visionary solution to the identified research challenges is proposed and briefly discussed. Based on this framework and its functional components, we conduct the analysis process to derive the system requirements for the framework.

2.2.4 A Framework-based Solution

As stated in Chapter 1, the main objective of this dissertation is *to design a framework with planning capabilities that acts as an integral part of a management system to assist fault recovery processes.* To achieve that goal, we need a framework that not only encapsulates the core capabilities of planning, but also considers the communication and information exchanges with external management components. Figure 4.17 offers an overview of the proposed framework and its interfaces to external management components.

The central part of this framework is *the planning component*, which provides the main functionalities to compose and schedule recovery action plans. An automated planning algorithm realises such functionalities by selecting recovery procedures while observing constraints that are applicable to the particular recovery activities. Scheduling of the recovery steps can be done either as a part of the planning process or as the separate procedure depending upon the particular planning algorithm under consideration.

The planning operation is based on information provided by several supporting components: *the knowledge base* is a structured repository which holds the recovery-relevant information like available actions and methods which are both described by proper annotations. Recovery actions are atomic steps that can be directly executed on the target system, for example, restarting the network interface is an atomic action that could be controlled by a script. Compared to recovery actions, recovery methods are high-level constructs of fault recovery activities which may include proper recovery actions or cascade with other methods. The recovery methods are not directly executable and need to be further refined. The separation of

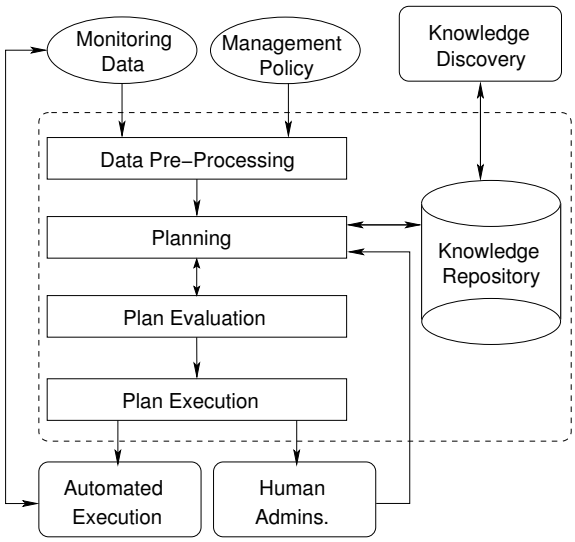


Figure 2.6: An Overview of the Planning Framework

recovery methods and recovery actions provides a flexible way to dynamically compose recovery plans. On the other hand, the construct allows to encode the best recovery practice, e.g., by properly composing the *method* construct.

To populate the knowledge base with data, the component provides an interface to external *knowledge discovery components*, which serves as a data source for discovery and enrichment of recovery knowledge. It is worth noting that the knowledge discovery itself is a highly complex and yet active research area in computer science; approaches such as data-mining, machine-learning and various reasoning methods [TLC+09] have been used to facilitate the knowledge discovery process. The planning component retrieves the necessary data from the knowledge base in order to achieve the planning objective.

To find applicable plans, the planning component needs to be sustained with a set of service states, these include the current state of the observed service and set of desired goal states to be expected after the recovery is done. The current state of the observed services can be provided by external monitoring facilities, which are available to the vast majority of management systems; therefore, the framework provides an interface to external monitoring components. Additionally, as we discussed, a feasible recovery plan should abide by and honour the management policies; these policies can be used as constraints to control the plan compositions. *The data pre-processing component* manipulates the data received from external components and translates them into a format that can be applied by the planning

process.

Additionally the composed plans can be evaluated and checked by *plan evaluation* for their correctness and validity. This procedure could be done in multiple ways, ranging from a simple verification of the recovery metrics to applying model checking approaches [FL00, RRD04]. Depending on the evaluation results, plans can be selected either by automated means according to a set of metrics or simply by human administrators. Finally, depending on whether the execution requires human involvement, recovery plans are either handed over to an automated execution mechanism or human operators for the actual implementations. *The plan execution component* is responsible for this procedure; it could integrate workflow engines to execute the designated recovery plan. The effect of each recovery step will be observed by the monitoring system and fed back to the framework to control the actual effectiveness of the plan. Figure 2.7 shows the interactions between components and their corresponding method calls of the framework in sequences diagram. More details on the design of framework and rationals of such design will be discussed in Chapter 4.

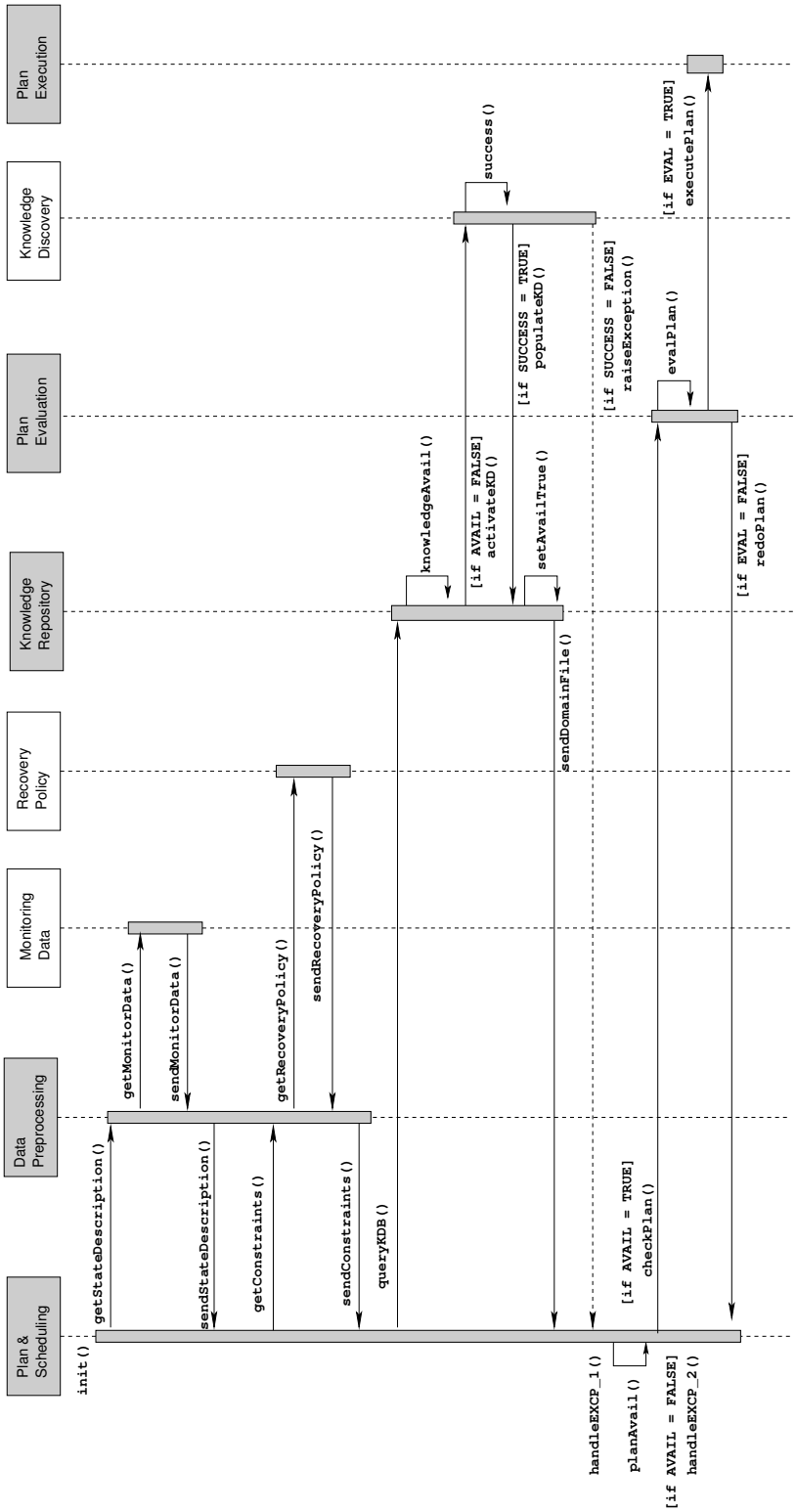


Figure 2.7: Interactions between Components of the Framework

2.2.5 Requirements Analysis

Having presented the scenarios and the sketch of the proposed framework, we discuss in this section the specific requirements for this framework. The purpose of the requirement analysis process is to further determine the functional and non-functional characteristics of the framework and of its comprising components in order to facilitate and to guide its further development. The derived requirements are not only concerned with the framework as a whole, but also focus on the essential capabilities of the core components introduced in the previous section. Requirements are derived based on the framework and its components.

Framework Requirements

General requirements illustrate properties that the framework as a whole has to fulfill. They are described in the following:

F1: Generic Framework The framework should be applicable to address the planning activities of generic IT services rather than being limited to specific application scenarios. It should not be bound to specific technologies and application domains as well.

F2: Scalability The framework should scale well. This means that the target framework should be adaptable to the complex recovery situations and can produce feasible recovery plans as the operation environment becomes intricate. The framework should offer acceptable performance characteristics, particularly in terms of time needed for the plan composition.

F3: Usability The framework should be easy to use for its users such as service operators. The intricate internal working principles should be transparent to the user of this framework; this implies that no steep learning process should be required in order to use it. Moreover, it should provide a friendly user interface, where results of the planning can be presented in a straightforward manner.

F4: Interoperability The framework is designed not as a stand-alone entity which works solely on its own, but rather as an enhancement to the current management system. Hence, it should possess the capability to interact with other management components which provide complementary planning information and functionalities to the planning framework. For example, to gain the current state of a service; it must not design an monitoring component if its own, since most of the management systems are coupled with monitoring capability.

F5: Adaptability The framework should be easily adaptable to an existing management infrastructure in order to cooperate with them. This

requires that the framework provides well-defined interfaces to facilitate the interoperability with external components.

Planning and Scheduling Requirements

As the core of the framework, the planning component performs actual planning procedures and schedules the plan executions according to the constraints and conditions provided. To achieve these objectives, the following requirements should be fulfilled by this component:

P1: Planning Capability The core function of the target framework is the automated planning capability. Based on the recovery knowledge, the planning function should be capable of fulfilling the following specific requirements:

- **P1.1: Automated Planning** This requirement depicts the basic functionality of the planning. Planning in this case is an informed problem-solving process; the algorithm operates on the available information and desired outcomes and tries to come up with one or a set of solutions to achieve the desired planning objectives.
- **P1.2: Generic Planner** The automated planner should be generic and independent of the use cases. However, this does not rule out the possibilities of situation-based optimisation of the planner to achieve better performance.
- **P1.3: Efficiency of Planning** The fault recovery plan should be composed automatically in an efficient way. The efficiency in this context means both timeliness and correctness of the planning results, i.e. the planning mechanism should find a feasible plan in a short time which is in alignment with the constraints and conditions posed on the recovery.
- **P1.4: Plan Optimisation** Optimising plans in terms of time and costs, e.g., some of the plan steps should be executed in parallel in order to reduce the total recovery time. The planner should integrate temporal information into the planning process so as to be able to recognise such opportunities as parallelization.
- **P1.5: Plan Adaptability** Changing plans adaptively to address the dynamics of the underlying system. The planner should constantly check the changing state of the system and adapt to the changes.
- **P1.6: Exception Handling** Handling exceptions during the planning. The planner operates on the *a priori* knowledge that is available; thus it is possible that no proper recovery plan could

be found by the planning process. The planner should be capable of catching and handling such kinds of exceptions as noted in Figure 2.7.

P1.7: Plan Granularity The composed plan should be fine-grained to the level that is directly implementable either by means of automation or with human assistance. All steps involved in the composed plan should be atomic actions that could not be further decomposed by the planner.

P1.8: Handling Different Levels of Recovery The planner should be capable of supporting recovery planning of different abstraction levels. As shown in the previous scenarios, recovery of services could be boiled down to recoveries on different levels, for example, in the Grid scenario, a recovery at the workflow level has vertical dependencies to the underlying services and resources; the recovery plan should take such dependencies into considerations.

P2: Scheduling Capability Complementary to the recovery planning, the framework should be capable of scheduling for executions of the selected actions. Scheduling is especially important when limited resources concerning recovery are available, for example, if multiple users are impacted by the service faults, obviously those users who have higher service class should have a corresponding priority during the recovery. Furthermore, if the recovery planning system has to process multiple recovery requests, the requests queue should be scheduled as well.

P3: Constraints Solving Capability Recovery activities are constrained by factors such as time, financial costs, personnel, technical resources etc. Those constraints have to be taken into consideration in order to compose a feasible recovery plan, therefore, the constraint-solving ability must be included in the core mechanisms of the targeted framework.

P4: Policy Integration Capability This capability is directly related to constraint-solving requirement of the target framework. Relevant management policies need to be translated into and represented as constraints so as to be processed during the recovery planning mechanism. An information model regarding the management policy needs to be established. Examples on the integration of those policies into the recovery process have already been given in the above scenarios.

P5: Fault Coverage Capability As one of the core capabilities, the framework should be capable of recovering the diagnosable faults. The cover-

age of faults is depending on the *a priori* recovery knowledge available to the system.

P6: Handling Multiple Information Types Planning a recovery is obviously an information-intensive task. To achieve a workable result, the planning mechanism should be capable of integrating multiple information resources. Existing management information such as monitoring data, management policies and alike need to be handled by the framework.

P7: Collaboration Capability To generate recovery plans, the planner needs various inputs from different data sources as shown in Figure 4.17 and Figure 2.7; thus it should possess the capability to collaborate and exchange information with those data resources.

Knowledge and Knowledge Repository Requirements

The following requirements describe properties that are related to the knowledge and knowledge repository. Essentially the knowledge repository provides the planning and scheduling component with the information for composition and scheduling of recovery plans.

K1: Identifying and Classifying Recovery Knowledge The term *Recovery knowledge* is a very general description. To build a planning-relevant recovery knowledge repository, we must firstly identify and specify what recovery knowledge is, i.e. what could be considered as recovery knowledge in the context of IT services? Those could be, for example, recovery procedures that are commonly used or specific actions applicable to a certain fault situation, etc. Additionally, we have to classify such knowledge into different categories, since recovery activities on different levels of service abstraction could be interpreted differently, for example, a recovery strategy such as job migration on the scientific workflow of the Grid computing scenario may be different to that on the resource level of the Grid. The classification of such recovery knowledge provides a way to organise and build relationships between them.

K2: Recovery Knowledge Repository The purpose of the repository is to provide the opportunity to persistently store the recovery knowledge and to facilitate knowledge reuses. It serves as an indispensable facility to enable the planning process. The term knowledge in this context denotes actions and methods that have been previously identified and are relevant to fault recovery for different services.

K3: Providing Structured Recovery Knowledge Recovery of IT services may require the knowledge from different application domains,

e.g., recoveries of web server, middleware, back-end systems or handling faulty scientific workflows. The persistently stored knowledge should be properly organised to avoid an unstructured "knowledge soup" situation, in which knowledge concerning different applications is mixed up into a mess and hampers the efficiency of the planning process.

K4: Interfacing with Knowledge Discovery Tools Although the knowledge elicitation is not the focus of the current research, there should be the possibility to access external knowledge discovery components. The purpose of the external knowledge elicitation and knowledge discovery is to enrich the recovery knowledge base, so that more fault cases could be covered.

K5: Modelling of Recovery Knowledge The representation and usage of the recovery knowledge is one of the keys to a successful recovery planning. In order to facilitate the automated planning process, an information model which describes different types of recovery knowledge and their characteristics should be devised. An information model of recovery knowledge provides sharable, reusable and structured information requirements on the prescriptions of recovery-related knowledge.

K6: Encoding and Reusing Recovery Methods Methods for the fault recovery should be encoded and reused in the fault recovery planning. Those methods should reflect the best practices and approaches of recovering particular services. However, as stated in both scenarios above, knowledge on recovery methods exist distributed around or across organisations in an informal way, therefore, collecting and eliciting such knowledge at the time of need is an arduous process. Consequently, such inefficient knowledge management unnecessarily prolongs the recovery period. To overcome this problem, recovery methods need to be properly encoded so as to facilitate the reuse of those procedures.

Data Pre-processing Requirements

The data pre-processing component is designed to translate and to format the data from external components into the one that can be utilised as input for the planning mechanism. It provides the planning and scheduling component with the ability to gather information which is external to the framework. Such information includes: current state of the faulty service and management policies that is relevant to the recovery.

D1: Translating relevant Knowledge into Planning Format The targeted framework should be capable of translating external information

into the format that could be applied by the planning operation. This requirement is directly related to the requirement of the information modelling, while the information model provided by the framework should cover the modelling of external inputs from other systems.

D2: Representing the States of an impacted Service The framework should provide ways to represent the states of the impacted service. The state of components of the service should be properly described. The state information is intended to be used to describe the initial states, objective states as well as state transformation of the fault recovery process.

D3: Representing the Recovery Policy as Constraints The composed plans should abide by any constraint that may be applied to the recovery. Those constraints could be related to the metrics such as time, costs and usage of resources. Such constraints are mostly expressed by the management policies, thus the planning and scheduling mechanism should provide the capability to represent such policies in the form of constraints, so that the planning and scheduling component could integrate those constraints into the planning process.

Plan Evaluation Requirements

Correctness is a critical factor for recovery plans. According to the IEEE Standard Glossary of Software Engineering Terminology [Boa90], *correctness* is defined as a characteristic of a software system that is free from faults and meets the specific requirements of the users so that their expectations of the system could be fulfilled. In the context of fault recovery, the user's expectation is to recover the impacted services with the generated recovery plans and thus the plans should be correct. To this end, the framework should provide possibilities to verify and evaluate the correctness and feasibility of the composed plans. The planning evaluation component is designed to serve this purpose. The following requirements should be fulfilled:

E1: Performance Evaluation To ensure the optimal recovery time and to minimise the MTTR factor, the generated plan should be checked against its performance. For example, if the scheduled recovery plan is optimised by means of identifying activities that could be overlapped during the execution or if the performance characteristics of all involved resources conform with the requirement of the recovery.

E2: Cost Evaluation A recovery activity in a real-world scenario is usually constrained by costs. The evaluation mechanism should provide the opportunity to evaluate the total costs of the plan in order to check if the automated generated plan conforms to the financial constraints provided.

E3: Temporal Evaluation The time-span for the planned recovery activity should be verified as well. The planned recovery procedures should comply with the temporal requirement of recovery, which is defined by the SLA.

Workflow and Execution Requirements

Workflow is a natural way to express the procedure-based recovery plans. For example, migrating and restarting a computing job to another node in a Grid system (scenario 2) could be described in a workflow form. Additionally, the fact that workflow management systems are widely deployed and used [AHA04] provides the targeted framework with the possibilities to execute the composed plan. The following are requirements concerning the workflow component of the framework.

W1: Execution Engine The execution engine should serve as an interface between the recovery planning framework and the underlying system. It implements the composed plan and actuates the target system with the steps described in the plan according to the specific schedule determined by the planner.

W2: Coordination of Execution Not all recovery steps could be automated and executed by machines. For some tasks, human interventions are indispensable for a successful completion of the recovery jobs. Thus, the execution mechanism should possess the capability to coordinate those tasks that could not be automated with those that could be executed without the human attendance.

W3: Compatibility to the existing Systems This component should be compatible with the tools and applications that have already been deployed in a management environment; therefore, generic interfaces should be provided by the execution engine to pass information on the recovery strategies to the underlying executing mechanism, when it is provided.

W4: Control over Executions The framework should possess the capability to have control over the execution of the recovery plans. As stated in requirements *F5* and *P1.5*, the framework must be adaptive to the changes of the underlying system or service. This implies that it should have some basic controls over the workflow executions, such as being able to halt the execution, restart or undo some of the specific plan steps etc. This is required when, for example, an instant change of the underlying system invalidates the current plan or, when an error during the execution of a particular step requires a retry of that step.

2.3 Summary

In this chapter, the concepts related to IT services, fault and fault recovery with their related terms as well as the terms for planning were first introduced to rule out any ambiguities in the rest of the discussion. Then two scenarios of IT services with different scales and purposes were presented to show the necessities of an automated planning framework to assist human administrators to compose a plan efficiently during the fault recovery processes. In each of these scenarios, a general overview was provided to allow a common understanding of the settings. Challenges and problems regarding planning of fault recovery were discussed in detail for each scenario as motivations of this research. Based on the scenarios introduced, an automated recovery planning framework was proposed as a solution and discussed as a seminal introduction to the succeeding research. In order to shape and to guide further research and development, requirement analysis was conducted in order to drive and define the specific characteristics of the framework. Requirement analysis was carried out based on the individual components as well as the framework as a whole. They encapsulate six categories: general requirements of the framework, planning and scheduling requirements, knowledge and repository requirements, data pre-processing requirements, plan evaluation requirements and workflow & execution requirements. The next chapter is intended to give an overview of the state-of-the-art of automated planning technologies and their application in IT management. The selected approaches will be evaluated against our requirements.

Requirements List		
Category	**ID**	**Requirement Details**
Framework	F1	Genericity
	F2	Scalability
	F3	Usability
	F4	Interoperability
	F5	Adaptability
Planning & Scheduling	P1	Planning Capability
	- P1.1	Automated Plan Composing
	- P1.2	Generic Planner
	- P1.3	Efficiency on Planning
	- P1.4	Plan Optimisation
	- P1.5	Plan Adaptability
	- P1.6	Exception Handling
	- P1.7	Plan Granularity
	P2	Scheduling Capability
	P3	Constraints Solving Capability
	P4	Policy Integration Capability
	P5	Fault Coverage Capability
	P6	Handling Multiple Information Types
	P7	Collaboration Capability
Knowledge & Repository	K1	Classifying and Identifying Recovery Knowledge
	K2	Recovery Knowledge Repository
	K3	Providing Structured Recovery Knowledge
	K4	Interfacing with Knowledge Discovery Components
	K5	Modelling of Recovery Knowledge
	K6	Encoding and Reusing Recovery Methods
Data Pre-processing	D1	Translating Knowledge into Planning Format
	D2	Service State Representation
	D3	Policy to Constraints Representation
Evaluation	E1	Performance Evaluation
	E2	Cost Evaluation
	E3	Temporal Evaluation
Workflow & Execution	W1	Execution Engine
	W2	Coordination of Execution Mechanism
	W3	Compatibility
	W4	Control over Executions

Table 2.4: An Overview on the Requirement Catalogue

3 Related Work

Contents

This chapter aims to provide an overview of the research endeavours regarding the applications of automated planning techniques in the management of IT services and other engineering disciplines. The main purpose of this chapter is to review and critically analyse the comparable approaches, especially those which are related to IT management. Instead of narrowing our views merely to the fault recovery aspect, we also investigate some of the most important applications and research efforts related to planning, which cover the other management areas such as configuration and change management as well. Followed by the detailed discussion of the related research, an evaluation of the mentioned approach is done accordingly. To the best of our knowledge, a comprehensive framework-based solution to solve the planning problem has been only marginally addressed in the research of IT management, therefore a component-based evaluation approach is considered to be more reasonable and fair under such circumstance. The

evaluations are carried out according to the requirements which were derived in the previous chapter.

In order to make the discussions in this chapter more comprehensive, a detailed investigation of the automated planning theory and different forms of planning paradigms is given before the related approaches are investigated and evaluated. To show the general applicabilities of the planning in the real-world applications, we also shred some light on the applications that are not directly related to the IT management. This chapter concludes with the discussion on the drawbacks of current approaches and how the research of this dissertation is going to endeavour towards improvements.

3.1 Fundamentals of Automated Planning

Since automated planning forms the cornerstone of the suggested framework, it is indispensable to provide some background knowledge on this technique first. Therefore, in this section we provide a formal introduction to the automated planning theory and investigate some fundamental planning paradigms which embody the automated planning theory in different ways.

The research on automated planning was brought up by the research of multi-agents systems (MAS), in which agents are deployed in an environment to perform pre-defined tasks. Due to the dynamics of the environment, agents must either adapt themselves to environmental changes (for example, autonomous robot) or take actions to regulate the environment (frequently used in control theory). The agents must make decisions themselves. These decisions help the agents to complete their tasks and reach their objectives. The decisions are frequently represented in the form of a sequence of actions, i.e. plans. To achieve this, the agents need to have the perception of their environment in order to make effective plans. This perception is frequently referred to as the environment model. The information necessary for building up the environment model is supplied by agents' sensors.

As previously stated, planning is a process of selecting and organising actions to be taken under certain conditions to achieve designated objectives. Automated planning is a study of such a process computationally. It is one of the classical research fields of artificial intelligence (AI), which concerns using well-established searching and reasoning techniques to generate plans according to the current state of the underlying dynamic systems and the pre-defined states.

Algorithmically, a planning problem in general has as input a set of actions, predictive model of an underlying dynamic system and a set of pre-defined goals. The output to a planning problem is one or more courses of actions that fulfill the designated goal. A planner operates based on the input and elaborates on a subset of actions, which transit the current

state of the system to the goal state. Advanced planners also schedules the executions of the selected actions, which are guided by the set of control rules.

Conceptually, the definition of a planning problem is similar to that of the Kripke Structure [CGP99] and could be denoted as :

$$P = (\Sigma, s_0, g)$$

where Σ is a representation of a state-transition system and s_0 denotes the current state or the initial state of the underlying dynamic system; g represents the goal state. A state-transition system is an abstract model of a planning system and is defined as a triple:

$$\Sigma = (S, A, \gamma)$$

where S is a set of states $\{s_1, \ldots, s_n\}$ and $A = \{a_1, \ldots, a_n\}$ is a set of actions to be chosen from. γ is a state-transition function, which takes the current state s_n as well as a selected action a_n as input and brings the system to the next state. In order to assist the reasoning process in the selection of actions, each action $a_i, i \in 1, \ldots, n$ in the action set is supplied with further descriptions:

$$a_i^{pre},\ a_i^{post} \subseteq S$$

The first notation a_i^{pre} denotes the pre-conditions of this action, which are conditions that have to be fulfilled before this action can be chosen. The a_i^{post} describes the post-condition of the action, in other words, the effect of the action. In some cases, a cost function $c(a_i, s_i) > 0$ is also needed to evaluate costs of the action a_i associated with the state s_i. This function is meaningful when more than one option is available for planner to choose from. For example, the costs involved for recovering a web server with restarting or with switching the server to a standby system are surely different. The cost could be, for example, financial cost or time costs etc., according to the application scenarios.

A solution to a planning problem P is a sequence of actions:

$$(a_1, \ldots, a_k), k \leq n$$

which transits a system from the initial state s_0 to the pre-defined goal state. The state-transition function γ maps actions to the new state:

$$s_1 = \gamma(s_0, a_1), \ldots, s_n = \gamma(s_{n-1}, a_n)$$

.

Provided with the above information, a planning algorithm finds a plan by searching in the space of possible states configurations. Different planning

algorithms utilise various methods to generate a plan. To illustrate the idea of automated planning, Algorithm 1 shows the algorithmic description of a very general and representative form of state-space-based planning, which will be introduced in the following section on planning paradigms.

Algorithm 1: The General Planning Procedure [NGT04]

Input : O: a set of actions;
s_0, g: initial & goal states.
Output: π: plan

```
1  s ← s_0;
2  π ← ∅;
3  while True do
4  |  if s satisfies g then
5  |  |  return π;
6  |  end
7  |  A ← {a|a ∈ O&precond(a) true in s};
8  |  applicable ← A;
9  |  if applicable = ∅ then
10 |  |  return Failure;
11 |  end
12 end
13 Choose a ∈ applicable;
14 s ← γ(s, a);
15 π ← π.a;
```

3.1.1 Planning Paradigms

Having been an important topic of research for decades, the automated planning research community has developed and proposed different planners and planning paradigms, ranging from those which could merely solve small-scale "toy" problems to those which are applied to solve real-world problems such as navigating unmanned vehicles and planning for military operations. We enumerate some of the planning paradigms and their characteristics.

State-Space Planning

State-space planning is the most straightforward form of automated planning paradigm. This breed of planners are based on exploring the complete search space. In this case, the search space is a subset of the state space, which contains all possible states of the environment (world).

As input, it takes the description of a planning problem **P**, which includes the initial state of the environment, the goal state and, most importantly, the description of the transition system. The planner starts from the initial

state a_0 and searches for the proper actions, whose pre-conditions are fulfilled by the descriptions of the current state. The search continues until either an solution, whose post-condition is equal to the goal state, is successfully found or otherwise it exits with a failure. Figure 1 shows the details of such a planning paradigm. An action a_i is said to be *applicable* if the pre-conditions of the action is a subset of the current state s_i, i.e. $a_i^{pre} \subseteq s_i$. A planning goal g is said to have been achieved with action a_k, if $a_k^{post} \subseteq s_g$.

One of the most important ingredients of state-space planning is the searching process for the proper actions. There are different flavours implemented for searching: *forward-search* starts by the initial states and works towards the goal, whereas *backward-search* begins with the goal state and works iteratively towards the initial state. Both search strategies produce sub-goals during the process if, at a certain stage of operation, a sub-goal is reached, which satisfies either the goal (by forward-search) or the initial state (by backward-search), the algorithm stops and returns the current set of action as solution. It has been proofed that state-space based approaches are sound and complete [NGT04]. The algorithm selects non-deterministically the next action as a candidate to be evaluated; however, this step could be done deterministically as well, for example, searching approaches such as A*, breadth/deep first or greedy search can all be applied.

An obvious drawback of state-space planning paradigm is its confined scalability. When the complexity of the planning problems scales up, e.g., when the size of the searching space grows, the time/memory costs of this approach become inadmissible, even if some optimisation methods have been tried. The naive version of the state-space planning algorithm has the complexity $O(n^3)$, where n is the length of the plan. Additionally, state-space planning also suffers from other problems, such as the Sussman Anomaly [Sus75] by solving some particular classes of planning problems. Finally, the state-space planning approach can only elaborate on linear plans, but plans are not always structured linearly [SC75]. Due to those reasons, the state-space planning approach is only confined to solving limited class of small size problems and has been hardly applied to real-world applications. However, as one of the earliest efforts in automated planning research, this approach builds a solid foundation for further research on other planning paradigms.

Plan-Space Planning

Compared to the state-space planning approach, the plan-space planning follows a different path towards plan composition. As its name suggests, the search space of the plan-space planning consists of partial plans, which are constructs involving multiple atomic action steps with specific causal relationships between those steps. The planning process starts with a partial plan and constantly refines this incomplete plan by adding new actions,

establishing new causal relationships between actions and applying ordering constraints to the existing plan until a well-articulated and executable plan is found. A partial plan π can be described as:

$$\pi = (A,\ \succ,\ B,\ L)$$

where A is a set of actions involved in the plan, $\succ$ represents the ordering of actions, B is a set of binding constraints and L denotes a set of causal relation between actions.

Different to state-space planning, plan-space planning composes plans by refining a partial plan until the pre-determined description of the goal state is satisfied. The planning process involves four types of operations:

i.) Inserting actions into the partial plan for the plan's refinement.

ii.) Establishing causal links between selected actions.

iii.) Imposing ordering constraints on the existing plan.

iv.) Binding variables and instantiating the plan.

Operationally, the planning process is proceeded by searching backwards from the goal state. The initial partial plan consists of two symbolic actions: a_{init}, a_{goal}, where a_{init} only contains post-conditions corresponding to the initial states and a_{goal} has only pre-conditions which are equal to the goal state. Unlike state-space planning, an action selected by the planner does not have to fulfill all the pre-conditions of its successor (searching from backwards); only partial fulfillment of the pre-conditions is required to make the selection decision on actions. In this way, the non-linearity of the plan structure could be covered and it allows the planning process to be conducted in a distributed manner as well. Actions are repeatedly inserted into the incomplete plan until the initial state (or the goal state, in the case of a forward search) is reached. The causal links between actions are also established during the planning process. Finally the plan is initiated as a totally ordered set of executable actions.

An important issue to be considered is the conflicts between the pre-conditions of the two selected actions that may result in an inconsistency of the plan. In planning, such inconsistency is called *threat*, e.g., if two selected actions have a causal relation $a_i \xrightarrow{p} a_j$, where p is the post-condition of a_i and pre-condition of a_j. Suppose there is an action a_c, which has a post-condition $\neg p$. We say that the action a_c is a threat to the causal link $a_i \xrightarrow{p} a_j$ if the ordering constraint $a_i \succ a_c \succ a_j$ is true. In other words, the action a_c eliminates the pre-condition of a_j and makes this partial plan inconsistent. Such inconsistency could be resolved by, for example, rearranging the ordering constraint. All potential threats are checked and resolved in the plan-space planning.

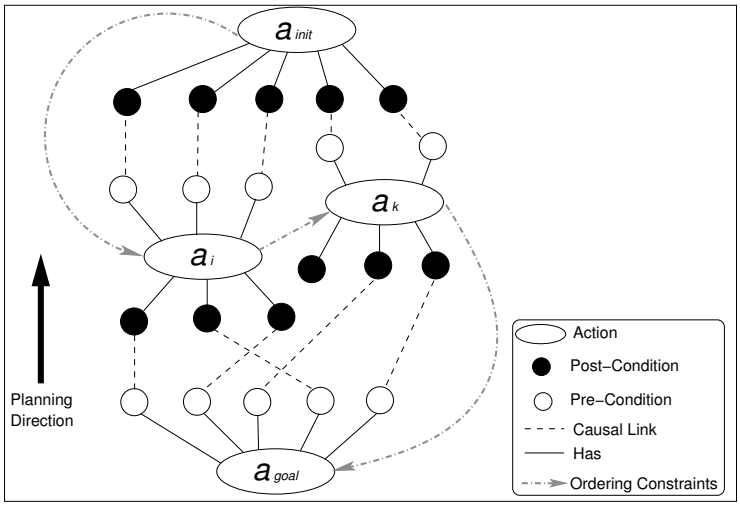


Figure 3.1: A Simple Example of Plan-Space Planning

Figure 3.1 shows a simple instance of plan-space operations. A partial plan π is said to be the solution to a planning problem, if and only if:

i.) Initial (goal) state is reached by the backward (forward) searching of actions;

ii.) No threat is involved in the plan π:

iii.) Ordering constraints and variable binding constraints are consistent.

Plan-space planning has some obvious advantages over the traditional state-space based approach: the property of working on partial plans imbues the planner with the capability to decompose a planning problem into subproblems, through which parallelization could be applied to increase the efficiency of the planning process. Thus, the solution plans do not need to be strictly linear in their structures, which enables the planner to solve more planning problem types.

The disadvantage of this sort of planner is that the partial plan refinement process as well as the detection and resolutions of the threats are computationally expensive operations. Without optimisation, the computing time would be a major influencing factor to the efficiency on the complete planning operation.

Graph-based Planning

Both planning approaches introduced above are not efficient enough to solve large-scale problems. They suffer from some common deficiencies: firstly,

both approaches are not scalable. As the problem size grows, the complexity of finding a solution rapidly becomes inadmissible. Secondly, the *branching factor* is a major problem which makes both approaches inefficient, since every action, including irrelevant ones, whose pre-conditions are fulfilled by the current state, will be tried, even if it may not finally lead to a viable final solution. Finally it is very hard to apply heuristic optimisation to those approaches without further ado.

To circumvent these problems, graph-based planning is proposed [BF97]. The idea of graph-based planning is to reduce the searching time by encoding the planning problem, with its constraints explicitly embedded, in a graph structure. The operations of graph-based planning consist of two stages: plan graph construction and solution extraction. In the first stage, a multiple-layered graph is built starting from the initial state of the planning problem, where the state is expressed in propositional logic.

The layers are built in an interleaving manner with states and actions. The even-numbered layers are states and odd-numbered layers contain applicable actions. The edges between layers connect pre-conditions and post-conditions with corresponding actions. Starting from an initial state s_0, the graph is expanded by applying actions that are relevant to the current state. The selected actions produce a new set of states which describe the expected post-conditions after those actions are executed. During the construction of the plan graph, it is of vital importance that the consistency within the action layers must be checked to see if there is any conflict in the current graph. A partial graph is inconsistent if one of its layers contains actions that are mutually exclusive, i.e. a valid plan could not possibly contain both of those actions. These mutually exclusive actions could be checked in two ways by the algorithm: checking for action *interference* and checking for *competing needs*, where interference means two actions are not independent of each other and competing needs means both actions have pre-conditions that are mutually exclusive. During the setup process of the plan graph, old states are modified by deleting propositions or adding new propositions from the current set of propositions.

Within each layer that contains state descriptions, a goal-check process is performed to see if the current set of state descriptions contains the goal state. If not, the graph will be further expanded, otherwise the graph construction phase is ended with success and the plan extraction phase starts to backtrack to the solution. The current state description $s_{current}$ satisfies the goal state s_{goal} if the propositional description of the goal state is its true subset ($s_{goal} \subset s_{current}$). Furthermore, the description of the current state may contain a solution to the problem if there are no mutually exclusive propositions appearing in this level. The solution extraction procedure starts with backtracking to actions that lead to the fulfillment of each sub-goal and proceed iteratively until the initial state is reached.

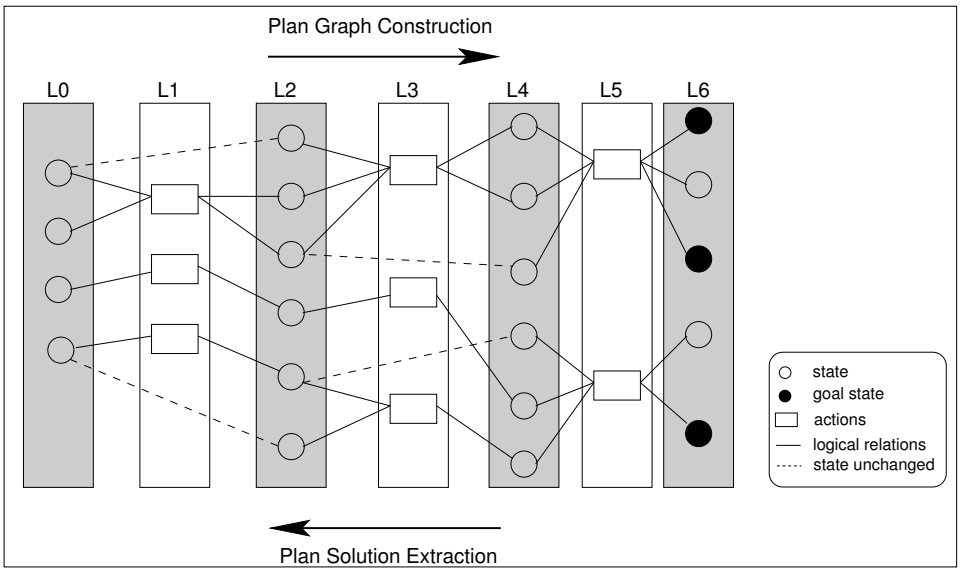


Figure 3.2: Plan Graph Planning

Figure 3.2 illustrates the graph planning. In this figure, the circles represent the propositional description of the state and the boxes represent the actions. The solid-lines connect actions with their propositional pre-conditions and corresponding post-conditions. The dashed-lines between some of the propositional descriptions of the states illustrate that the propositions would not be changed by the action, e.g., they remain consistent before and after the execution of the actions. Experimental results [BWE+94] show that the graph-based partial ordering planning approach achieves significant performance gain over state-space or plan-space-based planners.

Hierarchical Task Network

The basic idea behind Hierarchical Task Network (HTN) [EHN94] planning is the refinement of the planning tasks based on the decompositions of those tasks at different hierarchical levels. In the definition of HTN planning, there are three kinds of tasks: *decomposable tasks*, *primitive tasks* and *goal tasks*. As the name suggests, the decomposable tasks are those tasks that could be refined into more detailed levels and could not be directly executed. On the other hand, primitive tasks are directly executable operations; they are also called *atomic operations*. Finally, the goal tasks describe the desired final states of the targeted systems; they are identical to planning objectives of other planning paradigms.

Unlike the planners that have been discussed so far, the HTN planner introduces a new element called *method* into the planning procedures.

A method is a construct which contains the refinements of more specific operations organised in a form of partial network. Operations could include atomic actions or other methods. The planning operations is an iterative process of decomposing high-level tasks into lower-level operations until there are no more decomposable tasks. The planning process maintains a tree-like structure, where nodes are decomposable tasks and leaves contain atomic operations. The main purpose of HTN-based planning is not to achieve a defined goal, but rather to find out how to perform a given set of atomic or decomposable tasks.

Despite the difference in how planning operations are performed, the HTN-based planner has as input a similar definition to the planning problem description P that we previously discussed, which includes a state transition system Σ, the initial state description s_0 and the planning objective g. A method M for the decomposition of tasks is defined as follows:

$$M = (M_{ID}, M_{task}, M_{pre}, M_{network})$$

where M_{ID} denotes a unique name of the methods which distinguishes these methods from others. M_{task} describes this decomposable task; M_{pre} is a set of literals, which describes the pre-conditions of this tasks, and $M_{network}$ contains a partial network that includes the sub-tasks of these methods. Note that the sub-tasks could contain another set of decomposable tasks or primitive tasks, or both. A method is total-ordered if its corresponding subtasks are total-ordered. Algorithm 2 gives a description of the HTN-based planning paradigm. Note that during the task decomposition process, attention should be placed on avoiding conflicts between refined tasks and other existing tasks. With minor modifications to the planning methods, constraints could also be applied to the HTN planning; these constraints include ordering constraints, resource usage constraints, time constraints and so forth. The capability to handle different types of constraints greatly extends the applicability of the automated planner.

In practice, HTN-based planning is one of the most applied planning paradigms for solving real-world planning problems. Several reasons [NGT04] contribute to the popularity of HTN planning: first, the hierarchical structure of the plan structure resembles the natural way of humans solve practical problems. High-level tasks are resolved in a step-by-step manner into the low-level implementation details. Second, HTN planning provides ways to encode human knowledge into an automated process, which dramatically enhances the qualities of solution plans. Third, the HTN-based planners are domain-independent. Provided with different knowledge of the planning domains, HTN algorithms could be used to solve planning problems regarding those domains. Finally, the HTN-based planning paradigm is efficient, both theoretical studies [BWE+94] and empirical experience [FL02] show that such planners can achieve significant performance gains over other planners.

Algorithm 2: General HTN-Planning Paradigm

Input : P: definition of a planning problem;
T: the task network, where $t_{primitive}, t_{decomposable} \in T$;
M: set of methods.
Output: π: a sequence of primitive tasks.

```
1  π ← ∅;
2  Choose t ∈ T, where t_post−condition satisfies s_goal, s_goal ∈ P;
3  π = π.t;
4  while True do
5  |  if t ∈ π ∧ ∀t ∈ T_primitive then
6  |  |  Resolve conflicts in π;
7  |  |  if Success then
8  |  |  |  return π;
9  |  |  else
10 |  |  |  return Failure;
11 |  |  end
12 |  end
13 |  Choose m ∈ M;
14 |  Decompose t with m;
15 end
```

Additionally, the HTN-based planners proved to be sound and complete from an algorithmic perspective [NGT04, EHN94].

Other Planning Paradigms

The planning paradigms introduced in the previous section illustrate the basic forms and ideas behind the automated planning approaches. As multifaceted as the planning problems are, a rich set of approaches from other research perspectives have been proposed to solve planning problems. This section purposes to present different ideas for solving planning problem without delving too deeply into details.

Planning as Satisfiability. The idea behind planning as satisfiability is to encode the planning problem, including descriptions of initial and goal states, actions and state transformation functions, into propositional formulas. The transformed planning problem can then be solved using known algorithms that solves satisfiability problems, e.g., the *Davis-Putnam* [DP60] procedures. The planning as satisfiability shows that general reasoning techniques based on the first-order logic resolution and theorem proving could be used as means to solve planning problems; however, the performance of such approaches could not compete

with search-based planners. Therefore, planning based on satisfiability has not been widely applied to solve real-world problems.

Planning by Model Checking. Proposed by Cimatti et al. [CGGT97], the idea of planning by model checking is to transform a planning problem into a verification problem. Model checking is a formal way to verify the system properties and system behaviours based on a formalised state model, i.e. it checks whether a system would never reach some undesired states or that it would always reach desired states. Provided with the definition of a planning problem (encoded in Kripke model), a model-checking-based planner produces the solution by calculating whether the goal is true in the current model. One of the advantages of such kinds of planner is that they allow the planning domain to have a nondeterministic characters, which is reflected in many planning problems. The obvious disadvantage is that if the number of states increases, the model checking approach could be very inefficient. To solve such a problem, optimisations such as applying *symbolic model checking* [BCM+92] are proposed, which allow a more concise, less redundant representation of states with symbols.

Planning as Constraint Solving. Constraint satisfaction problems (CSP) have been the topic of computer science research for years. Many powerful approaches have been developed to solve CSP. Given a set of constraints with their domains and a set of variables with values that they may take, an algorithm to solve CSP means finding values for each of those variables, such that given constraints are satisfied. There are several ways to apply constraint-solving to planning problems: first, the planning problem could be converted into a constraint solving problem, e.g., encoding a given planning problem P into a CSP P', a solution to the planning problem P exists if a solution to the corresponding CSP P' is achievable by some well-known CSP algorithms, such as AC4 [MH86]. However, such an approach is only limited to certain classes of planning problem: the bounded planning problems [NGT04], which are restricted to finding a plan with certain solutions with length at most k and the integer value k must be provided as one of the parameters of a planning problem. The second way to apply constraint-solving techniques to planning problems is to use constraints to enhance the planning algorithms. For example, the plan-space and graph-based planning approaches use precedence constraints and consistency constraints to guarantee the correctness of the solution. To solve the planning problem with limited or concurrent resource and time conditions, constraint-solving techniques could be applied or integrated as well to augment the original planning algorithms [NFF+05].

Beyond all techniques introduced previously in this section, a rich set of alternatives has been proposed to solve planning problems, such as planning with Petri-Net reachability [NM02] and planning based on Markov Decision processes [KLC95, RPPCD08]. Each of these approaches has its pros and cons concerning efficiencies, flexibilities and complexities. A planner that works efficiently in one application domain may result in disaster in another. An application-domain independent choice of planning paradigms is therefore not rational without tailoring and customizing the planner to serve the needs of specific application areas. Having surveyed the general principle of planning with different paradigms, in the following section we investigate the applications of different automated planning methods and planning systems to solve real-world planning problems.

3.1.2 Applications of Planning

Automated planning methods have been successfully applied and integrated in different application areas, including space exploration, industrial applications, military operations etc. In this section we provide an overview of the application domains that are not directly related to IT management. The purpose of this section is to show the applicability of automated planning in a broader sense, on one hand and, on the other hand, the planning-related problems in those areas resemble the problems that our research intends to tackle; the commonalities and analogies of those applications can help us to leverage our further research.

- Automated Planning in NASA Mars Mission Operations [BJMR05]. The Mars Exploration Rover (MER) is an unmanned vehicle developed by NASA to carry out diverse scientific investigations on the surface of Mars. The MER possesses the capability to maneuver automatically on the ground and, equipped with a broad set of scientific equipment such as hazard camera, spectrometer, microscopic imager and rock abrasion tools, it is expected that the MER conducts different surface investigation missions according to the given high-level objectives. Due to many factors (such as the massive delay of signals with a round-trip time around 15 - 20 minutes) that limit the communication between ground control and the vehicle, it is impossible for ground control to communicate with the MER in a real-time manner. On the other hand, considering the complexity of MER's operation plan composition and unpredictability of the operation environment on Mars, it is impossible for operators on Earth to make a detailed operations plan in real-time. Consequently, the MER has to make decisions itself according to the research objectives given in the form of high-level descriptions. High-level mission objectives are provided regularly by the experts at ground control. The MER has to observe constraints in terms of time, energy

costs and priorities of tasks during the plan composition phase.

To address this challenge, researchers at NASA employed a planning-based framework called the *Mixed Activity Plan Generator (MAPGEN)* [NPV+05] to tackle the mission planning problem. The core of this framework is a HTN-like planner, which refines the high-level goal task into low-level implementation details. All decomposable tasks are termed *activities* and are defined in a structure called the *mission activity dictionary*. The core task for the planner is to decide whether a task with its sub-tasks needs to be included in the current mission plan and how conflicts of such mutually exclusive actions can be resolved. To serve such purposes, a planner implementation called *Extendable Uniform Remote Operations Planning Architecture (EUROPA)* [FJ03] is used to reason on solutions as well as to represent the plan. The unique feature of this planner is that it allows the information of solution time, usage of resources, mutual exclusion of actions and concurrency to be represented and integrated into the plan reasoning process. Parameters for temporal conditions of activities are expressed as constraints of the solution. A constraint-solving algorithm is applied in this planner to find the best solution while observing that no constraint is breached during the process.

- Planning in Robotics. Planning techniques have been traditionally applied in robotics to plan the path of the robot's movement in either a 2-dimensional or 3-dimensional world or to decide the movement of the robotic arms to achieve certain objectives. Therefore, planning in robotics is frequently referred to as *motion planning*. The planning process of robotic research differentiates slightly from the concept we have seen so far; the main difference is the description of the state space, which a planning algorithm observes. In robotics, the state space is a continuous space, in contrast to the state space we have discussed so far, which is discrete. The continuous planning space is usually represented by a set of mathematical equations to model the world, in which a robot moves. The general approach for the motion planning involves operations to define the 2D or 3D geometry models and transform these models. The results of the transformation forms the *configuration space*, which is comparable to the state space of planning in discrete state spaces. The dimensions of configuration spaces determine the freedom of the robot's movement and the motion planning in this stage could be regarded as searching in this high-dimensional space for the planning solution. To that end, it is necessary to transform the continuous model into a discrete model. There are multiple ways to perform such kinds of operations, e.g., *combinatorial motion planning* [Str00] and *sampling-based motion planning* [Lat91, BB05]. With a successful transformation, algorithms for discrete space could

be integrated into the motion planning approaches.

- Planning in Pervasive Computing. Pervasive computing, sometimes known as ubiquitous computing, ambient intelligence or smart environment, is a research area that targets enabling automated interaction between diverse embedded computing elements and the users of such systems. With assistance of such computing environments, users should be capable of achieving their desired objective with less efforts. To this end, it is required that the rich set of computing elements should have the capability to be self-managing and automated. The services in a pervasive computing environment usually comprise different sub-services offered by the computing elements and need to be dynamically assembled as the objective of the users may be constantly changing.

  Automated planning techniques have been recently proposed [PBA+07] and integrated [RC04, MU08] to achieve the dynamic composition of complex services according to high-level goal descriptions from the users. Rangananthan et al. [RC04] proposed a framework integrated with the planning capability. It perceives the state of the operating environment, generating the goal states according to the abstract objectives defined by users. The centrepiece of the framework is a meta-planning procedure, which heavily depends on the planner called *BlackBox* [KS85]. The BlackBox planner treats the planning problem as satisfaction problems (SAT). The input for the planner is a set of problem descriptions defined in the STRIPS format (Stanford Research Institute Problem Solver) [FN71]. The problem description will be converted into SAT format and solved as a SAT problem. The state of each device and service is represented by one or more predicates. Services and devices have interfaces through which their states could be queried by the planning framework. The meta-planning procedure gets the goal from users or other applications and generates a set of final states which satisfy the goal. The best final state is chosen based on the utility function. The BlackBox planner tries to find a solution plan according to the chosen final state. If no solution could be found, it switches to alternative final states, otherwise the solution is dispatched to the executing mechanism. The planing execution is constantly monitored by the framework for a possible failure in the execution. In case of failure, the action will either be tried or aborted. Marquardt [MU08] follows a similar approach, different types of planners are tested within the simulation environment set up by authors, including UCPOP [PW92], SGP [WAS98], BlackBox [KS85] and a self-implemented planner called TOP. The purpose of this work is to test and compare the performance of the given planner in the context of smart environments with different sized planning problems. Patkos et

al. [PBA+07] discussed the applications of distributed AI for ambient intelligence; the authors discussed the issue of collaborative problem-solving with distributed planning and multi-agent coordination, which is regarded as key to the planning operation across multiple domains. A detailed survey and discussion on the state-of-the-art and open issues are presented in their paper.

- Web Services Composition Planning. The W3C defines a web service (WS) as the following [BHM+04]:

  > ... a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialisation in conjunction with other Web-related standards.

  With the recent proliferations of WS and service-oriented architecture (SOA), which is built on the foundation of WS, it is no longer possible that a single WS could cover the intricate needs of users with complex objectives. This leads to the fact that different WS need to cooperate and interact with each other in order to fulfill the requirements of users. To this end, it is required that complex web services must be dynamically created and adapted to the needs of users [PTD+08]. As the services are constantly becoming more complicated, a manual approach to composing a complex web service is no longer capable of addressing the dynamically changing environment and, moreover, such an approach cannot adapt to constantly changing service goals. To address this issue, a wealth of proposals have been made recently to apply AI-based planning techniques for the automated service compositions.

  Carman et al. [CST03] formalised the web service composition problem as a planning problem. A semantic-type matching algorithm is proposed in their paper. Together with an interleave search and execution algorithm, the suggested approach allows a basic automated service composition. Similar research was conducted by Vukovic et al. [VR05], who discussed and applied the HTN-based planner to augment a context-aware application development process. Two different planning algorithms - Simple Hierarchical Ordered Planner 2 (SHOP2) and TLPlan - are evaluated and compared with regard to their abilities in terms of the domain description, planning control structures and capabilities to plan optimisation. Similarly Sirin et al. [SPW+04] proposed an approach which combines the application of the OWL-S [BHL+04] and the SHOP2 planner. They described an algorithm

to translate an OWL-S based service description into the planning domain for the SHOP2 planner. McDermott [McD02] suggested an estimated-regression-based planning approach for the composition of WS. While most of research on applying automated planning in WS composition are built on the premise that the behaviours of web services are deterministic and predicable, in most of cases, however, the WS and underlying environment are constantly changing in the real-world application scenarios, which limits the applicabilities of the classical planning approaches with a static model of WS and the underlying execution environment. To alleviate such problems, Doshi et al. [DGAV05] proposed an approach based on Markov Decision Processes (MDP) to model the planning environment as well as the behaviors of WS with uncertainty.

A great deal of work has been done to apply different types of automated planning techniques to solve real-world problems; the research mentioned above is just a slice of application instances of those application areas. With the rapid advances in the research on planning, this technique has been constantly extending its application domains, for example, by assisting the planning in military operations [BBB+04, AMB+08], supporting manufacturing [Nau95, TK00, RDF05], planning for the emergency evacuations [MAABN99, XSTK04] and gaming [YRB+04, AMP05], etc., to name just a few. In the next section, we will concentrate on the planning in the IT management domains.

3.2 State-of-the-Art: Planning Applications in IT Management

It was only recently that the IT management research communities discovered the powerful features of automated planning techniques to solve their problems. Prior to that, there had been scant research dealing with either the theoretical aspects of automated planning in IT management or practical management applications that utilise the features of planning. In this section, we investigate closely some of the related research of automated planning in IT management. Through the research of related work we confirm that, although being a promising technique, research regarding the application of planning techniques in IT management is still in its infancy . For the sake of completeness, we chose not to confine our investigation solely to the planning in fault management, but also to include functional management areas such as configuration management and change management. Additionally, the current status of planning in Grid computing is also included in this investigation. For each reviewed work a detailed discussion is included to analyse both the advantages and disadvantages of the approach.

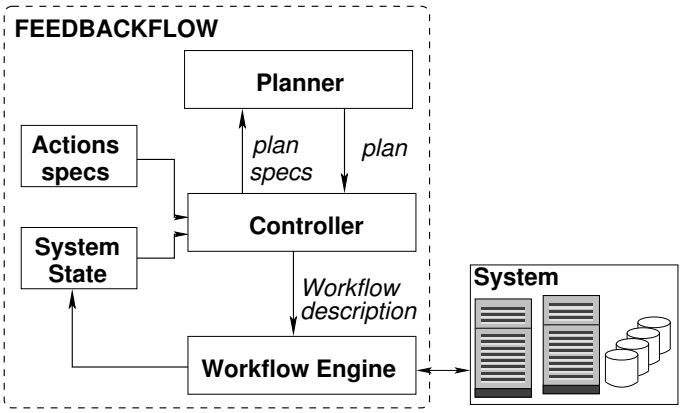


Figure 3.3: Architecture of FEEDBACKFLOW

Finally a summary of the evaluation against the requirements derived from the previous section is presented to serve as an overview of the related work.

3.2.1 Configuration Management

IT services and their underlying systems need to be properly configured to meet the objectives of business and individuals. Modern IT services usually consist of a large amount of sub-systems and components, including software and hardware. Manual configuration of such complex, multi-components systems is not only a costly, but also an error-prone task without an intelligent support mechanism. Automated planning in the context of configuration management could give operators flexibility to automatically find proper configuration actions and synthesise configuration workflows. Planning system should have the ability to transform the high-level configuration goal into a specific, executable workflow. During the planning process, it should also consider applicable configuration policies in order to guarantee that a solution conforms to the enterprise policies.

Andrzejak et al.[AA05] designed a prototypical framework called *FEEDBACKFLOW*, which incorporates an automated planning component to compose complex configuration tasks. As its name suggests, this framework is based on the closed-loop control with feedbacks, including planning, execution, results validation and replanning. As Figure 3.3 shows, the *FEEDBACKFLOW* framework consists of a planner, a controller and a workflow engine. The controller is responsible for taking the declarative action specifications from operators, which include atomic actions such as an installation of certain types of software components or setting parameters to certain values. The description of system is based on the system states, which are denoted with PDDL syntax. The controller triggers the

consecutive steps based on aforementioned inputs. The planner is supplied with plan specifications and it finds a configuration plan that is applicable. The generated plan is translated by the controller into XML format and forwarded to the workflow engine for execution. Each workflow has a state manager unit, which keeps tracking the execution status of actions, and a stop unit in case a non-recoverable failure takes place. A further unit is designed to represent the selected atomic actions. Upon termination of execution, the system manager unit updates the controller, which decides either to repeat the planning cycle or to terminate the whole process.

The planner in this framework is based on the Model-Based Planner (MBP) [BCP+01]. MBP synthesises plans in terms of *symbolic model checking*, in which a propositional formula is used to represent a finite state automaton. The advantage of using MBP is that it not only does plan composition, but also includes plan validation and plan simulation, where the generated plans are verified for their correctness according to the given property and their executions could also be simulated on a given domain model as well. Moreover, MBP can work with a partially observable planning domain, which means the status of domain is only available after the execution of the selected actions during runtime. In terms of configuration management, one could not expect the status of the planning domain to be fully observable during the planning phase, so some *sensing actions* need to be performed to detect the current state. Although MBP is claimed to be a highly efficient planner, integration in FEEDBACKFLOW shows that it suffers from a critical performance limit. The prototype of FEEDBACKFLOW only works well with small examples. Experiments have shown that it takes an excessively long time to compute a complex plan, which makes the practical value of the framework questionable. The major bottleneck of FEEDBACKFLOW, we believe, lies in the planner, where both domain model and plans are constructed with symbolic model checking; despite the performance improvement against explicit model checking approach, it is still a time-consuming approach. Other disadvantages of the suggested approach are: it does not provide an information model to describe the configuration actions, and representation is based on the simple PDDL descriptions. The work is merely built on the premise that the configuration actions and description of states are automatically available. Such assumptions seriously limit the feasibility of the approach in practice. Furthermore, for the framework as a whole, no effort is made towards the integration of the framework to the current management infrastructure. Since the publication of the work, no further research and development has been carried out and the framework remains a prototypical design.

The work from Arshad et. al [NA07] called *Planit* specifically targets on the optimal software deployment and reconfiguration issues. The motivation behind *Planit* is to cope with intricate software configuration tasks,

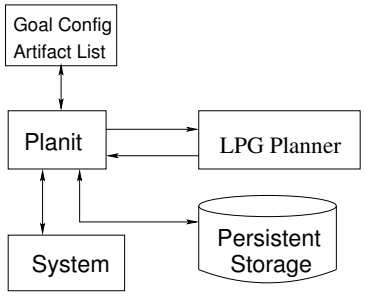


Figure 3.4: Architecture view of Planit

where many inter-related factors such as time constraints and states of applications etc. need to be considered during the process. The conventional configuration methods, which assume a static deployment environment and static plan, are neither effective nor scalable. The proposed approach deals with such dynamics based on the process called *sense-plan-act*. The architecture adopts a domain of reconfigurable systems, whose model consists of three types of entity: component, connector and machine. Component is a software entity, connector describes communication links and machine is where the component and connector are located.

As Figure 3.4 shows, the architecture is much like FEEDBACKFLOW introduced previously. As the mediator of the architecture, the *Planit* component consists of two core sub-components: the Reconfiguration Manager and the Problem Manager. Upon receipt of system events from sensor agents, the reconfiguration manager delegates them to the Problem Manager, where the current system state and goal state are derived. The goal state is computed by checking the explicit and implicit configurations, which denote a non-specific predicate of the system after plan is executed and information of detailed described artifacts of the configuration has been gathered. Together with the domain description file, the *Planit* component sends derived initial and goal states to the external planner. Note that all input files to the planner are written in a PDDL-conforming format. The *Planit* relies on the LPG planner[GS02] for plan syntheses; it searches iteratively for better plans, and plans are judged by metrics, such as time. The reconfiguration manager selects the best plan for execution on the system. The persistent storage is used to store contingency plans, which cannot be changed during the life of the system.

The LPG planner is based on the local search of planning graphs to solve planning problems. It enjoys a reputation of having a fast planning speed. The experimental results from Arshad et al. show that, with 60 configuration items including servers and network devices, the best plan could be found within 20 seconds. Furthermore, it considers the costs of actions

during the planning process, which plays a central role in the syntheses of the configuration plans. Moreover, the LPG planner could find multiple plans with which same configuration objectives could be achieved with different costs. This feature provides multiple possibilities for administrators to consider, in case the best plan fails, it could be decided whether to perform a re-plan process or to use alternative plans that are readily available. However, *Planit* still suffers from several drawbacks: for example, plan execution is done in a batch fashion; in the real-world scenario, system states are changing constantly, and a plan made several minutes previously could be invalid already; therefore, a plan and execution interleaving model would be more realistic. Despite the fast speed shown by experimental results, *Planit* still suffers from scalability problems. The performance bottleneck is the planner; the experiments conducted by the authors limit the number of configuration artifacts to 120. The external LPG planner is taken without any application-specific optimisations, for example, searching space could be reduced using constraints which are derived from management policies. The produced plan is a sequence of change actions that is to be executed; however, the aspect of the scheduling of the configuration activities is ignored. Scheduling is important in the configuration management when, for example, multiple configuration tasks need to be carried out with limited resources. Additionally, the proposal from Arshad et al. does not suggest how to integrate the framework into an existing management infrastructure. Finally, there are no information models considering configuration items, actions etc. provided, the prototype of the framework utilises a very naive way to model the configuration problems.

Srivastava et al. [SBS04] proposed a planning framework *Planner4J* [Sri04] for the IBM *Agent Building and Learning Environment (ABLE)* [BSP+02]. The ABLE framework is one of the major undertakings made by IBM to embody the Autonomic Computing [KC03, Kep05] concept. It provides a lightweight Java agent framework together with a library of intelligent software components and a set of development tools. The main purpose of the framework is to facilitate building multi-agent autonomic systems based on a set of toolkit and an agent platform. In this context, an agent is a software entity that performs different tasks, ranging from dynamically configuring applications from the library of software components regarding user requirements or automatically managing systems assigned to that agent. Planner4J is a framework designed to build and to reuse the planning components in applications. It provides a set of common interfaces that are essential for solving planning problems, such as actions, domain and plan descriptions as well as planning problem definitions. Planner4J is comprised of two major parts: the Planner4J-Core module and Planner4J-Classical module. A set of interfaces is defined in the core module, whereas the implementations of those interfaces are contained in the classical module. The implementa-

tion of the planner is based on the classical STRIPS-based [FN71] planner. The implementation follows the principle of the state-space searching technique and could solve the planning problems formulated by PDDL1 (the first version of the language PDDL). The advantage of separating planning interfaces and implementations allows a developer to implement different planners, whilst the interfaces to the users persist. Nevertheless, the type of planners that could be implemented is limited to the planning by heuristic searching. This limitation is caused by the static definitions of the interfaces. The Planner4J framework is prototypically applied to ABLE in order to enable the planning capability of the agent-building environment.

To summarise, instead of concentrating on the planning and scheduling itself, the Planner4J focuses more on how to facilitate the integration of the planning techniques by providing a set of pre-defined interfaces pertaining to the planning as well as data needed by planning operations. The functioning of the framework is completely dependent on the implementation of the interfaces as well as the system that is tightly coupled to the framework. The framework is, in fact, a software layer, which couples the planning techniques with the underlying target system. Therefore, it could not be properly viewed as a full-qualified planning framework.

Recently Levanti et al. [LR09] proposed another planning-based configuration and management of distributed systems. The work is based on the approach called *Stream Processing Planning Language (SPPL)* [RL05, RL06], which is proposed as an extension to the classical PDDL planning formalism by adding planning language constructs that allow multiple input and output ports for the components. Instead of focusing on the activities or actions, the SPPL concentrates more on the planning of the data streams. In SDDL, an action is an abstraction of a data processing components, which have certain requirements on the input and output data; those requirements are formulated as pre-condition and post-condition in SPPL terms. The planning operations are carried out based on the reasoning on the state of the data instead of the state of system.

Based on the SPPL formalism, the approach from Levanti et al. explicitly uses tags to express the configuration goals; those goals can be achieved by planning the interactions of different components involved, i.e. users of the planning system provide high-level management and configuration goals with a set of selected tags representing features and a final state of the desired system. According to those user inputs, the planning system finds the action plan, which is represented in a directed Acyclic Graph (DAG) form. One of the key concepts of this system is the reusable modules, which consist of *action modules* and *option modules*. The action modules include those executables that could change the state of the system, such as configure, restart and connect of a system component. The option module supplies values for the configurations such as the TCP-port number of a

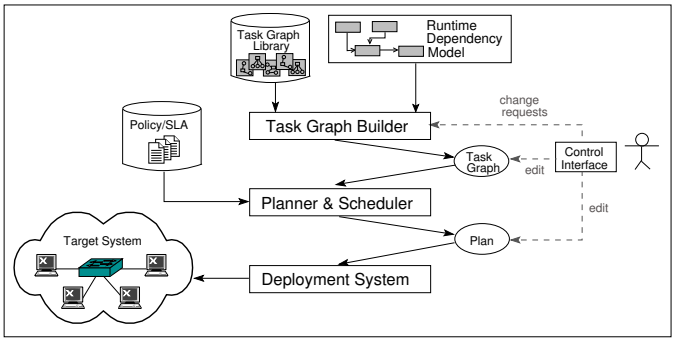


Figure 3.5: Overview of the CHAMPS Framework

service. The planning process completely depends on the state-search-based heuristic from SPPL. The final solution is expressed in terms of workflow and, consequently, the workflow will be handed over to the deployment system for execution. The approach proposed by the authors does not provide the capability to capture the dynamic of the underlying resources and infrastructure, e.g., the solution plan is composed in a offline fashion, which could cause the inconsistency between the solutions and the current states of the infrastructure. Additionally, it is not explained how the tags could be prepared for the planning operations.

3.2.2 Change Management

IT systems need to be changed in order to accommodate the shifting needs of users. Change activities include applying fixes, updating software and upgrading hardware and so forth with goals to enhance the performance of a target system. Change management is a process to manage those activities. Planning is one of the critical steps involved in change management in a highly complex and inter-dependent IT environment. Challenges regarding automating change planning involve consideration of dependencies of systems and components, incorporating policies in the planning process and capturing/reusing the best practices or change receipts. An effective change plan should minimize the time and cost of the change activities and, in the same time, be non-intrusive by reducing the impact of changes related incidents [Com07]. We look into several applications which address those challenges with automated planning feature.

The *Change Management with Planning and Scheduling (CHAMPS)* framework proposed by Keller et al. [KHW+04] undertakes a different approach to planning and scheduling of change activities. Figure 3.5 provides an overview of the architecture of the CHAMPS framework. The core part of the framework consists of two components: the *Task Graph Builder (TGB)*

and the *Planner and Scheduler (P&S)*. The main objective of TGB is to determine the temporal and location constraints for the planning tasks. TGB is also responsible to create reusable workflows from existing dependency model. The workflows are expressed in the form of a task graph with partial order, which transforms a system's state from the current state to the next workable state. The task graphs are meant to be reused as much as possible; therefore, based on the previous deployment experience, a rough estimation of the duration of the partial workflow is possible. This estimation is crucial for the P&S component to make a feasible scheduled plan.

As its name suggests, the P&S component is responsible for the planning and scheduling of the plan. Other than the aforementioned planning approaches, the P&S component of the CHAMPS system focuses on the optimisation rather than composition of the change plans. Based on the task graphs and constraints provided by the TGB component, the P&S component determines the order of the graphs and reasons to solve the constraints. Another task of this component is to bind the abstract plans to the physical resources according to financial and technical constraints. The final change plan produced by the planner will be submitted to the deployment system to perform the actual change operations.

The CHAMPS approach provides a rudimentary form of plan reuse in terms of partial plans expressed in task graphs. Additionally, an interface is provided to the human administrators to allow human interventions to control the change planning operation throughout the whole process. The planning concept presented in the research drifts away from the traditional definition of the planning concept, in which it concentrates more on the optimisation and binding of plans than searching and reasoning for the solution. However, the CHAMPS framework is built on the premise that the underlying targeting system is static and the execution of change plans is an error-free process, which is unrealistic in real-world change operations. Furthermore, no concept related to model change information and change items is provided in the design.

3.2.3 Fault Management

Although the application of automated planning in fault management is promising and the powerful features of planning techniques could relieve operators from arduous planning tasks, our survey in this area surprisingly only results in very few related works concerning applying planning techniques to assist fault management.

Arshad et al. [Ars06a, Ars06b] investigated the fault recovery in distributed systems and proposed a planning-based approach for dealing with fault recovery in such systems. The fault recovery model suggested by the authors is based on the *sense-plan-execute* phases, which is a simplified version of K-MAPE [KC03, GC03] model. In the *sense* phase, the states of the

underlying system are detected by a monitoring process. The monitoring process either listens to the heartbeat signal of the components or sends a control package to the components. In the case that the monitoring process detects failures in the components, the *planning* phase is initiated to automatically find a plan that could recover the fault while minimising the side-effects to other healthy components. Finally, in the *execute* phase, the plan is implemented by assembling the related recovery actions in terms of scripts.

The planning phase is the core part of the proposal. It comprises of four stages: in the first stage, the dependency of the system is analysed to determine the state of the components. The second stage is to find a configuration so that the failed system is expected to work correctly again with this configuration. The third stage targets at calculating a recovery plan that could fulfill the proposed configurations. Finally, the abstract plan is translated into a series of scripts which could be directly executed on the system. The cornerstone of the planning phase is the planner that automatically calculates a solution plan towards the goal configurations. To this end, the authors integrated the LPG planner [GS02], which finds a solution to planning problems based on the local search and graph analysis. During the plan execution, the current system state is constantly monitored [NA05]; in the case of failure during failure recovery, a new plan is computed accordingly.

Whereas the approach suggested by the authors is certainly interesting, there are obvious deficiencies to be observed. The monitoring component proposed is an abstract device; there is no indication given in their work as to how this component could be properly implemented in a real-world scenario. The sense phase is designed to find failures in the system; however, failure detection itself in large-scale systems is a hard problem for management. The authors' approach is merely based on the discussion of the small scenario where failure detection is relatively trivial. The same argument applies to the detection of dependency of the system; as the scale of the observed systems grows, the graph-based approach to describe the dependency of the system may not work. Like most other planning-based applications, the planning problem is defined in a domain file described by PDDL, the issue of how to transform the data from the monitoring data into a planning format is only discussed marginally. For the planning phase, the author proposed to use the LPG planner, which is an efficient planner; however, the strength and features of the planner are not fully exploited by the approach, for example, this planner allows the integration of a cost concept, which could be mapped to time costs or monetary cost of recovery actions, which are essential to the recovery plans. Additionally, the planning domain formalism is very rudimentary; temporal and cost metrics that are important to a recovery plan in a real-world scenario are completely missing. Directly related to that,

no scheduling capability is provided in the suggested approach because the planner is not capable of handling time constraints, which is essential for the scheduling of the plan. Furthermore, the aspect regarding how to integrate the proposed approach into the management infrastructure is also missing.

Gopisetty et al. [GBJ+08] designed an automated planning architecture for the purpose of storage provisioning and disaster recovery. The architecture has two separated planners: a provisioning planner and a recovery planner. The provisioning planner resolves the storage requirements from users and finds a set of configurations and corresponding implementation details, whereas the recovery planner is responsible for finding application-level recovery actions in cases of service failure. As the recovery planner is directly related to our research, in this section we will discuss the recovery planner instead of investigating the whole architecture. The recovery planner consists of several components collaborating together to fulfill the functionality: the *discovery engine* is designed to gather static information from the servers, storages, networks and so on that are associated with the storage service. Furthermore, it collects data on the software applications and their configuration from those associated devices that are involved in the service. The *knowledge base* is a repository which persistently stores the best recovery practices and suggestions from operators and the system designer. The knowledge is in the form of a recipe template, which describes best-practices such as configuration options and replication techniques that could be applied in cases of storage failure. The *match maker* component finds a set of replication technologies to fulfill the requirements of recovery. The match maker module computes the solution based on the pre-determined requirements or profiles from the administrators for the specific services. It takes such information as input and searches the knowledge base for the qualified solutions. There are two types of searches involved in the operations: searching in the solution templates for the popular strategies that could match the requirements or searching in the technology category. Finally, the search results are consolidated and combined as solutions by the optimisation and orchestration modules.

Although automated planning technique is not mentioned directly, the authors applied a searching-based method to find a set of suitable actions according to the templates in the knowledge base, which mimics the classic planning operations. The concept of a knowledge base is, on one hand, designed very close to real-world operations; on the other hand, it is a new construct that helps system designers to automate the storage management operations. The approach allows users to explicitly express their requirements on the recovery. However, this architecture is solely designed for the purpose of provisioning and recovery of storage systems; this limits the generality of the architecture. Additionally, the approach does not provide any scheduling possibilities for the recovery actions; in cases of multiple faults,

where resources for recovery may be severely limited, such feature is required in order to compose a feasible plan. The constraints are only partially considered by the plan optimiser. Although the concept of a knowledge base is proposed there is no discussion on how to model the different knowledge in such a repository, for example, best practises on recovery and technical recipes.

3.2.4 Grid Computing

To cope with the intricate tasks of managing large-scale services based on Grid computing infrastructures, one of the main challenges is to deal with the increasing size and complexity of not only distributed system infrastructure, but also the applications that run on top of it. As we mentioned in the Grid scenario, management of Grid infrastructure and applications today is growing beyond the human capability to cope them; issues such as efficiently composing the workflows for the scientific applications, addressing the dynamics of the underlying systems in the runtime as well as tackling the faults in computation jobs and infrastructure are all need to be addressed in the further research of Grid computing. Andrzejak et al. [ARSS05, AMF+08] identified and suggested that Grid need to be more adaptable and autonomic so as to address the dynamics in both infrastructure and applications. The authors proposed automated planning as one of the feasible ways to assist human operators and users to deal with the intricate planning and decision making process of Grid management.

Blythe et al. discussed the possible role of automated planning in Grid computing [BDG+03] and showed how the abstract scientific workflows are mapped onto the Grid infrastructure [DBGCK03]. They further proposed a knowledge-based approach to capture management knowledge and select computing resources and applications. The approach is based on the heuristic search and the purpose is to assist scientists to compose efficient workflows for data processing in the Grid environment [BDGK03]. In this series of work, the authors identified and mapped the scientific workflow composition problem in a Grid computing environment as the planning problem. The goal is to automate the intricate workflow composition procedures, which usually require knowledge from diverse areas, as much as possible and to relieve the burden and excessive knowledge requirements for the users of the system while the quality of the workflow is still guaranteed. The defined goals are end-data products specified by the user, who communicates with the planning system with a set of application-level descriptions of the desired final data-product. Based on the user inputs, the applied planning mechanism finds a feasible data processing workflows regarding criteria such as performance, reliability and resource usage. The knowledge base is a component that contains a set of operators, which are constructs that represent application elements that process the input data and pro-

duce data-products. The pre-conditions of the operators express the data dependency of that component and the post-conditions are the descriptions of the resulted data. The operators could also contain other relevant information as well, for example, the requirements for the physical resources that run that particular element. The description is formulated in a declarative language similar to PDDL. The planning procedure is the consumer of the knowledge and user descriptions described previously. The search process for the solution plan involves *local heuristics* and *exhaustive search*. The local heuristics is responsible for searching recursively for the steps to fulfill sub-goals that are generated locally as part of the planning objectives. After a step is chosen locally, a backtracking procedure is initiated by means of the exhaustive search to check if the current plan is still feasible with regard to the current choice. The control rule is applied to represents the local heuristics explicitly in the planner. This planning techniques is based on the *PRODIGY Planning Architecture* from Veloso et al. [VCP+95].

The approach from Blythe et al. has been modified and integrated into the Grid infrastructure [DKM+02] for the *Laser Interferometer Gravitational-Wave Observatory (LIGO)* project [AAD+92] at the California Institute of Technology. The goal of the LIGO project is to observe the gravitational wave, which was predicated by Einstein's theory of relativity. The integrated AI planner called *Pegasus* [GDB+04, GRD+07] is implemented based on earlier work by Blythe et al. to help scientists compose high-quality Grid job workflows to process massive data produced of the experiments. The concept of planning knowledge base is embodied as a pervasive knowledge base in the planning system applied to the LIGO project, which contains information such as policy, resource index, declarative knowledge on applications and components in the Grid infrastructures. Upon a user's request, the Pegasus systems builds a goal and current initial states, which it passes to the planner as input. The integrated planner accordingly finds a plan. Once the solution is found, the plan is transformed into a directed acyclic graph (DAG) for the execution sub-system of the middleware.

Several deficiencies could be identified in the proposed approach: first, it does not support the scheduling of the solution; temporal constraints are not considered in the model. Second, the approach is built on the premise that the data-processing on the components never fails, which is a relatively unrealistic assumption, since a Grid infrastructure comprises a large quantity of software and hardware components as well as inter-connections, which increases the risk of failure during its operation. In cases of failure, the Pegasus approach does not provide mechanisms to perform re-planning and recovery from the current failure. Third, the policies pertaining to planning tasks are not incorporated in the planning phases, for example, the policy regarding virtual organisation that authorises the use of a certain set of resources. Finally, plan reuse is not supported by the suggested approach.

To build a plan based on the reusable partial plan is a viable approach to increase the efficiency of the planning process, since the planner does not have to build a solution from scratch.

3.3 Evaluations of Existing Approaches

In this section, we present an overview of a comprehensive evaluation of the related work reviewed in the previous part. The evaluation is conducted according to the requirements we derived in Chapter 2 and is classified into several categories. Fulfillment of the requirements are represented by different symbols in Table 3.1 (on page 82), respectively. The symbols in Table 3.1 carry the following semantics:

- ●: the issue is addressed and corresponding discussions are provided in the reviewed work. The solution provided adequately fulfills the given requirement;
- ◐: the issue is addressed marginally, no in-depth discussion is evident or perceivable in the reviewed work or no suggestions for the solution is put forward to address the raised question;
- ○: the issue is not discussed at all. Even if the criteria are directly related to the particular related work, in this case, no data or discussion addressing those criteria are provided in the work available for this review.
- n/a: not applicable, this symbol denotes those requirements that are not relevant to the particular related work.

Note that, although we covered a wide range of research work in different management areas, the requirements are nevertheless generic enough to represent the common *sine qua non* to a framework with planning capability in IT management.

Having reviewed the state-of-the-art of the applications that feature automated planning techniques, we are now eligible to draw the following conclusions and to sketch a landscape of this particular research:

- Our review shows that AI-based planning is a promising field that is yet to be explored by the IT management research community. Since planning in general is an integral part of almost every management discipline, the planning techniques could solve many types of management problems. On the other hand, however powerful the planning techniques may be, they cannot replace human administrators in management operations, but they can be considered as a viable method to assist and augment the decision process in operations. Currently,

the application of AI-based planning in management area is concentrated more on configuration management and change management. Although there is potential, planning in fault management, especially fault recovery, has not been sufficiently represented and addressed in the current research landscape. This is exactly the gap that this dissertation is trying to bridge.

- Scheduling of a composed plan is often overlooked by most approaches, although a plan schedule is an essential part of planning. Frequently, planning activities are stopped when a plan is found; the question of when each step of the plan can be operated is often ignored. Furthermore, since the reviewed approaches concentrate on the planning structure that is only sequential, the opportunities for parallel executions of certain parts of the plan, which could lead to a shorter execution time, are not fully explored. This directly leads to the fact, that most of solution plans are not necessarily optimised.

- Besides very few exceptions, the most suggested planning systems are designed as stand-alone approaches which are isolated from the rest of the available management components. The issues of integration and collaboration with other management sub-systems are rarely addressed in the reviewed approaches. A planning system could be more useful and practical if it could work with existing management components in a cooperative manner, which could lead to several advantages: on one hand, the available system could provide planners with information necessary for the planning process; on the other hand, the planning results could be directly forwarded to the sub-systems that are capable of handling the planning solutions, for instance, implementation of the solution plan by the workflow execution system.

- The most suggested approaches are built on static world models, that means, it takes the initial states of the target system as input of the planner, and calculates the solution based on that snapshot of the underlying system. This ignores the fact that the states of the dynamic systems are constantly evolving; such a dynamic must be considered during the planning process in order to find consistent solutions.

- Almost all approaches we have reviewed so far use some types of automated planners as the core techniques to solve planning problems. However, the planners are used *as they are*; they are not tuned or optimised for the purpose of specific applications, even if a rich set of opportunities is available. Here the optimisation does not imply violating the generality of planner, but simply means to increase the efficiency of the planner. In other cases, the powerful features of the

planners are not fully exploited, even if they have potentials to perform better.

- The knowledge base for planning operation is frequently regarded as a sort of abstract component that is similar to an all-knowing oracle. Given the importance of this component, most of the reviewed work unfortunately failed to explicitly address the knowledge base. This includes what type of knowledge is necessary for planning and how such knowledge could be represented and stored. An information model for such knowledge is still missing. For the planning process, the actions, abstract control rules, constraints etc. are mostly expressed in PDDL form, which is developed for experts who have knowledge of automated planning and AI. However, we cannot expect that the actual users of the planning system have the capability to understand such formulations, and this fact could prevent the practical use of such systems. Therefore, a more approachable way to model the planning knowledge is desired, such that the complexity of such knowledge formulation could be hidden from the users.

3.4 Summary

This chapter provided reviews of the related research work. Given the importance of planning, this chapter began with a detailed overview of the principles of automated planning. We explained the planning theory as well as its different paradigms. In addition to the theoretical discussion, a broad range of planning applications was presented, including robotics, space exploration, pervasive computing, web-service composition, planning of military operations and emergency evacuations. Although such work is not directly related to ours, nevertheless, the problems that the listed approaches tackle reveal analogies to the problem that we are investigating; therefore, they could be regarded as useful references. A more detailed review was given concerning the current status of the applications of the planning techniques in IT management. The review was classified according to the functional areas of management, including change management, configuration management, fault management. The planning in Grid computing is also included. Finally a comparison was given based on the requirement catalogue derived in the previous chapter. Although the catalogue is derived based on the fault recovery planning, the planning problem in IT management shares some commonalities in nature. Based on the comparison, we drew conclusions regarding the current state of research on applications of automated planning in IT management.

Category	ID	Requirements	Feedbackflow	Planit	Planner4J	SPPL	CHAMPS	Arshad[a]	Gopisetty[b]	Pegasus
Framework	F1	Generic Framework	○	○	●	◐	●	●	○	○
	F2	Scalability	○	○	◐	●	◐	○	○	◐
	F3	Usability	◐	○	●	●	◐	○	●	●
	F4	Interoperability	◐	◐	◐	○	◐	○	●	●
	F5	Adaptability	○	◐	●	○	●	○	○	●
	P1	Planning Capability								
Planning & Scheduling	- P1.1	Automated Planning	●	●	○	●	◐	●	●	●
	- P1.2	Generic Planner	◐	◐	●	○	○	●	○	●
	- P1.3	Efficiency on Planning	○	◐	○	●	◐	○	n/a	○
	- P1.4	Plan Optimisation	○	○	○	○	○	○	●	○
	- P1.5	Dynamic Adaptability	◐	●	○	○	○	○	○	○
	- P1.6	Exception Handling	○	●	○	○	○	●	○	○
	- P1.7	Plan Granularity	◐	○	●	●	●	●	●	●
	- P1.8	Handle different levels of Recovery	n/a	n/a	n/a	n/a	n/a	○	○	n/a
	P2	Scheduling Capability	◐	○	○	○	◐	○	○	○
	P3	Constraints Solving	◐	○	○	○	◐	○	○	○
	P4	Policy Integration	○	○	●	○	◐	○	○	○
	P5	Fault Coverage	n/a	n/a	n/a	n/a	n/a	○	○	n/a
	P6	Handle Multiple Information	◐	○	○	◐	●	○	●	●
	P7	Collaboration	◐	○	◐	○	●	○	●	●
Knowledge Repository	K1	Classifying Planning Knowledge	◐	◐	●	●	●	○	●	○
	K2	Knowledge Repository	○	●	◐	○	●	○	●	◐
	K3	Structured Planning Knowledge	◐	○	●	◐	◐	◐	◐	○
	K4	Cooperating with Knowledge Discovery	○	○	○	○	○	○	◐	○
	K5	Modelling of Planning Knowledge	◐	◐	●	◐	◐	◐	●	○
	K6	Encoding & Reusing Planning Methods	●	○	●	●	●	○	●	○
Knowledge Processing	D1	Translating Planning Knowledge	●	○	◐	○	○	○	○	●
	D2	State Representation	◐	◐	◐	○	○	◐	○	○
	D3	Policy Representation	○	○	◐	○	●	○	○	●
Plan Evaluation	E1	Performance Evaluation	○	◐	●	○	○	◐	●	○
	E2	Cost Evaluation	○	○	●	◐	○	○	●	○
	E3	Temporal Evaluation	○	◐	○	◐	○	○	○	○
Workflow Execution	W1	Execution Engine	●	●	◐	●	●	○	●	●
	W2	Coordination of Execution	◐	●	○	○	○	●	○	●
	W3	Compatibility	○	○	◐	○	◐	○	●	●
	W4	Control over Execution	○	●	○	○	○	◐	○	○

Table 3.1: Evaluation Table of Related Work

[a] [NA05, Ars06a]
[b] [GBJ+08]

4

A Conceptual Model of the Recovery Planning Framework

Contents

As a key part of this dissertation, this chapter begins with a discussion of the transformation of an IT service recovery problem into a general planning problem. Such transformation is also referred to as *problem reduction*.

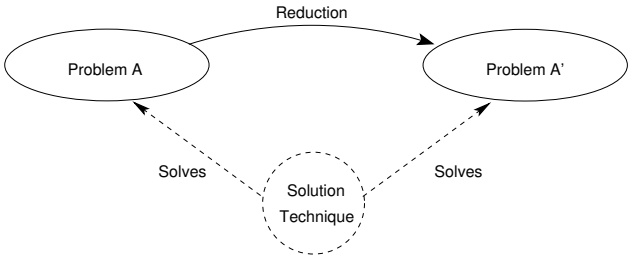


Figure 4.1: Problem Reduction

If a problem class $\mathcal{A}$ is reducible to another problem class $\mathcal{A}'$; it reveals that two problem classes are essentially equivalent with regard to the complexity of computing solutions. The reducibility also implies that the solutions of $\mathcal{A}'$ could provide instances of solutions to $\mathcal{A}$. The purpose of such transformation is to build a theoretical foundation to show that the service recovery problem can be transformed into the general planning problem and, therefore, the strategies which solve the general planning problem, could also be employed to solve IT service recovery problems. Figure 4.1 illustrates the idea of problem reduction.

In order to conduct the transformation operation, we first formalise the definition of an IT service recovery problem which captures essential elements of a recovery operation. The definition of IT service in the recovery problem is partially based on the MNM service model previously introduced. The definition of the general planning problem model is derived from the one proposed by Nau et al. [NGT04]. Further research can then be built based on the theoretical result.

To facilitate automated planning for fault recovery operations, it is essential to specify and formalise the relevant recovery knowledge for planning. To serve this purpose, we first identify the knowledge items and then specify recovery information with formal grammar. As a further step, we analyse the current available planning paradigms in order to make a rational selection of planning techniques that could serve our purposes. As we have shown in the previous chapter, the strength of a planning technique could not be unchained without further elaboration for a particular application field. Thus, adaptation and optimisation of the selected planning technique are required for an efficient planning application. Finally, we propose a design of an automated fault recovery planning framework with detailed discussions on its components.

The question of how to embed the planning framework into current management infrastructure is one of the key aspects to a successful integration. Conversely, the management infrastructure can also provide the planning framework with the necessary information to sustain the planning opera-

tions, for example, state of services and policies that are applicable to the particular recovery operations, etc. To this end, interfaces as well as information models for the exchange of planning information between them are extensively investigated in this chapter. To summarise, the key contributions of this chapter are the following:

- Presenting a formal model of the service fault recovery problem in Section 4.1.1;
- Reduction of the service fault recovery problem as general planning problem 4.1.3;
- Specifying and formalising the recovery-centric management knowledge and information models to facilitate information exchanges between the planning framework and the management infrastructure in Section 4.2;
- A translation mechanism to convert management knowledge into planning knowledge in Section 4.2.3;
- Choosing and adapting an automated planning paradigm for the design of the fault recovery planning algorithm in Section 4.3;
- Proposing a design for the planning framework in Section 4.4;

4.1 Fault Recovery as a Planning Problem

Before transforming the fault recovery problem into a general planning problem, it is essential to formalise the service fault recovery by building a generic problem model as a foundation for the transformation. *A conceptual problem model is simply a theoretical instrument for describing the main elements of a problem.* Although the model may depart from actual computational or algorithmic concerns, nevertheless, it could be indeed effective for explaining the basic concept, clarifying restrictive assumptions and capturing the essential ingredients of the problem. Based on the proposed model, we then cast the fault recovery problem as a planning problem and propose ways to solve it using automated planning techniques. The transformation between those two models follows the *endogenous* and *horizontal* transformation manner, with which the similarities and corresponding abstraction levels of those models are preserved.

4.1.1 Formalising Service Fault Recovery Problems

IT service fault recovery is a multifaceted management operation which involves many influential factors. Building a recovery problem model, which

is generic enough to cover the rich varieties of service types that are available today, is a non-trivial task. With that in mind, the proposed recovery problem model focuses on an abstraction level, which includes essential elements of the recovery operations while intentionally neglecting aspects that are specific to individual service or technology.

To properly define a service recovery problem, it is indispensable to model the IT service as a foundation of the discussion. The fundamentals of the service definition lend themselves well to the MNM service model introduced in Chapter 2. The definition of IT service is given as follows:

Definition 1 (IT Service)**.** *A service $\mathcal{S}$ is a tuple $\{s_j,\ \mathcal{R},\ \mathcal{C},\ \mathcal{F}\}$, whereas:*

- *s_j denotes a sub-service and $\mathcal{S} = \bigcup_{j=1}^{n} s_j$.*
- *$\mathcal{R}$ denote a set of resources r_i, where $r_i \subset \mathcal{R}$.*
- *$\mathcal{C}$ is a set of constraints that represent conditions imposed on a service. Such constraints could, for example, be derived from QoS requirements, management policy or SLAs etc., which are relevant to a particular service.*
- *Dependency functions $\mathcal{F}$, whereas $\mathcal{F} : \mathcal{S} \rightarrow \mathcal{R}$ and a reverse dependency function $\mathcal{F}' : \mathcal{R} \rightarrow \mathcal{S}$, which builds the inversed relationship between the resources and services.*

According to the MNM service model, an IT service is a set of functionalities, which a service provider offers to its customer [GHH+02]. To sustain the desired service functionalities, multiple sub-services are usually employed in the actual service implementation. The sub-services provide partial functionalities that are essential for fulfilling the desired service. In the above definition, a service is a summation of the partial functionalities that are offered by a set of corresponding sub-services, which could be denoted as $\mathcal{S} = \bigcup_{j=1}^{n} s_j$ where n is the number of sub-services that a service is comprised of.

Resources are software or hardware components which realise the specific functionalities required by a service or a sub-service. For example, the DNS service in the first scenario in Chapter 2 requires a DNS software package and a physical machine with a proper operating system (OS) installed to sustain the implementation.

Shared resources are resources that are used by more than one service, i.e. if $S_1 = \{r_1, r_2, r_3\}$, $S_2 = \{r_3, r_4\}$, where S_1, S_2 are two different service instances, then the shared resource is the result of a joint operation on the two sets, $S_1 \cap S_2 = \{r_3\}$. Shared resources are especially important in the virtualised service environment, where, for example, multiple virtual machines share same physical resources.

Constraint $\mathcal{C}$ is a set of rules that imposes conditions and restrictions on the characteristics of a service. The characteristics of services can be represented as a set of variables $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ and a set of corresponding value domains $\mathcal{D} = \{D_1, D_2, \ldots, D_n\}$, where $D_i \in \mathcal{D}$ denotes a set of valid assignments of variables relevant to the service character x_i.

If $\mathcal{X}' \subseteq \mathcal{X}$, that is $\mathcal{X}' = \{x_1, x_2, x_3, \ldots, x_j\}$, and let Ω be a subset of the Cartesian product of value domains relevant to variable x_i ($x_i \in \mathcal{X}'$), $D_1 \times D_2 \times D_3 \times \ldots \times D_j$, then the constraint $\mathcal{C}_{X'} = (\mathcal{X}', \Omega)$; it denotes a set of valid assignments of variables relevant to the service.

The dependency function $\mathcal{F}$ maps a service $\mathcal{S}'$ to a set of resources $\mathcal{R}_{\mathcal{S}'}$ that are associated with the service $\mathcal{S}'$. Since the service $\mathcal{S}'$ may associate with more than one resource and a particular resource may be used by more than one service, the function maps a non-injective relationship between those two domains. The inverse function $\mathcal{F}^{-1}$, in general, does not exist. However, to serve the management purposes, it is often required to have a function that maps the resources to get the services they are associated with. This is particularly important when shared resources are involved in different services, for example, in cases where we make a change to a shared resource, it is necessary to determine the services that depend on the resources and therefore will be affected as well.

With the definition of IT service provided, we now present a model for the service fault recovery; it is an integral part of management, which concentrates on reinstating the fault-impacted services into normal operation so that they could continue to offer the services that are agreed between customer and provider. In this definition of the fault recovery problem model, we concentrate on the essential and generic elements of a recovery activity rather than cover the technological specifics. In fact, given the richness of the manufacture-specific recovery solutions, it makes less sense to include those specifics in our recovery model, since it will deviate from our focus of providing a generic recovery problem model. The definition of the recovery model is given as follows:

Definition 2 (Service Fault Recovery Problem)**.** *A recovery problem model* $\mathbb{R}$ *is defined as a 4-tuple* $\mathbb{R} = \{\mathcal{S},\ \mathcal{Z},\ \mathcal{O},\ \mathcal{T}\}$*, whereas:*

$\mathcal{S}$*: defines a service.*

$\mathcal{Z}$*: denotes states of a service, where* $z_o, z_f \subset \mathcal{Z}$ *are subsets denoting the operational and faulty states, respectively.*

$\mathcal{O}$*: represents a set of recovery options that are currently available.*

$\mathcal{T}$*: denotes the types and root causes of fault.*

If $\mathbb{R}' = \{\mathcal{S}',\ \mathcal{Z}',\ \mathcal{O}',\ \mathcal{T}'\}$ *is an instance of a recovery problem model* $\mathbb{R}$*, the solution to* $\mathbb{R}'$*, given the root cause* $\mathcal{T}'$*, is a set of recovery options* o*, where* $o = o_i, i = 1, ..., n$ *and* $o \subset \mathcal{O}'$*, which fulfills following conditions:*

- $z_f \xrightarrow{o_1,\ldots,o_n} z_o$

- *The solution o satisfies c,* $\forall\ c \in C_{\mathcal{S}}$*, whereas* $C_{\mathcal{S}}$ *is a set of constraints regarding the service* $\mathcal{S}$*.*

The definition presented above provides a generic problem model for the fault recovery of IT services. The target for the recovery operation is a service $\mathbb{S}$, whose explanation is given in definition 1.

In this definition, we use the term *state* to represent the current situation of the observed service. Those descriptions are represented by a subset of $\mathcal{Z}$, which is a set of descriptions of the possible states of the service. Two special sets of states z_f *and* z_o represent the initial situation, which contains the fault (z_f), and final desired state (z_o), in which the service is reinstated. The state change is caused by the implementation of a recovery option $o_i,\ o_i \in O$; in this definition, we assume that each option o will cause changes of state. Note that at this stage, we consider the service state as a general concept and intentionally decouple it from specific implementation techniques. Those details will be discussed in the chapter on implementation in details.

Symbol $\mathcal{O}$ denotes a knowledge-base structure which contains recovery options, methods or actions. Operationally it is a part of the management knowledge that is essential to fault recovery procedures and encodes recovery-related information. Each data item in this repository is an individual action or set of methods that could be applied to appropriate fault scenarios. We assume that each recovery action will cause the change of states in the service $\mathcal{S}$, i.e. $z_i \xrightarrow{o_i} z_j$, where $z_i,\ z_j \subset \mathcal{Z}$ and $o_i \in \mathcal{O}$.

Given a definition of a service recovery problem $\mathbb{R}'$, we look for a set of actions in a proper ordering that transits the current state into the objective state, in which the expected service could be offered. Furthermore, the solutions should satisfy any constraint that may be required by that particular service. We are particularly interested in two types of constraint: *temporal* and *cost*.

Generally speaking, temporal constraints are conditions that are related to time. To be more specific, we distinguish three different types of temporal constraints in this work: *instance*, *interval* and *temporal relations*. An instance of time refers to some specific time points such as $t_1, t_2, ..., t_n$ and an interval refers to a period of time that leaps between two time instances, e.g., $[t_i, t_{i+1}]$. Finally the ordering constraints express the sequences between actions, e.g., $t_i \succcurlyeq t_j$ or $t_i \parallel t_j$. The capability to process temporal constraints is essential to solve real-world fault recovery problems which usually involve different time-related conditions such as scheduling of the recovery actions.

The cost constraints are conditions that are concerned with operational costs of a holistic recovery procedure. To recover a service from faults, it is possible that more than one set of options are available to achieve identical recovery objectives and, obviously, they are associated with various monetary requirements. Under such circumstances, the cost constraints

provide a possibility to select the right recovery options so that the solution would be in alignment with the provided management policy regarding cost of the operation.

As mentioned earlier, a valid solution to a fault recovery problem not only transits the current faulty state of a service into an operational state, it should also satisfy all applicable constraints. More details on the constraints process are given in the following sections.

In this section we define an IT service model relying on the MNM service model. A generic problem model for service fault recovery is then formalised based on the given service definition. The proposed problem model concentrates on the essential elements of a generic recovery operation rather than dig into technical specifics. Based on the problem model, we also define the solution model for the fault recovery problem, which is a sequence of recovery options that obey the applicable constraints. The capability of constraint processing is a particularly important aspect in dealing real-world fault recovery problems. In the following section, we discuss the general planning model, which is partially discussed in Chapter 3. The ultimate goal of those discussions is to provide a theoretical background for the problem reduction between the general planning problem and the service fault recovery problem. As long as the reducibility is shown, we are then justified in formally solving the fault recovery problem with methods that solve the planning problem.

4.1.2 A General Planning Model

In this section, we give an extended discussion on the formalisation of the general planning problem. The fundamentals of the planning model and some well-known planning paradigms were introduced previously in Chapter 3, hence this part of the discussion focuses on the further details of the planning model including its representations and applicable restrictive assumptions.

A planning endeavour of any kind is primarily concerned with achieving a set of predetermined objectives by taking actions in a particular order. To briefly recap on the definition, a classical planning problem P [NGT04] can be simply modelled as a tuple:

$$P = (\Sigma,\ s_0,\ g)$$

This model includes a state-transition system (also known as a *discrete event system*) Σ to model the underlying dynamic systems, an initial state of the underlying system s_0 and an objective state g, where $s_0,\ g \subset S$.

The planning problem model requires a sub-model to describe the underlying system. In this definition, the system is represented by Σ. It can be conceptually considered as a Kripke structure [CGP99], which is frequently used to describe and verify a dynamic system. In planning terms, the system

model is also referred to as the *planning domain*. A transition of the state in Σ is caused by implementations of some state-changing actions a_i, which are a subset of the available action set A, e.g., $a_i \in A$. A function γ notes such a transitional relationship between states:

$$\gamma \xrightarrow{a_i} S \times S$$

A planning problem is said to have a solution when there is a set of actions in proper orderings, which transform the current state of the underlying dynamic system s_0 into the desired states g, i.e.

$$\gamma(s_0, a_1) = s_1, ..., \gamma(s_{n-1}, a_k) = g$$

Restricted Planning Model

Given the formal definition of the planning problem model, we list and discuss applicable common restrictive assumptions [Nau07] regarding this conceptual model. Note that the presented assumptions are very strong for modelling complex real-world problems; nevertheless, those restrictive assumptions help us to concentrate on the essence of actual planning problems and abstract away details that are currently still unimportant to the problem definition. To adapt the model to the complex real-world planning problems, relaxations could always be made on those assumptions and corresponding techniques could be developed to solve the planning problems.

- **Assumption 1**: The planning domain Σ has a finite number of states.
- **Assumption 2**: Σ is a fully observable system. In this context, the knowledge of system states is completely available.
- **Assumption 3**: The behaviour of the system to a selected action is deterministic, meaning the implementation of a selected action will lead to another system state deterministically.
- **Assumption 4**: Static Σ. The state of the system would not change unless a selected action is implemented on the system.
- **Assumption 5**: The planner handles only the objectives that are specified in the goal state description g. The objective should be known to the planner and clearly specified in proper forms.
- **Assumption 6**: Plans are expressed sequentially.
- **Assumption 7**: Plans are done offline. This assumption is closely related to Assumption 4, which assumes that the system model is not changing during the plan composition phase.

- **Assumption 8**: Implementation of any selected action is without duration, i.e. execution time of the action is only given implicitly.

The above assumptions impose strong conditions on the planning problem model due to the fact that to emulate problem-solving skill in planning problems is a non-trivial task; therefore, constraints have to be placed on the problem model in order to make the problem tractable.

Among the presented assumptions, it is easy to notice that the majority of them (assumptions 1-4) impose constraints on the description of the planning domain Σ. Depending on the nature of target systems and the planning solutions we seek, planning problems could be categorised as *trajectory planning* or *discrete planning*. Those two different categories of planning problems fulfill the above assumptions differently.

Trajectory planning is frequently termed motion planning, which refers to a planning process that builds plans for non-linear dynamic systems and drives that system from initial state into the goal state. To solve such kind of planning problems, the planning domain, which includes the state space of the target system, has to be modelled in continuous time. The number of states in state space is uncountably infinity (e.g., velocity of the robot, rotation of the motors to control the movement of the robot). For the purpose of motion planning, the operation environment including obstacles and other rigid bodies has to be represented as geometrical models as well. The state space of a motion planning problem is called *configuration space* and a solution to the motion planning problem is a continuous path in the configuration space.

Compared to trajectory planning, the planning problems we investigate are more of discrete nature and the corresponding planning domains are formed differently as its counterpart. This class of planning problem usually has a countable finite number of states in Σ. Differential equations or geometric models are not required for the description of the discrete planning problems. Nevertheless, *Assumption 1* could be relaxed to include countable infinite number of states of a system, however, this relaxation requires that optimisation must be made during the solution-finding phase in order to increase the efficiency.

For both trajectory and discrete planning problems, *Assumption 2*, *Assumption 3* and *Assumption 4* imply that the underlying system is completely observable and static; the system or planning domain only changes its states when selected actions are implemented. They also suggest that all of the changes that are implemented in the system have deterministic effects. In other words, the planner works on a snapshot of the target system and assumes that the current system state is preserved during the planning. A static and isolated operation environment is indeed much easier for a planner to handle than a highly dynamic one due to the reason that the planner does not have to constantly update its knowledge of the systems

and deterministic outcomes of the selected actions raise no exception and thus avoid the replanning. Nevertheless, such idealised planning situations do not prevail in real-world applications, for which exceptions and constant changes of the underlying system are commonly to be expected. To relax those assumptions, the underlying planning domain Σ has to be modelled as a stochastic system, which is a non-deterministic state-transition system that allows probability to transitions between system. This technique is often called *planning under uncertainty*. With this extended model, the planning problem is then transformed into the *optimisation problem*, in which the planning algorithm is searching for the optimised solution in terms of a pre-determined utility function.

The planning problem model that allows stochastic system model is defined as:

$$\Sigma = (S,\ A,\ P)$$

where the definitions for S and A are similar to its deterministic counterparts; the symbol P is a probability distribution $P_a(s'|s)$, whereas a is an action; $a \in A$ and both s and s' are elements of state S, $s, s' \in S$. It denotes the possibility that if action a is implemented on the system with state s, then a will lead to state s' with that probability. If there exists an action $a \in A$ and state $s' \in S$, such that $P_a(s'|s) \neq 0$, then we have:

$$\Sigma_{s' \in S} P = (s, a, s') = 1$$

Techniques such as the Markov Decision Process (MDP) [Bel57], which are designed to deal with non-deterministic systems, probability and partial observability, could be employed to deal with planning under uncertainty.

Assumption 5 assumes that the solution to a planning problem is limited to reaching the goal state defined in $g, (g \subset S)$ and no extended goals are supported during the planning operation. The term *extended goals* refers to the conditions or constraints that are imposed on the state transitions *during* the planning operation, for example, the planner could be instructed not only to reach the objective, but also to avoid some particular state transitions that are not desirable. This assumption could be relaxed by integrating utility functions into the planning algorithm to allow the evaluations of actions and transitions. The planning problem with extended goals could be regarded as optimisation problems and the solutions to such problems is a set of state transitions that not only fulfill the desired objective but also considers the predefined constraints or conditions during the planning.

Assumption 6 limits the structure of the solution plan to be expressed sequentially. However, the plan in a real-world situation could take different forms, for example, conditional plans under non-deterministic situations and parallelism of certain steps of the solution to reduce the total implementation time of the plan. There are different ways to relax this assumption, for

example, by integrating scheduling techniques into planning operations, we could recognise the overlap of the planning steps and allow parallel plan structures.

Assumption 7 assumes that time is implicit in planning operations; temporal conditions or time constraints are abstracted away with this assumption. Each action and each transaction between states is considered instantaneous. However, under some planning situations, especially time-critical planning applications, relaxation of this assumption must be done. By allowing time durations of each action step and transaction, the classical planning algorithm could be extended to handle the explicit temporal constraints. Relaxation of this assumption also allows, for example, to apply scheduling techniques to planning as mentioned above.

Assumption 8 is closely related to *Assumptions 2, 3* and *4*, where a static domain is presumed for the classical model of planning problem. This assumption limits the planner to working on a *snapshot* of the underlying system and ignores the dynamic of that system. Since the planner works offline, no update of the system state is required during the planning phase. Turning an offline planning problem into an online planning problem is a non-trivial task; it requires the planner to constantly update the current situation s_0 of the system and a plan may be invalided and the planning process may constantly be interrupted due to the dynamics of the underlying system. Recently different approaches have been proposed for dealing with online planning problems, for example, Ross et al. [RPPCD08] investigated the online planning for a partial observable Markov process and Chan et al. [CFR+07] proposed an approach for applying online planning for the real-time strategy game.

To conclude, this section discussed the assumptions imposed on the restrictive planning problem model. The restrictions imposed on this model are very strong. They reduce the observed model to its essence, thus helping us to view the problem more clearly. Nevertheless, different strategies are allowed to relax those assumptions and make the problem model more practical. For each of those assumptions, we discussed the techniques that are applicable for reducing those restrictions. The downside of the relaxation is that they make the planning problem more difficult to solve, although those relaxations in different combinations are essential to properly model and represent real-world problems.

4.1.3 Problem Reduction

Based on the problem models built previously, we conduct a formal discussion in this section of the reduction between the planning problem and the fault recovery problem. The reduction operations could provide several useful insights [GJ79, CLRS01, Ski08] for the problem that we are trying to tackle:

- With reduction, we can show that the observed problems actually belongs to the same class of computational complexity as the source problem.

- Reduction can formally justify that techniques which solve the known problems class could also be applied to solve the problem for which we currently seek solutions.

Theoretically a problem P is said to be reducible to another problem P', if any instance of P can be formulated as an instance of P', which can provide solutions to problem P [CLRS01] by calling its routine procedures. If the transformation could be done by a function f within polynomial time, it is said that P is polynomial reducible to P', denoting $P \leq_f P'$. It implies that problem P is not more difficult to solve than P' and the solving procedures of P' can be used to solve P [Ski08].

The general idea of the reduction for our purposes is to take the planning problem model in its restricted form as the source problem $P_{planning}$, whose solutions are well-known, and try to map the instances of fault recovery problem $P_{recovery}$ onto the domain of $P_{planning}$.

The general planning model $P_{planning}$ is defined to include $\{\Sigma,\ s_0,\ g\}$, which are essentially the inputs for a planning problem. Σ denotes the underlying system including a set of state descriptions S and an applicable action set A. The transformation function γ implements the selected action and transforms the current state into the next state as indicated by the post-conditions of the selected action. Traditionally states of a system are usually described using some planning languages, for example, the Planning Domain Description Language (PDDL) [GHK+98], which predefines a set of proposition symbols mapped to each object of the observed system. The descriptions of system states are then expressed in a conjunctive form of the propositions. Theoretically if there are n symbols used in the description language, then there are $|S|$ possible subsets in disposal, where,

$$|S| = \Sigma_{k=1}^{n} \binom{n}{k}$$

Note that S includes *all* possible combinations of propositional descriptions.

Set A is a set of state-changing actions or operators that are applicable to states of the system. Each action has a set of pre-conditions that express requirements, under which this particular action could be chosen. Meanwhile, the expected effects of the action implementation is expressed in terms of post-conditions, which denote the specific changes done to the system by the selected action. The implementation of actions is realised by the function γ, which calculates the state transactions:

$$\gamma(a_i, s_i) \rightarrow s_{i+1},\ where\ s_i, s_{i+1} \in S$$

The transition function γ maintains an overview of the current status by means of deleting those predicate symbols that are changed by the action (denoted as $pcondition^-(a_i)$) and adding new symbols that are caused by this action (denoted as $pcondition^+(a_i)$), the new state s_{i+1} after implementation of the action can be formally expressed as follows:

$$s_{i+1} = (s_i - pcondition^-(a_i)) \cup pcondition^+(a_i)$$

Having the essential input for the planning problem presented, a planning algorithm is then used to find one or more paths in the state space between the current state and the goal state. Note that in this stage we consider the planning algorithm merely as a sub-program to be called with the given problem descriptions P. Techniques introduced in the planning part in Chapter 3 can be used as such sub-programs to solve planning problems. Also note that the planning techniques we discuss are domain-independent, which means they are not tailored or optimised for any particular application domains and are universally applicable. Note that for the purpose of proof, the planning problem in question is in the restrictive form with assumptions discussed in the previous section.

According to our definition, the input for a service recovery problem consists of several elements: a service $\mathcal{S}$, which is a target system under observation; a set of state descriptions $\mathcal{Z}$; a set of available recovery options $\mathcal{O}$ and types and root causes of a problem $\mathcal{T}$. Given an instance of service recovery problem $\mathbb{R}'$, we seek a set of recovery steps in $\mathcal{O}$ that could change the current fault state to the expected state; in this context, a normal working service, which could deliver the expected functionalities to the users or customers of that particular service.

There are two possible ways to show the reducibility between P_{source} and P_{target}:

- *Turing Reduction.* Assume that there is a well-known algorithm $\mathcal{A}$ that solves the source problem P_{source} and then the *Turing reduction* process tries to integrate the algorithm $\mathcal{A}$ into the algorithm $\mathcal{B}$ of the target problem P_{target}, which is still to be constructed. $\mathcal{A}$ is regarded as a sub-routine of the algorithm under construction and each time $\mathcal{B}$ tries to solve a target problem, the sub-routine $\mathcal{A}$ is called to provide at least a partial solution for the target problem. The final complexity of $\mathcal{B}$ is then equals to the complexity of $\mathcal{A}$ plus the complexity of the rest of $\mathcal{B}$. With the Turing Reduction, it is shown that $\mathcal{A}$ for the source problem could be used to solve target problem, and the complexity to solve P_{target} is *at least* as hard as solving P_{source}.

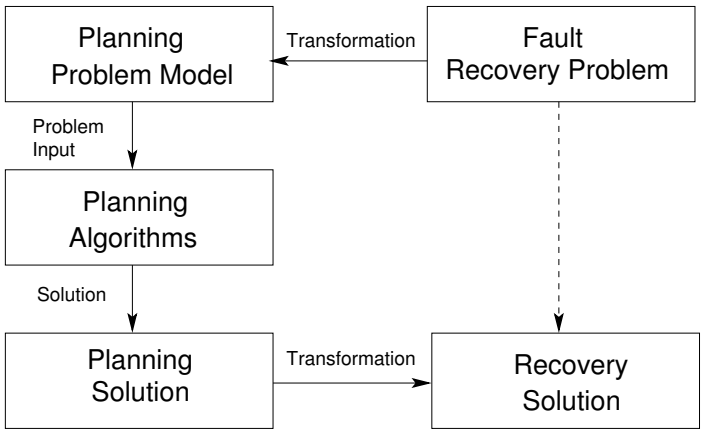


Figure 4.2: Problem Reduction

- Another more straightforward way is to transform every instance of the new problem P_{target} into an instance of P_{source}, and show that an algorithm $\mathcal{A}$ to P_{source} can directly operate on the instance of P_{target}. The solution provided by $\mathcal{A}$ is then the expected solution of P_{target}. The complexity of solving the target problem is equal to the complexity of transforming between instances (which is polynomial) and the complexity of algorithm $\mathcal{A}$. This type of reduction operation is also known as *Many-One Reduction or m-Reduction.*

To prove is the relation between the planning problem and service recovery problem, $P_{recovery} \leq_p P_{planning}$, meaning the fault recovery problem is polynomial reducible to the planning problem. The key steps of the proof follow the *Many-One Reduction* approach, with which each instance of the recovery problem is transformed into an instance of the planning problem and thereafter it is solved using selected available planning procedures. Figure 4.2 illustrates the reduction approach between the planning problem and the fault recovery problem with solution extraction. The general many-one reduction approach involves transformation of the problems under observation into decision problems in order to cover the diversity and heterogeneity of problem types. For our proof, however, this step is not required, because both of the problems are identical in terms of solutions and their structures.

Given the problems $P_{planning}$ and $P_{recovery}$, our goal is to prove the following theorem:

Theorem 1. *A service recovery problem is reducible to a planning problem and can be solved with a planning algorithm.*

Proof. In order to show the reducibility between the two problem models, as a first step we cast a service recovery problem as a planning problem.

A classical planning problem $P_{planning}$ with the set-theoretic representations of a planning domain [NGT04] can be denoted by a tuple $\{S, A, \gamma\}$. Let $\mathfrak{L}$ be a set of proposition symbols that can be used to describe states in S. If $|\mathfrak{L}| = n$, then the problem domain has a total of $\Sigma_{k=1}^{n} n\texttt{Choose}k$ possible states (which is equal to $2^{|\mathfrak{L}|} - \{\emptyset\}$). An action (an operator) $a_i \in A$ has pre-conditions and post-conditions that are both subsets of S. Function γ applies the selected actions and transforms the current state into a new state. Furthermore, s_0 and g denote the initial and pre-determined final states and they are subsets of S as well.

Suppose we use a set of proposition symbols $\mathfrak{L}'$ to delineate the states of a particular service, thus every state $z_i \in \mathcal{Z}$ is a subset of $\mathfrak{L}'$. The size of the complete state space therefore is equal to $2^{|\mathfrak{L}'|}$, including an empty state.

According to our previous definition of the service recovery problem model, the set $\mathcal{O}$ denotes possible recovery steps or operations that could be applied to repair a particular service. Each recovery operation has its requirements on a set of states, under which this operation could be chosen. Furthermore each operation is also associated with expected effects on the service states that this operation could cause. If o_i^{pre} and o_i^{effect} denote the requirements and effects of a particular recovery step, respectively, they could also be represented by subsets of $\mathfrak{L}'$, i.e. o_i^{pre}, $o_i^{effect} \subset \mathfrak{L}'$. The current state of the service containing faults is denoted as $\mathcal{T}$, which is a subset of state definition $\mathcal{Z}$. A goal state $z_g \subset \mathcal{Z}$ is a configuration of states which denote a repaired and operational service state.

With the above definitions, we could solve the fault recovery problem using classical algorithms for solving automated planning problems, such as state-space planning or plan-space planning etc. Specifically we construct a state space for the planning algorithms using the state definition of $\mathcal{Z}$ of the observed service. We use $\mathcal{T}$ as one of the inputs for the selected planning algorithm. It denotes the current state of the service that contains a fault. Additionally, a goal state z_o is any configuration of states that can be used to describe the service as functional. The selected planning algorithm operates on those inputs for the solutions which can transform the current faulty service state into a predefined state which satisfies z_o. Planning solutions that are returned by the algorithm constitute selected sequences of recovery actions that are given by $\mathcal{O}$. □

There are several notes regarding this proof that are worthwhile to be discussed here:

- For this proof, we use a subset of elements from the service fault recovery problem model, which are essential to cast the recovery problem as a planning problem. However, to achieve a more realistic planning process, further parameters from the problem definition could be added, for example, constraints $\mathcal{C}$ on the service could be used to fur-

ther optimise the searching process of the planning algorithm. Since we only considered the viability of applying the planning algorithm to solve service fault recovery problem, further parameters for the optimisation of the planning process are left out of the current proof.

- To extend the above discussion, the optimisation of the planning algorithm is beyond the focus in this context. The planner under consideration is solely used as an abstract problem-solving paradigm to operate on an input instance of the service fault recovery problem. In a more realistic usage context of automated planning, various optimisation approaches on the planning algorithms themselves are often used to improve the efficiency of planning procedures and the quality of the solutions.

- The assumptions on the restricted planning model discussed in the previous section are applicable to the planning in this proof to avoid unnecessary complications. However, as we will discuss in the later contributions, relaxations on these assumptions could be done for the constructions of a planner to make the planning more applicable to real-world planning problems.

Based on the previous discussion, we are now able to draw the following conclusions:

- The service recovery problem is reducible to planning problems; the two problem models share a very similar structure, which facilitates the transformations between their problem instances. This therefore justifies that using planning algorithms to solve the service recovery problem is a viable approach.

- Depending on how planning problems are represented, the complexity of automated planning ranges from class P to NP-Complete and in the class NEXPTIME under certain conditions; formal proofs are given in [ENS95, Byl91, LGM98, NGT04]. Therefore it is unrealistic to make a general statement on the complexity of automated planning. This implies that given a planning problem, we could not always expect the algorithms to find an optimal solution. As we use the planning approach to solve fault recovery problems, the planning algorithm may not always return the optimal solutions; however, in real-world application scenarios, a "good-enough" solution can serve the purpose in most cases as well.

- There are opportunities to optimise the planning process by using parameters that are specific to fault recovery problems. The purpose of using such parameters is to reduce the plan search spaces and guide the planning process.

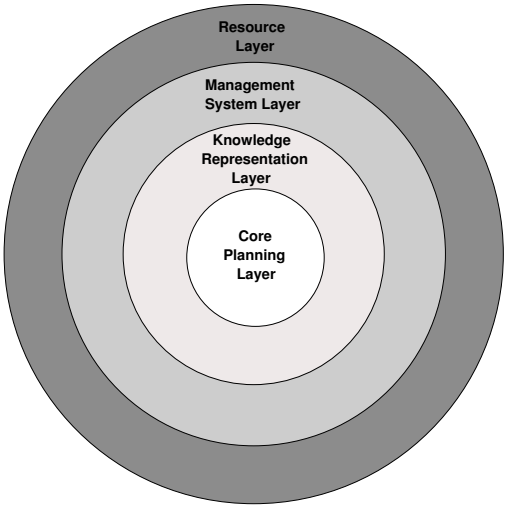


Figure 4.3: Relationships between the Planning Algorithm, Knowledge Representation and Management System

4.2 Recovery Knowledge

Instrumenting an AI-based planning mechanism to solve practical management problems is not an easy task; there is obviously a long way with many obstacles in between that need to be solved. One of the major issues that has to be tackled with paramount importance is to systematically specify, represent and encode management knowledge into a data model that could be understood and consumed by the planning mechanism. Among others, this issue is put forward due to the reason that planning knowledge is a crucial foundation for the operations of the automated planning algorithm. Even a good planning mechanism cannot compute viable solutions without properly formated high-quality input of planning knowledge obtained from different management sources.

To approach this research issue, a path of management information flow must be first of all clearly defined. Due to the fact that the planning mechanism is not intended to be a stand-alone management approach, it needs to collaborate with other underlying management components, such as monitoring system and execution system of management actions, in a cooperative manner. To serve this purpose, the data path for the planning inputs should cover the complete spectrum from management system to planner and vice versa.

To provide a more clear view of information process procedures and, at the same time, to follow well-established system design principles, we suggest a layered view to represent different abstraction and processing levels of the management information. Figure 4.3 illustrates those layers and

relationships between them. The knowledge processing procedures are herewith divided into four layers: *resource layer, management system layer, knowledge representation layer* and *core planning layer*. Each layer requires different degrees of capabilities of information processing.

Resource Layer The first layer represents resources or components, on which IT services are implemented (see definition 1). The resource layer consists of software or hardware entities as information sources that generate raw data upon different events during the service operation. As a sheer information provider, this layer does not provide any data processing capability. However, depending on the management system residing on this layer, it may be necessary to store management-relevant information locally.

Management System Layer This is the layer where management tools are deployed to perform corresponding management activities. A management system should be capable of collecting management data from underlying resource layers and supporting primary operations such as data correlation or filtering. Management decisions and corresponding management actions are determined based on those collected data. On the other hand, the management layer provides interfaces to interact with the underlying system; any management decisions and resulted actions could be capable of having effects on the resource through this layer.

Knowledge Representation Layer The information collected by the management system layer should be properly modelled and represented to support management operations such as fault recovery planning. Management decisions are made based on the available management knowledge, which could be categorised into two types: system knowledge and knowledge of the management operations. System knowledge includes information such as the current state of the system, dependencies of the system components etc. The latter type of knowledge is comprised of management actions which could be applied to the observed system.

Core Planning Layer The very inner ring is the core planning mechanism which bases its operation on the planning knowledge provided by the previous layers. The planning core combines different inputs and computes the proper solution management actions accordingly. As soon as an action plan is composed, the planning mechanism passes the solution back to the management system for the execution on the underlying system.

Knowledge representation in this context implies systematic ways to codify the knowledge that is relevant to the planning operations. Before we can

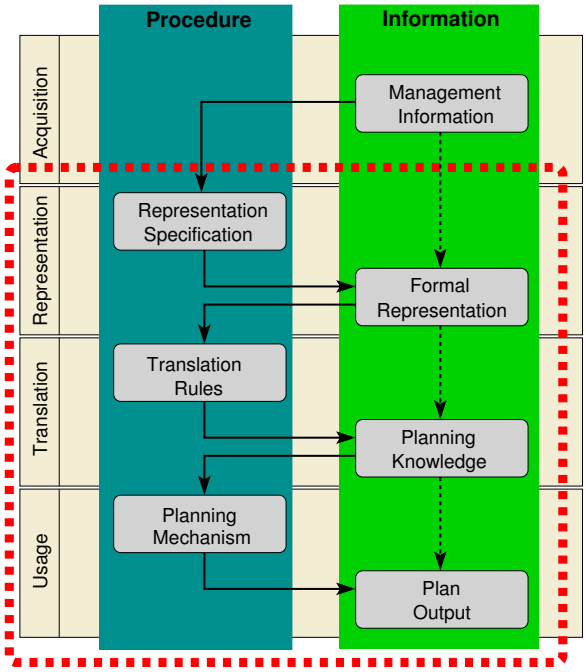


Figure 4.4: Dataflow: from management information to planning knowledge

proceed to plan automatically, formal means to encode as well as to represent planning knowledge have to be established in order to serve as a basis for later planning operations. This builds a bridge for data exchanges between the AI-based planning mechanisms and existing conventional management systems.

Figure 4.4 further refines the data path for planning operations from infrastructure to planning algorithm. The path for data processing is organised into four major phases which deal with knowledge processing in different layers:

- Acquisition: The knowledge acquisition phase focuses on extracting recovery-relevant information from management systems, for example, the monitoring data relating to the components of the IT service in question and management policies that are applicable to the recovery activities. Whereas abundant types of knowledge can be available from various management sources, a relevant set of planning-centric knowledge types has to be first determined and defined in order to facilitate the specification of such knowledge for the further processing. Additionally, the knowledge acquisition process has to deal with the heterogeneity of data formats rooted from various management resources, because each management system may apply different pro-

prietary data formats to represent the same piece of information.

- Representation: This phase intends to represent the acquired knowledge in a proper data model, which is independent from proprietary data formats from underlying management systems or technologies. The purpose of data representation is to separate the technology-independent representation schema from acquired management information which is technology dependent. To serve this purpose, formal specifications of the information model for the recovery knowledge have to be established.

- Translation: After the recovery knowledge has been formally specified, the next step is to translate such knowledge into the proper data format that could be consumed by the planning mechanism as input to produce meaningful plan solutions. To assist the operation of this phase, a translation mechanism has to be devised to emit the plan-centric knowledge. The translation process is guided by a set of translation rules. On the other hand, a solution plan elaborated by the planning mechanism also needs to be reversely translated into execution forms that could be understood by the execution component due to the fact that the execution component in our design is external to the planning core.

- Usage: The usage phase is comparatively straightforward. In this phase, the planner takes the prepared knowledge as input and feeds it to the planning core as the basis for the computation of solutions. The results of the operation are one or more viable recovery plans which could fulfil the recovery objectives. The resulting plans are forwarded to the execution components of the underlying management systems for execution.

The management systems or infrastructures provide the knowledge representation mechanism with the necessary information obtained from the managed components correspondingly. The knowledge representation mechanism formalises the obtained information and supplies the recovery planning algorithm with input. The planning algorithm operates based on these inputs and finds solution plans accordingly. The solution plans are delegated to management systems for further processing, e.g., execution of the plans. The *knowledge* itself, however, is a generic and pervasive term that needs to be further narrowed down to specific definitions. Therefore, the first step towards a proper representation is to determine what kinds of knowledge are specifically required for the fault recovery planning operation. In this section, we first identify types of knowledge that are essential to recovery planning, then we discuss issues on rules of how such kinds of

knowledge could be represented and converted into a planning-conforming data format.

We classify the term *knowledge* into two categories: *infrastructure knowledge* and *management procedural knowledge*. As its name suggests, infrastructure knowledge refers to the awareness of the information regarding the target *infrastructure* itself. It is inherently bounded to the components of a particular set of infrastructure including software, hardware, networks etc. Additionally, the complex relationships between those components are also one of the key items of the infrastructure knowledge which are frequently expressed as *dependencies*.

By *management procedural knowledge* we refer to knowledge documenting how management operations could be implemented, how managed elements could be controlled to achieve the desired effects or functionalities and what the workflows or processes are to perform management operations. *Management procedural knowledge* has more of an abstract nature and the acquisition of such knowledge is frequently empirical and based on human inputs. For instance, the procedures on how to re-configure an application server is one of many examples of such knowledge. The infrastructure knowledge provides the necessary information on the participating components such as server machine, network devices, software frameworks, operating systems etc. Furthermore, the control options of those components can also be made available through the infrastructure knowledge. Management procedure knowledge is mainly concerned with ways and processes of using those control options to compose a functioning application server according to some rules that may be applicable to the service. Additionally, in a real-world scenario, management policies in forms of constraints and conditions must be considered in the configuration process, thus they can also be regarded as part of management procedural knowledge. Effective and efficient management operations rely on obtaining as well as applying both infrastructure knowledge and management procedural knowledge in order to achieve the designated management objectives.

4.2.1 Specifying Recovery Knowledge

Having discussed the types and characteristics of management knowledge, this section intends to identify and analyse the specific knowledge that is highly relevant to the fault recovery planning operations. It addresses the knowledge specification issues as shown in Figure 4.4.

The following types of knowledge are specified for the purpose of fault recovery:

Managed Entity Managed entities are fundamental elements of infrastructure knowledge. A management entity could be a physical or abstract component or any other information items required to be managed.

According to the formal IT service definition presented in the previous section, an IT service consists of one or more managed entities as resources to sustain the actual functionality offered by the service. Note that a managed entity could be *atomic*, meaning management actions could be directly applied on this entity, for example, an Apache HTTPD software is an atomic entity, because implementable management actions such as restart, reload, stop, etc. could be directly applied to the entity. Contrarily, a 3-tier web application service is an abstract service entity, because in order to operate on it, management actions have to be resolved into specific implementable action sequences, for example, to stop a web host service, we may want to firstly backup current data, check-pointing the on-going transactions, redirect current traffic, stop the web front-end (HTTPD), and stop the web back-end (application server and DB).

Notification Notifications are generated events by different sources, e.g., by monitoring tools, or timely information provided by a human administrator. The purpose of notification is simply to convey the event information of a managed entity, such as a server going offline or a web application container has been configured. All those events imply state changes in the underlying system. Functionally, a notification is a carrier of management events from/to underlying managed entities.

State Information State-based representation is one of the most prevalent ways to model behaviours of a particular service [EMME+06, EKK+06]. To precisely describe states of a service, two types of information are required: *operational state* and *inter-dependencies*. The operational state information reveals the state of a service, e.g. *running, stopped, halting, restarting*, etc. Such information could usually be obtained either from monitoring data or by means of automated service state discovery mechanisms [ZMN05, FHT10]. Complementary to operational state information, inter-dependencies depict structural information of the observing service, e.g. server `srv1` is running `Linux OS` with `Apache` installed. Details on service dependency are given in the next section. Whereas most planning algorithms base their operations on states of the underlying system, the knowledge of states of a service can be mapped naturally into states in terms of automated planning. In IT operations, monitoring information gathered from a service could provide an overview of the state of that particular service.

Dependency Large-scale IT services often have a myriad of inter-dependencies between their components. As discussed previously, dependency information reveals how components are related to each other in order to provide desired functionalities. The knowledge concerning depen-

dencies of IT services could either be maintained manually or determined by automated means. In order to compose a proper recovery plan, various dependencies must be taken into consideration, e.g., dependencies between managed entities or dependencies within the managed entity. Dependency information could be obtained in different ways, for example, if CMDB is provided, dependencies between components and services are usually readily available there; on the other hand, automated mechanisms to determine dependencies information are also available [BKK02, Has01].

Management Actions A management plan is usually comprised of sequences of possible actions which lead to the desired objectives. Observing from a state-based point of view, an action either transits the current state of the particular service into a new state or alters conditions of a state internally. Generally management actions are designated with pre-conditions, which determine under what circumstances an action could be carried out, and post-conditions, which reveal what effects the actions could cause on the system and how a particular management action could influence the current states of a system or service.

Management Constraints Management constraints are special conditions, to which a service must comply. They are usually specified by management policies. Constraints could be expressed either numerically or in words, for example, constraints on time or monetary conditions are usually expressed using numbers and others constraints, for example, *stand-by failover server must be behind a firewall* can only be expressed in words.

Recovery Plan A plan in the context of IT management can be defined as a set of implementable actions in a particular order. A recovery plans is designed to achieve certain recovery objectives. A *feasible plan* from a management perspective is a plan that not only achieves designated management goals, but also fulfills management constraints that may be applicable to that particular activity. Additionally, a plan could have many forms regarding the orderings of the involved actions; in this work we consider the sequential and parallel form of plan actions, which are the two most frequent orderings of a management plan.

Management Objective In our context, a management objective describes a goal that is to be reached through implementation of a well-devised management plan. Specifically a management objective can be represented by a set of desired service states that a service needs to have after the execution of a solution plan. From a current state description of a service, e.g., the state in which the service is interrupted by faults, the management

objective provides a goal state in which the impacted service is reinstated and a composed plan describes a specific way to reach that state.

4.2.2 A Language for Recovery Knowledge

Automated planning mechanisms usually have their own representations for planning inputs; the previously discussed planning-relevant information obtained from management systems needs to be properly modelled to meet the planning requirements. In this section information models for the aforementioned planning knowledge are formally presented and discussed and then we propose a set of rewriting rules to convert management knowledge into the format that conforms to the planning.

Managed Entity

Definition Entities are virtual or physical components and information items that need to be managed. They are identified by an unique name. For practical purposes, names are organised in namespaces correspondingly to management domains. The description of a management entity should also contain the information on types which further describe the property of a managed entity, for example, whether an entity is a physical server, a software application or a configuration file etc. A further required property is the domain of the entity. By domain, we mean one or more services, to which an entity may belong. Note that a managed entity can be comprised of other entities; such intertwined relationships between them are coded in the dependency information. A managed entity without antecedent entities builds a *top-level plan domain*. Correspondingly those managed entities without dependents denote objects upon which management actions could be directly implemented. Note that depending on how management entities are related, they could be further identified as *abstract* or *atomic* entities.

Formal Grammar

```
entity  ::=  "entity{"
                "DOMAIN:" domain ";"
                    "ID:" identifier ";"
                  "TYPE:" entityType ";"
            "DEPENDENCY:" dep ";"

"}";

domain     ::= STRING;
identifier ::= STRING;
entityType ::= STRING;
dep        ::= ServiceDep;
```

Description An entity is described by its `ID`, which is a unique name for an entity in a given service domain, e.g., a web service domain or an email service domain. The information on service scope is given by the `DOMAIN` entry, which reveals the affiliation of the managed entity to the associated service domain. `TYPE` denotes the class of an entity, e.g., a DNS application can be an entity class. In the following example, the managed entity `dns1_app` belongs to the `web_service` domain and has type `dns_application`. The dependency of `dns1_app` is encoded in `dns1_app_dep` (see discussions on dependency on page 110 for more details).

Example

```
entity {
    DOMAIN: web_service;
        ID: dns1_app;
      TYPE: dns_application;
DEPENDENCY: dns1_app_dep;
}
```

Management Notification

Definition Notifications are events generated automatically, e.g., by monitoring tools, or timely information provided by a human administrator. The purpose of notification is to convey the management information of a managed entity. In addition to the actual notification message, the information model should contain the source of the notification, which is an instance of a managed entity. Furthermore each notification should be tagged with types, e.g., a notification could convey an state-changing event, or alert message etc. Finally the time point at which this event is generated should also be explicitly given in the notification.

Formal Grammar

```
notification ::= "notification {"
                 "ENTITY:" entity.ID ";"
                 "TYPE:" STRING ";"
                 "EVENT:"  STRING ";"
                 "TIME_STAMP:" DateString TimeString ";"
                 "}";
```

Description The `Entity` field gives information on the source entity, from which the notification comes. The name of the entity should conform with the ID of the identity given in the corresponding field of the `entity`. The `TYPE` entry denotes what kind of notifications is in question. The `EVENT`

entry denotes the associated event that is encapsulated in this notification. Finally the TIME_STAMP shows the time point when the notification is created. In the following example, the notification is generated by the DNS entity.

Example

```
notification {
    ENTITY: dns1;
      TYPE: state;
     EVENT: started;
TIME_STAMP: 2010-03-25T12:34:56.101;
 }
```

Another example shows a different type of event which notifies a state transition event:

```
notification {
     ENTRY: dns1;
      TYPE: transition;
     EVENT: started crashed;
TIME_STAMP: 2010-03-23T11:24:36.101;
}
```

State Representation

Definition Chandy and Lamport [CL85] defined a (global) state of a service (as a distributed system) as a combination of its sub-components and channel state. This definition can be applied well to IT services, since our notion of an IT service includes one or more sub-components which provide partial service functionalities, the corresponding states of the sub-components need to be aggregated to represent the state of the service in a holistic way. In the previous section (on page 104), we defined state knowledge to include two types of information: *operational state* and *dependency*; in this section we mainly concentrate on the representation of operational information.

Formal Grammar

```
EntityState ::= "state {"
                "ID: "nameOfEntity ";"
                "PRECONDITION:" pre-condition ";"
                "CURRENT_STATE: " stateOfEntity ";"
                "CONTROL_OPTIONS: " act ";"
                "[BEHAVIOR_ENTRY:]" API ";"
```

```
                "[BEHAVIOR_DO:]" API ";"
                "[BEHAVIOR_EXIT:]" API ";"
                "}";

nameOfEntity   ::= STRING;
pre-condition  ::= EntityState;
stateOfEntity  ::= STRING;
            act            ::= {MgtAction};
API             ::= {returnVal} identifier "("{parameters}")";
returnVal       ::= VOID|INTEGER|FLOAT|BOOL|STRING|LONG;
identifier      ::= STRING;
parameters      ::= returnValue paraID;
paraID          ::= STRING;
```

Description Operational states are always bounded to certain managed entities. As discussed in the section on managed entities (page 103), we distinguish between descriptions of states of atomic entities and those of abstract service state. The managed entity associated with the description of state information is given in the `ID` entry. To represent the state information of an atomic entity, we could use predicates to show the possible states of the entity, e.g., `running(serv1)`. Such information is expressed in the `CURRENT_STATE` in the definition language. State information should also document pre-conditions of possible state transitions which express the conditions of changes from the current state into other potential states. As soon as a set of pre-conditions of the current state is fulfilled, management actions could be triggered. State-changing actions are given in the `CONTROL_OPTIONS` field. Additionally, actions upon entering the current state, those within the current state and when exiting current state could be noted in the `BEHAVIOR_ENTITY, BEHAVIOR_DO, BEHAVIOR_EXIT`, respectively. We define the current state of an abstract service as aggregations of states of underlying atomic entities which belong to the domain of this service.

Example The simplified web-hosting service scenario given in Chapter 2 on page 26 is comprised of multiple sub-components (or sub-services) including a web server, database and DNS which are connected by networks to provide the desired service. The general state of this service, according to the above definition, could be defined by combinations of the individual state of those sub-components. The following example shows a state definition of a web server in running state.

```
        state = {
             ID : httpd;
   PRECONDITION : server_1 LinuxOS Apache ConfigFile;
```

```
 CURRENT_STATE : running;
CONTROL_OPTION : httpd-stop(server_1, Apache)
                 httpd-halt(server_1, Apache)
                 httpd-restart(server_1 Apache);
BEHAVIOR_ENTRY : start(ApplicationContainer)
                 connect_DB(dbs_1);
BEHAVIOR_EXIT  : graceful_stop(Apache)
                 disconnect_DB(dbs_1);
   }
```

Dependency

Definition Dependency information conveys the knowledge of how managed entities are interrelated to each other to provide aggregated service functionalities. Additionally, not only inter-entity dependency is of interest, intra-dependency is crucial for the management decision as well, as it contains the information on how different components within a managed entity are related together, for example, the operational environment on which certain type of applications are depending.

Formal Grammar

```
ServiceDep ::= "dependency {"
               "DEP_ID: " identifier ";"
               "DOMAIN: " domain ";"
               "TYPE: " typesOfDependency ";"
               "ANTECEDENT: " antecedent ";"
               "DEPENDENT: " dependent ";"
               "[PREREQUISITY]: " prereq";"
               "[COREREQUISITY]: " corereq";"
               "[EXREQUISITY]: " exreq";"
               "[DESCRIPTION]: " description";"
               "}";

identifier        ::= STRING;
typesOfDependency ::= STRING;
domain            ::= STRING;
antecedent        ::= {entity};
dependent         ::= {entity};
updatedOn         ::= dateString;
prereq            ::= {entity};
corereq           ::= {entity};
exreq             ::= {entity};
description       ::=  STRING;
```

Description The dependency information model provides views on how entities of the service in question are related to each other. Dependencies between components are labelled with unique identifiers, which could be numbers or descriptive strings; the entry `DEP_ID` serves such a purpose. The `DOMAIN` entry illustrates the domain of a service related to this particular dependency information. Also information on the types of dependency is essential; typical types of dependencies are, for example, inter-system dependencies or intra-system dependencies. `ANTECEDENT` and `DEPENDENT` fields describe the dependencies between components. In addition to this mandatory information, four optional sections are given for further auxiliary information regarding dependency. The `PREREQUISITY` contains the required components and services that are needed *before* a particular entity could be put into use, e.g., by installing an application a targeted physical server must be in running state and a certain type of operating system (OS) must be installed, and during the runtime a database query must be finished and returned before further processing is possible. `COREREQUISITY` denotes the components that must be run in parallel with the entity in question, for example, an application server; a database server must run in parallel with the web server in order to provide the desired service. `EXREQUISITY` gives information on the components that must be excluded. Finally the `DESCRIPTION` section provides supplementary information on the dependency.

Example

```
dependency{
       DEP_ID : dns_service;
         TYPE : intra;
       DOMAIN : web_service
   ANTECEDENT : LinuxOS Server01 BIND9 confFile;
    DEPENDENT : named;
 PREREQUISITY : LinuxOS Server01 BIND9 confFile;
COREREQUISITY : LinuxOS Server01;
  DESCRIPTION : "This provides intra-dependency information
                 of the dns service in web service domain.";
}
```

Management Actions

Definition A management action is an embodiment of a direct-executable management step. It is the basic ingredient of a management plan, which could include one or more management actions in a particular order. The management action exists in the form of, for example, system command or management scripts composed by human administrators of a service with certain management purposes.

Formal Grammar

```
MgtAction  ::= "action {"
               "ID: " identifie ";"
               "DOMAIN: " domain ";"
               "ASSOCIATED_MO: "association ";"
               "PARAMETERS: "paralist ";"
               "RETURN_VALUE: " rt_value ";"
               "EXCEPTION_HANDLING: " exceptions ";"
               "LOCATION:" loc ";"
               "SCHEDULED:" time ";"
               "DESCRIPTION: " desc ";"
               "CREATED:" date ";"
               "}";

identifier  ::= STRING|INTEGER;
domain      ::= STRING;
association ::= STRING[STRING];
paralist    ::= {type STRING};
rt_value    ::= VOID|STRING|INTEGER|FLOAT|BOOL;
exceptions  ::= MgtAction;
loc         ::= URI;
time        ::= TimeString;
desc        ::= "preconditions{" precond "}" \
                "effects{" effects "}";
precond     ::= EntityState [EntityState];
effects     ::= EntityState [EntityState];
author      ::= STRING[STRING];
date        ::= DateString;
```

Description The above grammar describes management actions which are implementable to specific components. An action is identified by the `ID` field with a unique name. The `PARAMETERS` entry gives a list of typed parameters that may be necessary for execution of the actions. Values that may be returned by an executed action are provided by the `RETURN_VALUE` field. An executed action on certain components could be either successful or failed. In cases of execution failure, the triggered exception may cause activations of further management actions; therefore the `EXCEPTION_HANDLING` field is defined recursively to address the exceptions that may be raised during the execution. The `LOCATION` field gives information on where actions could be possibly located. `DESCRIPTION` determines the pre-conditions and post-conditions of a particular action. The pre-conditions denote under which service state that particular action could be selected and used; it determines the conditions, which must hold before the execution of the action.

The post-conditions are effects that the action could cause, e.g., the effect of the reconfiguration action is that the DNS service is again in a configured state. Both pre- and post-conditions are described with `EntityState`, which was discussed previously. The last two fields `AUTHOR` and `CREATED` are both self-explanatory. They fulfill only a descriptive purpose rather than functional. The following example shows the description of a management action to reconfigure the DNS application.

Example

```
action{
                ID: dns-config;
            DOMAIN: email_service web_service;
     ASSOCIATED_MO: dns-application;
        PARAMETERS: configFile
      RETURN_VALUE: BOOL;
EXCEPTIONS_HANDLING: mgt_action_dns_exception;
          LOCATION: repo://mgt_server/dns/dns_apps/;
         SCHEDULED: 2010-04-01T02:00:00;
       DESCRIPTION: preconditions{
                       dns-application.stopped \
                       dns-machine01.running \
                    }
                    effects{
                       dns-application.configured
                    };
            AUTHOR: Joe;
           CREATED: 2010-01-01;
 }
```

The goal of the management action `dns-config` is to configure a DNS application. A DNS service may belong to the Email service domain or web service domain. The management entity this action is associated with is DNS software applications; other actions may be associated with DNS server machine. The `dns-config` action requires a configuration file as its parameter and may return a boolean value according to the result of execution. With an unsuccessful execution, an exception may be raised, which could be caught by other management actions that are especially designed for the exception handling. In this case, `mgt_action_dns_exception` is called upon exception. Additionally, the management action could be found in the URI given by the `LOCATION` field. The instance of the action could be a system command or a script that fulfills this particular purpose. The time point, at which this action is scheduled to run is given in the `SCHEDULED` entry with a specific date and time. In order to execute the configuration action,

certain pre-conditions must be met; in this case, the DNS application must be in the stop state and the hosting server of this service must in a running state. If executed successfully, the asserted effect of this action is that the DNS-application is in a configured state.

Recovery Plan

Definition A recovery plan is a sequence of management actions in a particular order that transit the current state of an IT service which contains a fault to an expected operational state. The operational state is defined by the plan objective, which is a set of pre-defined state descriptions of that service.

Formal Grammar

```
MgtPlan ::= "plan{"
"ACTIONS:" action ordering ";"
            "SCHEDULED_EXECUTE:" exec_time ";"
            "CREATED:" crt_time  ";"
            "ESTIMATED_TOTAL:" totalTime ";"
            "ASSERTED_OBJECTIVE:" obj ";"
            "}" ;

action   ::= MgtAction;
ordering  ::= "≺" | "‖" ;
exec_time ::=  DateString TimeString;
crt_time  ::= TimeString;
totalTime ::= TimeString;
obj       ::= objectives;
```

Description The `ACTION` field contains one or more management actions. The execution sequence is determined by the ordering of selected actions. In this work we consider two types of orderings of actions: sequential execution ($\prec$) and parallel execution ($\|$). `SCHEDULED_EXECUTE` reveals when the plan is scheduled to be implemented. `ESTIMATED_TOTAL` signifies the estimated total execution time of the plan. Finally, the `ASSERTED_OBJECTIVE` reveals the asserted final state of the targeted service after the executions of the plan.

Example

```
plan{
        ACTIONS: dns-stop ≺ dns-config
                ≺ dns-validate
```

```
                ≺ dns-start
                ≺ dns-test;
      SCHEDULED: 2010-04-02T03:00:00
        CREATED: 2010-04-02T02:30:00;
ESTIMATED_TOTAL: 00:05:30;
ASSERTED_OBJECTIVE: dns-application.running
                    dns-application.updated
}
```

Plan Objectives

Plan objectives should denote the desired final state of the observed state. They express the goal a management plan should achieve after the execution.

Formal Grammar

```
obj ::= "objective{"
        "ASSERTED_STATE:" EntityState|{EntityState} ";"
        "[TIME_REQUIREMENT:]" TimeString ";"
        "[COST_REQUIREMENT:]" cost ";"
        "};"

cost ::= INTEGER|FLOAT;
```

Description The `ASSERTED_STATE` denotes the desired asserted states; it can include one or more `ServiceState` defined previously. Optionally, requirements regarding the time and cost constraints could be explicitly expressed in the last two fields respectively. The plan objectives are either pre-determined or given by administrators during management activities.

Example

```
objective{
    ASSERTED_STATE: dns-app.running dns-app.updated
                    dns-app.onServer01
    TIME_REQUIREMENT: 00:25:00
}
```

4.2.3 Translation to a Planning Language

The reason for developing a knowledge translation mechanism is to reduce the impact of AI-planning technology on the planning system's users, who are, in most cases, non-AI specialists. Although the details of planning operations could be hidden from users, nevertheless, they are still required to

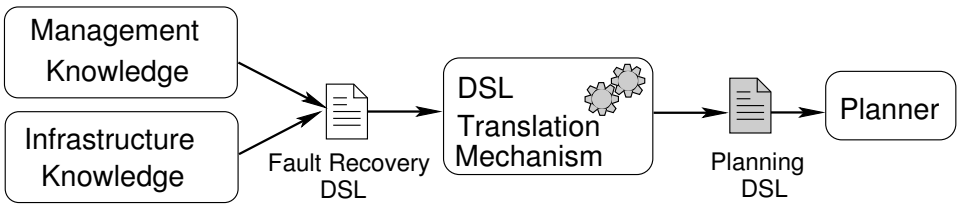


Figure 4.5: An overview of the translation process between DSLs

maintain, extend and continuously improve the planning knowledge. Therefore, a direct application of planning parlance would set a great barrier for them to grasp the system and to perform the required tasks.

On the other hand, the planner also needs to collect the system information from the infrastructure to support its operations; such information needs to be processed into a proper language before it can be handed to the planning algorithm. To bridge these gaps between the planning system, users and infrastructure information, we need to develop a mechanism that translates between them.

The knowledge translation mechanism is similar to a formal language translator, which interprets language **A** into language **B** without semantic bias. The formal languages are generally very high-level languages that are built and tailored for special purposes. They are also frequently known as *Domain-Specific Languages (DSL)*. Figure 4.5 providess an overview of the translation process with input and output. Management and infrastructure knowledge is aggregated and described in the recovery DSL using formal grammars.

Before we design a translation architecture, it is essential to understand the source and target languages; therefore, in this section we start by analysing both the input and output DSL of the translation mechanism. The focus will be however on the output DSL.

Input DSL In the previous section, we provided the formal specifications of the knowledge that is required during a recovery process. As the input DSL for the translation mechanism, such information needs to be properly parsed into another language that a planner can understand. The input DSL in fault recovery knowledge is a collection of information from different resources using the specifications defined above. Since the specifications and formal grammars with examples have already been given in previous sections, we will not go into great detail regarding the input language.

Output DSL The output DSL is a language specified to describe the knowledge necessary for the recovery planning operations. According to the

theory of automated planning discussed in section 3.1 in Chapter 3, a classic planning language should at least contain the following information:

- Action: describing actions that can change the state of the observed systems. The description should include at least a name of the action with the pre-conditions and post-conditions.
- Transitional system: representing behaviours of the observed system and its transitions between states.
- Planning goal: describing goals of a planning operations. It is usually represented by a subset of system state descriptions.
- Initial state: similar to the planning goal, only it illustrates the current system state.

An aggregation of all the knowledge items shown above is frequently referred to as the *plan domain*; a data structure that encompasses plan domain is usually called the *domain file*. Depending on the capabilities of particular planners, an advanced plan domain description can include much more information than shown above. For example, information on the constraints and preferences can be added to the plan domain description. If the planner is capable of doing un-deterministic reasonings, the probabilistic information on the success of a particular plan actions could also be inserted into the plan domain.

With the research and development of planning technologies, planning languages have continuously evolved and been improved by the AI community. The research effort resulted in an increasing degree of maturity of the languages in terms of expressiveness and completeness. New language constructs such as constraints and preferences have also been proposed to further extend the expressiveness of the planning languages. Meanwhile discussions and attempts regarding the standardisation of the planning language have also been made within the research community.

Due to the above reasons, designing a completely new planning language from scratch seems like *re-invent the wheel* and counter-productive; however, building a translation mechanism between the aggregated planning knowledge and planning DSL is still an essential procedure in designing a real-world planning application. Therefore, in this research, we intend to use an existing planning language as output of the translation mechanism and propose a translation process between the input and output DSLs. In the next section, we analyse several major types of planning languages.

Plan Domain Specification and Representation

The ultimate goal of a plan domain specification language is to formalise as well as represent and convey the planning knowledge to the planning

algorithm. Different approaches have been developed by the AI research community to formally specify the planning knowledge. These approaches can be categorised as follows:

- **Set-theoretic representation**. This approach uses a set of proposition symbols $P = \{p_1, p_2, ..., p_n\}$ to represent states of the world W, i.e $W \subset P$. Both the planning goal (objective) and initial state are subsets of the world W. The description of action is written based on the propositional logic, which usually has the form:

  $$a_{action} = \{pre(a), effect^{+}(a), effect^{-}(a)\}$$

  An action has pre-conditions and has two kind of effects: $effect^{+}$ adds new states to the current state of the world and $effect^{-}$ deletes the states that are no longer valid, i.e. states changed by the action. This representation scheme has the disadvantage that a large space for propositions is needed even when describing a very simple planning problem.

- **Classical representation**. This kind of representation allows the use of derivatives of first-order logic and planning operators to model the world and planning problems. States of the world are represented as sets of logical atoms, for example, the term:

  `Installed_OS(AppSer1,Linux)`

  conveys the information that Linux is installed on the application server (`AppSer1`) as the operating system. Note that terms must be *grounded*[1] and *function free* in the classical representation. Furthermore, the *closed world*[2] assumption is applicable to this kind of description logic. An action description in classical representation is comprised of pre-conditions and effects. However, instead of propositions, classical representation uses predicate logic with following the conditions:

  - Action name and list of parameters to uniquely identify the action.
  - Pre-conditions are conjunctions of positive and function-free literals.
  - Effects are conjunctions of function-free literals that describe post-condition of a particular action. Negative literals are allowed.

[1] Variables are not allowed in the terms.
[2] Unknown conditions are all false.

An action description example using classical representation:

```
action: Backup_Data(SRC, DST, File, size)
    precond: exist(File, SRC) ∧
             capacity_avail(DST, size) ∧
             network(SRC, DST)
    effects: exist(File, DST) ∧
             backup_copy(File)
```

The above example illustrates a very simple description of data backup action. In order to backup a particular piece of data using this action, the conjunction of conditions in the `precond` must be fulfilled, namely the target file must be on the source system (`SRC`) and there must be enough capacity (`capacity_avail`) in the backup system (`DST`). Finally there must be a network connection between the source and destination system. The `effects` field shows the consequences of this action.

- **State-variable representation**. This representation has an equivalent expressiveness to the classical representation. Instead of a relation, its representation uses *state-variable symbols* to denote *state-variable function*. The value of those functions forms characteristic attributes of a system state. Description of actions however still relies on the name, pre-condition and effects. Some attributes may not change during the planning operation; they are usually referred to as *rigid relations*, in other words they are invariants of the planning domain.

```
action: Backup_Data(SRC: sd, DST: da, File: f, Size: s)
    precond:  exist(File: f) = sd ∧ capacity_avail(d) = s
            ∧ network(sa, da)
    effects:  exist(f) = da ∧ backup_copy(f)
```

Theoretically classical representation and state-variable representation have an equivalent expressiveness. A planning problem represented by one of those representations could be translated into the other with linear increases in size of description [NGT04]. The set-theoretical representation is weaker than the other two. Its prohibitive requirement on the space for descriptions of complex planning problem limits its practical value in modelling real-world planning tasks.

Having discussed the representation paradigms of planning knowledge, we investigate several major planning languages that embody the ideas from the above-mentioned description paradigms. Those languages have been widely used in the practical planning systems either of an academic nature or in actual real-world contexts.

Situation Calculus Situation calculus is a predicate-based formalisation of states and actions with corresponding effects on the underlying system. It was originally designed for a logical deduction system for query questions. The system answers questions about the reachability of a goal state from the current state. The state transformations take place through a set of state-changing actions. In a strict sense, situation calculus is not a modelling language for the planning system; however, it provides theoretical fundamentals for the further development of the planning modelling approaches. Situation calculus formulates the state of the world based on the conjunction of first-order calculus. Actions are regarded as functions of the involved entities, for example, `migrate(server_1,server_2)` denotes the action to migrate `server_1` to `server_2`. Actions can also be represented as *schema* with *schema variables*, e.g., `migrate(x, y)` where both variables could be instantiated to constant symbols. In situation calculus, pre-conditions and effects of an action are theoretically described by two kinds of axioms: *possibility axioms* and *effect axioms*.

STRIPS The *Stanford Research Institute Problem Solver (STRIPS)* is a classical description language designed for the purpose of automated planning operations. The representation of the planning knowledge such as state information and plan actions rely on the classical representation rules. STRIPS describes states of the world also based on first-order logic; however, only positive literals are allowed in the state representation. Additionally, it is required that those literals are grounded and function-free. The close-world assumption is applied to the STRIPS. As plan goals are expressed as a set of desired states, the rules valid for state representation are also valid for plan goal representation. Plan actions are described by preconditions and effects. Pre-conditions are expressed as conjunctions of positive function-free literals and effects expressed as conjunctions of function-free literals.

ADL The *Action Description Language (ADL)* is a variant of STRIPS. ADL relaxes some of the constraints of STRIPS to increase the expressiveness of the language. The enhancements are based on the findings of Pednault [Ped87]. Compared to STRIPS, ADL is based on the *open world assumption* and it allows negative literals to appear in the state description. Furthermore, ADL supports quantified variables and typings of the variables in the descriptions of plan goals. Literals are formulated with conjunctions and disjunction. Finally, ADL supports conditional effects in the action description in the form of `A causes L if C`, meaning action `A` leads to effect `L` if the condition `C` is satisfied.

PDDL The *Plan Domain Description Language (PDDL)* is a language standardised by the AI planning research community. Originally in its first version, PDDL was built based on the STRIPS formalism with various extensions. By introduction of each new version, new extensions have been added to the language to enhance the expressiveness.

To describe a planning task, PDDL separates the domain information and the problem description. The domain information is included in a *planning domain file* and the problem description in a *problem file*. The planning domain file is comprised of predicates and action operators, which are reusable for various planning problems *about this domain*. The transition system is denoted as the planning domain file. Furthermore, a problem file includes involving objects, the initial state of the problem and the final desired states. The problem file denotes simply the current instance of the transition system given in the domain file. Syntactically, PDDL has a Lisp-like notation; a domain file has following structure:

```
(define (domain <name>)
(:requirements <req_1> ...<req_n > )
(:types <subtype_1> ...<subtype_n> - <type_1> ...<type_n>)
(:constants <cons_1>...<cons_n>)
(:predicates <p_1> <p_2>...<p_n>)
    (: action_1)
    ...
    (: action_n)
)
```

A plan domain is identified by a descriptive name. The `requirements` field denotes a set of representation features that could be used in the current domain specification. Since PDDL is a very generic declarative planning language, not all features that come with PDDL will be supported by a planner, therefore, it is essential to declare the set of features a planner supports during a planning domain compilation. Some basic features it supports include the following:

- `:strips` denotes that domain specification supports only basic STRIPS-style notation.
- `:typing` allows type names and declarations of variables.
- `:negative-preconditions` allow negation in goal description.
- `:disjunctive-preconditions` allow `or` in goal description.
- `:adl` denotes that the domain uses some or all of ADL features.

The `:types` field declares the types of objects and parameters involved in planning operations. Types of parameters could be defined as: `?x - WebServer`. Parameters of the same type can be written as: `?x ?y ?z - WebServer`. Declaration of object types can be done in the same way. Typed variables allow, for example, flexible definitions of actions. The `:constant` fields defines object symbols that will not be changed in a particular domain. All predicates that may be used in a domain are defined in the `:predicates` fields.

Finally in a domain description file, available plan actions are defined. The action description has its own syntax which is similar to the classical planning description paradigm. As shown in the structure below, each plan action description is identified by a unique name followed by a list of typed parameters. Both pre-conditions and effects are explicitly given using predicates defined in the corresponding domain file.

```
(:action <name>
    :parameters <pmtr_1> ... <pmtr_n>
    :preconditions <pre_1> ... <pre_n>
    :effect  <eff_1> ... <eff_n>
)
```

In addition to the domain description file, the problem description provides information about the current state of an observed transitional system. As shown below, a problem description structure includes a name that identifies the planning problem. The `:domain` field shows the particular planning domain, to which this planning problem is attached. The `:objects` field denotes the scope of all participating elements in this particular planning problem. Finally the `:init` and `:goal` fields denote the initial state and final desired state, respectively.

```
(define (problem <name>)
    (:domain <domain-name>)
    (:objects <obj_1>...<obj_n >)
    (:init <state_1>...<state_i> )
    (:goal <state_1>...<state_j>)
)
```

The above discussion illustrates the essence of the planning domain description structures and the planning problem structure. With the evolution of the PDDL standard, improvements have been constantly proposed and new constructs have been added, which make the language more expressive and have boosted the ranges and types of planning problems that could be modelled by PDDL. These improvements enable new language constructs to be integrated into the language. In the following paragraph, some important extensions are discussed, which are essential to model the fault recovery planning problem domain.

Operators A set of logical operators are supported by action descriptions with PDDL: `and (`$\wedge$`), or (`$\vee$`)` and `not (`$\neg$`)` can appear in pre-conditions, furthermore, logical quantifiers are also allowed to appear in the pre-condition part of an action description, including: `imply(` $\Longrightarrow$ `),` `exists(`$\exists$`)` and `forall(`$\forall$`)`. In action effects, the quantifier `and, not, forall` and `when` may be used.

Durative Actions The durative actions allow temporal concepts such as concurrency to be included in the modelling of plan actions. Two kinds of durative actions can be modelled by PDDL: *discretized-time* and *continuous-time* actions. Both types of action share a similar structure that is comprised of logical changes caused by the implementation of actions. Whereas the continuous-time actions are mostly used to model plan domains where continuously changing values are essential for actions, we will focus on the discretized-time actions. The modelling of discrete durative actions is done by means of *temporally annotated* conditions and effects. Such annotation of action conditions allows to express whether the associated proposition must hold *at the beginning* of the action, *during* the action or *at the end* of the action. Invariant conditions of a durative action can be explicitly expressed using the combination of three annotations. Temporal annotation of effects of an action express whether the effects take place immediately after the execution of the action or are delayed.

```
(:durative-action <name>
:parameters <p1>...<pn>
    :duration <duration>
    :condition <time-conditions>
    :effect <time-conditions>
)
```

The above example illustrates the structure of durative actions. An action is identified by a descriptive name that is unique in the plan domain. A list of parameters is then provided to declare the involved objects and their corresponding variables. The `:duration` field explicitly gives the timespan of an action, for example, `(:duration (?duration 5))`. Finally `:condition` and `:effect` list the conditions and effects, respectively, of a durative action connected by logical operators.

Plan Metrics PDDL allows plan metrics to be expressed in numeric form. This feature is especially useful for plan optimisations where plan qualities can be judged by numeric values such as time costs. Plan metrics can be written in the problem description file with an entry: `(:metric <keyword> <conditions>)`. According to the types of metric, dif-

ferent keywords can be applied in the metric entry, e.g., `minimise, maximise, length ...` etc.

Plan Preferences Conditions that the user of a planner would prefer to be satisfied through the execution of plan actions are referred to as *plan preferences*. However, the users are also ready to accept the fact that preferences are violated for various reasons, e.g., costs of the actions would be prohibitive if preferences are considered or due to some internal conflicts etc. Therefore, plan preferences can also be regarded as a kind of *soft constraints*. PDDL allows plan preferences to be added as a part of plan goal in the problem description. If preferences are applicable to all plan problems, they could be written in the domain description. It has the structure: `(:preference <name> <plan goal>)`.

Constraints Constraints are conditions that must be true during the execution of a plan. In PDDL, such constraints are referred to as *state trajectory constraints* which are described using temporal modal operators. PDDL specifies several temporal operators that could be applied both in the problem description file and the domain description file, e.g., `always, at end, within,` etc. Those temporal constraint operators can be brought in to a domain or problem description file with `:constraints` keyword. If constraints are found in a problem file, it implies that such temporal conditions must hold in every planning problem concerning this domain, otherwise, constraints are specific to individual planning problems.

The situation calculus lays important theoretical groundwork for the representation of planning knowledge; however, it is not a real language to be implemented in a real-world planning application. STRIPS and ADL represent the early attempts to model planning knowledge using programming language-like structures; however, their confined expressiveness limits their applicabilities to modelling real-world planning problems.

Comparing to the other modelling approaches for automated planning, PDDL provides a set of features that increases its practical values in modelling complex real-world planning knowledge. Its expressiveness allows domain experts to model rather complex planning problems before these problems can be handed to planning algorithms. Appendix A gives an overview of the formal grammar of the PDDL language. Additionally, PDDL is a language widely accepted by the research community for automated planning,[3]

Despite its wide acceptance as the standard language in the planning research community, it is still hard for a non-AI expert to compile a proper

[3]PDDL is originally designed for the international planning competition, which is an international venue for research teams to compete on the performance of their planners.

planning domain for automated planning operations for real-world applications by using PDDL directly. The somehow obscure Lisp-like syntax of PDDL indeed poses an extra challenge for users and maintainers of a planning system, because they are in most cases domain experts in certain application areas rather than AI specialists. As most current knowledge-based systems still require human input for the enrichment of crucial information as the foundation of decision-making processes, a knowledge modelling language with a steep learning curve and limited acceptance by its domain users could prevent the prevalence of the system and makes it less useful than it should be. In the next section, a translation architecture is proposed to bridge the gap between the recovery knowledge model suggested in the previous section and PDDL.

Translation Architecture

Issues regarding the specification and representation of the fault recovery knowledge are discussed in the previous section. Additionally, detailed analysis is also given on formal specifications of modelling approaches for automated planning. In this section, a translation architecture including translation rules is proposed and discussed to fill the gap between the knowledge representation and actual usage of the planning knowledge. The main topics of this section are: propose the architecture for the translation operations and determine translation rules that map the recovery-specific language to a planning modelling language. The discussion is, however, strictly focused on the design rationale and theoretical issues such as translation strategies as well as translation rules; specifics on instrumenting the translation operations will be presented in detail in the next chapter on implementation.

Architecture As shown in Figure 4.5, functionally, a language translation architecture reads a DSL and generates another DSL according to the given rules or models. In our case, the proposed translation architecture reads the language specified by the grammar presented in section 4.2.2 and produces a planning specification language such as PDDL. Together with the language specifications and translation approach, the architecture is a part of the automated planning framework which is designed to fulfill the requirements of the planning knowledge and repository identified in section 2.2.5, Chapter 2 (Table 2.4 on page 49 provides an overview of those requirements).

The design of the translation architecture depends on the underlying translation approaches. Several translation approaches are available for the translation process: syntax-driven translation, rule-based translation and model-driven translation. Figure 4.6 illustrates the workflow of knowledge translation of planning knowledge definition into PDDL descriptions for the planning operations.

The translation process is similar to that of building a front-end part of a compiler. Its main objective is to analyse the syntax as well as semantics of the input language and to generate the corresponding output, in this case, PDDL. The input language is defined by the context-free grammar discussed in the previous section. The first step of translation is the lexical analysis of the input file. During this phase, the token and its types are produced by a translation component called the *lexical analyzer* or *lexer*. The emitted token steams are then analysed in the syntax level by the syntax analyser, by which tokens are recognised and organised according to the given input language grammar. The syntax analyser produces its results in the form of intermediate representations (IR), for example, a syntax tree would be one of the representation possibilities. The result of the syntax analysis is then further forwarded to the semantic analyser to build IR of the output language; during this phase, different approaches could be applied according to the complexity of the translation, for example, the tree-rewriting technique re-builds the abstract syntax tree composed by the syntax analysis operations. Another alternative is to use the string template approach to fill out the predefined output template during the semantic analysis phase. Finally, the output language is generated and forwarded to the planning algorithm.

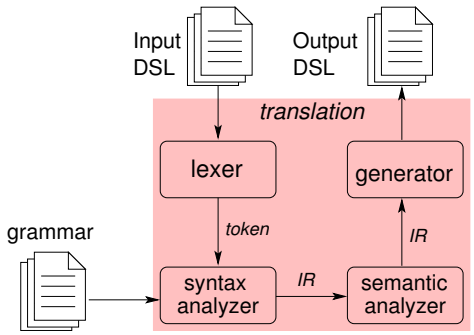


Figure 4.6: Translation Architecture

The multiple-stage knowledge translation architecture allows flexibility of building the target language for the planner. In cases that the requirements for the input language or output language are changed, individual components could be simply swapped out or modified to adapt to the new requirements, while essential parts of the translation architecture remain intact. For instance, if the input language is changed, only the lexer and syntax analyser need to be adapted to the new input language. Engineering details on the translation mechanisms will be given in the next chapter on the implementations of the proposed approaches. In next section, we discuss the translation rules.

Building and Translating Plan Domain A crucial step in the translation process is to define a set of translation rules which map the input planning knowledge onto the output language, in this case PDDL. The rules shall provide general guidances during the translation process. In other words, the translation rules try to associate the information models that are defined in the recovery knowledge with PDDL's planning knowledge model. Once the associations are determined, the translation could be done auto-

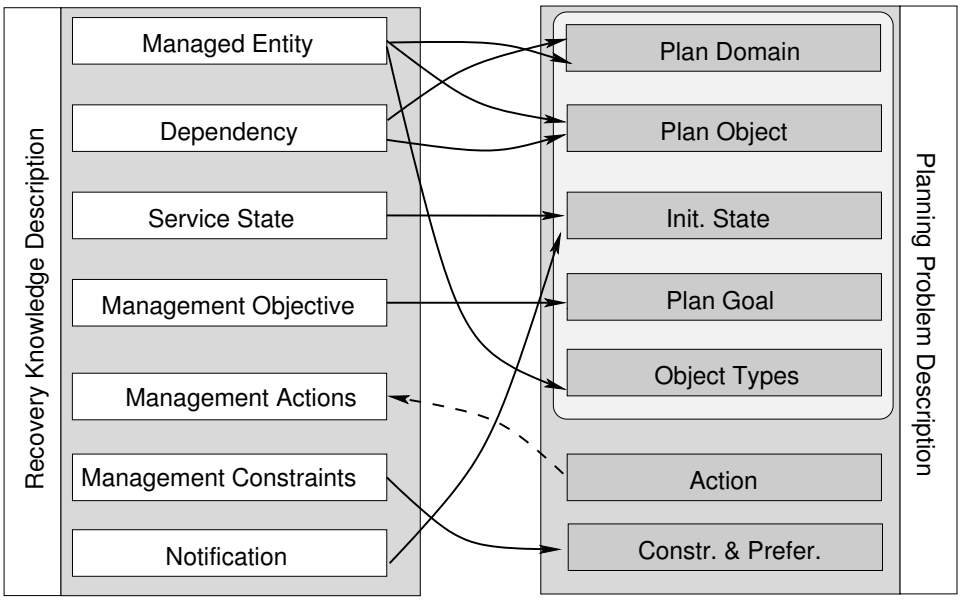


Figure 4.7: Illustration of translation rules that map between two different models

matically at syntactic level. Figure 4.7 provides an overview of the mapping between the two information models.

The translation rules can be classified into two main categories: *planning domain* and *augmented planning information*. The basic planning domain description should include *essential* planning information that is required by building an automated planning mechanism, whilst the augmented planning information extends the plan domain with *augmented planning knowledge* (e.g., temporal information, numerical constraints etc.) which enables more advanced planning operations. In Figure 4.7, the shaded area in the planning problem description represents the ingredients for building a basic planning domain.

Building Basic Planning Domain In this section, we discuss details of transformations between recovery knowledge description and plan the domain description, which also explains the fundamental principles of building a basic planning domain for service fault recovery planning. As discussed in the previous section, in order to build a basic planning knowledge domain, the following items are required:

- *Plan domain scope.* The plan domain scope encapsulates and consolidates relevant planning knowledge in a data structure. It provides a container gathering information that may facilitate the planner to

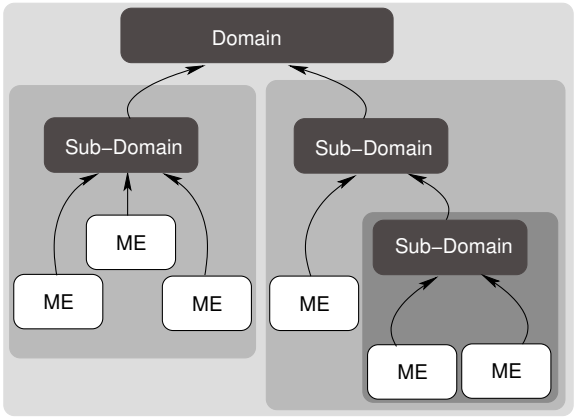


Figure 4.8: Plan domain structure

find solutions for problems in that particular domain. As given in the definition of IT service (Definition 1 on page 86), a complex service could consist of multiple sub-services in order to sustain its functionality. Therefore, a corresponding recovery plan domain can also be defined in a compound manner. For example, as given in Scenario 1 on page 26, a web-hosting service could consist of various components, which are (sub-)services themselves, in this case, web server, DB server and DNS server. Each of those sub-services can have its own domain. The plan domain for the web-hosting service is then composed of those three (sub-)domains. Figure 4.8 illustrates an exemplary domain composition. In this example, the top domain consists of two sub-domains, in which one of them contains a further sub-domain.

- *Plan object & Object types.* The building block of a service domain is the *Managed Entity (ME)*. There are two types of ME: *atomic entity* and *abstract entity*. Atomic entities are elementary components that a particular service is built upon. Compared to abstract entities, atomic entities are physical or virtual components, upon which management actions could be directly executed and take effect. For instance, a physical server which serves as part of a web-host platform is an atomic entity or an HTTPD daemon serving web request is also an atomic entity, because management actions could be directly acted on it so as to affect its state. Contrary to atomic entity, an abstract entity consists of more than one atomic entity. It also could contain other abstract entities to sustain its service functionality. For instance, a web-hosting platform is an abstract entity, because it consists of multiple atomic entities such as server machine, Apache HTTPD, DNS server, perhaps a load-balancer. To perform a management action on

an abstract entity, the action must be refined into sequences of actions that could be applied to the individual entities which the abstract entity is comprised of. Nevertheless, the abstract entity itself could also be part of another abstract entity, for example, a web-hosting service is a part of an online application platform. In Figure 4.8, all domains can be regarded as abstract entities.

Despite their differences, we define them both as entities in general to allow a unified definition and easy maintenance of such information, since only one format is valid for both types of entities. Later on we use algorithmic approaches to resolve their differences. Every ME is identified by its unique ID (e.g., `pcheger13`) and has a type associated with it (e.g., `AppServer pcheger13`). In planning parlance, the ME corresponding to the plan objects that participate in a plan domain as shown in Figure 4.8. Since varieties of MEs with different functionalities may participate in a service, typing is required to identify the classes of MEs which are mapped to the types of plan objects in a domain.

- *Actions.* In planning, actions are activities that can change the state of the underlying system. An action could only be selected and performed if all its required pre-conditions are fulfilled by the current state of the system. In the fault recovery planning problem, as defined previously on page 87, recovery activities are state-changing methods and actions that can be employed in recovery operations. Whereas in the formal definition of a management action (on page 111), a management action is characterised through multiple attributes, a subset of those aspects is required to build a basic domain for planning knowledge. Basic attributes such as action parameters, the associated management entity and action description including pre- and post-conditions etc. need to be included in the mapping process. Depending on the planners, the rest of the attributes can be used to support advanced planning features.

- *Initial state & Plan goal.* The current status of an underlying system is denoted by the initial state description. For the planning of fault recovery, the initial state is always a status of a service that contains one or more fault conditions that hamper the targeted service delivering its functionalities. The plan goal represents the desired state after a solution plan has been performed. In terms of service recovery, the goal state denotes the management objectives that should be met after the recovery plans are executed. Similar to that defined in planning parlance, both the initial and goal state in service fault recovery are technically the subsets of service state, and thus could be represented using the state representation definition of the recovery

knowledge description.

- *Plan* The product of a planning process is a set of actions in a particular orders, e.g., a plan. The formal definition of a recovery plan on page 114 contains information on types of action and the ordering of the plan. These elements could be directly translated from PDDL into the plan model given in the definition. Besides the sequence orders of the plan, the current model also considers actions that could be executed in a parallel manner. The time conditions provide extra features if, for example, temporal conditions and constraints are required.

Augmented Planning Knowledge Besides the basic planning knowledge, advanced planning attributes could be added to a plan domain description to augment more sophisticated automated planning operations. In terms of service fault recovery, we consider two types of attribute as augmented planning knowledge: *plan constraints* and *plan preferences*. Both attributes are introduced to improve the quality of plan results.

- *Plan constraints* PDDL has a concept of plan constraints (discussed on page 124), which asserts the conditions that must be true *in every step of the planning operation*. In terms of service fault recovery, such constraints are frequently documented in management policies. For example, in a management policy, it may state that during a fault recovery, the network interface `rt1-eth0` must not be affected and should always be available; in PDDL this could be written as:

  ```
  (:constraints (available (?rt1-eth0 - IFace)))
  ```

  Additionally, temporal information could also be integrated as constraints, for example, if a plan execution time should be less than 10 minutes in total, this can be expressed in PDDL as:

  ```
  (:constraints (< TotalTime 10min))
  ```

  As soon as the planner sees the `constraint`, the listed conditions are taken into consideration during the reasoning process.

- *Plan preferences* The concept *plan preferences* as discussed on page 124 can be regarded as *soft constraints* which express conditions that do not have to be fulfilled at any costs. In terms of service fault recovery, such attributes could be used to express the preference of the service recovery planner's user, for example, to express the preference "*failover site A is always preferred to site B* " could be written as:

```
(:constraints
    (preference P1
      (forall ?x - FailOverSite)
        (always ?m - Site_A ?n - Site_B)))
```

In this way, during the planning operation, the action that involves `Site A` is always preferred to another similar actions which involve `Site B` whenever possible. But if this rule is violated and other failover site is used in the plan, the results is still considered to be valid. Additionally, plan metrics (discussed on page 123) could be used to reinforce the plan results with preferences.

```
(:metric
  (+ 5 (is-violated P1)))
```

The above example expresses that if preference `P1` is violated, the plan is punished with extra weight. If the goal of the plan is to minimise the metric value while reaching the desired system state, increasing metric value makes this plan less preferable than those plans with lower value, even if they could transform the system into the same goal state.

Translation Algorithms Translation algorithms are implementations of the translation rules. All discussed algorithms operate based on the input files specified by the recovery knowledge specifications. The main purpose of the translation algorithms is to identify the knowledge types in the input file and to transform various attributes to the corresponding planning parlance. The input of the translation algorithms is the recovery knowledge description file compiled according to the given specifications. The output of the algorithms is the plan domain file and problem description file, respectively.

Plan domain scope. To extract plan domain scope from the recovery planning knowledge definition, input is required from the *Managed Entity (ME)* and *Dependency*. The algorithm scans through the input file to determine the plan domain as output. The scope includes the root service domain, intermediate sub-services domain and atomic objects as managed entities. As discussed previously, a compound IT service may encapsulate sub-services to sustain its functionalities. An intermediate sub-service may be a service domain itself and contains further services or atomic objects; the roles of a service domain and intermediate sub-service domain are therefore dependent on the scope of a particular service and could be used interchangeably according to the context. The atomic elements in a domain are those MEs, on which the *implementable* actions could be directly performed. The intertwined relationships between service, sub-service and ME are extracted from the available dependency information.

Algorithm 3: Determining the Scope of Plan Domain

Input : Read knowledge definition file f
Result: Determine root plan domains *dom*, intermediate domains & atomic objects

```
1  ReadInBuffer (f);
2  while !EOF do
3  |  Read (entity e in f);
4  |  if e → dependency have no antecedent then
5  |  |  e is a root plan domain;
6  |  |  Write (e → domain, dom);
7  |  end
8  |  else if e → dependency have no dependant then
9  |  |  mark e as an atomic object;
10 |  |  Call object & type processing procedures;
11 |  end
12 |  else if e → dependency has dependant & antecedent then
13 |  |  mark e as an intermediate domain;
14 |  |  Call object & type processing procedures;
15 |  end
16 end
```

Algorithm 3 shows the procedures to extract the scope of a plan domain. The algorithm reads a knowledge description file in a buffer and determines its scope by process entities and their dependencies. The algorithm differentiates between three cases:

i.) If an entity does not possess any antecedent, it is noted as a root plan domain scope (line 4 - 7).

ii.) If an entity does not possess any dependant, it is an atomic entity and the algorithm calls other procedures to process the entity (line 8-11).

iii.) If an entity has both an antecedent and a dependant, it is an intermediate sub-domain (line 12 -14).

The algorithm terminates when all the entities have been processed. It helps to build a hierarchical tree structure of a service landscape as the basis for further parsing operations. A tree represents a plan domain which is represented by its root. All abstract sub-services are represented by internal nodes of the tree and the leaves of the tree are atomic managed entities, on which management actions could be directly performed. Note that in the algorithm, the `Write` is written as a generic abstract process where actual parsing operations need to be integrated into the writing process (which will

Algorithm 4: Translating Plan Objects & Types

Input : Entity e
Output: Write e typed object in specification file *dom*

```
1 if e is atomic entity then
2     e → id as plan object obj;
3     e → type as object type type;
4     find the root domain dom, s. t. e ∈ dom ;
5     Write (obj, type, dom);
6 else
7     e is intermediate domain;
8     get e's dependant;
9     Call this procedure recursively until all descendants e_i(∀e_i ∈ e)
      are atomic entities;
```

be discussed in the following sections) to translate specific knowledge into planning language.

Plan objects & object types Algorithm 4 shows the translation procedures for objects and their types into the planning language. The algorithm takes the `id` of an atomic entity as an object name and its `type` as the class of the object. The processed entity will be finally written in a corresponding plan domain file as a typed object.

If the input e is an atomic entity, algorithm 4 extracts both the ID and type of that object and writes them to the plan domain file as `dom` corresponding entries (line 1 - 5). Otherwise if the input entity is an mediate domain (a sub-service), it processes the sub-domain recursively until all its descendants e_i are atomic entities and processes them correspondingly (line 6 - 9).

Actions Corresponding to the types of managed entities, two types of management action need to be distinguished: *abstract (non-primitive) actions* and *executable actions*. An abstract action is a management operation associated with a service domain. Those are operations that cannot be directly performed before they are resolved into specific implementable *executable actions*. For example, the management action `<Migrate Web Server A to B>` could be a recovery action associated with the web server management domain, however, this action could not be directly implemented before it is resolved into some specific implementable action such as copy content to server B, change default IP address, start site on server, etc.

To denote an action in planning parlance, we need to translate the ID of an action to symbolise an operation. An action is characterised by its pre-conditions and post-conditions, which describe its required operational

states as well as its effect after it has been implemented. An action effect could be a belief state of the underlying system where a particular management action is performed. Additionally, a management action usually requires one or more parameters to be associated with that particular action.

Algorithm 5: Translating Plan Actions

Input: Action entry a in file f
Result: Write a's entries in plan domain file *dom*

```
1 if (a → MO) ∈ e_non−atomic then
2     a is an abstract action (a task);
3     extract a → a^pre_description, a^post_description as pre- & post-conditions of a
      decomposable task;
4     extract a → a^para as a list of parameters of the task;
5     extract a → a^sub−tasks as a task network;
6     write a → ID as task name;
7 else
8     a is an implementable action;
9     parse a as a primitive executable action;
```

Algorithm 5 shows procedures for extracting action information from an input file. It determines whether the given action is an abstract non-primitive or executable in a plan domain; if it is an abstract action, it extracts necessary information from the given specification and establishes a partial sub-task network. The sub-task networks encode the procedures for the further refinement of the abstract non-primitive action.

States. A state description enables the planner system to gain an overview of the underlying service of its current states and, at the same time, allows the desired goal states to be expressed in the same way. In terms of service fault recovery, the current state is always the initial state with fault for the planner, and the goal state depicts a recovered service defined by an administrator. The corresponding translation algorithm should be capable of extracting individual states of managed entities involved in a service domain.

The algorithm distinguishes two types of state description as discussed previously. If the input state description is for an atomic managed entity, the algorithm extracts its ID as the name of an object in a plan domain. Then the type of object is to be determined with its ID, which could be matched in the e_{atomic}. Finally the algorithm extracts the states as a predicate of this plan object in a plan domain. If the s is a state description of an abstract entity, then the algorithm recursively extracts all states of involving

Algorithm 6: Translating State Information

Input: State description s in specification file f
Result: Write state definition in plan domain file *dom*

```
1 if (s → ID) ∈ e_atomic then
2     extract s → ID as an object in a plan domain;
3     find type of ID;
4     extract s → CURRENT_STATE as predicate;
5     Write CURRENT_STATE, Type, ID in dom;
6 else
7     s is description of state of an abstract service;
8     recursively extract all states from its dependants;
9     Write corresponding state with ID and Type in dom;
```

entities with their ID and types, and finally writes them to plan domain file *dom*. The plan goal has syntactically the same form as the current state description and can be expressed using predicates as well.

Plans A plan is a result of planning operations. Basically a plan is comprised of management actions with certain orderings. An advanced plan could also include temporal information to express its execution schedule and estimated total execution time. As the product of plan operations, the translation should be from PDDL to the specification language. An action in a basic PDDL form was introduced in the previous section (on page 122); an algorithm to translate a plan is designed as shown in Algorithm 7.

In Algorithm 7, the output of plan file is parsed to extract information to write into the specification file with format defined on page 114. In line 2-10, each action block is processed in the same order as given in PDDL. Corresponding information entries are mapped to the entries of the specification file *spec_plan*. The asserted service states denote the expected states that a service is in after the plan is executed; there are extracted and parsed in line 11-12 and finally written in the target file. If temporal and cost conditions are available, such as durations of action implementations and cost, they need to be calculated as well to indicate the further attributes of the computed recovery plan; however, this depends on the capability of planners of handling temporal information or any numeric information during plan composition.

Constraints & Preferences Management constraints and preferences can be expressed in the plan definition language. However, in IT service management, such terms are usually documented in legal papers such as SLA; to automatically translate documents as such into a formal language is semantically difficult, if not impossible. Therefore, if constraints and preferences

Algorithm 7: Translating Plan

Input : A plan file *pddl_plan* produced by planning algorithm
Output: Sequence of action specifications *spec_plan*

Read *Plan file pddl_plan*;
while *pddl_plan hasNextActionBlock* **do**
    get an action definition block $a \in pddl\_plan$;
    extract $a \rightarrow name$ as ID of an MgtAct;
    extract $a \rightarrow parameters$ as parameters of MgtAct;
    extract $a \rightarrow effects$ as effects of MgtAct;
    if *a parallelizable with previous action* a' **then**
        Write $a' \parallel a$ *in spec_plan*;
    else
        Write $a' \prec a$ *in spec_plan*;
get from *pddl_plan* the asserted states *states* after plan is executed;
parse *state* and write in *spec_plan*;
if *time duration & cost of actions are provided* **then**
    calculate estimated duration of the plan;
    calculate estimated plan costs;

must be considered in the planning, such terms could be directly edited by domain experts and then stored for future use. Details on which constraints and preferences could be expressed in the planning language are given in the discussion on page 130.

4.3 Recovery Planning Algorithm

Having the language specifications and the translation mechanism of recovery knowledge for planning operations presented, in this section we discuss design issues regarding the planning algorithm, which is the core component of the recovery planning framework. We lean on the HTN-based planning paradigm as our design guideline for the planning algorithm and argue that hierarchical-based problem-solving mechanism is appropriate for service fault recovery in IT management. We discuss at the beginning of this section the rationale of choosing HTN-based planning paradigm from different aspects, such as reduced complexity of HTN-based problem-solving techniques and expressiveness as well as easy extendability of an HTN-based knowledge representations. We then propose a design of an algorithm called the *Hierarchical-based Hybrid Management Activity Planner* (H^2MAP). Compared to the classical planning approaches, H^2MAP allows planning operations in a mixed initiative manner, where human operator can control and guide planning procedures. Despite the current concentration on IT ser-

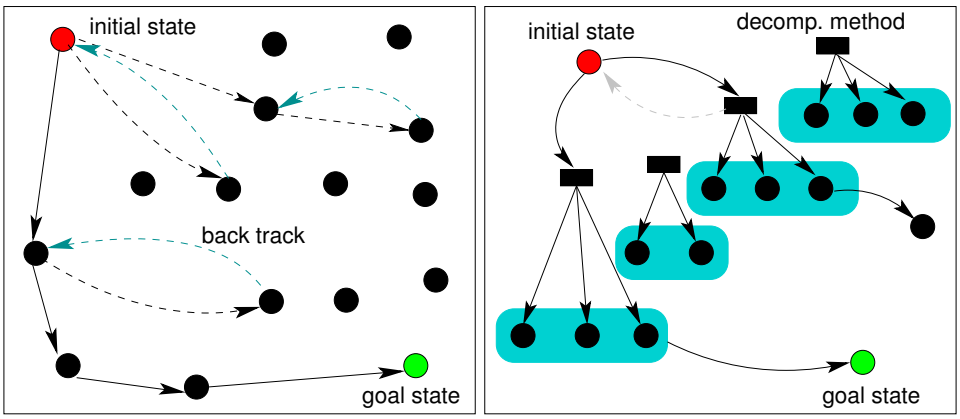


Figure 4.9: Comparison of state- and HTN-based planning operations

vice recovery planning, H^2MAP is designed to be generic enough for other management operations. With an easily modifiable knowledge base, the algorithm could be applied to other management areas, e.g., change management [TFL09]. We start the design of the algorithm with basic planning functionalities and by observing all assumptions of the automated planning theory discussion previously. Then more advanced capabilities are added to the planning algorithm by relaxing certain restrictions and therefore increasing the applicability of the planner.

4.3.1 HTN as Planning Paradigm

At the beginning of this chapter (Section 4.1), we dedicated an entire section to the discussion of the modelling of the service fault recovery problem as a planning problem. We also apply the problem reduction technique to show the transformability between the service recovery model and the general planning model. This justifies that automated planning can be used to solve the service recovery problem. In this section, we go a step further to determine a proper planning paradigm for the service recovery operations.

In Chapter 3, a list of different planning paradigms is presented in order to give an overview of planning technology. For planning of fault recovery operations, we selected the HTN-based paradigm as the guideline for designing our planning algorithms. The working principles of the HTN-based planning paradigm were introduced in Section 3.1.1 of Chapter 3. The rationale behind this decision is based on the following facts:

- Reduced Search Space with Hierarchical Problem Solving. It is well-known that applying the hierarchical problem-solving approach to complex problems with large search spaces can reduce the size of search

spaces and therefore significantly increases the efficiency of problem-solving processes [Min61, Kno91]. Both theoretical analysis and empirical studies [Kno91, Kor87, SSD77] showed that the hierarchical problem-solving technique can reduce the size of search space from exponential to linear. The HTN-based planning paradigm utilises this advantage by using concepts such as abstract methods, in which plan actions are organised into different abstraction levels.

- Domain Configuration Capability. The principle of the HTN-based planning paradigm applies highly reconfigurable planning knowledge which allows experts to write the domain-specific knowledge to compile a plan domain. Applying the domain-specific knowledge in the planning composition process can greatly improve the performance of a planning system [NGT04, SR02]. Despite the fact that domain-specific knowledge is used in the HTN-based planning paradigm, the planner itself still remains generic and domain independent, i.e. the planner is not bounded to any particular application domain. For example, an HTN-based planner could be used both in the planning operation for a fault recovery domain as well as for the change management domain with properly formulated planning knowledge. This feature makes an HTN-based planner highly flexible and reusable in various application situations. Most importantly, with configurable domain knowledge, HTN allows a certain degree of hybrid planning, meaning a user of the planning system is capable of influencing and controlling the planning operation, so that human user and computational methods could work cooperatively with each other.

- Expressiveness and knowledge reusability. The HTN planning paradigm is more expressive than other planning approaches, e.g., STRIPS-style planning [EHN95, NSE$^+$98]. This advantage allows a HTN-planner to be capable of representing a broader and more complex planning problems and domains [EHN96]. Despite the efforts that are needed to initialise a recovery knowledge base, all input knowledge in terms of recovery methods and recovery actions could be mostly reused as long as there are no fundamental infrastructure or policy changes. Reuse of management knowledge brings efficiency by increasing the information availability.

- Performance. Compared to other classic planning paradigms, such as state-space based and plan-space based planning, HTN planning can achieve significantly better performance through using domain-specific decomposition methods [NAI$^+$03]. Both theoretical studies [GN92, ST01] and empirical experiments [BK00] showed that HTN-based planners could quickly solve planning problems that are orders of magnitude more complex than classical planning methods. Figure

4.9 illustrates the difference between the state-space and HTN-based planning operations. Instead of searching the complete state space by randomly selecting actions that are adaptable to the initial state, an HTN planner searches a set of adaptable decomposition methods (symbolised with squares on the right hand of Figure 4.9) with integrated partially ordered plan segments. Due to its hierarchical organisation of tasks of different detail levels, the search space is greatly reduced in this way. Furthermore, the chances to backtrack because of the failure of the partial plan will be significantly reduced through using hierarchical task networks.

- Applicability. The HTN-based planning paradigm has been widely adapted in different application domains, such as high-critical military operation planning, space explorations and industrial manufacturing. Those application examples were introduced in Chapter 3 on related work. Multiple reasons have led to this wide acceptance of the paradigm: first, the hierarchical-based approach to problem-solving resembles how human operators resolve practical problems [SR02], in which problem-solving steps are hierarchically organised in different abstraction levels. Second, an HTN-planner allows more human control to influence and to optimise the planning process itself by integration of control knowledge [NSE+98], for example, restriction can be integrated to decomposition methods in order to guide the planning process. In this way, the plan results can be more realistic and applicable to real-world planning problems where restrictive conditions are common. Additionally soft constraints that represent preferences of users of a planning system could also be integrated during the planning operations.

- Extendability. The flexible domain configuration capability allows experts to extend domain knowledge. New planning tasks, actions, constraints (restrictive rules) etc. could be easily added to the domain knowledge. Moreover, compared to other planning paradigms, an HTN-based planner is also easy to be extended to handle more advanced planning requirements, for example, to integrate temporal information for the schedule of the plan executions. The hierarchical organisation of recovery knowledge with various detailed levels also allows domain the designer to focus on the detail level within her domain of expertise. For example, a domain expert, who is more familiar with management policies and SLAs, may concentrate on design abstract planning knowledge such as constraints and high-level tasks.

- Paradigm Maturity. The concept of the HTN-based planning paradigm has endured as a research topic for many years. Theoretical and practical experience achieved along the way have significantly improved

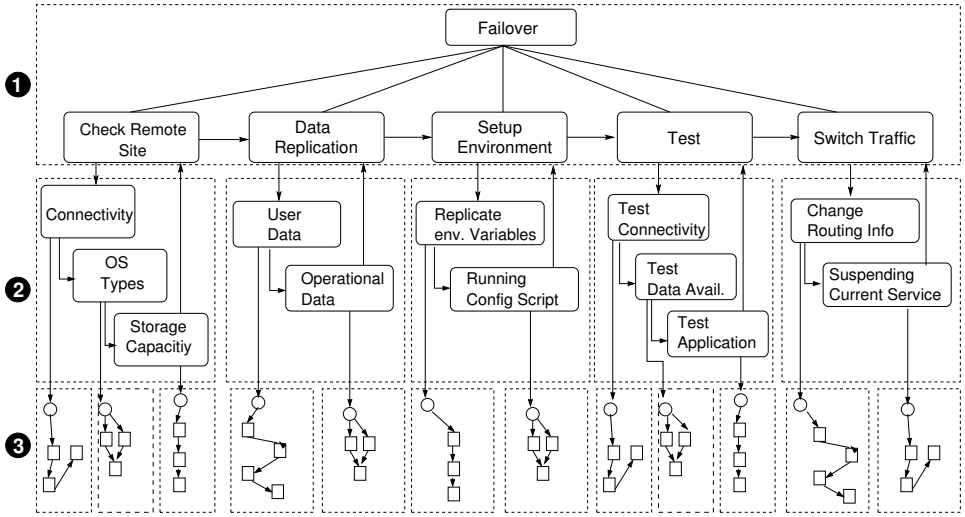


Figure 4.10: An Example of Failover Operation

the planning paradigm. Such improvements are essential driven factors towards a mature planner. The maturity of HTN-based planning approaches can be evidenced by their wide applicability in different areas [NAI+05].

The aforementioned facts are evidently enough to justify our decision to choose the HTN-based paradigm as our guideline for designing the planning algorithm for service fault recovery operations. When IT services become more intricate, management of such services also becomes a challenging task. To address this issue, one frequently employed approach is to break the management tasks into units with manageable sizes. Actually the hierarchical-based approaches have been frequently used in modelling and engineering technical systems, for example, the B-Method [Sch01, AA96] has been frequently used by system designers and engineers to model, understand and design technical systems, such as complex control architectures [LT04]. In the network management area, hierarchical-based approaches have also been applied [FLSDT10].

Composing a feasible recovery plan during a service being impacted by faults is a challenging task for a human administrator. Not only does he have to find a series of recovery actions that may transform the current state of a faulty service into an operational state, he also needs to consider factors such as the constraints of using resources, temporal conditions etc. The constantly growing size and types of services can only exacerbate such difficulties. To show the complexity of recovery actions, Figure 4.10 shows a recovery step - *failover*.

In this example, the task *failover* is designed to migrate the current ser-

vice to a remote stand-by system in order to guarantee service continuity in cases of fault. As shown in the figure, *failover* is an abstract recovery operation because it must be decomposed into specific tasks in order to be implementable. Each layer in the diagram represents a refinement operation and reveals more specific operations or actions that must be taken in order to implement their parent tasks. The final executable plan is shown in the last layer of the diagram (layer 3) where sequences of partial-ordered actions are organised into an executable plan. The *failover* example is a very simple recovery operation to ensure service continuity; however, from this simple example, we can see that even such a common recovery procedure can be refined and decomposed into a myriad of executable actions before it could be implemented. Additionally, the failover operation can merely be one of many recovery steps in order to achieve a recovery goal; in a real-world situation, many more similar operations must be integrated into a recovery plan which augments the difficulties of composing a feasible recovery plan manually, especially for time-critical management operations such as recovery. To choose *failover* a possible recovery action, one has to consider the set of pre-conditions that must be fulfilled, for example, a remote failover site must be designated before this task is chosen, e.g., in the state description the pre-condition *remote site availability* must be true. Not only pre-conditions could determine the usage of a certain action, other restrictive conditions could also affect the decision of whether a particular action could be chosen, for example, if there is a restrictive condition which states that the cost of the resulting recovery plan must not exceed a certain amount and if the application of the *failover* task will violate this policy then it cannot be chosen.

In parlance of HTN-based planning, the operation *failover* can be viewed as a compound task with multiple levels of decomposition operations. The knowledge about how to decompose such high-level tasks into executable tasks can be encoded in a data structure called *decomposition methods* or just simply *methods*. Specifically a method contains a partially ordered tasks network, which prescribes how a particular high-level abstract task could be refined into more specific tasks. Besides, a method as defined in the HTN approach, also includes the information about the pre-conditions under which this particular method could be chosen as a candidate of a plan. A method represents and encodes *standard procedures*, *best practices* or *empirical experiences* of management knowledge of conducting a particular high-level abstract management operation.

Figure 4.11 shows the operation *failover* in a perspective of task networks. Each compound high-level tasks must be decomposed into more specific tasks by applying appropriate methods. The gray line drawn across the decomposition process represents the selected *method* just as previously discussed. The primary task for a HTN-based planner is therefore

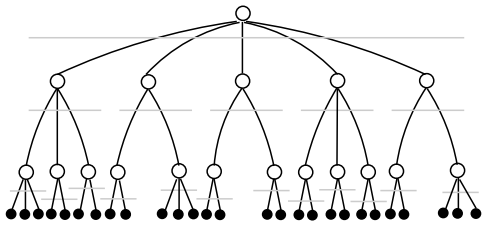


Figure 4.11: An abstract view of the failover operation based on HTN

to find a set of proper decompositions methods during its operation process. An advanced HTN planner should also be capable of observing any constraints and/or conditions that should be considered by composing a recovery plan, just like human administrators frequently do. A simple decomposition method has the following structure:

```
(
 :method <ID/Header>
 :parameter <param_1...param_j>
 :preconditions <⋂ pre_i, where i = 1, ...,n>
 :task-networks <t_1 ≻ t_2... ≻ t_m>
)
```

A decomposition method is identified by a name ID which is also commonly known as a header in planning parlance. Note that a method name could be shared by different methods, which have distinguished pre-conditions and different decomposition networks. ID is followed by a list of required parameters. The pre-conditions of the method denote under which circumstances a particular method could be employed. The task networks show how this abstract method could be decomposed or refined into more specific tasks. Note that in our fault recovery knowledge specifications, we also categorise recovery actions into two types: atomic and compound actions. Algorithm 5 (page 134) is designed to automatically identify and translate the recovery actions from recovery knowledge specifications into corresponding planning formats. The parsing and translation results of the failover operation in PDDL is shown in the following:

```
(:method failover
 :parameter ?failOverSite - Site
            ?currentSite - Site
            ?sIFACE - NetworkInterface
            ?tIFACE - NetworkInterface
            ?userDataLocation - DataLocation
            ?operationDataLocation - DataLocation
```

```
            ?setupScripts - DataLocation
 :preconditions(and
                    (avail ?failOverSite)
                    (OSRunning ?currentSite)
                    (conn ?sIFACE ?tIFACE)
                )
 :task-networks (ordered
                    (checkRemoteSite ?failOverSite)
                    (dataReplication
                        ?userDataLocation ?operationDataLocation)
                    (setupEnv ?setupScripts ?failOverSite)
                    (test ?failOverSite)
                    (switchTraffic ?currentSite ?failOverSite)

)
```

The above code snippet shows how the decomposition method could be represented in PDDL. Note that for the sake of conciseness, the preconditions and parameters are simplified in such a way that an overview could be easily given in this context; a real-world operations description encoded in the planning method as well as actions would surely be more complicated in terms of number of parameters and more restrictive preconditions.

4.3.2 Fundamental Planning Algorithm Design

Having examined the advantages of applying the HTN-based paradigm as the design guideline, in this section we propose the design of a planning algorithm for IT service fault recovery. We start with the discussion of a naïve version of the planning algorithm which observes all the restrictive assumptions presented previously in section 4.1.2 on page 90. The purpose of the naïve version is to show the design idea and working principles of an HTN-based planning algorithm by fault recovery planning. Then we present the next version of our planning algorithm with considerations of restrictive constraints such as temporal constraints and other numeric conditions like costs of the recovery steps. With consideration for the temporal constraints, we are allowed to integrate the scheduling feature into the planning algorithm. Finally, we consider the changing planning environment, e.g., underlying target systems are more likely to change rather than stand still. Finally, we propose ways to deal with such system dynamics as uncertainty information in planning.

Algorithm

In this section we propose the design of an initial version of the *Hierarchical-based Hybrid Management Activity Planner (H^2MAP)* for fault recovery planning. As its name suggests, the planner shall possess the capability to take plan instructions from its user; therefore the planning operation is done in a *hybrid* manner. We refer to H^2MAP at this stage as a fundamental version, since its current design observes all of the aforementioned assumptions regarding the planning domain and the underlying target system. However, the ultimate objective of the planner is to compute a recovery plan which is a set of recovery actions that could be implemented on the target system according to the observed current states. Selected actions, when executed, shall transit the current system to a pre-defined goal state by completing a set of pre-defined high-level tasks. Since we approach the planning issues in an incremental manner, the first design of the planning algorithm shall fulfill following requirements at this stage:

- Given a set of decomposition methods and initial tasks, the planner should find one or more plans which satisfy the desired goal state description.
- The ordering of the resulted plan sequence shall be retained as orderings of the plan execution. The plan ordering currently considered is strictly sequential.
- Users of the planner shall have the opportunity to give high-level abstract tasks to control the directions of plan operations.

Details on the Algorithm

The algorithm H^2MAP describes a framework of the computational operations of the planner, which is the core part of the recovery planning architectural framework. The operation of Algorithm H^2MAP is based on several input data, including:

- Initial task list T_{init}, which allows input of initial task or tasks to be decomposed; such initial tasks list can include both atomic actions and abstract (compound) tasks. It provides the user of the planning system with the opportunity to determine and control the planning operations.
- To conduct the planning operation, the planner needs to perceive the current state of the target service or system. This vital piece of information is described by the planning parameter i_0 which describes the

Algorithm 8: H^2MAP: A HTN-based Planner

Input : $T_{init} \subset M$: initial task(s);
i_0: description of current systems state;
i_g: description of desired goal state.

Data: M: a set of disposable methods *and* $a \in A$: atomic actions, where $(\sum_{i=1}^{n} a_{i,u}) \subseteq M_u$.

Output: One (or more) *ordered* fault recovery plan(s) Π, where $\Pi = \sum_{j=1}^{m} a_j$ *or* fails with no valid Π.

```
1  Π ← current plan, initiated with ∅;
2  if T_init contains only primitive tasks then
3      Π = T_init;
4      if GoalTest(Π, i_g, i_0) then
5          return Π;
6      else
7          return Failure;
8  else
       ► Comment: tests the solution hitherto ◄
9      if GoalTest(Π, i_g, i_0) then
10         return Π ;
11     else
           ► Comment: goal test fails and T_init = ∅ ◄
12         if T_init = ∅ then
13             return Failure;
14         else
15             Select t_0 ∈ T_init ;
16             if t_0 is a decomposable task then
17                 Find (M, i_0) ⇒ m ∈ M;
18                 Apply(m, t_0) ⇒ ⋃_{i=0}^{n} t_0^i ;
19                 change the (belief) state of the target system to i'_0 ;
                   ► Comment: Recursive call ◄
20                 H^2MAP(⋃_{i=0}^{n} t_0^i, i'_0, i_g);
21             else
                   ► Comment: In case t_0 is a primitive task ◄
                   Π ← Π ⋃ t_0;
22                 Apply(t_0, i_0) ⇒ i'_0 ;
23                 H^2MAP(T_init \ t_0, i'_0, i_g);
```

initial/current state of the system. Syntactically the state information is represented using the schema proposed in section 4.2.2 on page 108. Algorithm 6 (page 135) prepares the obtained information by translating it into the PDDL language which is ready for the planning operations. As discussed previously on page 104, the state information can be either obtained from components by automated means such as through monitoring mechanism or by input from operators.

- The desired state of the system *after* the computed solution is conducted is denoted using the planning parameter i_g. This parameter is above all necessary for controlling the correctness of the planning results during the planning process through comparisons with the (partial or complete) computed recovery plan.

The above inputs can be classified into category *infrastructure knowledge* as discussed previously in section 4.2 on page 102. However, in order to conduct successful planning operations, the planner needs to be supported with the recovery knowledge to sustain its planning operations. Such data is termed as *management knowledge*. A related discussion was conducted in the previous section (4.2) as well. Basically H^2MAP necessitates two types of supporting knowledge:

- A set of decomposable tasks M. The set M contains abstract recovery tasks that can be selected by the planner during its operation. The syntactics of method descriptions is given in section 4.2.2 (page 111). The translation of planning methods into the planning language is illustrated in Algorithm 5.

- Set of atomic (primitive) actions. Primitive actions are those that are directly implementable on the service. They are fundamental elements of a final recovery plan that is implementable on the impacted service. Their syntactic format for the input and translation algorithms are given in section 4.2.2 and Algorithm 5 as well.

This supporting information is an essential part of a recovery knowledge base. The planner needs to constantly query and consult those data during its operation in runtime so as to choose the right decomposition methods and actions to perform. We separate them from the standard input of the H^2MAP since, compared to the other input, they play merely an assisting role as invariant factors, despite their essentiality for the planner.

The final output is one or more executable recovery plans Π which are the results of decomposition operations. It contains a set of *total-ordered* executable atomic actions. A total-ordered plan implies that each action in the plan is bounded to an ordering relationship ($\preceq$) and the execution sequence of the plan is exactly in the ordering of the actions in the plan.

The H^2MAP algorithm starts with an initial empty plan Π. The planner first scans through the input and verifies the input initial task(s) T_{init} to determine if it only contains primitive actions. If this is the case, no decompositions are necessary. The planner conducts a goal test operation based on the initial task(s). The goal test should determine if the goal state of the target service or system could be reached with the initial task(s) in the plan. In case the goal test was passed, the initial task is returned as the solution plan Π, else the algorithm returns failure. This is the simplest case in the planning operation; where the sequence of primitive actions is already chosen by human operators, the planner just does a sanity check of the provided plan to assert the correctness of the solution.

Procedure GoalTest(Π, i_g, i_0)

1 **foreach** $t \in \Pi$ **do**
  ► Comment: t `satisfies current state` i_0 ◄
2 $t_{precond} \models i_0$;
  ► Comment: `apply` t `and change state` ◄
3 `Apply(`t, i_0`);`
  ► Comment: `update current state` i_0 ◄
4 $i_0 = t_{effects} \bigcup i_0 \setminus \overline{t_{eff}}$;
5 **if** $i_0 \models i_g$ **then**
6 **return** True;
7 **else**
8 **return** False;

The GoalTest procedure checks the given plan to determine if the actions comprised in the plan could eventually lead to a system state that satisfies ($\models$) the desired goal state (i_g). The assertion is conducted by updating the current state of the target service with the effects of t and excluding the changed states. The assertion process loops through the plan until no atomic action in the task list remains. The final asserted service states, which are reached by applying the actions, are compared with the desired goal state i_g; if all desired states are fulfilled then the procedure returns a boolean value `True`, otherwise it returns a `False` signifying that the given plan cannot fulfill the given planning objective.

Starting with line 9 in Algorithm 23, H^2MAP processes and decomposes the compound tasks accordingly. Line 9 and line 10 apply the GoalTest procedure as introduced above to test the partial result during the planning operations. In case the partial result could already fulfill the goal state, then no further decompositions are necessary. The planner stops its operation and returns the current partial plan as the final solution. H^2MAP also halts if there are no more tasks in the T_{init} and the goal state is still open; in this

case, the planning operations failed with no valid plan.

In line 15, the algorithm selects the next open task in T_{init} and determines if it is a decomposable task. If positive, the algorithm tries to find proper decomposition methods in order to refine the abstract task into more specific ones. A method uses task networks to guide the refinement operations. A decomposable task m_i is said to be applicable if $m_i^{pre} \subseteq i_0$, meaning the pre-conditions of the method m satisfy the current states of the observed service.

Procedure Find(M, i_0)

```
1 try m_i ∈ M ;
2 if m_i^pre ⊨ i_0 then
3 |   choose m_i as a solution candidate ;
4 else
5 |_  proceed to next method, if M ≠ ∅ ;
```

The Find procedure shown above illustrates how a proper decomposition method is chosen for the further planning process. In addition to the fulfillment of the pre-conditions of a chosen method by the current state description, in order to be chosen by the algorithm, a decomposition method must be *relevant* to the task. More specifically, *relevance* implies that there is a substitution θ, so that $\theta(task) = m_i$, meaning a task m_i is relevant to a compound task if they are unifiable under the substitution θ. The conditions for a method m to be selected as a proper decomposition are summarised below:

$$\begin{cases} m_i^{pre} \subseteq i_0, \ current\ state\ i_0\ satisfies\ pre-conditions\ of\ m_i; \\ \theta(t) = m_i, \ whereas\ \theta\ is\ a\ substitution. \end{cases}$$

Line 18 and line 22 in H^2MAP, the Apply procedure is used to update and to assert new belief state of the system as soon as a suitable method has been chosen. The current state information i_0 is correspondingly updated and used as a parameter in the recursive call of the algorithm. If t_0 is a primitive task, then it is added to the plan Π as an executable plan action and current state i_0 is updated according to the effects of the selected action. The open tasks in T_{init} with the updated current state description i_0' and goal state i_g are passed as parameters to the recursive call of H^2MAP. Note that in H^2MAP, the computed result plan is *total ordered*. In HTN parlance, a result plan is said to be totally ordered, if all of its sub-task networks are totally ordered. The execution of recovery actions will be exactly in the sequential order of the plan, e.g., optimisation mechanism in the plan such as parallel execution and interleaved plan execution is currently not considered in a total-ordered plan.

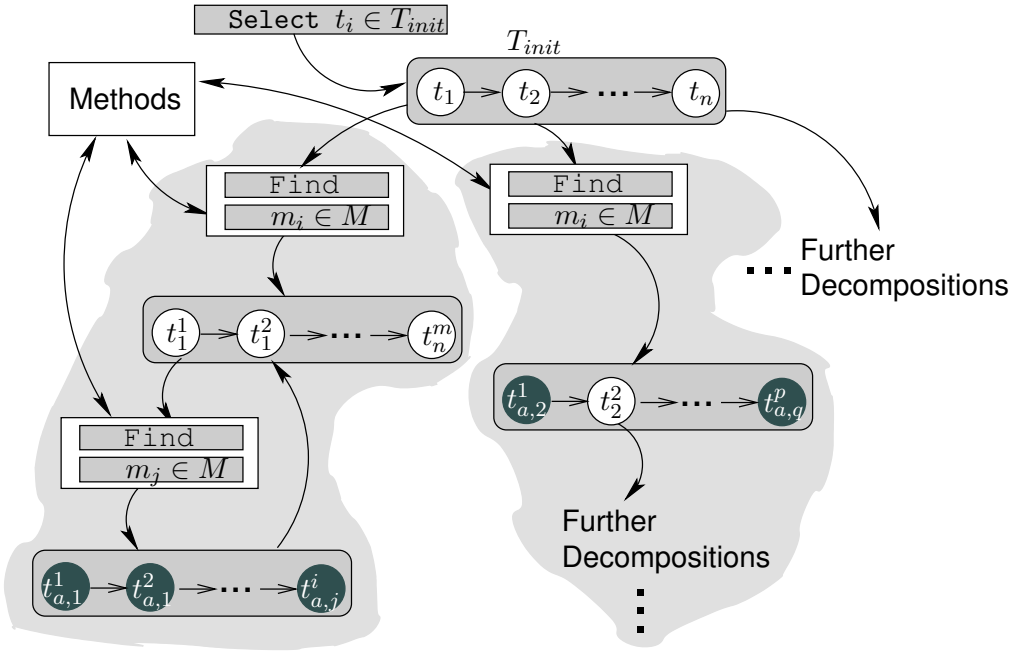


Figure 4.12: Illustration of H^2MAP Operations

Figure 4.12 illustrates the operations of the H^2MAP algorithm. The light-coloured cycles in the diagram represents decomposable tasks (e.g., t_n^m) and dark-coloured tasks represent non-decomposable atomic actions ($t_{a,j}^i$ where the a in the subscript denotes an atomic action), which are elements in the final recovery plan. Each shaded area illustrates a branch of recursive planning decomposition operations with corresponding decomposition methods. The Find function queries the method repository, which is part of the planning knowledge based introduced earlier. Decomposed tasks are total-ordered sub-plans with either a set of atomic actions or decomposable tasks or a mixture of both, as illustrated in the diagram.

Note that, to guarantee the termination of the algorithm in any case, we explicitly prohibit the recursive definition of a decomposition methods, meaning a method containing the same decomposable task is called by his predecessor which could lead to infinite recursive calls of decomposition. Such a situation makes the whole planning problem undecidable if not explicitly prevented. If we observe the decomposition structure as a tree, the recursive-free requirement leads to an acyclic tree structure.

4.3.3 Assured Properties of the Algorithm

Formal discussions regarding algorithmic properties of H^2MAP are necessary to show its general characteristics. We approach such analysis by

investigating two algorithmic properties: *soundness and completeness.*

Soundness. Theoretically the soundness property of an algorithm implies that the algorithm returns only valid solutions to an input problem. In other words, any solution provided by a sound algorithm is expected to be correct. This is a fundamental property that a reliable algorithm must possess. A valid solution in our case is a plan comprised of a sequence of implementable actions, which can transform the current service states into a set of desired states. If γ is a transition function, Π and $\imath_0$ are the solution plan and initial situation, respectively; a valid (correct) solution to planning problem P satisfies the following conditions: $\gamma(i_0,\ \Pi) \rightarrow i_j$. If $i_g \subseteq i_j$, we say that the plan solution Π is a valid solution to the planning problem P. Such correctness of the plan solution is examined by the `TestGoal` procedure in the suggested algorithm, which asserted the state of the system after the decomposition operations have been done, meaning there are only implementable actions that remain in the plan. The `TestGoal` procedure only releases the valid solutions if the solution satisfies the desired goal state, otherwise it returns failure. The soundness of the algorithm is therefore guaranteed by the choice of HTN-based approach.

Completeness The completeness of an algorithm suggests that if no solution can be found by the algorithmic procedure to a particular problem, then no solution to that problem exists. As long as a valid solution exists for the problem, an algorithm, which satisfies the completeness property, will find that solution for the input problem. In the case of H^2MAP, we show that the planning algorithm always finds a solution plan when one exists for the input problem based on the method repository M. We apply the complete induction proof to show the completeness of the H^2MAP algorithm; more specifically we induce the size of the decomposition method repository in order to substantiate our claim that the algorithm is complete with any size of M. In other words, if $|M| = n$ denotes the size of a decomposition method repository M, we induce n in our proof.

$|M| = 0$: In the base case, the size of the method repository is zero, representing the situation that no decomposition method is available. In this trivial case, the algorithm always returns `Failure` and there is no solution to the given planning problems either. Thus, the algorithm is complete.

$|M| = 1$: Under a situation in which only a single decomposition method m is available, the method could be either selected in the planning process as a valid decomposition or discarded by the algorithm. In both cases, the algorithm will return deterministically a positive or negative result accordingly. In the latter circumstance, the algorithm will deterministically return `False` as a result, since there is no further method currently disposable to the algorithm. If the method is chosen by the algorithm, exactly two possible results

will follow: If the result of the decomposition contains only executable primitive actions *and* the `GoalTest` procedure in the algorithm returns a `True` as an answer, then H^2MAP returns the decomposition result as answer to the planning problem. Otherwise, if the decomposed network contains other non-primitive tasks or the `GoalTest` procedure failed, a negative answer is guaranteed. Thus, in any case the algorithm is complete.

$|M| = n$: We construct the inductive hypothesis by assuming that with n decomposition methods available, the algorithm will return correct answers if there is a valid plan solution available based on the disposable methods and will terminate with false otherwise. In other words, the algorithm H^2MAP is complete under the condition $|M| = n$.

$|M| = n + 1$: In the induction step, we show that the algorithm is complete with a method repository of size $n+1$ as well. Note that methods cannot contain any non-primitive task which could lead to violation of the acyclic property of task networks. As discussed previously, this violation could cause non-terminating behaviours of the planning algorithm. To proof is the following lemma:

Lemma 1. *The algorithm H^2MAP is complete if $|M| = n + 1$.*

Proof. We show the completeness property of the algorithm H^2MAP by means of *proof by contradiction*. First, we assume that Lemma 1 is false and the algorithm is incomplete if the size of method repository M is larger than n. In this proof, we apply the previous induction assumption that the algorithm is complete when $|M| = n$.

Suppose P is the planning problem we intend to solve using the algorithm with $n + 1$ optional methods in the repository M'. By computing plan solutions, the algorithm either provides solutions or exits with failure. We discuss these potential results in a case-by-case manner.

If H^2MAP returns a failure, it signifies that no solution could be provided by the algorithmic procedures. However, due to the incompleteness of the H^2MAP as we assumed, suppose there would be a solution based on the current decomposition method repository M' (with size $n + 1$) which is unknown to the algorithm. We argue that such a situation will not occur, since it contradicts our previous hypothesis that H^2MAP is complete when $|M|$ has size n; a subset of M'. Obviously the additional method in M' could either contribute to the solution or be discarded by the algorithm during the planning process; nevertheless, it does not influence the completeness of the algorithm in both cases. The reason is obvious if the additional method is discarded during the computation (not contributing to a potential solution), then we have a repository with size of n, which is $|M|$. In case it contributes, the additional decomposition method will be applied as part of plan solution, if there is any; a situation in which the completeness of the algorithm is not violated as well. In the current case, the algorithm returns with a failure implying that the additional method does not contribute to

the planning process. Therefore, we have the case $|M'| - 1 = |M|$. Since the algorithm is complete with $|M| = n$, we can therefore conclude that the assumption, in which a plan solution is missed by the algorithm with $|M'| = n + 1$, contradicts our hypothesis.

In a further case, in which the planning problem indeed has no solution based on M' and the algorithm returns with a failure, then the algorithm automatically provides a correct result with $n + 1$ methods in this case and contradicts the assumption that the algorithm is incomplete. In a similar case, the planning problem has valid solutions based on $|M'|$ and the algorithm returns positive answer, which contradicts the assumption of the incompleteness as well. In a final case, the target algorithm returns a valid solution based on M' where the planning problem has actually no valid solution, which is obviously an implausible situation by nature; a further discussion of this case is thus not necessary.

To summarise, in the above proof we show on a case-by-case basis that by a method repository M' with size $n + 1$, the assumption that the suggested algorithm is incomplete could be contradicted in all possible cases. Therefore, we show that Lemma 1 holds.

□

With the above proof, we conclude that the algorithm is also complete in the induction case when $M = n+1$. Therefore, we argue that the algorithm H^2MAP is complete with any size of a method repository M.

4.3.4 Augmentations for IT Recovery Planning

With the basis planning algorithm H^2MAP in place, we develop in this section several extensions to extend the capabilities of the planner. With those extensions, we also try to relax several previously discussed restrictive assumptions generally imposed on the planner. The extensions of the algorithm concentrate on the temporal conditions as planning constraints as well as the planning in a stochastic domain.

Temporal Enhancements

Time plays a critical role in the fault recovery procedure of IT services, for example, a practical plan should conform to deadlines that set by service level specifications which are negotiated and signed between the provider and customer of the service. Violation of such time constraints will cause penalties on the service provider, albeit the recovered service. On the other hand, if every recovery action is extended with time conditions, we could exploit such information for optimising of the execution of the plan, provided that conflicts on the resource usages between actions are excluded. Furthermore, equipped with temporal knowledge, we could conduct reasoning operations to examine the consistency of actions in a recovery plan. An

extension of the temporal reasoning capability is essential for a practical planner. Our temporal enhancement of the H^2MAP planner will concentrate on the following aspects:

- Dealing with plan deadlines as plan constraints. With a time-critical service recovery operation, deadlines are usually defined to constrain the time needed for the recovery. In such cases, not only must a planner find a functioning recovery plan, it also has to observe the designated time windows of the recovery activities. In a very strict circumstance, any plan that violates the given deadline will be valuated as not useful.

- Integrating an advanced temporal reasoning mechanism to enable the partially ordered plan structure. In the basis algorithm, we assumed that all the partial plan networks involved in the operations are total-ordered, which could be a limitation when the planner is used in real-world situations, where an deal plan may include constructs such as parallel execution of the recovery actions when there are no conflicts or sharing of participating resources. We discuss therefore in the following section a planner extension which could process such structures during the planning operations.

- Securing temporal consistency of the plan actions. A consistent plan is the one without temporal conflicts between involved actions. To guarantee the applicability of a recovery plan, it is mandatory to ensure the temporal consistency of the plan. If we regard a plan as a network of operations with a set of particular ordering constraints existing between pairwise operations, a consistency check ensures that no constraint is violated plan-wide.

Planning with Deadlines Before the operation of the planner, a deadline can either be given by the user of the planner system or be extracted from other management-related documents like the SLA. A deadline can be regarded as a temporal constraint with beginning time-point and end time points. As long as a recovery deadline is given, the total amount of time that a valid plan requires should be within the defined time period. This imposes extra requirements on the planner to reason about time during its operation; the result should not only be a viable solution, but also a valid solution.

The idea of taking pre-designated deadline into consideration is to properly propagate the time constraints during the planning process into a task network hierarchy. A deadline defines a left and a right time boundaries of the whole plan, which can be written as an interval $[t_a, t_e]$, where t_a denotes the start time reference point and t_e is the required end time boundary of the recovery activity. The total time cost of a valid recovery procedure must

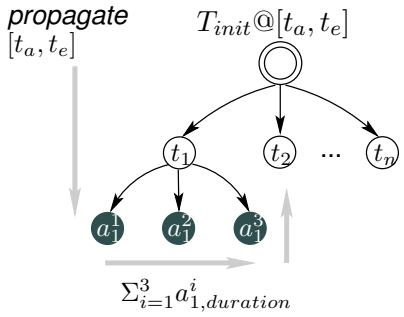


Figure 4.13: Illustration of time constraint processing

fall within the given time interval. During the planning operation, the selected tasks are decomposed into either further tasks or primitive actions. To facilitate such temporal capability, it is essential to give each primitive action a proper duration, which represents the time cost of that action. In the previous discussion, we integrated the time factor into the knowledge model of recovery knowledge, specifically as part of the information of the primitive recovery action. In the PDDL knowledge model, the actions with explicit time requirement are denoted as *durative actions*, to which our knowledge model could directly be mapped using the translation procedure. Note that we do not encode the time information into decomposition methods intentionally in order to retain the flexibility of writing decomposition methods.

We use $a_{i,j}^{t}$ to denote that the time requirement of the action a_i is t time unit. Note we use the generic term *time unit* rather than specific terms, such as second, minute or hour alike, in order to keep the discussion general. Together with previously defined time boundaries $[t_a, t_e]$, we say a plan satisfies its deadline constraints if $t_a < \sum_{i=1}^{n} a_i^t \leqslant t_e$, given a plan solution Π with $|\Pi| = n$. As the planning operation progresses, the deadline is propagated in terms of time constraints through sub-task networks. All accounted sub-tasks inherit the time boundary from the initial input and pass along it decomposition paths. Additionally, after each successful evaluation, the right-side boundary of the time constraint is modified in order to the keep the time limits up to date. A side-effect of processing temporal constraints is that, during planning, if the planner found that the actual time cost of a partial plan solution already surpasses the deadline, it causes the planner to backtrack and try to find alternative decomposition paths that can satisfy the given time constraints. The planner follows the principle of the *lazy evaluation*, which means that as soon as the violation of time constraint is found in a method during the operation, regardless how many sub-tasks or primitive actions in that method still remain to be evaluated, an immediate backtracking is triggered. The time constraints actually make the planning

operation more restrictive but bring the basis planner a step closer to a real-world operation scenario. The propagation paths of the constraints are symbolised with gray arrows showing the direction of the progresses.

Figure 4.13 shows a partial progress of constraint propagations in the HTN network. A vertical propagation pushes the given constraints through hierarchical levels until the level with primitive actions a_i^j is reached. As soon as the level with primitive actions has been reached, the algorithm begins to calculate the time cost required by each primitive actions in the level. The updated constraints after successful evaluations are propagated to the next task for further processing. The `TemporalConstraint` procedure describes related activities to process deadline constraints.

Procedure TemporalConstraint(a_i^t, Π)

1 $\Delta = a_j^t$;
2 $t'_e = t_a + \Delta + \sum_{i=1}^{n} a_i^t,\ \forall a \in \Pi$;
3 **if** $t'_e \leq t_e$ **then**
4 | a_j is valid for Π;
5 | $\Pi = \Pi + a_j$;
6 **else**
7 | backtrack to the higher level task t, where $a_j \in t$;
8 | find an alternative decomposition path;

In the `TemporalConstraint` procedure, an plan action a_j under question and the current (partial) plan Π are used as parameters. The time cost of a_j is extracted and noted as Δ to the time costs of the current plan. We used t'_e to record the consumed time that can be satisfied by the current plan Π. Note that the t'_e is computed based on the beginning time reference point t_a in the constraints. A comparison with the given deadline t_e is then conducted to see if the current actual time cost t'_e violates t_e. With a positive evaluation, the action a_j is inserted into the current plan as a valid action; otherwise the procedure triggers a backtracking operation. The backtracking process invalidates the high-level task, with which the action a_j associates, due to the violation of the given time constraint. A new search for an alternative decomposition method is triggered in order to find another plan path. Finally, to put the deadline reasoning mechanism into practice, the `durative` keyword must be appear in the description by preparing the plan domain knowledge with PDDL.

Integrating the Advanced Temporal Reasoning Capability Ideally, timely-optimised recovery plans usually includes not only sequential durative actions but also parallel activities to shorten the recovery time in general. However, our HTN-based planner H^2MAP currently merely considers the

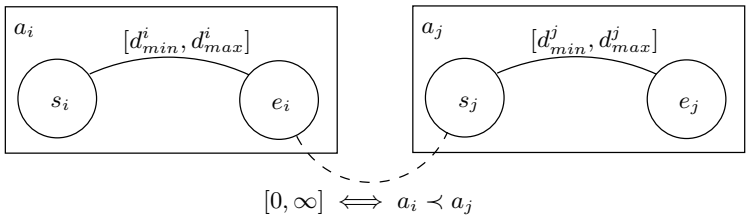


Figure 4.14: An example of detailed STP structure of HTN actions

totally ordered plans. That means the solution provided by our planner may solve the input problem, nevertheless, the solution is not temporally optimised. As a motivating example, consider a recovery plan involving two tasks *ReInstallDatabase* and *ReInstallWebServer*. Obviously if these two activities do not share the same or have any conflicts in resource usages, they are temporally independent from each other; thus the plan would be more efficient if they are conducted in parallel. Otherwise, special constraints must be used to prohibit parallelism. To deal with such a situation, which is common in real-world operations, we suggest in this section an integration of advanced temporal reasoning capability as a further extension to our H^2MAP planner.

To tackle the aforementioned issue, we suggest using the Simple Temporal Problem (STP) [Dec03], which is a constraint satisfaction problem specifically tailored for efficient temporal reasoning. We propose the integration of STP as an extension of our HTP planner to support both durative actions as well as parallel solution plans.

As defined in [Dec03], an STP problem is a triple $(X,\ D,\ C)$, where X is a set of time point variables, D defines the domain for every variable and, C is a set of binary constraints which are applicable between elements of X. An extra constraint $[a_{ij}, b_{ij}] \in C$ confines the relative distance between two time points by requiring $a_{ij} \leq t_j - t_i \leq b_{ij}$. Any temporal orderings could be represented either with *qualitative representation* or *quantitative representation*. Detailed information on both types of temporal representation is given in [Vil82, All83, VK86]. A qualitative ordering constraint between two time points, t_i and t_j can be expressed as $[a_{ij},\ b_{ij}] = [0,\ \infty]$. In order to sustain durative as well as parallel actions, we give every plan action start and end time points. Temporal relationships between these time points reflect quantitative and qualitative ordering constraints between plan actions.

We apply an STP data structure to capture not only qualitative ordering but also quantitative temporal relations. In Figure 4.14, two actions a_i and a_j with their start time points and end time points. A qualitative ordering constraint $[0, \infty]$ is imposed between a_i and a_j, which orders a_j after a_i.

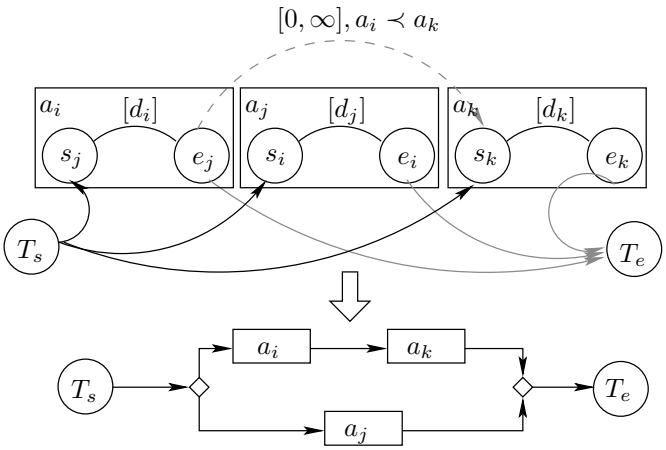


Figure 4.15: STP with workflow representations of HTN plans.

Durations of each action are given by the respective $[d_{min}, d_{max}]$ between the start and end time points. This illustrates how a solution plan and associated STP are related to each other. Figure 4.15 illustrates STP and corresponding workflow representations of the HTN plans. In the upper part of the figure, an STP is shown with global start (T_s) and end time points (T_e). Both qualitative and quantitative relations are considered in the STP diagram. The duration of actions a_i, a_j and a_k are denoted as $d_{\{i,j,k\}}$, respectively. A qualitative temporal ordering $[a_i \prec a_j]$ is enforced by the constraint $[0, \infty]$ between a_i and a_k, stating that actions a_i must precede a_k. The lower part of Figure 4.15 represents the translated plan from STP with parallel executions. Note that as shown in the STP diagram in the figure, the action a_j is temporally unconstrained.

As a recovery plan is continuously decomposed and refined by the algorithm, temporal constraints are added to the associated STP. Every plan Π has a corresponding STP with start and end time points associated with it. T_s and T_e denote the global start and end time points, respectively. Each time a task is decomposed by a selected method, all temporal constraints between its sub-tasks are posted to enforce the ordering structure of the sub-workflow as encoded in the decomposition methods. As a task is accomplished by a set of operators, the task's temporal properties are inherited by the operators. A durative action a_i with duration d can be encoded by the constraint $[d, d]$ between the time points s_i and e_i. A decomposable task τ with a deadline t generates the constraint $[0, t]$ between T_s and τ_f. We illustrate the process with the following procedure.

The Procedure TempConstraintProc decomposes tasks and forwards temporal constraints in an STP. It decomposes tasks p into a list $plist_{tasklist}$, which is comprised of either sub-tasks or actions. Additional ordering con-

Procedure TempConstraintProc(p_{task}, $C_{constraints}$, $plist_{tasklist}$)

```
1 foreach e ∈ plist do
      ► Comment: attach start and end points ◄
2     STP.add(e_s, e_f);
      ► Comment: e must start after the decomposed task p ◄
3     STP.addConstraint(P_s → e_s);
      ► Comment: e must end before the task p ◄
4     STP.addConstraint(e_f → p_f)
5 foreach (p^1 → p^2) ∈ C_constraints do
      ► Comment: add remaining ordering constraints ◄
6     STP.addConstraint(p^1_f → p^2_s)
```

straints C are inserted between the elements of *plist*.After updating the STP with new time points and temporal constraints, a path consistency algorithm can be applied to examine the consistency of the STP. Additionally, it can also compute the equivalent minimal network of the given STP. Checking of STP consistency for every potential new action or method decomposition in the preliminary plan can be accomplished with $O(n^3)$ time complexity. It helps to efficiently prune a large quantity of HTN search space and ensures that all plans provided by our algorithm are temporally valid. The Path-Consistency procedure illustrates a conventional path consistency checking algorithm proposed by [Dec03].

Procedure PathConsistency(C)

```
1 while not all constraints in C are stabilized do
2     forall the k, 1 ≤ k ≤ n do
3         forall the pairs(i, j):1 ≤ i ≤ j ≤ n, i, j ≠ k do
4             c_ij ← c_ij ∩ [c_ik • c_kj];
5             if c_ij = ∅ then
6                 Fail with inconsistency;
```

In this section, we extend our basis planning algorithm H^2MAP with temporal processing capabilities. Given the richness and importance of temporal conditions in the process of fault recovery and IT management in general, our two extensions consider deadline and temporal orderings of the recovery actions during the planning process, which are considered to be crucial especially for time-critical recovery actions. We apply temporal conditions as constraints of the planning procedure and propagate those temporal constraints in the planning hierarchy of our HTN-based algorithm. Further-

more, to ensure the correctness of the final plan, we use a path-consistency checking procedure to ensure that our result plans are consistent and valid.

Planning with Uncertainty

In our basis planning algorithm, we assume that the actions included in a result plan are the only source of change of service states, i.e. the planning environment is static and the recovery actions have deterministic outcomes. These reflect only a half-truth of a real-world service recovery scenario, in which diverse environmental dynamics and external unexpected events frequently appear. This could distort the current knowledge about the state of the target service possessed by the planner. In this discussion, we attempt to relax those restrictive assumptions and provide a theoretical framework for the extension of our basis planner. From a recovery planning perspective, it could be concluded that the uncertainties can be generally originated from the following sources:

- Environmental uncertainty. In an IT operational environment, fluctuations of the environment are frequent cases. During planning, unexpected events could be generated in the observed infrastructure and invalidate the planner's previous knowledge of the states, on which the computation of recovery plan is based. Such a dynamic could invalidate the results of the planning algorithm and renders the recovery plan as infeasible.
- Non-deterministic execution outcomes of plan actions. Executions of plan actions could be unsuccessful due to unexpected situations, which can complicate the recovery process. Non-nominal outcomes of recovery actions are important and sometimes highly critical. Therefore, it is necessary to model the possible failure of an recovery action similar to an exception-handling mechanism, for example, possible failure to restart a server process needs to be considered in the planning.
- Partial observability of the planning environment. In a large-scale planning environment with many components and sub-services, gaining a complete knowledge of the current state of the system is difficult, if not impossible. During the planning process, part of the state variable could simply be unavailable, for instance, if a recovery plan requires a remote failover site, but during the computation, the state variables regarding the remote site are not automatically available; some sensing operations are normally required to get them. In planning terms, observability refers to the visibility of state variables that describe the targeted system Σ (the generic planning system model).

Tackling the uncertainty properties in planning is a hard problem, one of the possible ways to deal with the aforementioned uncertainty issues is

to integrate Markov Decision Processes (MDP) [Bel03, HJ00, CKL95] into planning. With an MPD the change flow (evolution) of the planning environment is modelled as a Markov chain. An utility function is designed to reward the execution of certain actions for each state. It is required that the underlying system is modelled as a stochastic system, which is a non-deterministic state-transition system that assigns probabilities of state transitions. With the stochastic system, the uncertainty regarding the outcomes of plan actions can be modelled with a probability distribution function. To adapt the MDP to the planning, the system model Σ is modified as $\Sigma = (S, A, P)$ [NGT04], where S is a finite set of system states and A is a set of actions as previously defined in the generic planning model. P in this stochastic-based system model is a modified transitions system (corresponding to γ in the deterministic system model). It defines a probability distribution for every state transition. Given system states $s, s' \in S$ with $a \in A$, the probability distribution $P_a(s'|s)$ denotes the probability of state transition $s \rightarrow s'$ with the action a. The solution of an MDP is called the *policy*, usually denoted as π. The policy includes the best action to be taken in each state transition, e.g., $\pi : S \times A$. A utility function u evaluates a selected action a_i, given a current state s_j, and assigns a value to the choice of action a_i in state s_j, thus $u(a_i, s_j)$. The characteristics of Markov decision processes determines that the MDP is memoryless, which means the computation of the next optimal action step is only referred to the current state of the system, not the states that prior to that state.

The expected value of executing the policy from state s can therefore can be formalised as:

$$V^*(s) = \max_{a \in A(s)} [u(a, s) + \sum_{s' \in S} P_a(s'|s) V^*(s')]$$

An optimal policy π is therefore a policy which maximises the utilisation value and denoted as:

$$\pi^*(s) = \arg \max_{a \in A(s)} [u(a, s) + \sum_{s' \in S} P_a(s'|s) V^*(s')]$$

Note that depending on the types of optimisation and their applied metrics for the utilisation function, arg max could be replaced by arg min if, for example, we are trying to minimise the cost of the plan. Two classical algorithms are frequently applied to solve MDP problems by finding an optimal solution policy with the cumulative maximal expected utilisation π^*: *value iteration* and *policy iteration* [Put94].

With the *policy iteration* algorithm, the current policy (plan) is constantly improved by searching plan actions in each state that have higher reward value than the action that is currently chosen for that state. The

initial policy is, however, chosen at random. The algorithm terminates when no further improvement can be discovered.

The *value iteration* algorithm for solving an MDP takes a different approach in finding optimal policies. Instead of striving to iteratively *improve* the reward values for each state as a policy iteration, the value iteration algorithm produces a solution policy in a constructive manner, in which the optimal policy is computed with successively growing length. In other words, the algorithm iterates on the length of the policy starting with $V^*_{n=0}(s) = u(s,a),\ \forall s \in S$ and for further iteration steps, computes:

$$\pi^*_n(s) = \arg \max_{a \in A(s)} [u(a,s) + \sum_{s' \in S} P_a(s'|s) V^*_{n-1}(s')]$$

The above notation denotes that at each step n, the algorithm computes a reward value based on the utility function. u with consideration on the previous evaluation V^*_{n-1}. A threshold value ϵ can be defined to terminate the iterations with the following condition:

$$\max_{s \in S} |V_n(s) - V_{n-1}(s)| < \epsilon$$

Specifically, when the maximum change in values between the current and previous value function converges to a value below ϵ, the algorithm terminates.

A planning problem can be modelled as an MDP to deal with uncertainties by computing plan solutions. More specifically, an MDP considers the situation where outcomes of plan actions can variate and the results are not always corresponding to what the planner expects. To enable the capability to deal with such uncertainty in the planning realm, it is essential to extend the modelling of plan actions to incorporate probabilistic values. A viable approach is to explicitly declare probabilistic effects of executions in the PDDL action model [YL04] as follows:

$$(\textit{probabilistic}\ p_1\ e_1\ p_2\ e_2\ p_3\ e_3\ ...\ p_n\ e_n)$$

Outcome e_i of an action a occurs with probability p_i, where

$$p_i \leq 0\ \textit{and}\ \sum_{i=1}^{k} p_i = 1$$

Nevertheless, a probabilistic-effect pair can also be left out if empty; in this case the effect description is identical to the conventional PDDL definition.

Also, MPD-based planning provides a suitable way to include planning metrics in the planning processes. An utility function $u(a,s)$ needs to be defined to reward each choice of action. When the cost and rewards both need to be considered by the same utility function, then a corresponding

optimal solution is the one that maximises the following value, given a cost function $c(a_i, s_{i-1})$:

$$\arg\max[\pi^*(s) - \sum_{a \in A, s \in S} c(a_i, s_{i-1})]$$

Action description of an MPD-based planner should also allow the representation of Markovian rewards associated with corresponding state transitions. This could be accomplished by simply adding the keyword *reward* to an extended action description scheme. Furthermore, we can define explicitly in the plan the goal description as a metric to maximise the utilisation value or minimise the total plan cost.

As mentioned above, the sources of uncertainty during planning stem not only from probabilistic outcomes of plan actions, but also from the fluctuations in the environment which induce the uncertainty of an observed target system. This phenomenon is commonly known as *partial observability* or *incomplete plan information*. Two possible approaches could be applied to deal with the partial observability: *Partial-Order Markov Decision Processes (POMDP)* [PB01, Bou02] and *Interleaved Planning and Execution*. We briefly discuss those two approaches as potential extensions to our planner theoretically.

POMDP builds directly on the extension of planning with an MDP with further considerations on the uncertainties of the observed planning environment. To facilitate that, it is required to incorporate a set of observations in the planning model. Thus, a simple planning system model is modified based on the MDP system model as follows:

$$\Sigma = (S_{[blf]}, A, P, O)$$

In the model above, we define the states of a system as a set of belief states $S_{[blf]}$. Thus, a belief state $b \in B$ is defined as a probabilistic distribution over the states. The probability assigned to each state s is denoted by $b(s)$. The definitions of elements A and P remain identical to those of the MDP system model. The additional element O is a set of observations, for each $o \in O$ and $a \in A$ there is a known probability $P_a(o|s)$ which gives the probability of observation o after executing the action a in state s, thus $\sum_{o \in O} P_a(o|s) = 1$. Together with the state transition probability distribution P, the observation O describes a further aspect of the system dynamics. Additionally, a reward function similar to that of an MDP can be defined to give numeric values to each calculated plan step. The uncertainty in this context implies that it is impossible to determine the current state with complete certainty. In ideal cases, at any given time point, the target system is assumed to be in some known state s_n; with introduction of the uncertainty concept, instead of being in s_n, we say the target system in a

belief state with probabilistic distribution b_n, which can be represented as follows:

$$b_n = P(s_n|o_n, a_t, o_{t-1}, a_{n-1}, ..., o_1, a_1), where\ o \in O, a \in A$$

The above representation shows the probability that at time n, the observed system is in state s_n given the history $o_n, a_t, o_{t-1}, a_{n-1}, ..., o_1, a_1$. The probability of observation $o \in O$ after executing action $a \in A$ can be computed as:

$$b_a(o) = \sum_{s \in S} P_a(o|s)b(s)$$

With the above equation, we can compute the probability with Bayes' rule that the system will be in state s after applying action a in belief state b and observing o:

$$b_a^o = \frac{P_a(o|s)b_a(s)}{b_a(o)}$$

With the above mathematical definitions, a POMDP planning problem is therefore a tuple $P = (\Sigma, b_0, G)$, where $b_0 \in B$ is the initial belief state (a set of possible initial states) and $G \in B$ is a set of states that is only comprised of goal states.

As its MDP counterpart, the goal of solving a POMDP is to find a policy/plan that maximises the total rewards defined by a reward function. However, the further consideration of the state probability distribution makes it extremely computationally expensive; therefore, a more efficient heuristic than the classic approach such as value iteration is needed in future research. Due to its demanding computational intensity, using POMDP is currently only suitable for solving small-scale planning problems, nevertheless, it provides an excellent formal foundation for planning under environmental uncertainty.

Besides modelling the planning domain as a stochastic system and solving the planning problem with probabilistic approaches like MDP and POMDP, another way to deal with uncertainty is through the active adaption of the plan, which means that an initial plan will be composed on-the-fly and the plan executed as a classical planning approach. However, during the execution, the planner keeps sensing and tries to detect environmental changes and unexpected events. Upon receiving any events that may invalidate the current plan, the planner is activate to compute or to adapt the current plan to the new situation.

Figure 4.16 shows the process of plan adaptation through environmental sensing and active replanning. In this planning model, a planning algorithm, such as H^2MAP proposed by this thesis, computes a plan according to the planning elements suggested by the conventional planning model. The

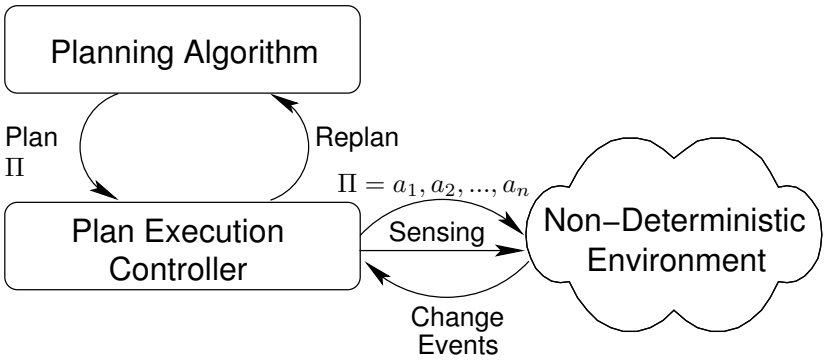


Figure 4.16: A Model for Active Planning Adaptation with Replanning and Environmental Sensing

planning results are delivered to the *Plan Execution Controller* element for implementation of the plan actions on the target system. The plan controller executes the plan in a step-by-step manner and, meanwhile, senses any environmental changes. It is important for the controller to detect any unexpected events that deviate from the expected state. The unexpected events could be caused by two reasons:

- Any events caused by the plan action that is not in accordance with the expected events of that action.
- Any events caused by influences that are external to the plan, which corresponding to the fluctuations of the system under observation.

Since any of these events could invalidate the original plan solution; therefore as soon as such events take place, the controller detects them and sends the planner the signal to trigger the replanning operation. In such cases, the planner re-computes or repairs the current plan according to the new state of the underlying system and forwards the adapted or repaired plan to the controller for a new round of execution.

One of the important considerations of applying adaptive planning is that the time costs of planning operations are a critical factor which can effect the usability of this approach. Specifically the time required by the planning operation needs to be less than the average change rate of the observed system. If this condition is not fulfilled, e.g., the environmental change rate is much faster than the planning computation, then the planning operation may not converge.

There are two potential approaches that can help to remedy such a problem:

- Use a more aggressive and fast planning heuristic. This approach increases the planning efficiency and lowers the time requirement of the

planning operations. To facilitate a faster planning, we need to use more aggressive plan searching procedures and integrate them into our H^2MAP algorithm. Approaches such as Anytime A* [HZ07] and Anytime Repairing A* [LGT04] can be adapted in our algorithm as sub-procedures to search for promising decomposition methods in the planning procedure. These time-sensitive algorithms poccess the capabilities to conduct a searching operation when the time available for the operation is limited. The initial solution is monotonically improved if the time for the computation is still available.

- Isolate and "freeze" the system state. Another way to solve the time problem is to isolate the target system and freeze its states as the planner gets the initial state description. That means, no external influences will be accepted and the system is artifically prevented from evloving. We force the system to be in a state which is deterministic and guarantees that the plan actions are only sources of state changes. Nevertheless, the planner still needs to track variable outcomes of plan actions as we discussed in the section regarding MDP. Although we could not prevent the system from evolving internally, through isolating and freezing the system, we reduce the sources of change to a more controllable quantity.

Concluding Remarks

In this section, we propose and provide detailed discussions of two possible extensions to our basis planning algorithm H^2MAP. The first extension we consider is to enable the planner to do temporal-based reasoning during the plan operation. We use temporal conditions as constraints of the plan, and propagate those constraints in the planning phases to guarantee those constraints are met in the final solution plan. Furthermore, we use an STP data structure to model the interval as well as the point time conditions, which is essential for scheduling and optimising the execution sequences of the plan. As we modelled the temporal constraints as STP, we could not only reason on the improved execution sequence, but also check the consistency of the plan using well-known constraint-solving techniques such as the Path Consistency (PC) algorithm. To further relax the restrictive assumptions of our planning algorithm, we also discuss in details the planning under uncertainty which is a common case regarding IT service fault recovery. We depict two possibilities to enable the planner to do reasoning under uncertainty: *Markov Decision Processes (MDP)* and *Partial Markov Decision Processes (POMDP)*. Both require the modelling of a planning environment as a stochastic system and the reasoning is conducted using probability distributions of action outcomes as well as the probabilistic distribution of belief system states. Due to the computation complexity in a

complex application scenario with large quantities of belief states, we further propose two alternative approaches: *active adaptive planning* and *isolate & freeze the target system*. Both approaches provide architectural solutions for planning under uncertainties while requiring minimum modifications to the original planning algorithm.

4.4 Recovery Planning Framework

To put a planning algorithm to practical use requires a properly designed framework to support its computation. Issues such as information exchanges between sub-systems, interactions with the external environment and users, persistent storage of planning knowledge, etc. are all considerations that have to be carefully made during the design phase of a framework. Additionally, a framework is going to be used by different application developers on different systems; therefore, it also needs to be flexible and extendible enough to address such dynamics.

Having the planning algorithm already in place, in this section we propose a detailed design of a software framework for IT service fault recovery planning. The framework intends to support the operations of the H^2MAP planner as well as a set of assisting components that revolve around it. It also provides an extensible development architecture for the developer to elaborate the planning applications in the recovery domain without sacrificing flexibility, extendability and simplicity. We begin our discussion with a review of a set of requirements related to the framework in order to rationalise our design. Then we show the overall architecture of the proposed framework followed by individual discussions on the layout and functions of each involved sub-system component. To ensure maximal flexibility and extendability of our framework, we apply well-established design patterns as general guidelines of our proposed framework to characterise the ways of interactions and distributions of responsibilities between involved components in the framework.

4.4.1 Overview of the Framework

A framework manifests and dictates the architecture of an application. The following is a definition of a framework as given by Gamma et al. [GHJV95]:

> [A framework is] a set of cooperating classes that makes up a reusable design for a specific class of software. A framework provides architectural guidance by partitioning the design into abstract classes and defining their responsibilities and collaborations ...

The fault recovery planning framework is thus designed to define an overall structure of the planning framework and to partition different re-

sponsibilities into diverse components. As a guiding design principle for any complex framework, we partition our design into several cooperative sub-systems. Thus, the framework design also illustrates how individual sub-systems collaborate with each other in a thread of plan operation control. Note that in the further discussions regarding the framework, we use the terms *sub-system* and *component* interchangeably. Our design of an IT fault recovery planning framework should fulfill the requirements derived in Chapter 2 on page 41 *Framework Requirements*. The requirements related to the fault recovery framework are briefly recapped as follows:

- Generic Framework. The framework shall not be bounded to a specific application scenario.
- Adaptability & Extendability. The framework shall be flexible. Flexibility in this context means that the design should be easy to be adapted and extended as the system evolves. A new and improved planning algorithm shall easily to be plugged into the framework without major modifications to the framework. On the other hand, the adaptability requirement shall also ensure the easy adaptation of the framework to other assisting management components. Changes of sustaining components shall be non-intrusive to the whole framework.
- Usability & Interoperability. The framework shall be easy to use for users. Complex internal planning principles and working details must be hidden from users of the planning system. A user-friendly interface shall be supported to show the plan results in a straightforward and easy-to-comprehend way for the user to scrutiny as well as verify the solutions.

To sustain the fault recovery operations, the framework consists of the following components:

- Planner component for plan generation. As the central element of the framework, the planner component provides the core planning functionality supported by the algorithm. The operation of the planning algorithm needs to be assisted by an additional set of operations, which facilitate information exchanges and inter-component cooperations between the planner and other sub-systems in the framework. As the automated planning technology is constantly evolving, new planning algorithms need to be easily adapted to the framework; therefore, the planner sub-system should provide the flexibility to allow easy adaptations of potential improved or new algorithms into the framework.
- Data processing component. The data processing component is designed as a mediator between different data formats exchanged in the

planning system. The main goal of the data processor, on one hand, is to parse and translate input data into a target language that could be comprehended by the planner. On the other hand, the results of the planner need to be communicated back to the external system which may have different requirements on the representations of the plan solutions; therefore the mutual communication needs of the planner require a flexible design, with which new system requirements regarding the representation could be easily adapted. Instead of hard-coding all operations into the component, a flexible design can provide interfaces to the lexical analyser, parser and translator that streamline the processing of input data from different supporting components into the format that is supported by the planner. In our case the output language is PDDL which is the *de factor* language standard for automated planner as already discussed in Section 4.2.3. With this design principle, an implementation of an abstract interface could be easily swapped according to the representation requirement of the target system without disturbing the rest of the system.

- Knowledge repository for persistent planning data stores. Fault recovery is obviously a knowledge-intensive management operation. As we have seen in the algorithm, the planner needs to use available recovery knowledge to synthesise viable recovery plan solutions. To fulfill this purpose, recovery information needs to persistently stored and easily be called by the planner during the planning operation. Furthermore, recovery knowledge items should be easily added, modified and updated. Details on types of recovery-relevant knowledge were already discussed in Section 4.2.

- Planning System interfaces. Two types of interfaces should be considered:*interface to execution system (I2ES)* and *interface to user (I2U)*. The I2ES interface provides the planning system the possibility to access execution mechanisms of the management of target service, so that the recovery action steps could be implemented on the target service, such execution system could be some workflow execution engine, where recovery actions are automatically implemented on the target system or it could also be some hybrid system where human assistance and human control could be involved. Unlike the I2ES interface, the I2U is the interface design for the planner to show its solution to the user of the planning system, so that the user has opportunity to exam the correctness of the plan results. The user should have chance to even modify, correct or artificially adapt the plan results, if necessary. The I2U interface facilitates the interactions between planning system and its user.

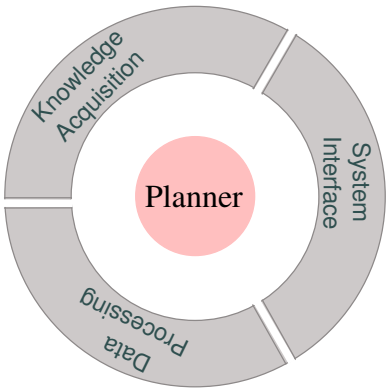


Figure 4.17: An Overview of the Planning Framework

Figure 4.17 provides an overview of the framework. As shown in this figure, all components revolve around the planner components and provide services to sustain the recovery planning operations. Each component has its in- and outwards functionalities to be fulfilled, for example, the system interface has as one of the outward tasks to represents plan results and to enable interaction between the planning system and its user. The system interface should track the plan executions and report and unexpected events from the target services. In the following section, we discuss each components and their design in details. We propose our framework design based on the object-oriented design approach [RBP+91, Boo82]. Figure 4.18 shows a more detailed view of the proposed recovery framework in UML.

As shown in the figure, the framework maintains interfaces for information exchanges between the planning system and external entities; the rest of the framework components are encapsulated inside the system.

4.4.2 Planning Component

As the core component of the proposed framework, the flexibility of the planning functionality should be ensured as we discussed in the requirements of the framework. In our context, the flexibility can be interpreted as the capability of easy adaptations of new planning algorithms without any major modifications to the framework itself. As new or improved planning algorithms appear, the framework must readily integrate them into its planning component. Additionally, in the runtime of the planning operations, the user should have the opportunity to dynamically select the available planning algorithms. This implies that, instead of considering the H^2MAP as the only algorithm to fulfill the core planning functionality, the system must be open to scores of similar algorithms. On the other hand, a user of the planning system could switch between different algorithms during the run-

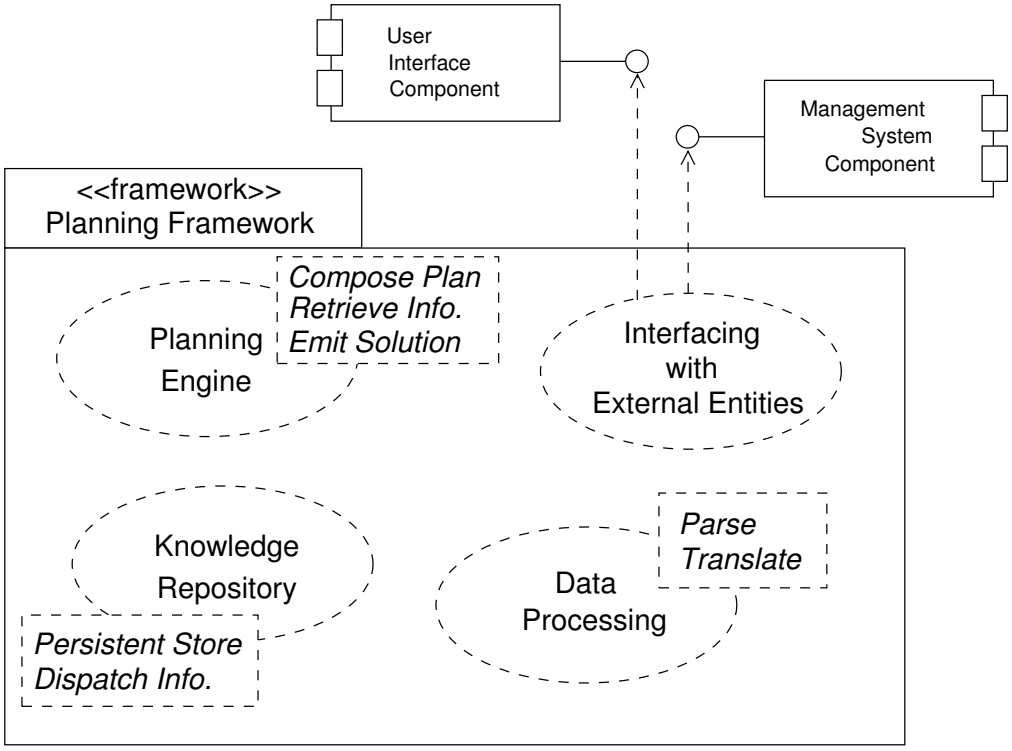


Figure 4.18: An UML View of the Recovery Planning Framework

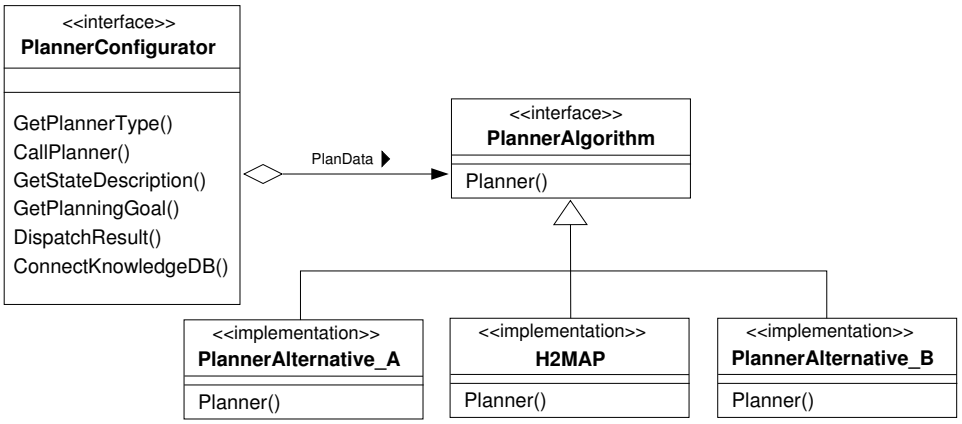


Figure 4.19: Planning Component

time and therefore achieve different plan solutions for consideration. This approach increases the possibility to obtain viable plan solutions.

To address these requirements, we apply the well-known design practice *Strategy Pattern* [GHJV95] to guide our design of the planning component. Given a set of algorithms which serve similar purposes, the strategy pattern suggests to encapsulate the algorithms and make them interchangeable. This approach allows the user to vary algorithm usages independently and, thus ensures the flexibility and extendability of our design. Figure 4.19 shows the design of the planner sub-system

As shown above, the planner sub-system maintains an interface *PlannerAlgorithm* that is common to all candidate planning algorithms, e.g., H^2MAP and *PlannerAlternative_A, PlannerAlternative_B* in this case. Concrete implementations of the planning functionality are therefore realised in the actual planning algorithm codes written in each individual algorithm. The *PlannerConfigurator* interface is an single entry point for the caller code (client) to access the planner algorithm. It forwards the requests of the planning operations to the planner and the user could also determine through the *PlannerConfigurator* interface which planner is to be used for the current planning operation. Additionally the *PlannerConfigurator* takes care of the house-keeping tasks by preparing data required by the planner. This design constellation allows flexible swapping of the planning algorithm without major modifications to the sub-system during the runtime of the planning system. Moreover, as the provider of the core planning functionality, the planner algorithms are detached from assisting tasks and thus kept as simple and clear as possible. The use of interfaces allows a separation of concerns in the system design, so that as the planning system evolves, changes made to part of the system will not seriously damage the holistic system design.

Component	Functions Related to Planner
Knowledge Repository	The planner component should be capable of connecting to the repository and retrieving planning knowledge when necessary.
Data Processing	The planner provides the solutions in its own data format and forwards them to the data processing unit to translate into other data formats. Those formats are either suitable for executing the actions in the system or suitable for presenting them to the user for inspection of solution plans.
System Interface	The planner gets and dispatches the plan solution indirectly to the system interface, which is responsible for presenting to the user or executing the plans on the target system. The indirect communication between the planner component and system interfaces is bridged by the data processing components, as data need to be mutually parsed and translated into corresponding formats.

Table 4.1: Interfaces of the Planning Component

In order to support the selected planning algorithm, the *PlannerConfigurator* needs to maintain a set of supporting functional operations to other supporting components to collect and prepare data. The list of the core operations of the planner component and additional operations concerning interactions with other components are listed in Table 4.1:

The planner sub-system delegates all those tasks that are not directly related to the planning algorithm to the *PlannerConfigurator*, such as data preparatory jobs. It works as a proxy between the system user and the planner algorithms. All those required functions for other components are embodied in the operations listed in the *PlannerConfigurator* interface as shown in Figure 4.19. Table 4.2 shows the operations encapsulated in the configurator interface with a description of each of these operations.

In this design, the planner itself is completely encapsulated in the framework with no direct contact with external entities. This prevents the user of the system, who in most cases is not a planning expert, from accidentally altering the core planning algorithms while still giving them the flexibility to try different planners. The operations listed in Table 4.2 cover the required functions listed in Table 4.1. The operations *ListPlannerTypes* and *SetPlannerTypes* facilitate the use of different planning algorithms. The

Operation	Description
`SetPlannerType`	The caller (client) of the planning algorithms uses this operation to determine types of the planner it wants to apply for the current planning operations.
`ListPlannerTypes`	The caller of the planner could get knowledge of the available planner using this operation. It lists all optional algorithms with their corresponding characteristics for users to chosen from.
`CallPlanner`	Calls the planner algorithm determined by the user with all necessary parameters such as descriptions of current system states, desired goal states, etc.
`GetStateDescription`	The configurator get the state description and forwards it as one of the parameters to call the planner algorithm.
`GetPlanningGoal`	The configurator gets the desired goal for the current planning operations and forwards it to the planner algorithm.
`DispatchResult`	Plan solutions are dispatched to system interfaces for presentations. The results could be represented as executable action commands or simply to show the results to the user in proper human-readable ways.
`ConnectKnowledgeDB`	Establishes the connections to the knowledge database (knowledge repository) and passes this connection to the planner, so that the planner can directly interact with the knowledge repository *during* the planning operations.

Table 4.2: Operations in *PlannerConfigurator* interface

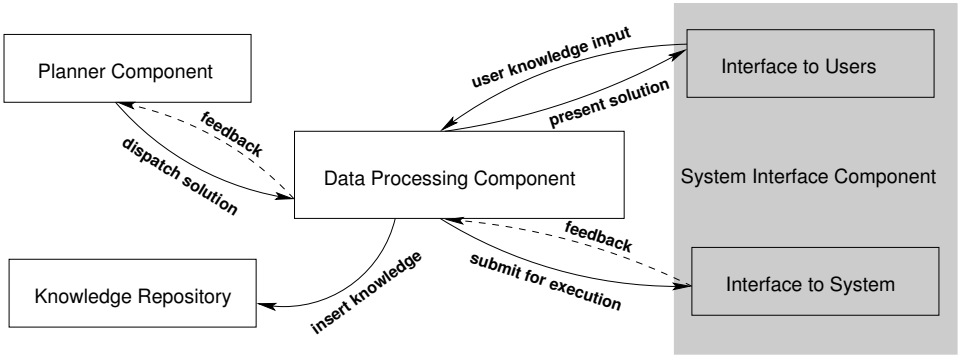


Figure 4.20: Types of data translation operations

CallPlanner operation sends the chosen planner with necessary data and triggers its operation. *ConnectKnowledgeDB* prepares the access for the planner to the domain knowledge and handles established connections to the planner. The rest of the operations are designed to handle communications with the sub-systems responsible for getting and translating the external information such as the description of current states and desired planning goals.

4.4.3 Data Processing Component

The data processing component is mainly responsible for parsing and translating planning information between the planner sub-system and other sub-systems. This sub-system is a host of parsing algorithms presented in the previous section on knowledge translations. The translation process should consider the data translation and reformatting of input and output domain-specific languages (DSL) mutually in two different directions: from the planner sub-system to the rest and from other knowledge sources to the planner. Figure 4.20 illustrates various types of data translation directions needed to be covered by the data processing component.

The translation process from the planner to the rest of the components concerns translating the planning solution into a format that a user wants. There would be two alternatives in the current consideration: presenting the solution to the user interface or submitting the plan solution to the external management system for implementation. The user should have the opportunity to determine in which ways he wants the solution to be presented. If the solutions are intended for direct execution on the target service, the data processing component needs to parse and translate the current plan solution into a format that is known to the execution mechanism which is attached to the planning framework. The targets of the translated format may vary according to the user's requirement, for example, a user may require a check

to be made of the solution plan with a web-based presentation; in this case the data processing component has to parse the plan format into HTML or some other web-based presentation forms. In other cases, a solution plan may need to be executed on a execution platform that is attached to the planning system; in this case, the data processing component needs to translate its input from the planner sub-system into a language that is supported by the attached execution mechanism. For example, if some workflow system is attached to the planner system, solutions of the planner have to be translated into a workflow language that is compatible with the system.

Furthermore, the data processing component also translates the user input into the language that is understandable to the planner. This process is especially crucial for the enrichment of the planner knowledge in the knowledge repository. For this purpose, we designed a set of simple configuration language specifications, as presented in Section 4.2.2, with corresponding translation algorithms (shown in Section 4.2.3), which parses and translates the knowledge encoded in the configuration language. User input knowledge is forwarded by the data processing component to the knowledge repository for future reference.

Optionally, as shown in Figure 4.20, the planner components may need instant feedbacks from the execution system to monitor the progress of the plan implementations and reacts accordingly if any unexpected events occur. In this case, the system interface sends an event notification message back to the planner through the data processing components, where system-specific messages are translated into a format that is comprehensible for the planner. The data processing component forwards the event notifications to the planner after the translation operation.

Depending on the translation directions, the input and output DSL may vary in applications. The data processing components must be flexible enough to cover such dynamics by allowing easy adaptation and extension of new translation approaches and algorithms for the cases that desired output or input changes. In our design, we assume that the input and output language of the planner components is invariably PDDL, since it is the *de facto* standard planning description language as we discussed in the previous section. Thus, all information exchanges between the data processing component and the planner component are conducted using PDDL.

Our translation architecture, as defined in Section 4.2.3, consists of lexical analyser (lexer), syntactic and semantic analysers (as parser) as well as target language generator. Based on the fact that the output language is invariantly PDDL, the operations in the data processing component are shown in Table 4.3.

Figure 4.21 shows the essential operations included in the data processing sub-system. The above half of the workflow illustrates a data process stream from the planner component to the system interfaces. The plan

Direction	Target System	Description
Planner to Interfaces	Execution System	Use PDDL-specific lexer to analyse the solution from the planner. Use parser that is specific to the target execution system (e.g., workflow system) to translate solution into the target format.
	User Interface	Use PDDL-specific lexer to analyse the solution for representation; use front-end-specific parser to translate and generate the target language to show the results on the user interfaces (e.g., web presentation).
Interface to Planner	(User to) Data Proc.	Use the lexer specific to the front-end to get user input and translates it into PDDL language.
	(System to) Data Proc.	Optionally, translate the execution event caught by the execution system into a format understandable by the planner component.
Data Proc. to Knowledge Repo.	Knowledge Repository	Insert translated user input into the knowledge repository.

Table 4.3: Directions of translations and operations

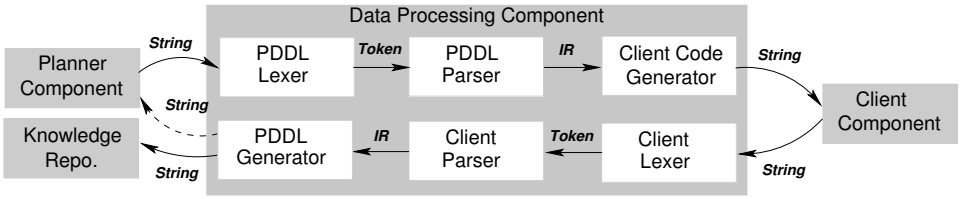


Figure 4.21: Detailed views on the translation processes in both directions

solution produced by the planner component is initially analysed by the lexical analyser, through which all tokens of the input plan solution are extracted. The following parsing operation performs semantic analysis using the parsing algorithms proposed in the previous discussion. The parsing operations emits the given plan input in an Intermediate Representation (IR) form, such as abstract syntax tree (AST) for code generation mechanisms. Finally the code generator operation generates the final codes in a format that is required for the representation. These codes could be destined for the representation of the plan solution in certain types of user interface representation (e.g., HTML for a web-based presentation) or directly submitted to the target system for execution (e.g., workflow system). To produce the suitable target code for the target system, general code generation approaches such as the tree-walking technique [LASU06] can be employed if the IR is presented as AST. For each target DSL, there is a corresponding coder generator provided.

Additionally users of the planning system should have the opportunity to dynamically configure and control the productions of target codes in different DSL. To fulfill this requirement, we need to provide users with a set of *target code selection* operations to allow them to choose among different code generation mechanisms for various target DSL. The target code selection operation determines a proper code generator for the target system. In this way, we separate the analysis process of the input DSL from the code generation process, so that the analysis part of the process can remain unmodified even if the types of the target system are changed. For example, if instead of a web-based presentation, the users want the plan solution to be presented in a user-specific application such as a desktop-based client, they only have to develop a suitable code generator for their own applications and the rest of the translation process can therefore remain unmodified. This design gives the users of the planning framework maximal flexibility to adapted the planning system to their own application by simply re-targeting the output generator. Users can easily extend the framework in their desired ways. The same principle is also valid for the generation of the system-specific DSL codes for the target systems so that the data processing component, hence the planning system, is not bounded to any specific target execution system and is freely configurable.

The lower part of the translation process stream takes the opposite direction, in which the user input or, optionally, system events in forms of DSL are translated into the plan description language. The process is similar to its counterpart. Depending on the sources of the input, the lexer analyses the structure of the input DSL and gets all tokens extracted. A client-specific parser then produces the results of the semantic analysis in terms of IR for the coder generator to produce the target language, in our case, the PDDL description language. The translation process in this direction is

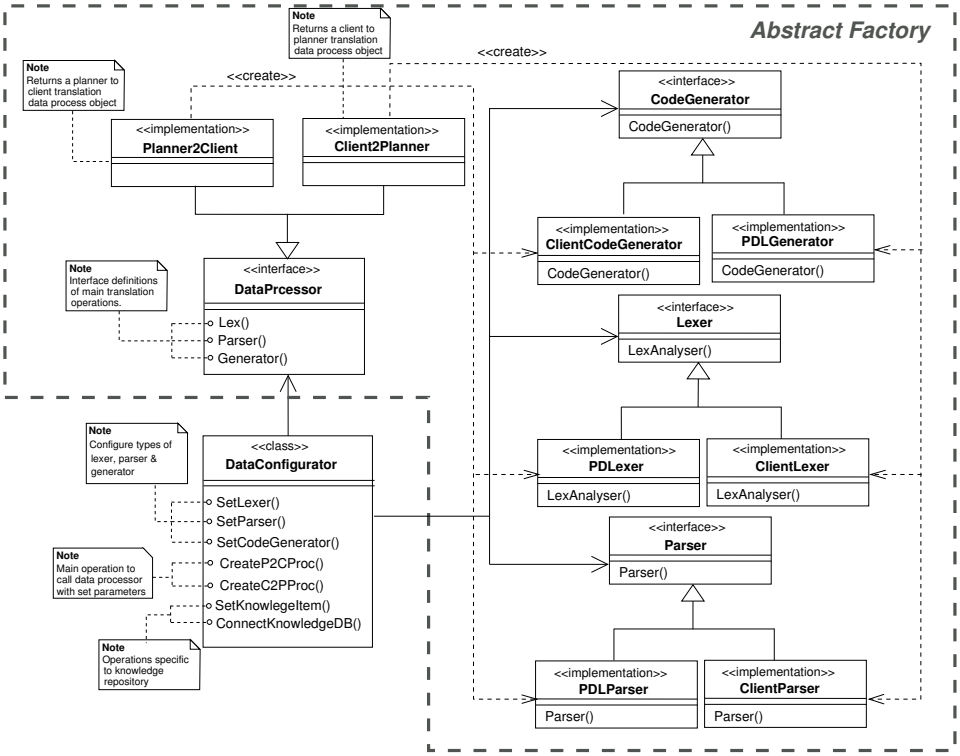


Figure 4.22: An UML view of the data processing component design

especially important for the enrichment of planning knowledge, where knowledge regarding fault recovery is translated from a user- or system-defined DSL into a format that is comprehensible for the planner. After a suitable translation, the input is inserted into the knowledge repository for future planning operations. For interactive planners, where instant feedbacks of the target system are required, the translated planning information such as system event notifications can also be directly forwarded to the planner component itself, which may influence the current computation. In such case, although the consumer of the data processing component is changed (e.g., different planners), nevertheless, the translation process in the data processing component remains unaffected.

As shown in Figure 4.22, the system design provides its user with possibilities to configure the data processing sub-system of the framework by dynamically determining the different parser, lexer and coder generator types. The user can also use default translating components if no particular preferences are given. The fact that there are no hard-coded translation processes in the design allows the holistic planning system to be easily adaptable to the different target languages and therefore their corresponding target

systems. The proposed design of the data processing component uses the *Abstract Factory Pattern* [GHJV95], where desired *products* are assembled in a component factory and details of the creations of the system are hidden from users, since they merely get the desired systems as objects delivered to them from the factory. Nevertheless, we alter the original pattern by allowing users to determine different translation component to their own preference; however, the concrete implementation of components and how the final objects are assembled is still hidden from them as originally proposed by the pattern.

The main participants in the pattern are shown in the box with dashed lines in Figure 4.22. Different translation components, e.g., lexer, parsers and code generators, are prepared as a set of interfaces for the object creations. These interfaces generalise and hide the concrete implementation codes, so that potential modifications of the implementations or introductions of new translation mechanism will not affect the system as a whole. System developers could therefore improve current or add new component implementations into the abstract factory without affecting the user's codes, since the implementation is hidden from the user. Furthermore, the users of those components are not exposed to the complex internal working details.

As the main component of the sub-system, the *DataProcessor* defines the rules for the object creation processes, where core functionalities are declared as a list of abstract operations. Conceptually a data processor represents either the translation of DSL from planner to client or vice versa, as shown in Figure 4.21; therefore, the interface is implemented by two concrete creation processes: *Planner2Client* and *Client2Planner* which represent both translation directions. Each of the created translation processes includes a set of working translation components including lexer, parser and code generator of various types. Both implementations of the *DataProcessor* interface consequently produce and return the user-desired translation mechanism in each corresponding direction, respectively. The creations are depend on the set of translation components as listed on the right-hand side of the diagram. However, the actual creation processes as well as implementations of the components are hidden from the user.

To trigger the generation processes of translation objects, the caller of the data processing sub-system uses the *DataConfigurator* class to gain access to the factory by initiating the object production mechanism. Since *DataConfigurator* embeds the operations to initiate the requests of the object creations, it can be regarded as a client of the abstract factory and the returned objects as the products of the factory. The client relies on *CreateP2CProc* and *CreateC2PProc* operations to drive the object creation processes. If the user has preferred translation components, he can use the series of *set* operations, e.g., *SetLexer, SetParser* and *SetCodeGenerator*, defined in the class to select his desired component types. Otherwise, if

no specific preferences of component types are given, the system applies a default set of components defined by the system developer. The returns of both operations are objects with concrete implementations of translation components encapsulated. Provided the proper input data as parameters, the caller can use these objects to perform desired translation processes and generate target output languages accordingly to the translation directions. Albeit its role as initiator, the *DataConfigurator*; however, does not have control over how these objects are created, since the complex internal workings are hidden in the factory, which is a positive effect of our design, because the details of the creation processes are for the users of the system not germane. Still the users are left with the possibility to manually determine component types involved in the returned objects.

Not only is the *DataConfigurator* applied as the caller of the factory, it also determines other operations that are not directly related to the translation processes but are still pertinent for the data processing component. In the current design, we consider the interaction of data processing subsystem with the knowledge repository sub-system by connecting the knowledge database and inserting input recovery knowledge. These operations are carried out by *ConnectKnowledgeDB* and *SetKnowledgeItem* operations. In order to insert the obtained recovery knowledge into the repository component, it is essential to first establish the connection to the database that persistently stores the information. Due to the fact that various types of the storage mechanisms may be applied as knowledge repository, the *ConnectKnowledgeDB* operations wraps the connection operation details and returns the user to an established database connection object, through which the data processing component could directly dispatch the translated knowledge into the target knowledge repository, for example, if the Java^{TM}is used in the implementations of the framework, the *ConnectKnowledgeDB* operation could establish data connections using JDBC API [JDB] which facilitates the data processing component to access the target relational database with operations such as update, query and insert into the database. However, if an other database such as CouchDB is used as the storage mechanism, then the *ConnectKnowledgeDB* should prepare the database connection using approaches such as jCouchDB [JJCD] to bridge the communications between the database and data processing component.

In the proposed design, connection details are hidden from the users of the system. The *ConnectKnowledgeDB* operation wraps all complex operation details and returns an established database connection to its user regardless of the type of database applied underneath. After the connection handle is returned, the data processing component uses the returned handle to insert the newly obtained knowledge into the knowledge repository. This operation is simply performed by the *SetKnowledgeItem*.

To summarise, the goal of the data processing component is to provide

DSL translation mechanisms between the components involved in the framework. Due to the variable input and output DSL, flexibility is an essential factor to be considered in the design. Furthermore, the component should be easy to use for other components as users and the complex working details of translation mechanisms should be hidden from the user. To address these requirements, we applied in our design the *abstract factory* pattern, with which the internal working details of the creations of translation objects are opaques to the user. To invoke the creation operations, the user can simply rely on an class which serves as the access method, which initiates and triggers the creation operations. The details of the creation are done internally in the factory, where different implementations of translation components are dynamically composed into an object. The object which encapsulates the desired components is returned to user for further application. Furthermore, the component also prepares database connections to the knowledge repository. To address different types of information storage system that the planning framework may have, the data processing component wraps the details of connection establishment into an operation, which simply returns a connection handle for its user to perform database-specific operations.

To conclude, the proposed design has the following advantages:

- The translation object required by the user can be dynamically created either with a default set of translation components or with user-specified types.

- The design allows the developer to control the creation of user-specific translation object creations in an actual application and, therefore, enforces the consistency among the created objects. Because it encapsulates the creation processes and separates the user of the system from the actual implementations of the components, the user can only manipulate the created instances through the defined abstract interfaces.

- The actual creation processes are hidden from the user. As a result, the user simply gets a translation object returned from the creation process to address his requirements.

- Due to the abstract interfaces in the design, it is easy for the developer of the planning system to modify the implementations of diverse translation components, such as lexer, without influencing the user code.

- As the target systems evolve, new translation components may be needed to address such changes. The proposed design allows the system developer to easily add new components targeted at new DSL and thus new systems.

4.4.4 Knowledge Repository

The overarching goals of the knowledge repository are relatively straightforward: first, it provides a unified central data storage mechanism for the recovery knowledge and related information for the framework; second, it facilitates other components to retrieve and manipulate knowledge entries when necessary.

The knowledge repository interacts with the planning component and the data processing component, whose designs already include operations that reflect their usages of the knowledge repository. During the operation, the planner establishes a connection to the knowledge repository for retrievals of information that is relevant for its current planning operations. To reduce the data communication overhead and to suppress the frequency of data exchanges between the components, the planning information is organised in the form of plan domain files as we discussed previously, which include recovery abstract tasks, decomposition methods, and actions that are available to the current recovery task. By doing so, we allow the planner to fetch a complete plan domain description file with the necessary planning knowledge with a single operation. The data entries in the knowledge repository are description files for certain plan domains. Domain descriptions files are text-based documents with keywords which identify specific recovery information such as *action, decomposition method*.

Other than the planner component, the data processing sub-system acts as an interface for the users to manipulate knowledge entries in the repository. Therefore, a data store mechanism of the knowledge should allow some data manipulation primitives, such as update and insert the newly acquired knowledge into the database in order to allow users to enrich and extend data inventories in the repository.

The planning knowledge is encoded in PDDL format for the current design, which is a document-based data structure as defined by its specification. Therefore, any data storage system that is capable of persistently retaining semi-structured data types is more appropriate to be applied as possible implementations of the repository, since they can handle semi-structured or even unstructured data types. Moreover, to facilitate the operations related to information retrievals, the select data storage mechanism should also support basic database-like operations which allow the client of the repository to simple query, update and insert operations. Despite the fact that there are currently many possible candidates available for the implementation of a knowledge repository, for example, document-based data storage approaches for NoSQL data storage systems [Bar10, CDG$^+$08, LM10] which are currently gaining prominence among different applications, our discussion is not focused on any particular kind of storage mechanism or system in order to keep the component as generic as possible. Decisions on applying which specific storage mechanism as the

Operation	Functional Description
`Update`	The user invokes this operation to update the currently planning knowledge such as decomposition methods, action description.
`Insert`	This operation adds new planning knowledge to the repository for certain planning domain description files. The user could also insert new planning domain into the repository with this operation.
`Delete`	The user could actively deletes any obsolete planning knowledge if it is no longer required.
`Get`	Retrieve planning domain description files that are required by the planner.

Table 4.4: Generic Operations for Knowledge Repository

implementations of the knowledge repository is therefore left to the developer. To that end, we focus our current discussion on the generic operations that are required to be supported by the knowledge repository to fulfill its designated functionalities.

Table 4.4 lists four generic primitive operations that must be supported by the knowledge repository. With those operations, clients of the knowledge repository (either the user or system component) can interact with the repository to either retrieve or manipulate knowledge entries. The planner components use the *Get* operation to retrieve the domain descriptions with planning knowledge from the repository and caches the retrieved knowledge in the local storage for instant access during planning. The *Update, Insert* and *Delete* operations allow a user (e.g., domain expert) to manipulate the knowledge items in the repository. Table 4.23 shows the communications processes between knowledge repository and other relevant components.

Figure 4.23 (a) offers an overview of the interactions between components that are related to the knowledge repository as discussed previously; part (b) of the diagram provides a detailed view of the inter-component activities in the form of a sequence diagram.

A more advanced design of the knowledge repository can also include features such as a syntax-based check of the input recovery knowledge to rule out possible errors in the input knowledge, so that we can reduce the possibility of false positives in results of the planning operations. Furthermore, the repository can also be used to log plan history for future references, with which approaches such as case-based planning [CMAB05] can use the logged plan history to perform planning operations.

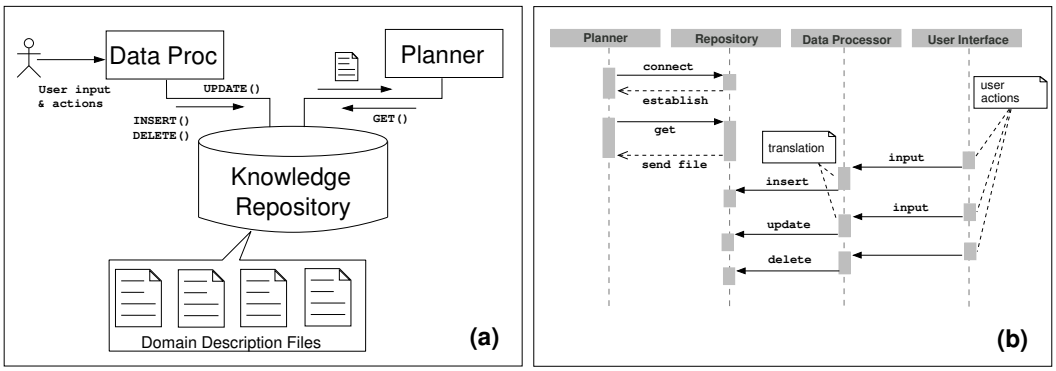


Figure 4.23: (a) Repository operations and interactions with other components; (b) Sequence diagram of cooperations between knowledge repository and relevant components

4.4.5 Planning System Interfaces

The planning system interfaces are the border and mediator between the planning framework and external systems. They provide ways to facilitate the participating systems to exchange data and to communicate with each other in the well-defined manner. From the external systems perspective, the interfaces provide them with planning solutions that are ready to be implemented to fulfill the designated management goals. From the framework perspective, the interfaces allow the planning system to acquire new planning knowledge from the user input. If the planning algorithm is featured with interaction capability or it requires constant feedbacks from the plan execution, all such events will also be required to communicate to the planner through the system interfaces.

As previously shown in Figure 4.18 (on page 170), the planning component faces two types of external systems: the *user-side systems* for presentations of plan solutions and knowledge acquisitions as well as the *attached management/execution systems* of the target services for plan executions.

A user-side system refers to any system that users of the planning framework apply to present solutions in their specified ways, for example, while one may prefer the plan solution to be presented in a web-based system, others may want the solution to be integrated into their applications. Additionally, our system design should also allow users as domain experts to edit, update and insert planning knowledge from the user interface, so that the knowledge repository could be enriched for more planning tasks.

An attached management system acts as the execution mechanism for the planning framework; it is responsible for implementing the devised solutions in their pre-determined order. On the other hand, the attached system could also monitor the execution during the execution process and provides

the planning system with any feedbacks of the execution if required. For example, a workflow system can be used as an execution mechanism for plans and a monitoring system can track the execution status of the current plan and propagates any system event back to the planning framework if necessary.

Depending on the initiating side, the communications and data exchanges between participating systems take place in a bi-directional manner. We classify the operations into two major categories as the following:

- Planning-framework-initiated communications. Messages or data generated by the planner are internally processed by the planning framework and sent back to external systems for either direct execution or presentations. The planning framework initiates the communications with the system interfaces to perform the tasks of calling corresponding external functions for the desired purposes.

- Client-systems-initiated communications. The client systems, either user-specific or system-specific, send event messages and data back to the planning system. The client system should be capable of performing such tasks through the defined operations within the system interfaces component. However, the working processes internal to the framework are hidden from the calling client.

Based on the discussions above, we extract and define a set of generic operations required to facilitate communications and data exchanges between the planning framework and external systems. The operations are discussed separately according to the types of external systems.

Table 4.5 shows a set of operations defined for the user-side system interface. These operations allow the planning framework to render plan solutions and get user input for the plan knowledge repository. A user can apply this interface for their application-specific implementations to show solution plans and manipulate knowledge items. In order to perform the knowledge manipulation operations, a caller needs to specify the types of knowledge items he wants to update in the call as parameters, e.g., if the knowledge to be inserted is a decomposition methods or management constraints or management actions. Additionally, the user also needs to specify the name of a domain file, in which the plan knowledge is to be edited. The same principle also applies if the user wants to create or delete certain plan domain files.

Table 4.6 lists operations designated for the mutual communications between the planning framework and external management systems. The operations allow the planning framework to execute the composed plan on the system and, furthermore, enable the system to get feedback events back to the framework if required. After the computation of a recovery plan as a solution to a recovery problem, the data processing sub-system is triggered

Operation	Descriptions
`RenderPlan`	This operation in the interface intends to wrap in a user-specific way to render a plan in a user-specific ways. The caller of this operation must encapsulate the platform-specific rendering method for its applications.
`UpdateDomain`	This operation allows a caller to update planning knowledge in a designated planning domain file in the repository. The caller must specify the target plan domain and specific knowledge items it wants to update with this operation.
`DeleteDomain`	Allows a user to delete a specific planning domain file.
`CreateNewDomain`	Allow a user to create a new planning domain.

Table 4.5: Generic operations to the interface for presentation of planning solution and knowledge acquirement

Operation	Description
`ExecutePlan`	This operations allows the planner to call the executing mechanism of a user-specific system.
`DispatchSystemEvents`	If required by the planner algorithm, the external management system can optionally dispatch system events back to the planning framework

Table 4.6: Generic operations to system-specific interface for external management systems

to convert the plan solution into a system-specific DSL and ready for execution. The framework uses the `ExecutePlan` operation to implement the actions in the plan on the target external system. By calling the operation, the planning framework does not have to be aware of the type of target system and how the plans are actually executed. Types of different end-system should be hidden from the planner. The operation `DispatchSystemEvents` works in a different direction to the plan execution operation. By calling this operation, the external system tries to send a system-generated event during the planning execution to the planner. The transmitted events messages are passed by the system interface component to the data processing component, where the data are translated into the planning language that is comprehensible to the algorithm.

While the internal workings of the planning system can remain relatively constant, meaning the involved components and formats for exchanges are not changed very frequently, types of external systems may, however, vary drastically according to the application scenario. Some aspects of such dynamics have been addressed in the design of the data processing component; nevertheless, the discussion only concentrates on the data models and translation of DSL between systems both internal and external to the framework without consideration of the system aspect of the end-system dynamics. We deliberately left the discussion of the end-system dynamics to the current section on the system interfaces component to follow and to advocate the principle of separation of concerns in the design.

Other than the data processing sub-system, which deals with the conversion of different planning data, the system interface component has to cope with incoherent external system types users may apply. It is responsible for calls from or to external systems and takes care of data communications, regardless of the syntax and semantic of the data that are transported between the systems. The system interfaces should also facilitate the external system in order to send information to the framework.

To address the issues of the external system dynamics in the design, the system interfaces component should be defined in such a way that changes to the underlying system would not lead to significant modifications of the framework.

To ensure such flexibility, we apply the principle of the *Adapter Pattern* [GHJV95] in our design. The pattern[4] suggests using a wrapper instance to translate an interface of the system into another interface expected by a foreign system. It proposes a way for two different systems to work together that otherwise could not be done because of incompatible interfaces. The working principle of the adapter mechanism is illustrated in Figure 4.24. With this design principle, the adapter is going to be the only changing

[4]There are two types of adapters: *class adapter* and *object adapter*. In our design we use the object adapter approach to avoid multiple inheritance.

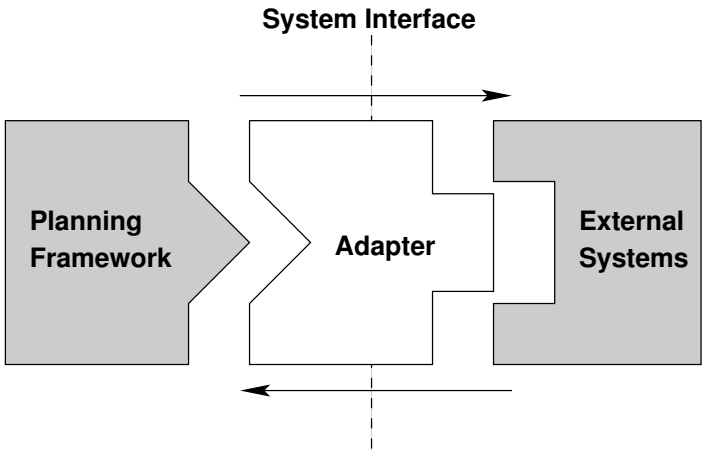


Figure 4.24: Working principle of the adapter pattern

element in the design; systems that intend to communicate with each other on both sides of the adapter are not aware of the specific interfaces to each other. The main advantage of the adapter pattern is that through the use of abstract interfaces, the planning framework can always be reused with minimal changes to the framework itself. If the underlying systems are changed, only the interface needs to be adapted to the new systems; the existing internal components in the planning framework can thus remain intact for the new application. A user application simply calls the wrapped operations which map the interface of one system into the target system interface. In order to apply this design pattern, we assume that the primitive operations such as performing a plan execution and rendering a solution are operations that are generic enough to be available on the target systems.

Before we design the interfaces, we first determine the operations that need to be adapted to the end-systems (also known as the *adaptee*). From all operations listed in Tables 4.5 and 4.6, we identify two operations to be wrapped into the adapter: `RenderPlan` and `ExecutePlan`. For both operations, external end-systems, either user-specific or system-specific, take the role of adaptee. We base our decision on the fact that the external systems are sources of changes in most cases. In other words, the data and operations within the planning framework remain relatively constant. Therefore, only operations that need to trigger the external systems must to be adapted in our case; the rest of the generic operations which trigger the planning framework require almost no changes if the external systems are changed. To perform these operations, the external systems can directly call them as needed. For example, if a user-specific system wants to modify planning knowledge in the repository through a web-based form, he could simply call the `UpdateDomain` operations to achieve his goal; if the user-

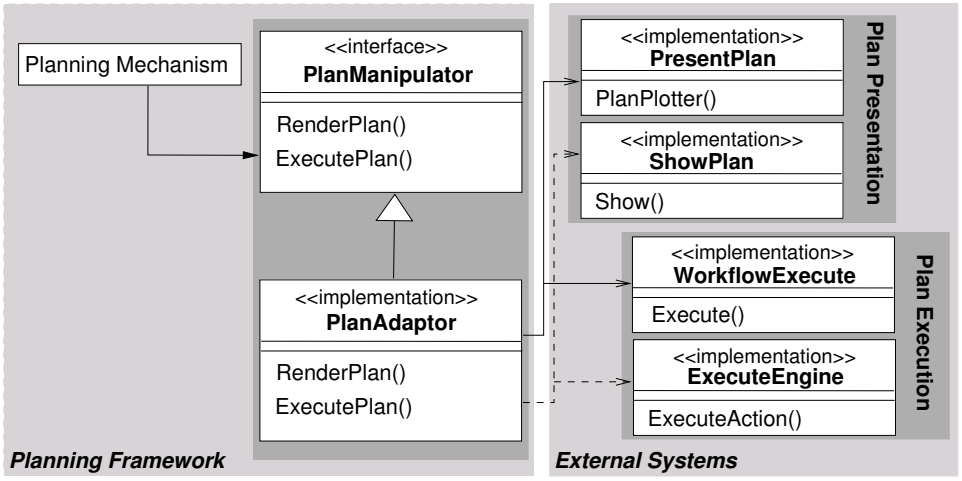


Figure 4.25: Design and working principle of the adapter for the interface component

side system is changed to other desktop applications, he could just execute the same call without any change in the call procedure. Nevertheless, if a planning framework intends to dispatch a plan solution to different types of end-systems, it has to alter the calling procedure to meet the requirement of the changing systems.

Figure 4.25 shows the adapter design in a UML class diagram. As shown in this design, the planning framework does not have to have the specific knowledge on the target external systems regarding their procedures to perform the desired operations. As access point, the interface *PlanManipulator* defines the generic operations that need to be adapted. To conduct operations on the external systems, the framework simply uses the generic operations to meet its needs. In the actual implementation, the *PlanAdapter* realises the *PlanManipulator* interface; when it gets an operation call from the planning component, it delegates the calls to corresponding implementations of the external systems. In the external systems shown in the right-hand side of the diagram, system-specific implementations are listed as examples. For the plan presentation operations, two operations are available for two different user systems, as examples, one uses the `PlanPlotter` operation and other relies on the `Show` method to present the plan solution for the systems specified by the user. The same principle also applies to the external plan execution systems as well. The adapter relies on the composition principle to dynamically tie together the necessary system-specific implementations.

To conclude, the design of the system interface component allows the information-hiding principle to be applied by encapsulating plan operations into the framework. In most cases users of the planning framework are not

interested on how planner operates; rather they care more about how their designated systems interact with the framework, for example, how the plan solutions are presented and implemented in their designated applications. To this end, at the border of the planning framework, the system interface component defines a set of operations to enable data exchanges and communication between the planning system and external systems, without revealing the internal working principles of the planning mechanism. With this set of operations, users could not only easily retrieve and display planning results on their specific systems, but also input their specific planning knowledge into the repository with their platform. To execute plans on the system, the operations defined by the interface allows the framework to easily talk to different execution systems and monitor any relevant system events during execution. To ensure the flexibility in the design, especially regarding the dynamics towards the end-systems, we apply the adapter pattern in the design of the operations with a need to trigger the foreign methods on the external systems. The adapter decouples the internal planning mechanism from the actual operations of external systems. If external systems are changed over time, only minimal modifications to the adapter are required. For the adapter approach to work, we assume that the end-systems possess operations that semantically match the generic operations defined in the interface.

4.5 Summary

As the core part of this research, this chapter delivers contributions ranging from theoretical investigations of solving the IT service recovery problem with planning techniques to the specific design of the planning algorithm based on the HTN-based planning paradigm. To support the operation of the proposed planner, a range of supporting components is required, which fulfill different purposes in order to sustain the planning operations. Based on these components, we design a software framework which encapsulates the working principles of the planning algorithm and decouples the users of the planning system from the actual planning operations. To guarantee the extendability and flexibility of the framework, design patterns are applied in our design. The main contributions of this chapter can be abstracted as the following:

- Theoretical investigations of solving the IT fault recovery problem as a planning problem. To provide a solid theoretical foundation for the further research to be built on, we argue and formally show that automated planning techniques is a viable approach in fault management of IT services, especially in the planning of fault recovery operations. In order to support this argument, we first build a generic model for

the IT service fault recovery problem by formalising the essential factors in the model. Then we rely on the problem reduction technique as the formal method to show the reducibility between the proposed fault recovery problem model and the generic planning model. The reduction operation is performed by transforming the fault recovery problem model into a generic planning problem model. Then the transformed recovery problem can be solved with a planning technique.

- IT service fault recovery is a knowledge-intensive operation. To support the automated planning process, we define and model a set of knowledge types that are required to be served as the basis for the planning. We provide the language specifications using the formal grammar for the different recovery knowledge elements. The language is a Domain-Specific Language (DSL) for the description of recovery information.

- Knowledge translation mechanism from user input to PDDL. After investigating diverse description languages for expressing planning knowledge, we choose the Plan Domain Description Language (PDDL) as the input for the planner. PDDL is the *de facto* planning language standard which has been developed and maintained by the international research community. Its expressiveness allows descriptions of complex planning knowledge. Nevertheless, despite its expressiveness, this language is still hard for the non-AI specialist to grasp, therefore, to compile a reasonable planning domain description poses a challenge to users. To reduce such impact, we propose a translation process between PDDL and the proposed DSL. This allows the users to interact with the planning mechanism and to build a planning domain without explicit knowledge of PDDL. The complex details of building a planning domain are performed by a set of translation algorithms which analyse and compile the planning domain description according to the user input.

- Design of the recovery planning algorithm. As one of the core contributions of this chapter, a planning algorithm called H^2MAP is proposed. The H^2MAP is designed based on the HTN-based planning paradigm. Compared to other planning paradigms, the HTN-based planning approach has advantages in terms of performance, knowledge reusability, domain configuration capability and extendability which make it a widely accepted planning paradigm for solving real-world planning problems. The proposed algorithm operates based on the iterative refining of high-level abstract tasks into low-level planning actions. Users of the planning algorithm could guide the planning process by explicitly providing one or more high-level tasks and the algorithm computes the plan solution according to the current states

of the target system. The algorithm is shown to be sound and complete. To improve the capability of the H^2MAP, we also design extensions that handle temporal planning knowledge and uncertain plan information during the operation.

- Design of the planning framework. With the planning algorithm in place, a framework design is provided, which revolves around and sustains the operations of the planning algorithm. To ensure the flexibility of the design, our proposal of the framework and its components is guided by a set of design patterns. The sub-systems involved in the design cover the data translations and persistent storages of reusable recovery planning knowledge. The design intends to encapsulate the complex planning operations and interactions of the components within the planning framework while still providing its users with well-defined and simple interfaces for data exchanges.

5

Prototypical Applications and Evaluations

Contents

This chapter starts with discussions of the essential aspects regarding instrumentations of the core components in general. The focuses are on the language translation mechanisms and the planning component. We outline and suggest how these components in the framework could be technically realised. The discussion on the implementation is, however, generic and independent from specific technologies.

To demonstrate applicabilities and the practical values of the approach, we instantiate and prototypically implement the framework to several application scenarios. For that purpose, we choose to show how the planning framework can be instantiated, adapted and integrated to deal with practical recovery tasks in managing the service recovery in a Cloud computing infrastructure. Not only do we address the issues regarding fault recovery management, to proof the portability of our idea for other management disciplines, we also show an example regarding the planning for change management. This demonstrates that our framework and the underlying mechanism in general are well suited for many management tasks that require planning.

In the final section of this chapter, we summarise and evaluate the framework against the requirements derived in the Chapter 2 and its degree of fulfillment is discussed.

5.1 Implementations of Core Components

In this section, we provide insights regarding the implementation outlines and instrumentation aspects of the core sub-systems. The focuses are on the planning component and data processing component, since they provide the most essential services in the entire framework. Whereas the goal of this chapter is to show how specific system components can be technically realised, we provide implementation suggestions and methodologies rather than providing full-fledged programs. These instructions can be applied as guidelines in the development to construct artifacts of the framework.

5.1.1 Knowledge Translation Components

The translation architecture was explicitly discussed in Chapter 4 in Section 4.2.3 (on page 125). The main working principle of the translation architecture is to first lexically analyse the input description in the form of the DSL encoded with recovery knowledge from user input. After the analysis has been done, we parse the results and map the corresponding items into the planning knowledge structure. Finally the language generator automatically constructs the target output language, in our case PDDL. To implement this translation architecture, we need to construct three components, including *Lexer*, *Parser* and *Generator*, which are similar to those for building a compiler for programming languages. To instrument the translation component we used the *antlr tools*[1], which is an open-source tool with an integrated development environment for the construction of language compilers.

[1] antlr, Another Tool for Language Recognition, `http://www.antlr.org`.

Figure 5.1: Syntax diagram of the input planning document

Lexical Analyser

The lexical analyser (lexer) is the first processing unit of the input language. Its main responsibility is to scan through the input language sentences and decompose the input into a sequence of tokens. Specifically, a lexer reads the characters in the input stream and groups them into lexemes together with their attribute values. Those tokens and corresponding values will be used by a parser for the structure analysis. Additionally the lexer also performs house-keeping tasks such as stripping out the write spaces and comments in the source codes which are not directly related to the semantics of the input information.

In our context, the lexer reads a set of user input specified by the grammar of the recovery knowledge descriptions (as described in Section 4.2.2) and extracts the tokens in the input stream. The user input is a text-based document with different types of recovery knowledge organised in blocks. Figure 5.1 shows a syntax diagram of an input planning knowledge (*pkd*) document as discussed earlier. The document is headed by a description label which denotes a domain, to which all of the blocks in the document belong. The concept of the domain was previously discussed in Section 4.2.3. The start of the input is signified by the `BEGIN` keyword and is ended with the `END` keyword. The loop in the diagram denotes that the definition of the domain contains at least one block.

Note that to simplify the implementation so that the key idea of the translation is more clear, instead of automatically extracting domain information from the block definitions and clustering the blocks that belong to the same domain, we explicitly give the domain in the beginning of document. Nevertheless, an automated sorting and clustering of the blocks according to their types, as already defined in Algorithm 3 (on page 132), can be easily added in the future implementation.

A block is a typed data structure, i.e. each block contains the recovery information entries which are specific to the corresponding knowledge types, e.g., recovery action, decomposition methods, description of system states etc., as they are specifically defined in Section 4.2.2. The information types and detailed descriptions are discussed in Section 4.2.1. The type information is given at the beginning of the block definition by a label, which signifies the expected knowledge types that are presented inside the block. For processing of type-specific blocks, a set of sub-rules are defined. Appendix B.3 shows an example grammar definition of the action block.

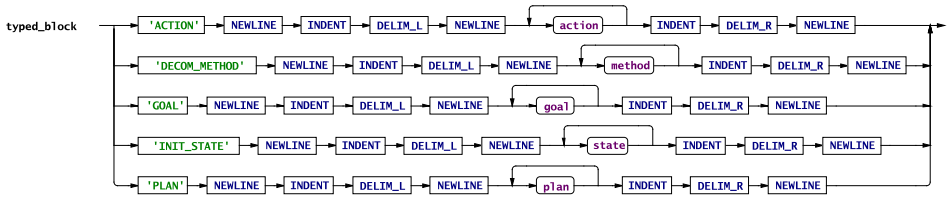


Figure 5.2: Syntax diagram of typed blocks. A block contains multiple entries of type-specific expressions

Figure 5.3: An example of a grammatical structure of action block in syntax diagram

Figure 5.2 shows as an example syntax diagram which contains different typed information definitions in the form of typed blocks. The different typed blocks have similar grammatical structures. Depending on the type information, corresponding sub-rules are invoked if they are correctly recognised to process the input information. The translation mechanism determines which grammatical sub-rules to follow by recognising the labels that convey the type information.

For a block definition, the translation mechanism expects a pair of delimiter symbols that are used to mark the start and the end of a block input. Inside a block, a set of description entries specific to that information type is given. Types of entries are, as already discussed, given in the corresponding grammatical sub-rules. An information entry represents a block attribute, which is an expression labelled with a keyword with its corresponding values. It is separated by a special symbol and ended with a semicolon.

As can be seen from Figure 5.3, a definition of a block entry starts with a double indentation (`INDENT`) as syntax sugar; each entry definition contains an expression which is a combination of keywords in terms of literals and their corresponding values, separated by a colon. Each block contains at least one such entry.

With the clearly defined document structures, a lexer is created to analyse the input stream and to find all tokens involved in the document. Details of the implementation of the lexer are provided in Appendix B, Section B.1. The implementation code is written in the Java programming language.

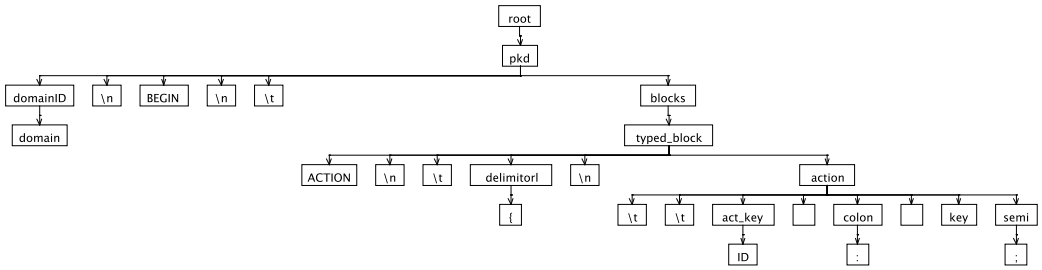


Figure 5.4: A Partial Parser Tree

Parser

After the lexical analysis phase, a set of tokens involved in the input document is produced by the lexer. All of the recognised tokens are saved in a data structure called a *symbol table*[2]. Based on the extracted tokens and defined grammatical rules, a parser performs further operations to analyse the syntax structure of the inputs by organising the tokens according to the rules defined in the grammar. The parser must recognise not only the keywords in the input document, for example identifiers used inside the block definitions, it also needs to parse the structure and sub-structures of the input information. Additionally, the parser scrutinises the input language to determine whether it conforms to the syntax of the given grammar as defined in the previous discussions. It raises exceptions in cases of mismatches and errors. The main task of the parser is, however, to identify the structure of the language and to associate the extracted tokens with their corresponding values. The parser emits as result an intermediate data structure called *Abstract Syntax Tree (AST)*, which reveals the processed syntax structure of the input information.

As an example, Fig. 5.4 shows a partial AST[3] generated by analysing a simple information block. ased on the grammatical specification, the parser recognised that the input data is the planning knowledge definition (*pkd*) which has, as defined in the grammar, a domainID called *domain*. The current definition contains a block typed as plan *action* by recognising the label *ACTION* at the beginning of the block. This event triggers the grammatical sub-rule to process the recognised structure as an action block with information types defined for recovery actions. The implementation code of the parser is listed in Appendix B. The AST data structure generated by the parser builds an essential basis for the next processing step to generate the target language.

[2]In the current implementation, tokens extracted by the lexer are stored in a file called `PKDL.tokens`.

[3]A complete AST is too large to be presented on this page.

Generating the Target Language

The last phase of the translation mechanism is to generate the target language that conforms to the format expected by the planner. The translation process requires fully comprehending of the input phases, then selecting an appropriate output format or construct, and finally filling it with the language elements from the input model. In our implementation, the input language is the knowledge specifications written with the defined grammar. In order to perform the translation operation, we need to define a set of rules to guide the generation process. In the current implementation, we take PDDL as the target output language. Thus, based on the previous language analysis results and grammar, we define a set of rules that produce the PDDL code.

Translation approaches can be roughly classified into several major categories [LASU06, Par09]: *syntax-driven translator, rule-based translator* and *model-driven translator*. With a syntax-driven translator, the output is immediately emitted as the translator reads input, while a rule-based translator uses a rule engine to specify the mapping between input and output language elements. The model-driven translation approaches commonly build up an internal data structure of the input language before the target languages are generated. The internal representation of the input languages is an universal and independent of the output, for example, an AST is one of many representation possibilities. Compared to other approaches, model-driven translation is more flexible since the rules that define the generation processes are not hard-coded. The separation of the language model and output mechanism allows us easily adapt the translation component to other output models as the requirements of output languages change. The separation of concerns of the model-driven approaches make the whole translation process easy to maintain and to debug.

In the current implementation, we instrument the StringTemplate [Par04] engine to assist the target language generation. StringTemplate is one of the many model-driven translation approaches; nevertheless, compared to other approaches, the StringTemplate engine enforces a strict separation of models and views. In other words, the logics concerning the input language analysis and building of intermediate representation data are kept in the code, and the output language is encoded in one or group of templates. In this way, as soon as a requirement on the output language is changed, we could simply swap out the current template with the one designated for the new output language; the rest of the translation procedures remain intact. In this way, the code for language analysis phase can be written once and always be reused, regardless of the target languages. The separation-of-concerns concept with StringTemplate is in alignment with our design principles of the knowledge processing component, whose gory details were discussed in Chapter 4, Section 4.4.3 (on page 174).

A template encodes a set of structures of the output language, which can be used to map the analysed input language elements to the output structures. The mappings of language elements between the input language and target language were discussed in detail in Chapter 4, Section 4.2.3 (on page 126). The templates also dictate and instantiate the translation rules, which means the translation logics expressed in the translation algorithm could all be embedded in the templates. StringTemplate is a programming library which can be used to create templates of the output language. The template defines the keywords and structure of the output. The values that need to be inserted into the output document are controlled by a set of attributes defined inside the translation logics.

For a simple translation, the translation logics in terms of template rules can be directly embedded as part of the input grammar; the corresponding rules will be triggered if the input stream conforms to the grammar. Nevertheless, it is not a very flexible way to use the library, especially in terms of translating and generating large, complex, cascaded inputs. The set of the translation rules could be gathered in groups and put into a file, which only contains such rules. In the grammar, the grammar rules are matched to the translation rules defined in the file to generate output. The actual values in the input stream are extracted by the template and passed to the generation rules as parameters to set the attributes of a particular generation rule. For each typed block of the input, we define a set of corresponding translation rules and group all those rules together in an independent template file.

For example, to translate a parameter list from an input action block, we define the rule in the template file:

```
param_header() ::= "(:parameters (and ";
param(v, t) ::= "?<v> - <t>";
param_tail() ::=  ") "
```

In this definition, `param` is used as a rule name which has an attribute `pms`. In the body of rule definition, it produces a parameter definition expression compatible with PDDL specification, which is:

```
(:parameters (and ?var1 - type ?var2 - type))
```

The generation rule is embedded inside the grammar definition of the corresponding block. The following code shows the integration of the rules.

```
var     : 'VAR' WS varname WS type WS
                -> param(v = {$varname.text}, t = {$type.text});
param : 'PARAMETERS' -> param_header()
            WS COLON WS (var SEMI)+
            NEWLINE -> param_tail();
```

Upon recognising a parameter entry inside an action block, the `param` rule is triggered to perform the generation procedure. With this approach, if the target language is changed, we simple adapt the definitions of the translation rules inside the rules file; the rest of the translation approach does not need to be changed at all. For a complete implementation, a set of generation rules needs to be defined for each typed block inside a recovery knowledge description file. For a more complex target domain descriptions file, for example, if quantifiers such as `Forall, Until` etc. are included, sophisticated techniques such as AST tree-walks may be required to facilitate the language generation process, which will be integrated as part of our future work.

5.1.2 Planning Algorithm

The planning algorithm performs operations to compose viable plan solutions. A correct solution plan is expected to transform the current state of the target service, which contains faults, into a set of fault-free states. The main operational character of the H^2MAP planner, as proposed, is to refine the high-level tasks recursively into finer sub-tasks. The refinement operations continue until the low-level tasks result. Compared to the classical planning approaches, the suggested algorithm searches and selects suitable decomposition methods instead of pure low-level actions. This property makes the planning with hierarchy more efficient and practical for real-world planning tasks, which many involve a large quantity of selectable actions to be chosen from.

The operation of the H^2MAP planning algorithm is sustained by a set of auxiliary functions. The most essential ones in those functions, as stated in Algorithm 23 in Chapter 4, are: `Apply()` and `Find()`. The `Apply()` function is responsible for implementing selected actions in a partial result on the current infrastructure. Depending on whether the planner is implemented in interactive or non-interactive mode, this function either simulates the results of an selected action on the infrastructure model or executes directly on the real infrastructure by using the system interface.

With the non-interactive simulation of the effects of the selected action, the planner holds globally a list data structures called *CurrentState* (denoted as i_0 in Algorithm 23). The list contains a current state descriptions of the infrastructure which can be dynamically altered during the course of action executions. As defined, the recovery action model contains a set of effects, which are the expected state changes as soon as the action is executed, the *CurrentState* is thus updated correspondingly to simulate the expected results. Positive effects listed in the expected results of an action are inserted into the *CurrentState* and corresponding negative effects are removed from

the global list[4]. With each call of the function with state-changing actions, the global *CurrentState* is modified by adding and deleting terms in the list.

```
(CurrentState
              ...
              (running serv1)
              (installed httpd)
              (configured serv1 httpd)
              (stopped web_service)
              ...
              )

(list (:action (start_web_service)
       :precondition ( (running serv1)
                       (installed httpd)
                       (configured serv1 httpd)
                       (stopped web_service))
       :effects ( (add started(web_service))
                  (delete stopped(web_service)))))
```

The codes above show a simple example encoded in Lisp style to illustrate how the plan action changes the current state. The current state list may contain more states; nevertheless, for the action *start_web_service* the explicitly stated pre-conditions are fulfilled by the current state of the system; therefore we preform the action, which has a positive effect that turns a web service into started state. Furthermore, the stopped state is deleted by executing this action, as explicitly defined in the action description.

If the there are no more actions to be performed in the plan then the planner checks to see whether all conditions expressed in the goal description are satisfied; if so, then the plan is evaluated as a valid solution. In non-trivial cases, previously discussed constraints regarding temporal and cost condition also need to be checked against the corresponding values during the evaluation.

If the planner is implemented in the interactive mode, the selected actions are directly executed on the system, in which case the changes of the system states reflect the actual implementation results; thus the planner needs to wait for the system feedbacks and must duly react to the new system state.

The function *Find()* is responsible for finding the next decomposition method to refine the current tasks. The function is designated to perform two core tasks: computing a set of valid variable bindings and searching for the next decomposition method. Since the decomposition methods and actions are encoded generically with parameters denoted in variables and their types, during the computation of a plan, these variables must be correctly bound to a set of instance values that are in alignment with those defined in the methods or actions. For example, if in the current state description,

[4]A plan action maintains an *add list* and a *delete list* to denote state changes.

a state of the server can be expressed as: `(CurrentState ... (running LinuxServ1 server))`, this means that in the current system state, an constant `LinuxServ1` of type `server` is in a running state. In the pre-conditions of an action or a decomposition method, there may be a condition expressed as `(running serv1 server)`. In this case, the planner should compute and bind the variable in the action `serv1` with the constant `LinuxServ1` of type server. To improve the performance of the planning process, it is also possible to apply the *least-commitment* technique, which delays the binding of variables after the primitive plan solution is produced [TEHN96]. With deferred variable binding, the CSP algorithm can be used to postpone the computation of variables until absolutely necessary [YC94].

By searching decomposition methods and performing refinement operations, different strategies could be applied; some candidates are, for example, iterative deepening search, breadth-first search, etc. The planner chooses the next decomposition method by evaluating its pre-conditions to check if the method is unifiable with current state descriptions. If iterative deepening searching is applied, the `find()` function first finds a decomposition method that refines the current abstract task into a abstract task network and performs the deepening search with the left-most node of the networks; it iteratively repeats its search until all results in the deepest level of the left-most node are executable actions. The operation forms a tree-like structure and halts if all leaves in the tree are non-abstract actions. In the application example for fault recovery planning of service in a Cloud computing infrastructure, we use a modified iterative deepening technique which only works on the most promising node to improve the performance. In the following section we implement the planner with its essential features for the application scenarios.

5.2 Applications

Having discussed the implementation aspects, we demonstrate in this section instantiations of the planning framework for different application purposes. Before we dive into the concrete examples, we first propose a generic provision and deployment process, which serves as a guideline for the analysis, development, testing and improvement of the application. To perform the tasks for different stages of the process, we define several participating roles with different specialised responsibilities to support these activities.

Depending on specific scenarios, the instantiations of the original framework are *adapted* to the corresponding applications. Therefore, the concrete implementations of components involved in the specific applications might deviate slightly from the generic design from case to case. Nevertheless, the guiding principles and working mechanisms of planning operations still reflect and are in accordance with our original design. For each example, we

implicitly map the instances of components in the implementations to the corresponding components in our proposed framework. As the first example, we discuss an instantiation of the application of management fault recovery of virtual services in a cloud computing infrastructure. The flexibility of the proposed planning approach design allows applications that cope with other management disciplines requiring planning. To demonstrate this property, we also apply our approach to change management with only minor modifications.

5.2.1 Suggested Deployment and Provisioning

We discuss in this section general rules and issues regarding the deployment and provisioning of the planning system in operational environments. These are guidelines that reflect deployment patterns of the framework. To apply the planning framework in an operational environment, we divide the deployment process into four phases: *analysis and provisioning*, *testing & calibrating*, *deployment of the planning system* and *maintenance & continued improvement*. Different deployment tasks involved in these phases need to be supported by different specialised roles. To this end, we define in the next section those expert roles.

Participating Roles

Before we go on to explain these phases, it is essential to define roles that are required to involve in this process. Roles represent persons who have certain special capabilities to perform a particular set of tasks. Hence we specify the following roles: *system developer*, *knowledge engineer* and *plan system user*.

System developers are responsible for the implementation and development tasks of the framework. Before they commence with their tasks, they make decisions on unifying the development tools to use throughout the phase, such as programming languages and integrated development environment (IDE) etc. Also they have to know the allocation and distribution of the components, given a set of resources and devices. For the internal system , they write codes for the planner algorithm and implement the planner's interfaces to allow information exchanges with other sub-systems. To facilitate the persistent storage of the planning information, they ought to choose a suitable database technology for the instantiation of planning knowledge base. They should decide on the internal representation of the plan description formats, so that the translation mechanism including lexer, parser and language generation rules can be correctly developed. For the system externals, the system developers should carefully analyse the system interfaces to the management system, to which the planning framework is attached. For this task, they should closely cooperate with the plan system

users, who are familiar with the operation environment, to extract and determine the system interfaces so that the planning system and management system can be properly coupled.

Knowledge engineers are experts who are familiar with knowledge elicitation and documentation of the management information. They are responsible for transforming such information into reusable units for the planner in a suitable way. Experts with this role need to work closely with system administrators and operators, or even managers of the target service to distill and to classify the recovery knowledge of the target service. They need to organise different levels of tasks according to their abstraction levels and draw clear distinctions between them. Finally together with operators they encode the knowledge into the specified format usable to the planner. The ultimate goals of their tasks is to create the initial knowledge base with well-encoded entries and continuously improve the quality of the knowledge.

Plan system users are end-users of the framework. They are operators who want to manage recovery processes of the target service with the planning system. In the preparation phase, they should work together with the system developers and the knowledge engineers to build up system components. Their participation in the development is crucial for the success of the final developed system. They help the system developers to identify important system interfaces and system calls from the perspective of the target services, so that the planner system can be correctly coupled with the management system. They also need to work with the knowledge engineers to extract and aggregate management knowledge in the specified format.

To construct a proper implementation of the recovery framework, experts with those three specialised roles need to cooperate with each other throughout the different stages of development.

Implementation Phases

We now look closely at the four phases of the deployment process of a planning system into an operational environment. For this discussion, we assume that the target services are already running and recovery experiences are already gained.

Analysis and Provisioning. The initial phase of the deployment of a planning system is to prepare the system for operation. In this phase, the system developers and the plan system users are suggested to determine the target services or systems that are needed to be put under the control of the planning system. Questions regarding the range of the services, their participating managed elements and their current corresponding management systems need to be firstly answered after a close investigation of the operational environment. Information sources that may be needed by the planner to collect required information need to be determined, for example, in order to get a snapshot of the states of the target service, monitoring data

need to be collected and aggregated; the sources (providers) of such data must be pinpointed at this stage. To collect the service information, the Service-MIB suggested by Sailer [Sai07] is a feasible approach to aggregate the critical information related to a particular service.

In addition, relevant function calls of the current management system need to be extracted, so that the interfaces of the planning system could be correctly implemented with the assistance of those calls. For example, for the execution of a plan solution, system calls that are related to a pre-existing workflow system may be required. They also determine the format and the information representation issues regarding the data processing procedure. Given a set of available resources, the system developers decide on the allocations of these available resources and distributions of the system component, for example, where the software package for the planner component should be installed and which server should host the knowledge base.

In this stage, the knowledge engineers need to cooperate with the experienced operators, perhaps also with the service manager, for the purpose of recovery knowledge elicitation. They should document the available information and classify it according to their abstraction levels. Hierarchical tasks networks are expected to encode different levels of recovery knowledge. Furthermore, tasks, decomposition methods and actions should be carefully annotated, for example, an operator may tell in which situations a particular recovery action could be performed to reach a certain set of goals; this is corresponding to the pre-conditions and post-conditions of that action. Furthermore, recovery constraints that are applicable to the target service also need to be extracted and documented in a data structure that is usable to the planner.

Testing & Calibrating. Before the planning system is put into practical use in the operational environment, it needs to be tested and, if necessary, debugged. In this phase, experts with all of the defined roles should work together to test, verify and calibrate the planning functionality. The tests can be performed by operating the planning system in a confined test environment. Simulated faults are injected as test cases against the current knowledge base in order to verify the correctness of the planning knowledge and solution. The computed plan solutions are then compared with standard solutions; if the solutions deviate from the correct ones, either the planner component or the knowledge base must be calibrated in order to rule out any error. On the other hand, since compiling knowledge is a complex process, it is difficult to prevent errors in encoding the recovery knowledge to sneak in; therefore, a careful debugging of the knowledge base is absolutely necessary.

Deployment. During the precedent phase, the system engineers draw the requirements and develop the components of the framework. At the same time, the recovery knowledge is elicited, formatted and documented by the

knowledge engineers and operators. In this phase, the system is consequently deployed in the operational environment. Deployment in this context means that the developed planning system is attached to the management system and ready to be functional, i.e. the system interfaces of the framework are implemented accordingly as specified, including system interfaces and user interfaces. The system developers should ensure that internally all components can communicate with each other without any problem and externally the planning framework should be able to exchange information with external systems. Additionally, all components are installed or initialised on the devices assigned to them and they are ready to be operational. For example, the planner component is encapsulated into a software packet installed on a computationally powerful server machine. A knowledge base could be hosted using a database, with which the recovery knowledge and information can be persistently stored, queried and updated. Since the planning knowledge is represented in a semi-structured data structure, a NoSQL-based approach seems to be more suitable for the storage implementation than a conventional relational database management system.

Maintenance & Continued Improvement As the underlying infrastructures and services may be modified and new service elements may be introduced into the current service landscape, planning knowledge needs to be updated correspondingly. Out-dated information has to be adapted to the new situations to guarantee the relevance of the current knowledge. Additionally, during actual operations, new management knowledge may be gained. To reflect such system dynamics, it is essential that the knowledge base has to be continually improved and maintained. This phase is to ensure that the planner produces high-quality solutions. To facilitate the operations involved in this phase, the framework design provides interfaces to update the planning information. The defined interfaces are access points to the internals of the planning framework, which means that the maintenance and improvement action can be done non-intrusively of the deployed planning system.

The deployment and operation of a planning system is admittedly a non-trivial process. Close cooperations between system developers, knowledge engineers and service operators are key to a successful deployment and operation of the system. The maintenance and continued improvement requires discipline to diligently document any modification of the service infrastructure in order to ensure the relevance of already-captured information. Nevertheless, we believe that once such jobs have been done correctly, the long-term benefits of computing and suggesting recovery solutions in efficient and effective manner will make all efforts invested pay off in the long-run.

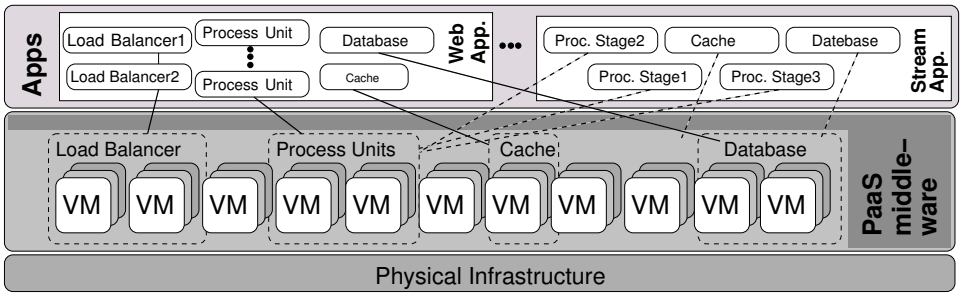


Figure 5.5: A Cloud Infrastructure Scenario

5.2.2 Recovery Planning in Cloud Computing

Figure 5.5 illustrates a typical setting of services being provisioned based on the PaaS (platform-as-a-service) model, as known from e.g., Google AppEngine, Heroku and the efforts toward data centre consolidation as well as service re-centralisation. The PaaS/virtualization middleware maps physical resources to virtual service components and offers flexible allocation and deployment of virtual resources (e.g., VMs) for the hosted services and provides software stacks for the hosted IT services, e.g., applications, libraries and DB systems. Virtual machines are configured and provisioned as disposable elements of the service platform. Figure 5.5 exemplifies two usage scenarios: the *Web App* providing dynamic web services based on service elements such as a load-balancer to normalise the network traffic during peak time, a database for dynamic content and process units to serve HTTP requests, supported by cache elements to accelerate identical queries. Similarly, the *Stream App* provides a pipelined/staged process service relying on large numbers of processing elements governed by a load balancer and database component.

Faults can occur on all layers of this setup, in a *horizontal view*. Services are sustained by multitudinous components. Servers, storage, database and network components form complex interdependencies, in which a single faulty component could degrade or even corrupt the whole service. In scenario, a defect physical server could bring down VMs and deteriorate the associated services.

In a *vertical view*, multiple layers participate in service delivery. The stack of components across all layers increase the potential for faults with different characteristics, depending on the impacted layer. For instance, the corruption of the virtualization/middleware layer will impact the services running on top of it, even if the physical server remains intact.

The rate of concurrent failure events in such a large-scale infrastructure overwhelms the operators due to high service density, operator-to-server ratio and growing customer expectations on reliability; thus, human-centric

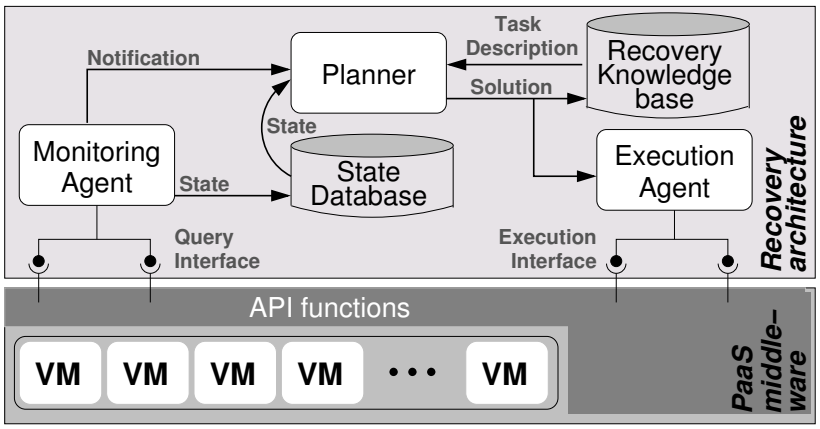


Figure 5.6: Design of a planning-based recovery architecture

fault management is rendered all but infeasible. What is worse, time-critical and highly stressful fault recovery leads to an increased rate of human error both in terms of decisions as well as during the execution of a management action. Hence, a solution for fault recovery must support a quick and effective recovery process.

The proposed solution revolves around the architecture sketched in Figure 5.6, which incorporates a *planner* as the core component, as well as the information sources necessary to the recovery planning process, as stipulated by the requirements.

The communication between the central planning system and the PaaS middleware is facilitated by monitoring and execution agents. The *monitoring agent* is responsible for collecting data and writing them into the *state database*, as well as triggering the planner if anomalies are detected. The *execution agent* implements selected management actions on the underlying infrastructure. The *knowledge database* contains two types of information: task descriptions (i.e. fault recovery recipes as either abstract or atomic actions) and system descriptions containing the dependencies between applications and virtual resources. Ideally, such information is obtained by discovery mechanisms; lacking that, it may be provided by administrators.

Upon receiving a notification, the planner computes a set of solutions based on information retrieved from the state database (the current system status) and the knowledge database (the corresponding task and system descriptions). The result is written back to the recovery knowledge base and saved as history data for documentation.

Once plans have been computed they may be examined by human administrators with respect to the feasibility and correctness. (To support such interaction, our prototype includes a web-based front-end.) Finally a selected plan is enacted by the execution agents on the API of the underlying

middleware. The execution process is constantly supervised by the monitoring agent (in terms of partial state changes), which trigger re-planning in cases of secondary faults or exceptions.

Figure 5.7 shows the adapted data models applied for the recovery planning operations in the cloud scenario. In this application, the planning knowledge is encoded in the JSON (JavaScript Object Notation, RFC 4627) [5] format for its simplicity, since we want to rapid-prototype the application.

The planning algorithm is accompanied by two assisting functions: EVALUATE and APPLY. The `EVALUATE` function analyses the current global state (provided by the `status` parameter) and classifies the states of individual virtual resources into one of the fault states: *hangup*, *idle*, *stopped* or *missing*. These potential states (simply represented by integer values in our implementation) are determined by symptoms apparent in monitoring data collected from the virtual infrastructure. More sophisticated evaluation heuristics or utility functions could be applied, see, e.g., [Fis70], to get more precise classification results.

The algorithm iteratively searches for viable recovery tasks for the involved virtual resource taking into account the pre-conditions provided as part of recovery knowledge. If a selected task is decomposable, it is then further refined according to the "hints" (in the form of a task network) present in the task description.

The `APPLY` function commits the selected atomic actions to the current state model and transits the model into a new state. Effects of atomic (executable) tasks that would change the current state of the observed virtual resource are reflected as state changes in the `statuslist`. This new state is passed to the `EVALUATE` function to achieve a new set of possible configurations. The configurations with the best estimation values are then further extended by the recursive call of the planning function (`dig`). The recursion ends when the estimated value is below the pre-defined threshold (i.e. there are no further candidates).

Note that at each level of expansion, only the recovery configuration with the best evaluation is processed further by the algorithm. The rest of the possible configurations will be pruned for the purpose of efficiency.

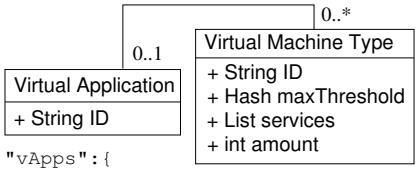
The evaluational experiments are based on simulations performed on data collected from an operational virtualized environment. The tests were performed on a Linux machine with a 2.2 GHz CPU and 3 GB memory. For each test in different test groups, around 10 tests are conducted. Figure 5.8 shows the evaluation results with standard deviations for each data point.

We test the scalability of our approach with respect to three dimensions: number of VMs, types of faults and length of plan.

In the first experiment, we scale from 15 to 300 VMs and inject 6 different types of faults into the test data. In the first test configuration in

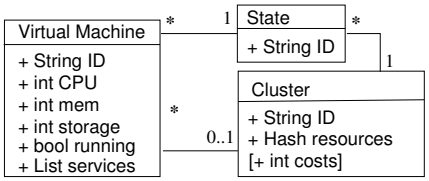
[5] `http://tools.ietf.org/html/rfc4627`

Service information denotes semantics and dependencies of the services running on a virtual infrastructure. A Virtual Application is associated with one or more VM–Types.



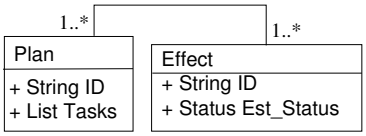

```
"vApps":{
   "WebApp": {
    "procUnit":{
      "amount": 3,
      "clustered": "deny",
      "MAX_threshold":{
      "cpu": 95,
      "mem": 90,
      "hdd": 90, ... },
      "services":["app", "nginx"] },
    "database":{...},
    "loadbalancer:{...}" }  }
```

State information encodes the current status of the underlying infrastructures. Essential information such as CPU and memory utilization as well as available storage space taken by a VM are included as part of VM state information.



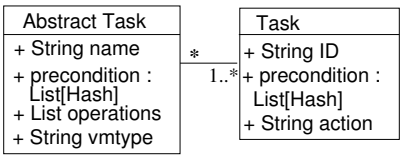

```
"ID":{
   "cluster": "1",
   "type": "ProcUnit",
   "vapp": "WebApp",
   "cpu": 34,
   "mem": 25,
   "hdd": 70,
   "running": True,
   "services":["nginx", "app"] }
```

Plans denote the solutions computed by the automated planning algorithm. A final solution plan consists of primitive tasks that are no longer decomposable. Each plan has a set of effects which are expected post–conditions after the execution of the solution.



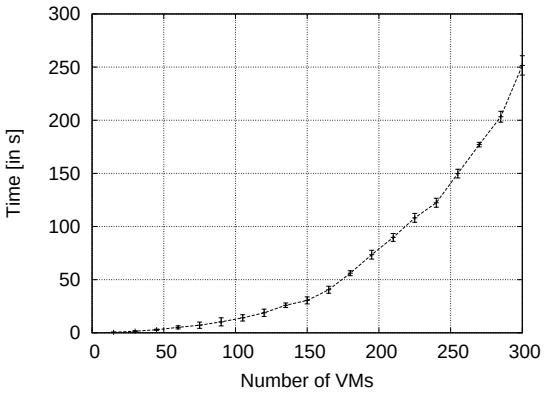

```
"PlanID": {
   "actions":[
   {TASK_1},
   {TASK_2},
    ...],
   "effect":[
   {NEW_STATUS} ] },
```

Management Tasks encode knowledge on recovery actions. Two types of tasks are presented in this model: Abstract Task denotes high level tasks that could be further refined into more specific actions; Task describes primitive action that is atomic and directly implementable.

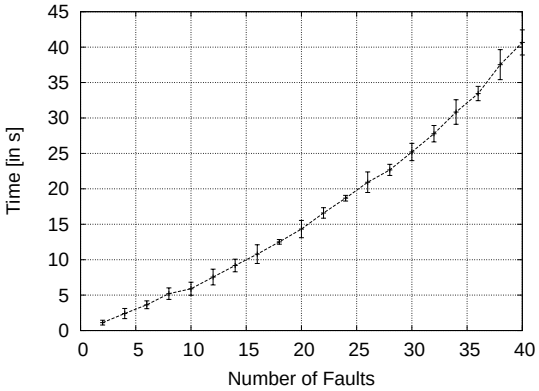



```
"TASK_ID": {
"preconditions": \
 [{"overload": true}],
"todo": \
["clone", "migrate"],
"type": 0,
"name": "CreateNewDB",
"vmtype": ["database"]
},
```

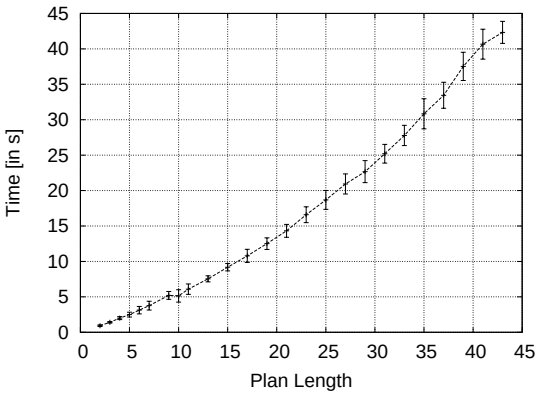
Figure 5.7: Data Models for Planning Operation used in the Recovery Applications



(a) *Variable number of VMs*

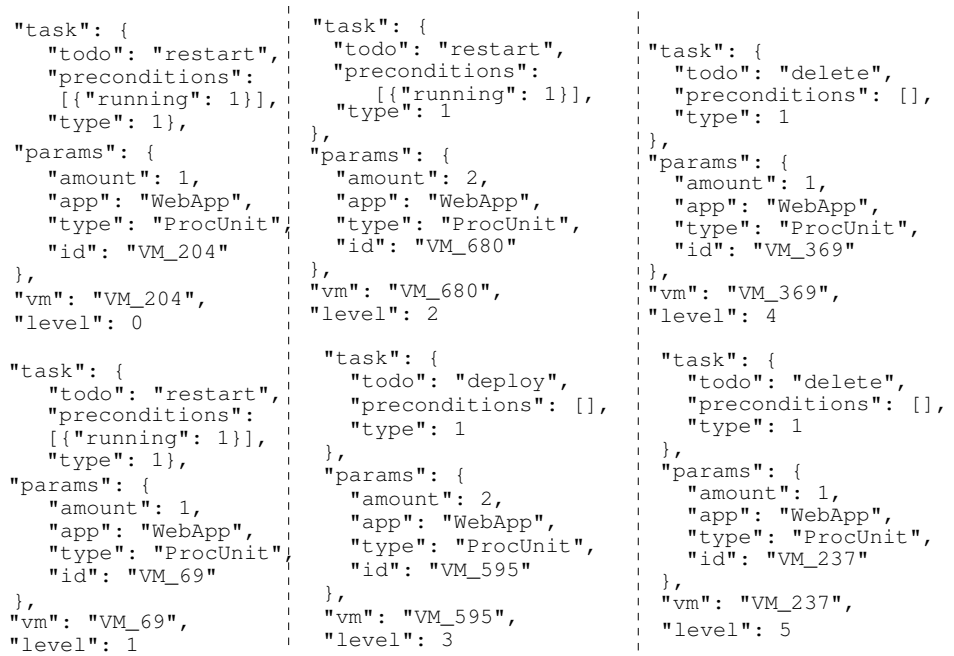


(b) *Variable number of faults*



(c) *Variable plan length*

Figure 5.8: Evaluation results of various test configurations



```
"task": {
   "todo": "restart",
   "preconditions":
    [{"running": 1}],
   "type": 1},
"params": {
   "amount": 1,
   "app": "WebApp",
   "type": "ProcUnit",
   "id": "VM_204"
},
"vm": "VM_204",
"level": 0

"task": {
   "todo": "restart",
   "preconditions":
   [{"running": 1}],
   "type": 1},
"params": {
   "amount": 1,
   "app": "WebApp",
   "type": "ProcUnit",
   "id": "VM_69"
},
"vm": "VM_69",
"level": 1
```

```
"task": {
  "todo": "restart",
  "preconditions":
     [{"running": 1}],
  "type": 1
},
"params": {
  "amount": 2,
  "app": "WebApp",
  "type": "ProcUnit",
  "id": "VM_680"
},
"vm": "VM_680",
"level": 2

 "task": {
   "todo": "deploy",
   "preconditions": [],
   "type": 1
 },
 "params": {
   "amount": 2,
   "app": "WebApp",
   "type": "ProcUnit",
   "id": "VM_595"
 },
 "vm": "VM_595",
 "level": 3
```

```
"task": {
  "todo": "delete",
  "preconditions": [],
  "type": 1
},
"params": {
  "amount": 1,
  "app": "WebApp",
  "type": "ProcUnit",
  "id": "VM_369"
},
"vm": "VM_369",
"level": 4

 "task": {
   "todo": "delete",
   "preconditions": [],
   "type": 1
 },
 "params": {
   "amount": 1,
   "app": "WebApp",
   "type": "ProcUnit",
   "id": "VM_237"
 },
 "vm": "VM_237",
 "level": 5
```


Figure 5.9: An example recovery plan generated by the algorithm. The generated example plan output is encoded in JSON format

Figure 5.8(a) the algorithm has a steep growth on the landmark of approximately 150 VMs in a service. This behaviour is caused by the fact that the algorithm initially expands an increasing number of search nodes as the number of machines grows. A possible remedy to this problem is using dynamic programming or caching techniques. We consider the runtime of approximately 4 minutes to find a fault recovery solution in a service involving about 300 VMs to be an acceptable timeframe in real operations.

Figure 5.9 shows an example output of one of the test cases, which involves *WebApp* running on a PaaS platform. The injected faults simulate cases in which three virtual Processing Units (PU) do not process the user requests but still are in running state and showing exceptionally high CPU and memory usages. Another two PUs have unknown and unretrievable states. These faults are injected into the state description data. The algorithm rates the first three PUs as corrupted and decides to select `restart` as recovery actions. For the latter two PUs, the algorithm decides to deploy two instance PUs of the same type to replace the missing ones and deletes the unresponsive instances. A complete description of the implementation of the proposed recovery framework is published in our paper [LDK10].

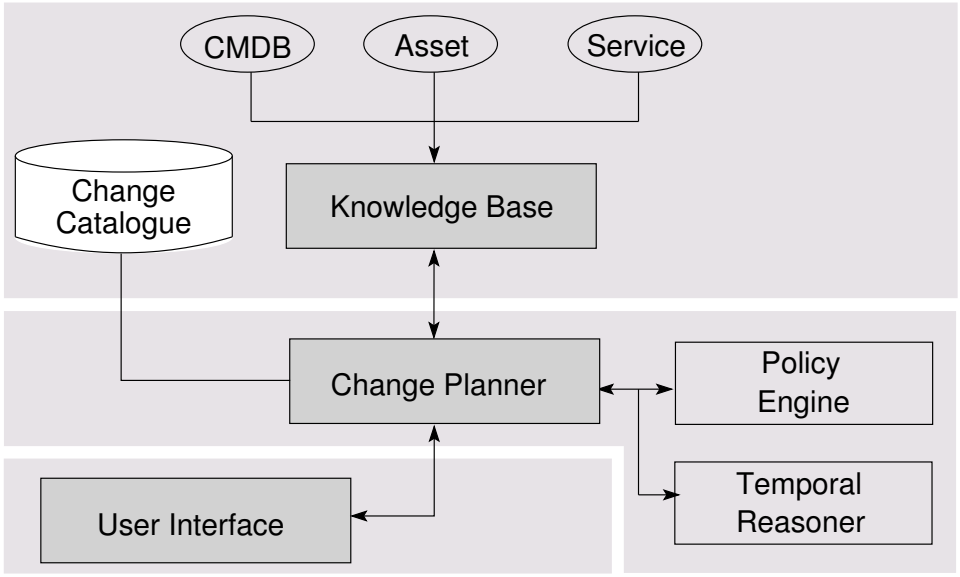


Figure 5.10: Architecture of Change Management Planner

5.2.3 Portability of the Framework

The proposed planning approach is generic in design so that with minor modifications, it could be applied to management tasks other than fault management. To justify such portability, we demonstrate with an example regarding change management, which uses the planning technique to assist the complex change management process.

Planning for Change Management

Change management is a management process which ensures that all requests for changes of IT services are carefully designed, evaluated, prioritised and scheduled, not only according to technical resources requirements, but also in alignment with applicable business rules. Without automated decision support and knowledge management mechanism, handling a large volume of change requests can cause a bottleneck and become a source of inefficiency in the management process. To tackle this challenge, we instantiate and adopt the planning approach to providing a machine-supported design of plans for change actions. Specifically our solution assists the process to refine high-level change requests into implementable change actions which can be directly executed.

Figure 5.10 illustrates the instantiation of the planning framework for the design of the change process. The architecture includes the following components:

- *Change Planner.* The change planner corresponds to the planning component of our framework. It applies HTN-based planning to refine high-level change requests into executable change plans. Recursive invocations of the planner take as parameters a reference to the current state of the knowledge base, (partial) tasks to be refined and the variable bindings from the previous invocations. Based on these input, the planner chooses a suitable decomposition method for the abstract change task that is still to be refined. Before the next invocation of the planner, the temporal reasoner and policy engine are invoked to check the consistency as well as the alignment of the partial plan to the applicable policies. In case of failure in the search, backtracking operations are initiated.

- *Change Catalogue.* It is part of the knowledge repository, which persistently stores the change templates and best-practice models. In HTN terms, the change catalogue corresponds to abstract tasks, decomposition methods and operative actions.

- *Temporal Reasoner.* Since real-world planning operations frequently involv parallel and durative actions, a Temporal reasoner is used as an enhancing approach to the fundamental HTN algorithm to deal with plan operations that require explicit considerations of time factors. The temporal reasoner maintains a Simple Temporal Problem (STP) data structure to enhance the planner with temporal reasoning capability for durative and parallel executable change actions.

- *Policy Engine.* Provides rules governing the change procedures. For every plan operation, the planner queries the policy engine to check if the intermediate plan conforms to the management policies. Any off-the-shelf policy engine such as Drools system can be used as an implementation of the policy engine component.

- *Knowledge base.* The knowledge base aggregates the information regarding the target infrastructure. It acts as an abstract interface between the planner and different types of IT operation databases, such as CMDB. The knowledge base provides the planner with a snapshot of the states of infrastructure at a given time point. Therefore, the knowledge base provides ways for the planner to query the database and assert information if necessary. In the implementation, the knowledge base component implements the CIM-based IT operation model. We use the Hibernate [6] object-relational persistence storage service as an object-oriented database for configuration items. Queries of the database are performed using the Hibernate Query Language (HQL).

[6] Hibernate Relational Persistence `www.hibernate.org`

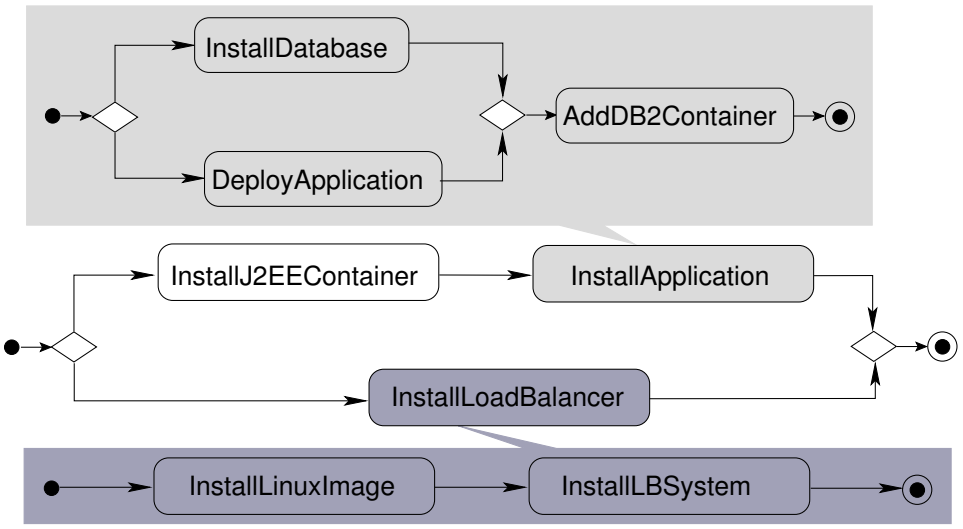


Figure 5.11: An Example on planning of installation of a J2EE application

- *User Interface.* The user interface is where the operator or user of the change plan system could interact with the planner to give instructions or scrutinise the solution plans.

The participating components and the layout of the architecture are derived from the instantiation of the framework we proposed. Components can be easily mapped to those in the original design. The data processing component is not mapped in this instantiation of the framework, since we experiment to pass state information and other planning-relevant knowledge with object-oriented descriptions.

With this architecture, several real-life examples of change are modelled, including installation, migration of services, specific maintenance operations, etc. As an example, we demonstrate a simple case of planning an installation of a J2EE application, which requires configurations of a runtime environment, installation of the applications and configuration of a load-balancer component. The knowledge of the plan refinements exists as change templates in the repository. Figure 5.11 illustrates the change template of the goal task *InstallLBSJ2EEApp* with two decompositions for the sub-task *InstallApplication* and *InstallLoadBalancer*.

To adapt our approach to planning for change activities, we introduce formalised sets of best change practices. We propose the notion of Change Catalogues for the implementation. It is a subset of predefined task networks similar to that of HTN planning. The catalogue is comprised of a set of high-level IT changes that are available to the requesters of a particular change activity. For example, a Request for Change (RFC) contains an instance of

a task network.

The change planning approach is capable of generating valid change plans for realistic enterprise examples in the order of seconds or minutes in a *fully automated mode* without any human intervention, depending on the scale of the input IT model. The experiments operate on a knowledge base with 100 to 1000 configuration items and change catalogues with 20 to 40 tasks with 3 to 10 methods each. The planner also supports a *mix-initiative* mode of operations, where the user can interact with the planner by determining the search space for the planner to explore. The user is then presented with a set of feasible decomposition methods or implementable change actions for the refined tasks. In this mode, viable plans are completed in a matter of seconds due the human assistance during the planning operation. Since by combining user decisions into operations of the automated planner, the search space is actively pruned by the user decisions, which offload the computational burdens of the planner. Since the multiple queries have to be made during the operations, the knowledge database becomes a bottleneck. This problem needs to be solved in our future research. More details on the adapted approach are described in our published work [TFL09].

With the application of the planning framework in the change management, we show the portability of our proposed approach to other management disciplines. This portability is ensured by the generic design of the framework. While the planning operation is an integral part of IT management in general, we are confident that, due to the growing scale, variety and complexity of IT service, our planning-based approach will find more applicable areas in the management to simplify the management tasks and relieve the service operators from the burdens of their daily tasks.

5.3 Evaluation

To show whether our proposed recovery planning framework and its components attain their designated goals, we discuss the fulfilment of the approach against the requirements derived in Chapter 2. For each of derived requirement, we provide a brief discussion to justify its degree of fulfilment.

5.3.1 Framework

F1 Genericity The framework design is not limited to specific scenarios or applications, but can be applied generically to management activities that require planning. As we showed in the application examples, not only can the planning framework to be applied to deal with fault recovery planning tasks, with minor adaptation, it could also be used to cope with other management activities such as planning for change management. The current planning algorithm and the layout of the components contribute to the generic design of the framework.

F2 Scalability The suggested planning framework scales well. The planning algorithm can handle a large mount of planning information and generate a usable plan in acceptable time intervals. As shown in the application example, it can handle as many as hundreds of VMs in a relatively large virtual infrastructure. In the change management example, we test the framework with more than one thousand configuration items.

F3 Usability The framework is design with usability in mind. To reduce the impact of the planning technique on end-users, who are not experts in compiling a plan domain, we developed a configuration language that could facilitate the user to encode the planning knowledge. The language is designed explicitly and simply enough for the users of the planning system to input basic planning knowledge as part of the knowledge base. For the complex translation processes between user input and planning knowledge, we propose a flexible compiling mechanism that takes care of the complex translation details. Furthermore, to assist the interaction between users and the planning system, the framework is equipped with an adaptive user interface components which is responsible for the presentation of solutions and interactions.

F4 Interoperability The framework provides the well-defined interfaces for interacting with external systems. The main task for the interface is to facilitate the information exchanges between external management system, to which the framework is attached, with the framework interns. The interface design is system-independent which allows flexible inter-operations between the planner and various external systems.

F5 Adaptability The design of the system interface allows the planning system to be easily adaptive to different underlying management systems. Regardless of the types of the attached management systems, the abstract operations defined in the system interface can be readily re-implemented without modifying any internal workings of the planning mechanism. The interface can be regarded as an adaptor which lies between systems and translates system calls for the systems on both sides of it.

5.3.2 Planning & Scheduling

P1 Planning Capability Planning capability is the core functionality provided by the planning framework. Sustained by the components revolving around the system, the planner components generates recovery plans according the current state of to the target system/service.

P1.1 Automated Planning The automated planning capability is offered by the H^2MAP algorithm, which is designed based on

the HTN-based paradigm. The algorithm could be operative in two modes: fully automatic mode or hybrid mode, depending on whether human assistance during the planning operation is given. An administrator could give the high-level abstract recovery tasks as initiation and the planning computes, according to the current situation and different constraints, viable implementable recovery actions.

P1.2 Generic Planner The algorithm is designed in such a way that it is not dependable on any recovery use case. The selection of a domain-configuration planning paradigm as guideline for the algorithm design augments the general applicability of the planning algorithm.

P1.3 Planning Efficiency As shown in both of our examples, the planning algorithm is efficient at dealing with large-scale recovery planning tasks. Depending on the operation mode, the plans could be found in magnitudes from seconds to minutes.

P1.4 Plan Optimisation We augment the fundamental planning algorithm with the capability to deal with temporal conditions and uncertainty information during the planning operations. The capability to process temporal information allows the planner to compose a plan that can be parallel executed instead of merely in a pure sequential manner. With considerations on the durative actions, the planning algorithm is extended to reason and optimise on time conditions.

P1.5 Plan Adaptability With the extension to the fundamental algorithm to treat uncertain environmental information, the result plan could be actively adapted to the changing state. By modelling the observed services as stochastic systems, we are capable of using approaches such as MDP or POMDP to compute the optimal plan solutions. Furthermore, we suggested other approaches based on the using a more aggressive planning heuristic or even isolating and freezing systems during the planning period to guarantee the correctness of the computed solution.

P1.6 Exception Handling If no plan is found, the planner would simply return to its user with no solution provided. The completeness property of the algorithm expresses that the algorithm will return no solution if none exists based on the currently possessed planning knowledge. In this case, the planner falls back on the human administrator for other recovery alternatives.

P1.7 Plan Granularity The final solution plan generated by the planner is a set of fine-grained, executable actions which can be directly implemented on the system.

P1.8 Different Levels of Recovery The proposed planner can handle the plan with different abstract levels. A high-level task is expressed as abstract actions in the planning knowledge base and it could be refined by refinement methods, which could be regarded as *recipes* to carry out the abstract task. The final plan is a set of implementable actions in a certain orders, which could be directly implemented on the systems.

P2 Scheduling Capability With the temporal reasoning ability, the planner can perform scheduling operations regarding the execution of planning actions. The planner is capable of arranging sequences of the execution according to their temporal requirements, so that the plan execution is optimised. The scheduling for the resources usage is currently not considered in our approach.

P3 Constraints Solving Capability Temporal-related constraints could be solved by the planner, which models the planning tasks as Simple Temporal Problems (STP). Thus, the planner applies constraint solving algorithm to arrange both durative and parallel actions in solution plans.

P4 Policy Integration This requirement is directly related to the capability of constraint-solving. If the policy is properly translated as constraints, the planner considers those constraints during its planning operations. As policy constraints can be expressed either numerically or in words, the planner currently considers the numerical terms, for example, temporal or monetary conditions. Constraints expressed in words is much more difficult to handle because of the semantic problems.

P5 Fault Coverage The planner could handle the recovery of diagnosable faults. The general capability of the planner is limited to the set of recovery knowledge that is currently available in the knowledge base. The planner is designed as an assistant management system for human administrators rather than being completely autonomic.

P6 Handling Multiple Information Types During its operation, the planner aggregates different types of information as part of its computation. As for infrastructure knowledge, the planner observes the current state of the system as the initial planning state; as for management procedural knowledge, the planner's operation is based on the recovery methods, abstract tasks and executable actions available in the knowledge base.

P7 Collaboration Capability The planning framework provides the planning component with a set of sub-systems that facilitate the collabo-

ration with external systems and the administrator. The external systems could collaborate with the planning system through well-defined interfaces, which are flexible regarding being adapted to different external management systems. Additionally, users can also interact with the planning system through different user interfaces; the planning system could be easily adapted to them.

5.3.3 Knowledge & Repository

K1 Identifying & Classify Recovery Knowledge We classify recovery knowledge into infrastructure knowledge and management procedural knowledge. Based on this classification, we specify the data models to describe identified knowledge so that it can be persistently stored and used by the planner. The knowledge base is a repository for the management procedural information, which includes the descriptions of abstract tasks and the decomposition methods. The infrastructure knowledge is mainly concerned with the information regarding the target system, e.g., a service that is impacted by faults, which includes information such as the current state of the target system and dependencies of the components that the system consists of. The planner requires such knowledge in order to make situation-aware plan decisions. Our planner aggregates all such knowledge into its computing process.

K2 Recovery Knowledge Repository The framework is designed with a recovery knowledge base as a persistent storage of the management procedural knowledge. It provides a well-defined interface for the planner to query the corresponding recovery knowledge. Additionally, it also provides the opportunity for the knowledge to be reused, since the knowledge base can be enriched with user input so that the planner could cope with more planning problems.

K3 Structured Recovery Knowledge The HTN-based approach allows the recovery planning knowledge to be represented in different abstraction levels. High-level recovery activities can be encoded as abstract tasks which require suitable decomposition methods to refine them into tasks that are less abstract. The implementable actions could be simply encoded as atomic actions in the knowledge repository. This capability is sustained by the proposed data models, which enable the different abstraction levels of knowledge to be encoded.

K4 Interfacing with Knowledge Discovery Tools In the current design, we provide well-defined system interfaces for the planning system to interact with external elements. For the planning knowledge elicitation, we currently consider administrators as the main source

of gaining recovery knowledge. For this purpose, an adaptive interface is provided to different user-side systems. Nevertheless, thanks to the flexible design of the interface, knowledge elicitation and discovery tools can be connected to the planning system through this interface without major efforts.

K5 Modelling of Recovery Knowledge To sustain the planning operation, we not only specify the types of knowledge required by the planner, but also provide data models for different knowledge types. The data models describe items that are included in the particular kind of knowledge and how such items could be represented. The data models provide a sharable, reusable and structured representation of the recovery knowledge.

K6 Encoding and Reusing Recovery Methods The recovery method encodes reusable decomposition recipes which denote how abstract recovery actions can be resolved into implementable recovery actions. According to the state of the target system and other constraints, the planner can reuse those methods while it computes a recovery plan.

5.3.4 Data Processing

D1 Translating relevant Knowledge into Planning Format To reduce the impact of applying an AI planning system on users, we design a simple and straightforward description language for the input of system information and recovery knowledge. Since the planner uses PDDL as input and output format as default, the relevant knowledge is translated by the data processing components between the planning component and external systems. The translation architecture allows flexible adaptations of the different source formats, e.g., user input, system events etc.

D2 Representing the Service State The data models we presented include a model which describes the system state for the planner. The input of system states can be provided by an external monitoring system. The translation component parses the input state information and generates a PDDL-based description which contains the state information. The translated information can then be used by the planner as its input.

D3 Representing Policy as Constraints Policies regarding temporal or any other numerical constraints can be represented in PDDL, which is a *de facto* standard knowledge description language used by most of the automated planning approaches. In PDDL, they can be simply represented as either soft constraints or planning preferences as discussed previously in Section 4.2.3 (on page 130). The planner could

be instructed to take those constraints into consideration (see requirements P3 and P4).

5.3.5 Plan Evaluation

E1 Performance Evaluation Augmented with the capability to reason on temporal constraints and other numerical constraints such as the monetary cost of recovery actions, the planner is able to evaluate the performance of the plan solution. Such evaluation processes can be either embedded during the planning or after the candidate solution plans have been generated. During the planning, the planner observes the given conditions and iteratively selects to expand the partial solution recovery plan if the given conditions, such as the deadline, and cost conditions have not been violated. Tasks with their associated actions, which violate the given constraints, will not be considered as a valid step. They cause the planner to backtrack in order to find alternatives. On the other hand, the planner can also be instructed to find all possible recovery plans during its operation, regardless of the given constraints. At the end of the planning phase, the planner could check the cumulative costs to filter out the plans that violate the given constraints.

E2 Cost Evaluation Cost evaluations are possible with the proposed planning approach. The cost could either be conducted, as discussed previously, during the runtime of the planner or after all possible alternative plans have been generated. Regarding the representation of the plans on the user systems, an operator also has the possibility to check and evaluate solution plans regarding their cost before they are handed over to the execution mechanism.

E3 Temporal Evaluation Fulfillment of requirements P2, P3 and P4 allows the planner to perform time-based plan evaluations. Parallel execution possibilities could be identified in order to shorten the recovery time during the plan execution phase.

5.3.6 Workflow & Execution

W1 Execution Engine An execution engine is intentionally not implemented in the framework, since a specific execution engine makes it difficult for the entire planning framework to be adapted to the different underlying management systems, which may have various requirements for the execution mechanism. Nevertheless, our planning framework provides an interface for the execution of the plan on the diverse execution platforms. The interface defines operations the planner can call, regardless of what execution engines are used in the practices.

The detailed information on the execution mechanism is hidden from the planner. To execute a solution plan, the planner just calls the well-known execution operation defined in the interface. The actual system-specific execution mechanism is wrapped in the implementation of the interface.

W2 Coordination of Execution With user interfaces, the planner could present a computed recovery plan solution to users. Users have the opportunity to scrutinise the suggested plan and decide whether it is ready for execution on the system based on their judgement. Additionally with user systems attached to the planner framework, the users could decide which plan actions could be executed fully automated and which may need human attendance for their implementation.

W3 System Compatibility The framework applies the information-hiding principle in its design. The design with abstract interfaces to the external systems makes it easy to adapt the generic framework to the specific underlying systems which may dynamically change from application to application.

W4 Control over Execution The planning framework delegates the actual controls of the plan executions to the execution mechanism. With any exceptions during the plan execution, the planner expects the underlying execution mechanism to register these events and notify the planner, so that the planner could react to the new situation accordingly. The current design does not allows the planner to directly control the progress of the plan execution; however, if the planning algorithm requires control operations, such as halt or stop the execution, they simply be added to the interface definitions. The interface component can wrap the specific control operations. The adaptive and extensible characteristic fulfilled by the currently design enables an easy extension of the planning framework without massive modifications to the design.

5.3.7 Overall Fulfilment of the Requirements

Table 5.1 summarises and provides an overview of the fulfillment of the requirements of the suggested recovery planning framework. For each derived requirement, we represent the degrees of fulfillment with the following symbols:

- ● the requirement is completely fulfilled.
- ◐ the requirement is partially fulfilled, but still needs improvement.
- ○ the requirement is not fulfilled at all.

As can be clearly seen from the results, our proposed planning algorithm and framework fulfill the vast majority of the requirements we derived. The framework design defines a generic, adaptive and at the same time usable architecture for a platform-independent planning system. Instantiations of the framework show the applicability of using automated planning not only as a viable approach for fault management, but also as a useful technique for other management activities that require planning.

As the core of the framework, the planning algorithm employs a generic planning approach augmented with scheduling and constraint processing capabilities as they are essential to real-world fault recovery operations. Nevertheless, to increase the efficiency of the planning algorithm, there is still room for theoretically and experimentally testing some heuristics to accelerate searches for plan alternatives. On the other hand, the current HTN-based algorithm considers those constraints that could be expressed in numerical form, research is still needed to increase the variates of constraints that can be incorporated in the planning process, for example, to the express the condition such as *the server must be behind a firewall*.

The planning is a knowledge-intensive management process. The planning algorithm relies on the supporting knowledge for performing its operations. Thus building a knowledge repository that could persistently store recovery-related information is essential to sustain the computation of plans. The knowledge-related parts in the design provide mechanisms which act not only as a central storage but also as mediator between the planner and users, through which users could actively enrich the knowledge based with acquired informations.

To facilitate communication and information exchanges between the framework and external systems, we include the data processing component to our design. The proposed translation mechanism converts the user input knowledge into a form that can be used as input for the planner; this approach reduces the issues of composing a plan description domain for the system users, who most possibly are foreign to the planning techniques. On the other hand, the translation mechanism can process the system events and state descriptions that are part of the input for the planner. Additionally, the translation mechanism is designed in a way to make it flexible enough to be adapted to other target and goal formats. Although a generic interface is provided, work still needs to be done to specify the operations that allows the repository to connect to the knowledge discovery tools other than through human input, for example, machine-learning approaches that could discover patterns from execution traces from operators could be regarded as a possible source of recovery knowledge.

The evaluation of a result plan is currently carried out implicitly during the planning phase. Considerations of the temporal conditions allow a partial plan already to be judged by the planning algorithm. The possibility to

integrate the mechanism to evaluate the cost of the recovery plan is given. Although not explicitly discussed, the cost conditions could be simply expressed as numerical terms in the planning knowledge; a simple summation operation at the end of the plan operation can add together the costs involved in the solution currently found by the algorithm, so that information regarding the total cost of the solution could be computed.

For the execution of the plan solution, we design an interface that is generic enough for the planner to call different underlying execution systems without changing the mechanisms inside the framework. The planner does not have to have explicit knowledge regarding how the solution plan is executed; it just calls the generic operations defined by the interface, regardless of the implementations of the interface. Further possibilities can be added by extending the interface with further operations supported by the underlying execution and management systems.

Compared to the approaches proposed by the related work listed in Chapter 3 (on page 82), our approach can obviously accomplish more in terms of planning capabilities, constraint processing and flexible architecture design. Most importantly, the framework design is generic enough to be adapted to other management systems, so that only minimum modifications are required when the underlying system, to which the planning framework is attached, is changed.

5.4 Summary

This chapter first discussed the principles on the prototypical implementations of the essential components in the planning framework. Specifically we discussed the instrumentation possibilities of the knowledge translation component and the planning component. For the knowledge translation mechanisms, we showed how to implement the lexical analyser, parser and language generator which are designed to translate the user input information into the planning knowledge. As examples, we engineered those crucial components with *antlr* tools, which is an open-source tool that is frequently used for crafting of computer language compilers. For the realisation of the planning algorithm, we discussed the technical possibilities to implement the two main functions that sustain the operations of the algorithm.

To justify the applicability of the framework, we instantiated the proposed framework for two application scenarios. In the first application, we applied the framework to assist service fault recovery in a Cloud computing infrastructure. If a service hosted by the Cloud infrastructure is impacted by a fault, the planner is expected to compute possible technical solutions as remedies to the fault state. The evaluation of different test configurations shows encouraging results and suggests that automated planning is, indeed, a viable technique to be used in IT management. The prototype

appears to scale well to a higher number of (potentially faulty) resources but is sensitive to the number of actual concurrent faults. We note that effective plan-based management depends not only on a planning algorithm but also on the availability and quality of management knowledge; effective monitoring and actuation components are a prerequisite.

In order to show the portability of the planning framework to other management disciplines, we adapt the framework to change management as the second application scenario. Based on the high-level change requests, the planner algorithm computes the viable change actions by observing management constraints that are applicable to the change process. Specifically, we instrument the framework to assist in the planning of the change actions. A working example based on the Java programming language is developed to demonstrate viability of adapting the framework to change management. The developed prototype was tested with real-life examples.

In the final part of this chapter, we evaluated the degree of fulfillment of the requirements derived at the beginning of this work. The results showed that our proposed approach fulfills the vast majority of requirements. For those requirements not directly satisfied by the current approach, the framework offers possibilities for extensions or improvement, thanks to the flexible and modular design of the framework architecture.

Requirements List			
Category	**ID**	**Requirement Details**	**Fulfilment**
Framework	F1	Genericity	●
	F2	Scalability	●
	F3	Usability	●
	F4	Interoperability	●
	F5	Adaptability	●
Planning & Scheduling	P1	Planning Capability	●
	- P1.1	Automated Plan Composing	●
	- P1.2	Generic Planner	●
	- P1.3	Efficiency on Planning	◐
	- P1.4	Plan Optimisation	●
	- P1.5	Plan Adaptability	●
	- P1.6	Exception Handling	◐
	- P1.7	Plan Granularity	●
	P2	Scheduling Capability	◐
	P3	Constraints Solving	●
	P4	Policy Integration	◐
	P5	Fault Coverage	●
	P6	Handle Multiple Information	●
	P7	Collaboration	●
Knowledge & Repository	K1	Classifying & Identifying Recovery Knowledge	●
	K2	Recovery Knowledge Repository	●
	K3	Structured Recovery Knowledge	●
	K4	Cooperating with Knowledge Components	◐
	K5	Recovery Knowledge Modelling	●
	K6	Encoding & Reusing Recovery Methods	●
Data Pre-processing	D1	Translating Planning Knowledge	●
	D2	State Representation	●
	D3	Policy Representation	●
Evaluation	E1	Performance Evaluation	●
	E2	Cost Evaluation	◐
	E3	Temporal Evaluation	●
Workflow & Execution	W1	Execution Engine	◐
	W2	Coordination of Execution	◐
	W3	Compatibility	●
	W4	Control over Executions	◐

Table 5.1: Fulfillment of Requirements

6 Conclusion and Future Work

Contents

This chapter concludes this research with a summary of the contributions made in this work. As an outlook, several research directions are presented for future investigation and development.

6.1 Conclusion

6.1.1 Putting All Pieces Together

Faults are a fact of life; the very same principle applies to IT systems as well. Despite advances in software and hardware technology, faults of different kinds still inevitably happen to all types of engineered systems, both simple and complex alike. Instead of treating them as exceptions in operations, they should be taken as facts in the life-cycle of a technical system.

Once it has been impacted by faults, the service needs to be recovered by operators as efficiently as possible. Nevertheless, given the scale, complexity and variety of today's service operation environments, a completely human-centric recovery process will overwhelm most operators. To address this problem, we propose an automated recovery planning framework to assist human operators composing viable recovery plans.

To achieve that goal, we employ a concept that originates from automated planning research, which is an integral part of artificial intelligence. The power of automated planning and its practical values were recognised

only recently. Many mission-critical applications that require planning are applying this technique to achieve self-managing capabilities.

The main working principle of a planner is to compute the sequence of actions that are expected to transform the current states into a set of desired states expressed as the plan goal. During operation, the advanced planning algorithms also need to consider different applicable restrictive conditions in order to compose more sophisticated solutions.

Based on the automated planning theory and the general planning model, we begin our discussion by justifying that fault recovery problems can be reduced to planning problems, hence they can be solved by techniques that solve planning problems. To conduct the proof, we first formalise the definition of fault recovery problems, which catches essential elements required for recovery. The proof is conducted based on the Turing reduction, in which the recovery problem model is mapped and transformed into the general planning model. After the relationship between models is established, the recovery problem is solved using a planning technique.

With the theoretical groundwork in place, we construct a planning algorithm to provide the core functionality. To design the algorithm, we need to choose a planning paradigm as our guideline. While there are a number of paradigms available, the classical planning paradigms, such as state-space based planning, are not suitable for solving complex real-world problems because of their formidable complexities for large problem instances. Therefore, we choose the Hierarchical Task Network (HTN) planning paradigm as the guide principle to construct our planner. Compared to the traditional approaches, the HTN-based planning paradigm employs sophisticated knowledge representations known as hierarchical task networks. Functionally, it concentrates on searching of the decomposition methods, which refine the high-level tasks into low-level executable actions, instead of querying the complete action spaces for solution. The HTN-based paradigm decomposes the planning problem into manageable sizes and solves them iteratively until fine-grained plan actions result, which is similar to how humans solve complex problems. The HTN planning paradigm allows encoding and reusing of plan knowledge in different hierarchical and abstract levels. On the other hand, it allows to elicit and codify human knowledge into a reusable and machine-readable format, so that such knowledge could be used to make computer-aided plan decisions.

We design and develop an HTN-based planner called H^2MAP, which not only computes plan solutions, but also considers different temporal constraints and uncertainty information. Such observations are crucial for real-world fault recovery scenarios. The operations of the planner are based on the recovery knowledge base, initial tasks, current states of the observed service as well as predefined goal states. A user of the planning system can determine the directions of the planning operation by providing the

H^2MAP planning with a set of initial abstract tasks. The planner iteratively searches for suitable decomposition methods from the knowledge base until a viable plan is found which transforms the initial states into goal states. With considerations on the temporal information, it is capable of optimising the action execution sequences, which not only consider sequential but also recognise parallel opportunities. This makes executions more efficient.

Planning is a knowledge-intensive operation. To support the planner, we model and specify types of relevant recovery information. We define a set of formal grammars for a recovery information definition language, which is straightforward for users to understand and to use. From the planner perspective, we choose PDDL as the input language, since it is a de facto language developed and maintained by the planning research community. Since the vast majority of planners are compatible with PDDL, selecting it has the advantage that if we change the current planner to other PDDL-compatible planners, nothing has to be changed in our framework.

In order to reduce the impact of planning technology on users, a translation mechanism is developed with intention to automatically translate the user input knowledge into the description language for the planner. Based on the user input, the translation mechanism first lexically analyses the input stream and extracts all tokens from the document. Then a parser analyses the results and tries to comprehend structures of the different input knowledge types. At the final stage, the target language is generated according to the set of translations rules. We provide a set of translation algorithms to direct the operations. The translation mechanism has the advantage that if the source or target language is changes, only minor modifications are required in order to adapt to the new language.

With the modelling and specifications of recovery language as well as the planning algorithm in place, we design a framework which dictates the architecture of the plan applications. The framework is divided into four functional sub-systems including the planning component, the data-processing component, the knowledge base and the system interfaces. The planning component offers core plan functionality sustained by the other system components that revolve around it. It is a host of the planning algorithms. To ensure the flexibility in the design, we apply the strategy patterns with generic access point in the form of the interfaces. This future-oriented flexible design approach enables the developer to integrate new compatible planners without any major modification to the framework itself.

Equally important to the framework is its knowledge processing component. This component is mainly responsible for information translations between the planner component and other sub-systems. The target and source of the translation mechanism are expressed in Domain-Specific Languages (DSL). As input and output may vary from application to application, in

order to keep our design generic, we employ the abstract factory pattern in our design. To invoke the translation operations, the user can simply rely on the access point of the concrete translation operations. All internal working details are hidden from the users. In this way, it is easy to change specific implementations of the component without influencing the user.

The knowledge repository is a component which serves to persistently store the planning knowledge and related information. It is a unified storage mechanism in the framework which facilitates other components to retrieve and to manipulate knowledge entries when necessary. The current design of the knowledge repository supports the primitive operations such as update, insert and query information.

The system interface defines a set of operations to facilitate data exchanges and communication between the framework and external systems. The external systems may be different from use case to use case; therefore, in order to ensure the flexibility of the framework as a whole, we use the adaptor class which decouples the internal planning mechanism from the external systems and makes the framework independent.

Putting all of the pieces together, we present a complete system framework for fault recovery planning. The involved components revolve around and sustain the planning algorithm. Besides providing its designated functionalities, flexibility is one of the paramount criteria of our design, considering the dynamic of the external systems. This consideration is behind the design idea of each single sub-system as well as the framework as a whole. To address this issue, we employ system design patterns to guarantee the maximal flexibility of the planning framework. We sketch and discuss the technical aspects of implementations of some of the components of the framework. Finally as proof of the concept, the framework is instantiated to two distinguished application scenarios. In the scenario of service fault recovery in a Cloud computing infrastructure, management actions are planned to recover the impacted service running in the infrastructure into a normalised state. To show the portability of our idea to other management disciplines, our further application example scenario concentrates on change management. The main objective is to use the framework instance to plan for the change requests. The requests are refined from the abstract form into concrete implementable change actions. Even if the planning for change management has different goals and tasks, through this application example, we show that our generic framework is well suited for change management as well.

Through the applications as proof of the concept, we show the practical value of our approach, not only as part of fault management, but also for other management disciplines that require planning as an integral part of their management activities. All examples show that planning is indeed a viable approach to be integrated as part of future management technology.

At the current stage, operators of IT services may not completely rely on the planner to make management decisions as fine-tuning of the solution may still be needed; nevertheless, we believe that with the automated approach providing them with sketches of solutions to their management problems, the time-to-solution will be dramatically reduced, especially for intricate large-scale management scenarios.

6.1.2 Research Questions Answered

At the beginning of this work we envisioned our solution to manage intricate fault recovery problems. Our vision was boiled down to several detailed research questions. Now with the solution in place, we retrospectively review the research questions that we have answered in this work. We map the contributions of this thesis to the questions previously raised (section 1.1.3).

Research question 1 (**Q1**) is mainly concerned with modelling and specification of various types of recovery information. In our solution, we specify the types of information that are required by an automated composition of the recovery plan. Then we design a language for simplified descriptions of such knowledge from users. For the planner, we select a planning description language that is expressive enough to encode rich types of information for the planning process. To bridge the gap between those two languages, we present a translation mechanism that is flexible enough so that even if the target and source languages are changed, only minimum modifications are required for the adaptation.

To answer research question 2 (**Q2**), we include the knowledge base in our framework design. Technically a knowledge base is a data storage component that has similar functionality to a conventional database. Only instead of being a full-fledged DBMS system, it is required to fulfill some basic functions which allow querying, inserting and updating of its data entries. The data entries are planning knowledge described in semi-structured data types. The knowledge base defines a set of operations which not only allow the planner to query during its operations, but also facilitate the user to enrich the content by inserting new planning information and updating existing data entries.

Question 3 (**Q3**) is concerned with automated composition of a plan. As one of the main research issues of this work, we apply the AI-based automated planning as a technical means to find recovery plans. More specifically, the algorithm design relies on the HTN-base planning paradigm as guideline. In addition to its sophisticated knowledge representation, the HTN-based planning paradigm has as the advantage of being capable of solving complex practical planning problems. Additionally the planner is augmented with extensions of temporal reasoning capability as well as the ability to deal with uncertainties during its operations.

To answer research question 4 (**Q4**), we discuss in section 6 the adaptive

approach of planning, which processes the real-time feedbacks from the execution of the plans and reacts to the events accordingly. Also uncertainty information models dynamics of the underlying target systems without external information.

Research question 5 (**Q5**) deals with the correctness of the plan. In the current approach, the correctness is ensured in two ways: first a plan is considered to be correct if its temporal constraints are satisfied. The algorithm could also be easily extended to handle the cost information in the planning process, when actions are modelled with, for example, cost information. These could be expressed as plan preferences as discussed previously. The other way to ensure correctness is to allow human inspections as solutions are found by the algorithm. To serve this purpose, we design in our framework an interface to present the solution to the users, so that they could have the chance to inspect the solution manually and make any modifications if necessary before it is executed.

The final question (**Q6**) concerns the integration of management knowledge into the planning process. In our approach, this is guaranteed by characteristics of the planning algorithm, which integrate the different types of recovery knowledge into its planning operations. The framework also provides a system interface to aggregates external information into the plan knowledge base as part of the framework. Furthermore, to solve the formats between the planner and external information sources, we design an automated translation process to convert between different formats. The flexible design of the translation process ensures that only minimal modifications are necessary if the requirements for the target and source formats are changed.

6.2 Future Work

The current work lays a solid foundation for further research to be built on. There are several directions, in which the current planning approach can be further extended and investigated.

- Fault diagnosis as planning. Whereas the current work focuses on using planning technology to find recovery actions, we assume the availability of fault diagnosis results at the time of planning. Fault diagnosis is nevertheless an equally important phase of the whole fault management process and should not be neglected. It seems to be a viable idea to integrate the diagnosis as part of fault recovery planning. One of the possible ways to achieve this goal is to apply *diagnostic problem solving* [Mci94] which collectively refers to approaches that consolidate diagnosis, test and repair tasks. Diagnostic problem solving involves reasoning about the evolution of the target dynamic system. Given observations of aberrant system behaviours, which are inconsistent to the expected, the diagnostic problem solving technique

generates a series of candidate diagnoses from inconsistencies between observations and specified system behaviours. At this stage, with a proper transformation between problem models, we could solve the diagnosis generation problem with a planning algorithm. Some pioneering theoretical work was done recently by Sohrabi et al. [SBM10], who tried to establish the relationship between the generation of fault diagnosis of a dynamic system and the automated planning problem.

- Investigate new types of planner for the fault recovery. Albeit its practical value for real-world applications, to find an optimal plan solution, the HTN-based planner still requires a significant amount of computation time. To tackle this issue, we could reuse past plan solutions to solve current planning problems, if they express a certain degree of similarity. This approach is called *Case-based Planning (CBP)* [CMAB05]. Case-based planning maintains a special memory to remember past solutions, so that they could be reused and adapted to solve new planning problems. With the CBP approach, not only the successful plan will be remembered, but also faulty plans will be stored. In this way, the planner learns how to reuse successful plans and how to avoid failures. The goal of using a case-base approach is clear: we want the planner to remember the plan work it has done before as experience. When a new problem appears, it can reuse its past experience to find a solution. Rather than building a plan from scratch, the CBP-based planner reuses and modifies its past plan.

- More interactions with plan users. The HTN-based H^2MAP planner provides a certain degree of interactions between the user and the planner. Nevertheless, more complicated interactions are still required to build a more practical planning system, especially when it comes to mission-critical planning tasks. The user needs to be presented with partial solutions even during the planning, so that more controls can be given to the users. When it comes to the decision points, the planner helps the user to understand trade-offs between different plan actions or decomposition methods, so that the user could make more rational and informed choices. For example, both decomposition methods A and B could lead to identical results, but A is less time-consuming but may involve more risk and method B is more stable but more costly. Under such a situation, human interventions are required to guide the automated planning process.

- Self-repairing and replanning. The presented framework provides interfaces to interact with the execution system. In a real-life system, fluctuations happen frequently, either caused by the underlying system dynamics or induced by the plan execution itself. To increase plan stability, as soon as exceptions during the plan execution occur, suitable

mechanisms are needed to repair or re-plan the original solution. There are two possible ways to handle such exceptions: self-repair the current plan or re-plan from scratch. Efficiency trade-offs and the practicality of these approaches need to be investigated in future research.

- Handle more sophisticated scheduling requirements. In the current work, we consider scheduling from a temporal perspective. In more realistic fault recovery cases, scheduling based on resource usages is also an important aspect that needs to be carefully considered. Involving reasoning of resource usages during the planning phase allows the planner to make the right choice by selecting correct alternatives among resources. From the planner's perspective, it means to incorporate more constraints regarding the amount of resources and their time of usage.

- Recovery planning in distributed, inter-domain environments. Recovery in an inter-domain environment involves different organisations, for example, in Grid computing, large amounts of resources are distributed among several inter-organisational domains. Planning in such inter-domain scenarios requires the planner to aggregate different sources of planning knowledge and state informations. Multiple domain descriptions and constraints need to be used as input to the planner in order to find a globally valid recovery solution. Additionally, a recovery process must also be coordinated among participating institutions which may have conflicting constraints in terms of recovery policies. How to efficiently resolve such conflicts is a challenging task for future research.

- Advanced knowledge organisation and representation. In the current research, we assume that planning knowledge such as actions, decomposition methods, constraints etc. are expressed in unified terms in the description. In a real-world operations, this may not be the case, since descriptions may be used in different literals with the same semantics (meanings) but in different syntactic forms (representations). In the worst case, this could seriously disrupt the normal operations of the planner. To bridge this gap, technologies such as the semantic web, e.g., RDF [MMM04] and OWL [MVH$^+$04], could be investigated and applied in future research to offer a unified representation form.

List of Figures

List of Algorithms

List of Tables

Appendices

A

Grammar of PDDL v3

Following text shows the grammar definitions of Plan Domain Description Languange (PDDL) v.3[1] :

```
grammar Pddl;
options {
    output=AST;
    backtrack=true;
}

tokens {
    DOMAIN;
    DOMAIN_NAME;
    REQUIREMENTS;
    TYPES;
    EITHER_TYPE;
    CONSTANTS;
    FUNCTIONS;
    PREDICATES;
    ACTION;
    DURATIVE_ACTION;
    PROBLEM;
    PROBLEM_NAME;
    PROBLEM_DOMAIN;
    OBJECTS;
    INIT;
    FUNC_HEAD;
    PRECONDITION;
```

[1]Reference: ANTLR v3 Grammar List `http://www.antlr.org/grammar/list`

```
    EFFECT;
    AND_GD;
    OR_GD;
NOT_GD;
IMPLY_GD;
EXISTS_GD;
FORALL_GD;
COMPARISON_GD;
AND_EFFECT;
FORALL_EFFECT;
WHEN_EFFECT;
ASSIGN_EFFECT;
NOT_EFFECT;
PRED_HEAD;
GOAL;
BINARY_OP;
UNARY_MINUS;
INIT_EQ;
INIT_AT;
NOT_PRED_INIT;
PRED_INST;
PROBLEM_CONSTRAINT;
PROBLEM_METRIC;
}

/************* Start of grammar ******************/

pddlDoc : domain | problem;

/************* DOMAINS ***************************/

domain
    : '(' 'define' domainName
      requireDef?
      typesDef?
      constantsDef?
      predicatesDef?
      functionsDef?
      constraints?
      structureDef*
      ')'
      -> ^(DOMAIN domainName requireDef? typesDef?
                constantsDef? predicatesDef? functionsDef?
```

```
                constraints? structureDef*)
    ;

domainName
    : '(' 'domain' NAME ')'
     -> ^(DOMAIN_NAME NAME)
    ;

requireDef
: '(' ':requirements' REQUIRE_KEY+ ')'
-> ^(REQUIREMENTS REQUIRE_KEY+)
;

typesDef
: '(' ':types' typedNameList ')'
  -> ^(TYPES typedNameList)
;

typedNameList
    : (NAME* | singleTypeNameList+ NAME*)
    ;

singleTypeNameList
    : (NAME+ '-' t=type)
  -> ^(NAME $t)+
;

type
: ( '(' 'either' primType+ ')' )
  -> ^(EITHER_TYPE primType+)
| primType
;

primType : NAME ;

functionsDef
: '(' ':functions' functionList ')'
-> ^(FUNCTIONS functionList)
;

functionList
: (atomicFunctionSkeleton+ ('-' functionType)? )*
;
```

```
atomicFunctionSkeleton
: '('! functionSymbol^ typedVariableList ')'!
;

functionSymbol : NAME ;

functionType : 'number' ;

constantsDef
: '(' ':constants' typedNameList ')'
-> ^(CONSTANTS typedNameList)
;

predicatesDef
: '(' ':predicates' atomicFormulaSkeleton+ ')'
-> ^(PREDICATES atomicFormulaSkeleton+)
;

atomicFormulaSkeleton
: '('! predicate^ typedVariableList ')'!
;

predicate : NAME ;

typedVariableList
    : (VARIABLE* | singleTypeVarList+ VARIABLE*)
    ;

singleTypeVarList
    : (VARIABLE+ '-' t=type)
      -> ^(VARIABLE $t)+
    ;

constraints
: '('! ':constraints'^ conGD ')'!
;

structureDef
: actionDef
| durativeActionDef
| derivedDef
;
```

```
/************* ACTIONS ***************************/

actionDef
: '(' ':action' actionSymbol
      ':parameters' '(' typedVariableList ')'
          actionDefBody ')'
       -> ^(ACTION actionSymbol typedVariableList actionDefBody)
    ;

actionSymbol : NAME ;


actionDefBody
: ( ':precondition' (('(' ')') | goalDesc))?
  ( ':effect' (('(' ')') | effect))?
  -> ^(PRECONDITION goalDesc?) ^(EFFECT effect?)
;

goalDesc
: atomicTermFormula
| '(' 'and' goalDesc* ')'
          -> ^(AND_GD goalDesc*)
| '(' 'or' goalDesc* ')'
          -> ^(OR_GD goalDesc*)
| '(' 'not' goalDesc ')'
          -> ^(NOT_GD goalDesc)
| '(' 'imply' goalDesc goalDesc ')'
          -> ^(IMPLY_GD goalDesc goalDesc)
| '(' 'exists' '(' typedVariableList ')' goalDesc ')'
          -> ^(EXISTS_GD typedVariableList goalDesc)
| '(' 'forall' '(' typedVariableList ')' goalDesc ')'
          -> ^(FORALL_GD typedVariableList goalDesc)
    | fComp
              -> ^(COMPARISON_GD fComp)
    ;

fComp
: '('! binaryComp fExp fExp ')'!
;

atomicTermFormula
```

```
: '(' predicate term* ')' -> ^(PRED_HEAD predicate term*)
;

term : NAME | VARIABLE ;

/************* DURATIVE ACTIONS ****************************/

durativeActionDef
: '(' ':durative-action' actionSymbol
      ':parameters' '(' typedVariableList ')'
          daDefBody ')'
       -> ^(DURATIVE_ACTION actionSymbol typedVariableList daDefBody)
    ;

daDefBody
: ':duration' durationConstraint
| ':condition' (('(' ')') | daGD)
    | ':effect' (('(' ')') | daEffect)
    ;

daGD
: prefTimedGD
| '(' 'and' daGD* ')'
| '(' 'forall' '(' typedVariableList ')' daGD ')'
;

prefTimedGD
: timedGD
| '(' 'preference' NAME? timedGD ')'
;

timedGD
: '(' 'at' timeSpecifier goalDesc ')'
| '(' 'over' interval goalDesc ')'
;

timeSpecifier : 'start' | 'end' ;
interval : 'all' ;

/************* DERIVED DEFINITIONS ****************************/

derivedDef
: '(' ':derived'^ typedVariableList goalDesc ')'!
```

```
;

/************* EXPRESSIONS ****************************/

fExp
: NUMBER
| '(' binaryOp fExp fExp2 ')' -> ^(BINARY_OP binaryOp fExp fExp2)
| '(' '-' fExp ')' -> ^(UNARY_MINUS fExp)
| fHead
;

fExp2 : fExp ;

fHead
: '(' functionSymbol term* ')' -> ^(FUNC_HEAD functionSymbol term*)
| functionSymbol -> ^(FUNC_HEAD functionSymbol)
;

effect
: '(' 'and' cEffect* ')' -> ^(AND_EFFECT cEffect*)
| cEffect
;

cEffect
: '(' 'forall' '(' typedVariableList ')' effect ')'
  -> ^(FORALL_EFFECT typedVariableList effect)
| '(' 'when' goalDesc condEffect ')'
  -> ^(WHEN_EFFECT goalDesc condEffect)
| pEffect
;

pEffect
: '(' assignOp fHead fExp ')'
  -> ^(ASSIGN_EFFECT assignOp fHead fExp)
| '(' 'not' atomicTermFormula ')'
  -> ^(NOT_EFFECT atomicTermFormula)
| atomicTermFormula
;


condEffect
: '(' 'and' pEffect* ')' -> ^(AND_EFFECT pEffect*)
| pEffect
```

```
;

binaryOp : '*' | '+' | '-' | '/' ;

binaryComp : '>' | '<' | '=' | '>=' | '<=' ;

assignOp : 'assign' | 'scale-up' | 'scale-down' | 'increase' | 'decrease' ;


/************* DURATIONS  ***************************/

durationConstraint
: '(' 'and' simpleDurationConstraint+ ')'
| '(' ')'
| simpleDurationConstraint
;

simpleDurationConstraint
: '(' durOp '?duration' durValue ')'
| '(' 'at' timeSpecifier simpleDurationConstraint ')'
;

durOp : '<=' | '>=' | '=' ;

durValue : NUMBER | fExp ;

daEffect
: '(' 'and' daEffect* ')'
| timedEffect
| '(' 'forall' '(' typedVariableList ')' daEffect ')'
| '(' 'when' daGD timedEffect ')'
| '(' assignOp fHead fExpDA ')'
;

timedEffect
: '(' 'at' timeSpecifier daEffect ')'
    | '(' 'at' timeSpecifier fAssignDA ')'
| '(' assignOp fHead fExp ')'
;

fAssignDA
: '(' assignOp fHead fExpDA ')'
;
```

```
fExpDA
: '(' ((binaryOp fExpDA fExpDA) | ('-' fExpDA)) ')'
| '?duration'
| fExp
;

/************* PROBLEMS ****************************/

problem
: '(' 'define' problemDecl
  problemDomain
      requireDef?
      objectDecl?
      init
      goal
      probConstraints?
      metricSpec?
      ')'
      -> ^(PROBLEM problemDecl problemDomain requireDef? objectDecl?
       init goal probConstraints? metricSpec?)
    ;

problemDecl
    : '(' 'problem' NAME ')'
    -> ^(PROBLEM_NAME NAME)
    ;

problemDomain
: '(' ':domain' NAME ')'
-> ^(PROBLEM_DOMAIN NAME)
;

objectDecl
: '(' ':objects' typedNameList ')'
-> ^(OBJECTS typedNameList)
;

init
: '(' ':init' initEl* ')'
-> ^(INIT initEl*)
;
```

```
initEl
: nameLiteral
| '(' '=' fHead NUMBER ')'          -> ^(INIT_EQ fHead NUMBER)
| '(' 'at' NUMBER nameLiteral ')'   -> ^(INIT_AT NUMBER nameLiteral)
;

nameLiteral
: atomicNameFormula
| '(' 'not' atomicNameFormula ')' -> ^(NOT_PRED_INIT atomicNameFormula)
;

atomicNameFormula
: '(' predicate NAME* ')' -> ^(PRED_INST predicate NAME*)
;

goal : '(' ':goal' goalDesc ')' -> ^(GOAL goalDesc) ;

probConstraints
: '(' ':constraints'  prefConGD ')'
  -> ^(PROBLEM_CONSTRAINT prefConGD)
;

prefConGD
: '(' 'and' prefConGD* ')'
| '(' 'forall' '(' typedVariableList ')' prefConGD ')'
| '(' 'preference' NAME? conGD ')'
| conGD
;

metricSpec
: '(' ':metric' optimization metricFExp ')'
  -> ^(PROBLEM_METRIC optimization metricFExp)
;

optimization : 'minimize' | 'maximize' ;

metricFExp
: '(' binaryOp metricFExp metricFExp ')'
| '(' ('*'|'/') metricFExp metricFExp+ ')'
| '(' '-' metricFExp ')'
| NUMBER
| '(' functionSymbol NAME* ')'
| functionSymbol
```

```
    | 'total-time'
| '(' 'is-violated' NAME ')'
;

/************* CONSTRAINTS ****************************/

conGD
: '(' 'and' conGD* ')'
| '(' 'forall' '(' typedVariableList ')' conGD ')'
| '(' 'at' 'end' goalDesc ')'
    | '(' 'always' goalDesc ')'
| '(' 'sometime' goalDesc ')'
  | '(' 'within' NUMBER goalDesc ')'
| '(' 'at-most-once' goalDesc ')'
| '(' 'sometime-after' goalDesc goalDesc ')'
| '(' 'sometime-before' goalDesc goalDesc ')'
| '(' 'always-within' NUMBER goalDesc goalDesc ')'
| '(' 'hold-during' NUMBER NUMBER goalDesc ')'
| '(' 'hold-after' NUMBER goalDesc ')'
;



/************* LEXER ****************************/


REQUIRE_KEY
    : ':strips'
    | ':typing'
    | ':negative-preconditions'
    | ':disjunctive-preconditions'
    | ':equality'
    | ':existential-preconditions'
    | ':universal-preconditions'
    | ':quantified-preconditions'
    | ':conditional-effects'
    | ':fluents'
    | ':adl'
    | ':durative-actions'
    | ':derived-predicates'
    | ':timed-initial-literals'
    | ':preferences'
    | ':constraints'
```

```
    ;


NAME:    LETTER ANY_CHAR* ;

fragment LETTER: 'a'..'z' | 'A'..'Z';

fragment ANY_CHAR: LETTER | '0'..'9' | '-' | '_';

VARIABLE : '?' LETTER ANY_CHAR* ;

NUMBER : DIGIT+ ('.' DIGIT+)? ;

fragment DIGIT: '0'..'9';

LINE_COMMENT
    : ';' ~('\n'|'\r')* '\r'? '\n' { $channel = HIDDEN; }
    ;

WHITESPACE
    :   (   ' '
        |   '\t'
        |   '\r'
        |   '\n'
        )+
        { $channel = HIDDEN; }
    ;
```

Implementation of the Language Translation Components

B.1 Lexer Code

Following is the code list for the Lexer implementation.

```
1
2  import org.antlr.runtime.*;
3  import java.util.Stack;
4  import java.util.List;
5  import java.util.ArrayList;
6
7  public class PKDLLexer extends Lexer {
8    public static final int EOF=-1;
9    public static final int T__9=9;
10   public static final int T__10=10;
11   public static final int T__11=11;
12   public static final int T__12=12;
13   public static final int T__13=13;
14   public static final int T__14=14;
15   public static final int T__15=15;
16   public static final int T__16=16;
17   public static final int T__17=17;
18   public static final int T__18=18;
19   public static final int T__19=19;
20   public static final int T__20=20;
21   public static final int T__21=21;
22   public static final int T__22=22;
23   public static final int T__23=23;
24   public static final int T__24=24;
25   public static final int T__25=25;
26   public static final int T__26=26;
27   public static final int T__27=27;
28   public static final int T__28=28;
29   public static final int T__29=29;
30   public static final int T__30=30;
31   public static final int T__31=31;
32   public static final int T__32=32;
33   public static final int NEWLINE=4;
34   public static final int IDENT=5;
35   public static final int WS=6;
```

```
36  public static final int STRING=7;
37  public static final int INTEGER=8;
38
39  public PKDLLexer() {;}
40  public PKDLLexer(CharStream input) {
41      this(input, new RecognizerSharedState());
42  }
43  public PKDLLexer(CharStream input, RecognizerSharedState state) {
44      super(input,state);
45
46  }
47  public String getGrammarFileName() {
48      return "/users/wiss/liufeng/work/diss/mydiss/tools/antlr/PKDL.g";
49      }
50
51  public final void mT__9() throws RecognitionException {
52      try {
53          int _type = T__9;
54          int _channel = DEFAULT_TOKEN_CHANNEL;
55          {
56          match("BEGIN");
57          }
58          state.type = _type;
59          state.channel = _channel;
60      }
61      finally {
62      }
63  }
64
65  public final void mT__10() throws RecognitionException {
66      try {
67          int _type = T__10;
68          int _channel = DEFAULT_TOKEN_CHANNEL;
69          {
70          match("END");
71          }
72          state.type = _type;
73          state.channel = _channel;
74      }
75      finally {
76      }
77  }
78  public final void mT__11() throws RecognitionException {
79      try {
80          int _type = T__11;
81          int _channel = DEFAULT_TOKEN_CHANNEL;
82          {
83          match('{');
84          }
85
86          state.type = _type;
87          state.channel = _channel;
88      }
89      finally {
90      }
91  }
92
93  public final void mT__12() throws RecognitionException {
94      try {
95          int _type = T__12;
96          int _channel = DEFAULT_TOKEN_CHANNEL;
97          {
```

```
98              match('}');
99              }
100
101             state.type = _type;
102             state.channel = _channel;
103         }
104         finally {
105         }
106     }
107
108     public final void mT__13() throws RecognitionException {
109         try {
110             int _type = T__13;
111             int _channel = DEFAULT_TOKEN_CHANNEL;
112             {
113             match(';');
114             }
115
116             state.type = _type;
117             state.channel = _channel;
118         }
119         finally {
120         }
121     }
122
123     public final void mT__14() throws RecognitionException {
124         try {
125             int _type = T__14;
126             int _channel = DEFAULT_TOKEN_CHANNEL;
127             {
128             match(':');
129             }
130
131             state.type = _type;
132             state.channel = _channel;
133         }
134         finally {
135         }
136     }
137
138     public final void mT__15() throws RecognitionException {
139         try {
140             int _type = T__15;
141             int _channel = DEFAULT_TOKEN_CHANNEL;
142             {
143             match("ACTION");
144             }
145
146             state.type = _type;
147             state.channel = _channel;
148         }
149         finally {
150         }
151     }
152
153     public final void mT__16() throws RecognitionException {
154         try {
155             int _type = T__16;
156             int _channel = DEFAULT_TOKEN_CHANNEL;
157             {
158             match("DECOM_METHOD");
159             }
```

```
160
161             state.type = _type;
162             state.channel = _channel;
163         }
164         finally {
165         }
166     }
167
168 public final void mT__17() throws RecognitionException {
169     try {
170         int _type = T__17;
171         int _channel = DEFAULT_TOKEN_CHANNEL;
172         {
173         match("GOAL");
174         }
175
176         state.type = _type;
177         state.channel = _channel;
178     }
179     finally {
180     }
181 }
182
183 public final void mT__18() throws RecognitionException {
184     try {
185         int _type = T__18;
186         int _channel = DEFAULT_TOKEN_CHANNEL;
187         {
188         match("INIT_STATE");
189         }
190         state.type = _type;
191         state.channel = _channel;
192     }
193     finally {
194     }
195 }
196
197 public final void mT__19() throws RecognitionException {
198     try {
199         int _type = T__19;
200         int _channel = DEFAULT_TOKEN_CHANNEL;
201         {
202         match("PLAN");
203         }
204
205         state.type = _type;
206         state.channel = _channel;
207     }
208     finally {
209     }
210 }
211
212 public final void mT__20() throws RecognitionException {
213     try {
214         int _type = T__20;
215         int _channel = DEFAULT_TOKEN_CHANNEL;
216         {
217         match("ID");
218         }
219
220         state.type = _type;
221         state.channel = _channel;
```

```
222         }
223         finally {
224         }
225     }
226
227     public final void mT__21() throws RecognitionException {
228         try {
229             int _type = T__21;
230             int _channel = DEFAULT_TOKEN_CHANNEL;
231             {
232             match("DOMAIN");
233             }
234
235             state.type = _type;
236             state.channel = _channel;
237         }
238         finally {
239         }
240     }
241
242     public final void mT__22() throws RecognitionException {
243         try {
244             int _type = T__22;
245             int _channel = DEFAULT_TOKEN_CHANNEL;
246             {
247             match("PARAM");
248             }
249
250             state.type = _type;
251             state.channel = _channel;
252         }
253         finally {
254         }
255     }
256
257     public final void mT__23() throws RecognitionException {
258         try {
259             int _type = T__23;
260             int _channel = DEFAULT_TOKEN_CHANNEL;
261             {
262             match("DESCR");
263             }
264
265             state.type = _type;
266             state.channel = _channel;
267         }
268         finally {
269         }
270     }
271
272     public final void mT__24() throws RecognitionException {
273         try {
274             int _type = T__24;
275             int _channel = DEFAULT_TOKEN_CHANNEL;
276             {
277             match("SUB_TASKS");
278             }
279
280             state.type = _type;
281             state.channel = _channel;
282         }
283         finally {
```

```
284         }
285     }
286
287     public final void mT__25() throws RecognitionException {
288         try {
289             int _type = T__25;
290             int _channel = DEFAULT_TOKEN_CHANNEL;
291             {
292             match("CURRENT_STATE");
293             }
294
295             state.type = _type;
296             state.channel = _channel;
297         }
298         finally {
299         }
300     }
301
302     public final void mT__26() throws RecognitionException {
303         try {
304             int _type = T__26;
305             int _channel = DEFAULT_TOKEN_CHANNEL;
306             {
307             match("ASSERTED_STATE");
308             }
309
310             state.type = _type;
311             state.channel = _channel;
312         }
313         finally {
314         }
315     }
316
317     public final void mT__27() throws RecognitionException {
318         try {
319             int _type = T__27;
320             int _channel = DEFAULT_TOKEN_CHANNEL;
321             {
322             match("TIME_TOTAL");
323             }
324
325             state.type = _type;
326             state.channel = _channel;
327         }
328         finally {
329         }
330     }
331
332     public final void mT__28() throws RecognitionException {
333         try {
334             int _type = T__28;
335             int _channel = DEFAULT_TOKEN_CHANNEL;
336             {
337             match("COST_TOTAL");
338             }
339
340             state.type = _type;
341             state.channel = _channel;
342         }
343         finally {
344         }
345     }
```

```
346
347   public final void mT__29() throws RecognitionException {
348       try {
349           int _type = T__29;
350           int _channel = DEFAULT_TOKEN_CHANNEL;
351           {
352           match("ORDERING");
353           }
354
355           state.type = _type;
356           state.channel = _channel;
357       }
358       finally {
359       }
360   }
361
362   public final void mT__30() throws RecognitionException {
363       try {
364           int _type = T__30;
365           int _channel = DEFAULT_TOKEN_CHANNEL;
366           {
367           match("SCHEDULED");
368           }
369
370           state.type = _type;
371           state.channel = _channel;
372       }
373       finally {
374       }
375   }
376
377   public final void mT__31() throws RecognitionException {
378       try {
379           int _type = T__31;
380           int _channel = DEFAULT_TOKEN_CHANNEL;
381           {
382           match("TOT_TIME");
383           }
384
385           state.type = _type;
386           state.channel = _channel;
387       }
388       finally {
389       }
390   }
391
392   public final void mT__32() throws RecognitionException {
393       try {
394           int _type = T__32;
395           int _channel = DEFAULT_TOKEN_CHANNEL;
396           {
397           match("OBJECTIVE");
398
399
400           }
401
402           state.type = _type;
403           state.channel = _channel;
404       }
405       finally {
406       }
407   }
```

```
408
409   public final void mWS() throws RecognitionException {
410       try {
411           int _type = WS;
412           int _channel = DEFAULT_TOKEN_CHANNEL;
413           {
414           match(' ');
415
416           }
417
418           state.type = _type;
419           state.channel = _channel;
420       }
421       finally {
422       }
423   }
424
425   public final void mIDENT() throws RecognitionException {
426       try {
427           int _type = IDENT;
428           int _channel = DEFAULT_TOKEN_CHANNEL;
429           {
430           match('\t');
431
432           }
433
434           state.type = _type;
435           state.channel = _channel;
436       }
437       finally {
438       }
439   }
440
441   public final void mSTRING() throws RecognitionException {
442       try {
443       int _type = STRING;
444       int _channel = DEFAULT_TOKEN_CHANNEL;
445       {
446       int cnt1=0;
447       loop1:
448       do {
449        int alt1=2;
450        int LA1_0 = input.LA(1);
451
452        if(((LA1_0>='A' && LA1_0<='Z')||
453            (LA1_0>='a' && LA1_0<='z'))) {
454            alt1=1;
455        }
456
457        switch (alt1) {
458     case 1 :
459     {
460      if((input.LA(1)>='A' && input.LA(1)<='Z')||
461         (input.LA(1)>='a'&&input.LA(1)<='z'))
462     {
463           input.consume();
464     }
465
466     else {
467       MismatchedSetException mse =
468                               new MismatchedSetException(null,input);
469       recover(mse);
```

```
470         throw mse;
471         }
472       }
473           break;
474       default :
475           if ( cnt1 >= 1 ) break loop1;
476                   EarlyExitException eee =
477                       new EarlyExitException(1, input);
478                   throw eee;
479           }
480           cnt1++;
481       } while (true);
482
483       }
484
485       state.type = _type;
486       state.channel = _channel;
487   }
488   finally {
489   }
490   }
491
492   public final void mINTEGER() throws RecognitionException {
493       try {
494           int _type = INTEGER;
495           int _channel = DEFAULT_TOKEN_CHANNEL;
496           {
497           int cnt2=0;
498           loop2:
499           do {
500               int alt2=2;
501               int LA2_0 = input.LA(1);
502
503               if ( ((LA2_0>='0' && LA2_0<='9')) ) {
504                   alt2=1;
505               }
506
507
508               switch (alt2) {
509             case 1 :
510                 {
511                 matchRange('0','9');
512
513                 }
514                 break;
515
516             default :
517                 if ( cnt2 >= 1 ) break loop2;
518                       EarlyExitException eee =
519                           new EarlyExitException(2, input);
520                       throw eee;
521               }
522               cnt2++;
523           } while (true);
524
525
526           }
527
528           state.type = _type;
529           state.channel = _channel;
530       }
531       finally {
```

```
532         }
533     }
534
535     public final void mNEWLINE() throws RecognitionException {
536       try {
537         int _type = NEWLINE;
538         int _channel = DEFAULT_TOKEN_CHANNEL;
539         {
540         int cnt3=0;
541         loop3:
542         do {
543             int alt3=2;
544             int LA3_0 = input.LA(1);
545
546             if ( (LA3_0=='\n'||LA3_0=='\r') ) {
547                 alt3=1;
548             }
549
550
551             switch (alt3) {
552           case 1 :
553               {
554            if ( input.LA(1)=='\n'||input.LA(1)=='\r' ) {
555                 input.consume();
556               }
557            else {
558             MismatchedSetException mse =
559                                     new MismatchedSetException(null,input);
560              recover(mse);
561              throw mse;}
562               }
563               break;
564
565           default :
566               if ( cnt3 >= 1 ) break loop3;
567                     EarlyExitException eee =
568                         new EarlyExitException(3, input);
569                     throw eee;
570             }
571             cnt3++;
572         } while (true);
573
574
575         }
576
577         state.type = _type;
578         state.channel = _channel;
579     }
580         finally {
581         }
582     }
583
584     public void mTokens() throws RecognitionException {
585         int alt4=29;
586         alt4 = dfa4.predict(input);
587         switch (alt4) {
588             case 1 :
589                 {
590                 mT__9();
591                 }
592                 break;
593             case 2 :
```

```
594                 {
595                 mT__10();
596                 }
597                 break;
598             case 3 :
599                 {
600                 mT__11();
601                 }
602                 break;
603             case 4 :
604                 {
605                 mT__12();
606                 }
607                 break;
608             case 5 :
609                 {
610                 mT__13();
611                 }
612                 break;
613             case 6 :
614                 {
615                 mT__14();
616                 }
617                 break;
618             case 7 :
619                 {
620                 mT__15();
621                 }
622                 break;
623             case 8 :
624                 {
625                 mT__16();
626                 }
627                 break;
628             case 9 :
629                 {
630                 mT__17();
631                 }
632                 break;
633             case 10 :
634                 {
635                 mT__18();
636                 }
637                 break;
638             case 11 :
639                 {
640                 mT__19();
641                 }
642                 break;
643             case 12 :
644                 {
645                 mT__20();
646                 }
647                 break;
648             case 13 :
649                 {
650                 mT__21();
651                 }
652                 break;
653             case 14 :
654                 {
655                 mT__22();
```

```
656                 }
657                 break;
658             case 15 :
659                 {
660                 mT__23();
661 
662                 }
663                 break;
664             case 16 :
665                 {
666                 mT__24();
667                 }
668                 break;
669             case 17 :
670                 {
671                 mT__25();
672                 }
673                 break;
674             case 18 :
675                 {
676                 mT__26();
677                 }
678                 break;
679             case 19 :
680                 {
681                 mT__27();
682                 }
683                 break;
684             case 20 :
685                 {
686                 mT__28();
687                 }
688                 break;
689             case 21 :
690                 {
691                 mT__29();
692                 }
693                 break;
694             case 22 :
695                 {
696                 mT__30();
697                 }
698                 break;
699             case 23 :
700                 {
701                 mT__31();
702                 }
703                 break;
704             case 24 :
705                 {
706                 mT__32();
707                 }
708                 break;
709             case 25 :
710                 {
711                 mWS();
712                 }
713                 break;
714             case 26 :
715                 {
716                 mIDENT();
717                 }
```

```
718                 break;
719             case 27 :
720                 {
721                 mSTRING();
722                 }
723                 break;
724             case 28 :
725                 {
726                 mINTEGER();
727                 }
728                 break;
729             case 29 :
730                 {
731                 mNEWLINE();
732                 }
733                 break;
734         }
735     }
736
737     protected DFA4 dfa4 = new DFA4(this);
738      static final String DFA4_eotS =
739       "\1\uffff\2\22\4\uffff\11\22\5\uffff\10\22\1\61\13\22\1\75\7\22\1"+
740       "\uffff\13\22\1\uffff\5\22\1\125\1\22\1\127\1\22\1\uffff\4\22\1\uffff"+
741       "\2\22\1\137\3\22\1\143\1\22\3\uffff\1\145\2\22\2\uffff\2\22\1\uffff"+
742       "\1\152\1\22\2\uffff\1\154\1\uffff\4\22\1\uffff\1\22\1\uffff\6\22"+
743       "\1\uffff\1\170\1\22\1\uffff\1\172\1\uffff\1\173\2\uffff";
744      static final String DFA4_eofS =
745       "\174\uffff";
746      static final String DFA4_minS =
747       "\1\11\1\105\1\116\4\uffff\1\103\1\105\1\117\1\104\1\101\1\103\1"+
748       "\117\1\111\1\102\5\uffff\1\107\1\104\1\124\1\123\1\103\1\115\1\101"+
749       "\1\111\2\101\1\122\1\102\1\110\1\122\1\123\1\115\1\124\1\104\1\112"+
750       "\1\111\1\101\1\111\1\105\1\117\1\103\1\101\1\114\1\124\1\uffff\1"+
751       "\116\1\101\1\137\1\105\1\122\1\124\1\105\1\137\2\105\1\116\1\uffff"+
752       "\1\117\1\122\1\115\1\122\1\111\1\101\1\137\1\101\1\115\1\uffff\1"+
753       "\104\1\105\2\137\1\uffff\1\122\1\103\1\101\1\116\1\124\1\137\1\101"+
754       "\1\116\3\uffff\1\101\1\125\1\116\2\uffff\1\111\1\124\1\uffff\1\101"+
755       "\1\105\2\uffff\1\101\1\uffff\1\114\1\124\1\116\1\111\1\uffff\1\104"+
756       "\1\uffff\1\105\1\137\1\107\1\126\1\137\1\104\1\uffff\1\101\1\105"+
757       "\1\uffff\1\101\1\uffff\1\101\2\uffff";
758      static final String DFA4_maxS =
759       "\1\175\1\105\1\116\4\uffff\1\123\2\117\1\116\1\114\2\125\1\117\1"+
760       "\122\5\uffff\1\107\1\104\1\124\2\123\1\115\1\101\1\111\1\172\1\101"+
761       "\1\122\1\102\1\110\1\122\1\123\1\115\1\124\1\104\1\112\1\111\1\172"+
762       "\1\111\1\105\1\117\1\103\1\101\1\114\1\124\1\uffff\1\116\1\101\1"+
763       "\137\1\105\1\122\1\124\1\105\1\137\2\105\1\116\1\uffff\1\117\1\122"+
764       "\1\115\1\122\1\111\1\172\1\137\1\172\1\115\1\uffff\1\104\1\105\2"+
765       "\137\1\uffff\1\122\1\103\1\172\1\116\1\124\1\137\1\172\1\116\3\uffff"+
766       "\1\172\1\125\1\116\2\uffff\1\111\1\124\1\uffff\1\172\1\105\2\uffff"+
767       "\1\172\1\uffff\1\114\1\124\1\116\1\111\1\uffff\1\104\1\uffff\1\105"+
768       "\1\137\1\107\1\126\1\137\1\104\1\uffff\1\172\1\105\1\uffff\1\172"+
769       "\1\uffff\1\172\2\uffff";
770      static final String DFA4_acceptS =
771       "\3\uffff\1\3\1\4\1\5\1\6\11\uffff\1\31\1\32\1\33\1\34\1\35\34\uffff"+
772       "\1\14\13\uffff\1\2\11\uffff\1\20\4\uffff\1\27\10\uffff\1\11\1\12"+
773       "\1\13\3\uffff\1\24\1\23\2\uffff\1\1\2\uffff\1\10\1\17\1\uffff\1"+
774       "\16\4\uffff\1\7\1\uffff\1\15\6\uffff\1\21\2\uffff\1\22\1\uffff\1"+
775       "\25\1\uffff\1\26\1\30";
776      static final String DFA4_specialS =
777       "\174\uffff}>";
778      static final String[] DFA4_transitionS = {
779           "\1\21\1\24\2\uffff\1\24\22\uffff\1\20\17\uffff\12\23\1\6\1\5"+
```

```
780         "\5\uffff\1\7\1\1\1\15\1\10\1\2\1\22\1\11\1\22\1\12\5\22\1\17"+
781         "\1\13\2\22\1\14\1\16\6\22\6\uffff\32\22\1\3\1\uffff\1\4",
782         "\1\25",
783         "\1\26",
784         "",
785         "",
786         "",
787         "",
788         "\1\27\17\uffff\1\30",
789         "\1\31\11\uffff\1\32",
790         "\1\33",
791         "\1\35\11\uffff\1\34",
792         "\1\37\12\uffff\1\36",
793         "\1\41\21\uffff\1\40",
794         "\1\43\5\uffff\1\42",
795         "\1\44\5\uffff\1\45",
796         "\1\47\17\uffff\1\46",
797         "",
798         "",
799         "",
800         "",
801         "",
802         "\1\50",
803         "\1\51",
804         "\1\52",
805         "\1\53",
806         "\1\54\17\uffff\1\55",
807         "\1\56",
808         "\1\57",
809         "\1\60",
810         "\32\22\6\uffff\32\22",
811         "\1\62",
812         "\1\63",
813         "\1\64",
814         "\1\65",
815         "\1\66",
816         "\1\67",
817         "\1\70",
818         "\1\71",
819         "\1\72",
820         "\1\73",
821         "\1\74",
822         "\32\22\6\uffff\32\22",
823         "\1\76",
824         "\1\77",
825         "\1\100",
826         "\1\101",
827         "\1\102",
828         "\1\103",
829         "\1\104",
830         "",
831         "\1\105",
832         "\1\106",
833         "\1\107",
834         "\1\110",
835         "\1\111",
836         "\1\112",
837         "\1\113",
838         "\1\114",
839         "\1\115",
840         "\1\116",
841         "\1\117",
```

```
842         "",
843         "\1\120",
844         "\1\121",
845         "\1\122",
846         "\1\123",
847         "\1\124",
848         "\32\22\6\uffff\32\22",
849         "\1\126",
850         "\32\22\6\uffff\32\22",
851         "\1\130",
852         "",
853         "\1\131",
854         "\1\132",
855         "\1\133",
856         "\1\134",
857         "",
858         "\1\135",
859         "\1\136",
860         "\32\22\6\uffff\32\22",
861         "\1\140",
862         "\1\141",
863         "\1\142",
864         "\32\22\6\uffff\32\22",
865         "\1\144",
866         "",
867         "",
868         "",
869         "\32\22\6\uffff\32\22",
870         "\1\146",
871         "\1\147",
872         "",
873         "",
874         "\1\150",
875         "\1\151",
876         "",
877         "\32\22\6\uffff\32\22",
878       "\1\153",
879       "",
880       "",
881       "\32\22\6\uffff\32\22",
882       "",
883       "\1\155",
884       "\1\156",
885       "\1\157",
886       "\1\160",
887       "",
888       "\1\161",
889       "",
890       "\1\162",
891       "\1\163",
892       "\1\164",
893       "\1\165",
894       "\1\166",
895       "\1\167",
896       "",
897       "\32\22\6\uffff\32\22",
898       "\1\171",
899       "",
900       "\32\22\6\uffff\32\22",
901       "",
902       "\32\22\6\uffff\32\22",
903       "",
```

```
904         ""
905     };
906 
907     static final short[] DFA4_eot = DFA.unpackEncodedString(DFA4_eotS);
908     static final short[] DFA4_eof = DFA.unpackEncodedString(DFA4_eofS);
909     static final char[] DFA4_min =
910                                 DFA.unpackEncodedStringToUnsignedChars(DFA4_minS);
911     static final char[] DFA4_max =
912                                 DFA.unpackEncodedStringToUnsignedChars(DFA4_maxS);
913     static final short[] DFA4_accept =
914                                 DFA.unpackEncodedString(DFA4_acceptS);
915     static final short[] DFA4_special =
916                                 DFA.unpackEncodedString(DFA4_specialS);
917     static final short[][] DFA4_transition;
918 
919     static {
920         int numStates = DFA4_transitionS.length;
921         DFA4_transition = new short[numStates][];
922         for (int i=0; i<numStates; i++) {
923             DFA4_transition[i] = DFA.unpackEncodedString(DFA4_transitionS[i]);
924         }
925     }
926 
927     class DFA4 extends DFA {
928 
929         public DFA4(BaseRecognizer recognizer) {
930             this.recognizer = recognizer;
931             this.decisionNumber = 4;
932             this.eot = DFA4_eot;
933             this.eof = DFA4_eof;
934             this.min = DFA4_min;
935             this.max = DFA4_max;
936             this.accept = DFA4_accept;
937             this.special = DFA4_special;
938             this.transition = DFA4_transition;
939         }
940         public String getDescription() {
941           return "1:1:␣Tokens␣:␣(␣T__9␣|␣T__10␣|␣T__11␣|␣T__12␣|
942 T__13␣|␣T__14␣|␣T__15␣|␣T__16␣|␣T__17␣|␣T__18␣|␣T__19␣|
943 T__20␣|␣T__21␣|␣T__22␣|␣T__23␣|␣T__24␣|␣T__25␣|
944 T__26␣|␣T__27␣|␣T__28␣|␣T__29␣|␣T__30␣|␣T__31␣|␣T__32␣|
945 WS␣|␣IDENT␣|␣STRING␣|␣INTEGER␣|␣NEWLINE␣);";
946         }
947     }
```

B.2 Parser Code

```
1 
2 import org.antlr.runtime.*;
3 import java.util.stack;
4 import java.util.List;
5 import java.util.ArrayList;
6 
7 public class PKDLParser extends Parser {
8     public static final String[] tokenNames = new String[] {
9         "<invalid>", "<EOR>", "<DOWN>", "<UP>", "NEWLINE", "IDENT", "WS", "STRING",
10         "INTEGER", "'BEGIN'", "'END'", "'{'", "'}'", "';'", "':'", "'ACTION'",
11         "'DECOM_METHOD'", "'GOAL'", "'INIT_STATE'", "'PLAN'", "'ID'", "'DOMAIN'",
12         "'PARAM'", "'DESCR'", "'SUB_TASKS'", "'CURRENT_STATE'", "'ASSERTED_STATE'",
```

```
13          "'TIME_TOTAL'", "'COST_TOTAL'", "'ORDERING'", "'SCHEDULED'",
14          "'TOT_TIME'", "'OBJECTIVE'"
15      };
16      public static final int EOF=-1;
17      public static final int T__9=9;
18      public static final int T__10=10;
19      public static final int T__11=11;
20      public static final int T__12=12;
21      public static final int T__13=13;
22      public static final int T__14=14;
23      public static final int T__15=15;
24      public static final int T__16=16;
25      public static final int T__17=17;
26      public static final int T__18=18;
27      public static final int T__19=19;
28      public static final int T__20=20;
29      public static final int T__21=21;
30      public static final int T__22=22;
31      public static final int T__23=23;
32      public static final int T__24=24;
33      public static final int T__25=25;
34      public static final int T__26=26;
35      public static final int T__27=27;
36      public static final int T__28=28;
37      public static final int T__29=29;
38      public static final int T__30=30;
39      public static final int T__31=31;
40      public static final int T__32=32;
41      public static final int NEWLINE=4;
42      public static final int IDENT=5;
43      public static final int WS=6;
44      public static final int STRING=7;
45      public static final int INTEGER=8;
46
47          public PKDLParser(TokenStream input) {
48              this(input, new RecognizerSharedState());
49          }
50          public PKDLParser(TokenStream input, RecognizerSharedState state) {
51              super(input, state);
52          }
53
54      public String[] getTokenNames() {
55              return PKDLParser.tokenNames; }
56      public String getGrammarFileName() {
57              return "/users/wiss/liufeng/work/diss/mydiss/tools/antlr/PKDL.g"; }
58
59      public final void pkd() throws RecognitionException {
60          try {
61              {
62              pushFollow(FOLLOW_domainID_in_pkd11);
63              domainID();
64
65              state._fsp--;
66
67              match(input,NEWLINE,FOLLOW_NEWLINE_in_pkd13);
68              match(input,9,FOLLOW_9_in_pkd15);
69              match(input,NEWLINE,FOLLOW_NEWLINE_in_pkd17);
70              match(input,IDENT,FOLLOW_IDENT_in_pkd19);
71              pushFollow(FOLLOW_blocks_in_pkd21);
72              blocks();
73
74              state._fsp--;
```

```
75
76              match(input,10,FOLLOW_10_in_pkd23);
77
78              }
79
80          }
81          catch (RecognitionException re) {
82              reportError(re);
83              recover(input,re);
84          }
85          finally {
86          }
87          return ;
88      }
89
90
91      public final void blocks() throws RecognitionException {
92          try{
93              {
94              int cnt1=0;
95              loop1:
96              do {
97                  int alt1=2;
98                  int LA1_0 = input.LA(1);
99
100                 if ( ((LA1_0>=15 && LA1_0<=19)) ) {
101                     alt1=1;
102                 }
103
104                 switch (alt1) {
105               case 1 :
106                   {
107                   pushFollow(FOLLOW_typed_block_in_blocks32);
108                   typed_block();
109
110                   state._fsp--;
111                   }
112                   break;
113
114               default :
115                   if ( cnt1 >= 1 ) break loop1;
116                         EarlyExitException eee =
117                             new EarlyExitException(1, input);
118                         throw eee;
119               }
120               cnt1++;
121           } while (true);
122
123
124           }
125
126       }
127       catch (RecognitionException re) {
128           reportError(re);
129           recover(input,re);
130       }
131       finally {
132       }
133       return ;
134   }
135
136
```

```
137     public final void delimitorl() throws RecognitionException {
138         try {
139             {
140             match(input,11,FOLLOW_11_in_delimitorl44);
141 
142             }
143 
144         }
145         catch (RecognitionException re) {
146             reportError(re);
147             recover(input,re);
148         }
149         finally {
150         }
151         return ;
152     }
153 
154     public final void delimitorr() throws RecognitionException {
155         try {
156             {
157             match(input,12,FOLLOW_12_in_delimitorr53);
158 
159             }
160 
161         }
162         catch (RecognitionException re) {
163             reportError(re);
164             recover(input,re);
165         }
166         finally {
167         }
168         return ;
169     }
170 
171     public final void semi() throws RecognitionException {
172         try {
173             {
174             match(input,13,FOLLOW_13_in_semi62);
175             }
176 
177         }
178         catch (RecognitionException re) {
179             reportError(re);
180             recover(input,re);
181         }
182         finally {
183         }
184         return ;
185     }
186 
187 
188     public final void colon() throws RecognitionException {
189         try {
190             {
191             match(input,14,FOLLOW_14_in_colon70);
192 
193             }
194 
195         }
196         catch (RecognitionException re) {
197             reportError(re);
198             recover(input,re);
```

```
199         }
200         finally {
201         }
202         return ;
203     }
204 
205 
206     public final void keyword() throws RecognitionException {
207         try {
208             {
209             int cnt2=0;
210             loop2:
211             do {
212                 int alt2=2;
213                 int LA2_0 = input.LA(1);
214                 if ( ((LA2_0>=STRING && LA2_0<=INTEGER)) ) {
215                     alt2=1;
216                 }
217                 switch (alt2) {
218               case 1 :
219                   {
220                   if ( (input.LA(1)>=STRING && input.LA(1)<=INTEGER) ) {
221                       input.consume();
222                       state.errorRecovery=false;
223                   }
224                   else {
225                       MismatchedSetException mse = new MismatchedSetException(null,input)
226                       throw mse;
227                   }
228                   }
229                   break;
230               default :
231                   if ( cnt2 >= 1 ) break loop2;
232                         EarlyExitException eee =
233                             new EarlyExitException(2, input);
234                         throw eee;
235               }
236               cnt2++;
237             } while (true);
238 
239             }
240         }
241         catch (RecognitionException re) {
242             reportError(re);
243             recover(input,re);
244         }
245         finally {
246         }
247         return ;
248     }
249 
250     public final void input() throws RecognitionException {
251         try {
252             {
253             int cnt3=0;
254             loop3:
255             do {
256                 int alt3=2;
257                 int LA3_0 = input.LA(1);
258                 if ( (LA3_0==STRING) ) {
259                     alt3=1;
260                 }
```

```
261                 switch (alt3) {
262               case 1 :
263                   {
264                   match(input,STRING,FOLLOW_STRING_in_input107);
265                   }
266                   break;
267               default :
268                   if ( cnt3 >= 1 ) break loop3;
269                         EarlyExitException eee =
270                             new EarlyExitException(3, input);
271                         throw eee;
272                 }
273                 cnt3++;
274             } while (true);
275             }
276         }
277         catch (RecognitionException re) {
278             reportError(re);
279             recover(input,re);
280         }
281         finally {
282         }
283         return ;
284     }
285
286     public final void domainID() throws RecognitionException {
287         try {
288             {
289             int cnt4=0;
290             loop4:
291             do {
292                 int alt4=2;
293                 int LA4_0 = input.LA(1);
294                 if ( (LA4_0==STRING) ) {
295                     alt4=1;
296                 }
297                 switch (alt4) {
298               case 1 :
299                   {
300                   match(input,STRING,FOLLOW_STRING_in_domainID163);
301                   }
302                   break;
303               default :
304                   if ( cnt4 >= 1 ) break loop4;
305                         EarlyExitException eee =
306                             new EarlyExitException(4, input);
307                         throw eee;
308                 }
309                 cnt4++;
310             } while (true);
311             }
312         }
313         catch (RecognitionException re) {
314             reportError(re);
315             recover(input,re);
316         }
317         finally {
318         }
319         return ;
320     }
321
322     public final void typed_block() throws RecognitionException {
```

```
323         try {
324             int alt10=5;
325             switch ( input.LA(1) ) {
326             case 15:
327                 {
328                 alt10=1;
329                 }
330                 break;
331             case 16:
332                 {
333                 alt10=2;
334                 }
335                 break;
336             case 17:
337                 {
338                 alt10=3;
339                 }
340                 break;
341             case 18:
342                 {
343                 alt10=4;
344                 }
345                 break;
346             case 19:
347                 {
348                 alt10=5;
349                 }
350                 break;
351             default:
352                 NoViableAltException nvae =
353                     new NoViableAltException("", 10, 0, input);
354 
355                 throw nvae;
356             }
357 
358             switch (alt10) {
359                 case 1 :
360                     {
361                     match(input,15,FOLLOW_15_in_typed_block177);
362                     match(input,NEWLINE,FOLLOW_NEWLINE_in_typed_block179);
363                     match(input,IDENT,FOLLOW_IDENT_in_typed_block181);
364                     pushFollow(FOLLOW_delimitorl_in_typed_block183);
365                     delimitorl();
366 
367                     state._fsp--;
368 
369                     match(input,NEWLINE,FOLLOW_NEWLINE_in_typed_block185);
370                     int cnt5=0;
371                     loop5:
372                     do {
373                         int alt5=2;
374                         int LA5_0 = input.LA(1);
375 
376                         if ( (LA5_0==IDENT) ) {
377                             int LA5_1 = input.LA(2);
378 
379                             if ( (LA5_1==IDENT) ) {
380                                 alt5=1;
381                             }
382                         }
383 
384                         switch (alt5) {
```

```
385                         case 1 :
386                             {
387                             pushFollow(FOLLOW_action_in_typed_block188);
388                             action();
389 
390                             state._fsp--;
391 
392                             }
393                             break;
394 
395                         default :
396                             if ( cnt5 >= 1 ) break loop5;
397                                 EarlyExitException eee =
398                                     new EarlyExitException(5, input);
399                                 throw eee;
400                         }
401                         cnt5++;
402                     } while (true);
403 
404                     match(input,IDENT,FOLLOW_IDENT_in_typed_block192);
405                     pushFollow(FOLLOW_delimitorr_in_typed_block194);
406                     delimitorr();
407 
408                     state._fsp--;
409 
410                     match(input,NEWLINE,FOLLOW_NEWLINE_in_typed_block196);
411 
412                     }
413                     break;
414                 case 2 :
415                     {
416                     match(input,16,FOLLOW_16_in_typed_block201);
417                     match(input,NEWLINE,FOLLOW_NEWLINE_in_typed_block202);
418                     match(input,IDENT,FOLLOW_IDENT_in_typed_block204);
419                     pushFollow(FOLLOW_delimitorl_in_typed_block206);
420                     delimitorl();
421 
422                     state._fsp--;
423 
424                     match(input,NEWLINE,FOLLOW_NEWLINE_in_typed_block208);
425                     int cnt6=0;
426                     loop6:
427                     do {
428                         int alt6=2;
429                         int LA6_0 = input.LA(1);
430 
431                         if ( (LA6_0==IDENT) ) {
432                             int LA6_1 = input.LA(2);
433 
434                             if ( (LA6_1==IDENT) ) {
435                                 alt6=1;
436                             }
437                         }
438 
439                         switch (alt6) {
440                       case 1 :
441                             {
442                             pushFollow(FOLLOW_method_in_typed_block211);
443                             method();
444 
445                             state._fsp--;
446 
```

```
447                             }
448                             break;
449
450                         default :
451                             if ( cnt6 >= 1 ) break loop6;
452                                 EarlyExitException eee =
453                                     new EarlyExitException(6, input);
454                                 throw eee;
455                         }
456                         cnt6++;
457                     } while (true);
458
459                     match(input,IDENT,FOLLOW_IDENT_in_typed_block215);
460                     pushFollow(FOLLOW_delimitorr_in_typed_block217);
461                     delimitorr();
462
463                     state._fsp--;
464
465                     match(input,NEWLINE,FOLLOW_NEWLINE_in_typed_block219);
466
467                     }
468                     break;
469                 case 3 :
470                     {
471                     match(input,17,FOLLOW_17_in_typed_block224);
472                     match(input,NEWLINE,FOLLOW_NEWLINE_in_typed_block226);
473                     match(input,IDENT,FOLLOW_IDENT_in_typed_block228);
474                     pushFollow(FOLLOW_delimitorl_in_typed_block230);
475                     delimitorl();
476
477                     state._fsp--;
478
479                     match(input,NEWLINE,FOLLOW_NEWLINE_in_typed_block232);
480                     int cnt7=0;
481                     loop7:
482                     do {
483                         int alt7=2;
484                         int LA7_0 = input.LA(1);
485
486                         if ( (LA7_0==IDENT) ) {
487                             int LA7_1 = input.LA(2);
488
489                             if ( (LA7_1==IDENT) ) {
490                                 alt7=1;
491                             }
492                         }
493
494                         switch (alt7) {
495                       case 1 :
496                           {
497                           pushFollow(FOLLOW_goal_in_typed_block235);
498                           goal();
499
500                           state._fsp--;
501
502                           }
503                           break;
504
505                       default :
506                           if ( cnt7 >= 1 ) break loop7;
507                               EarlyExitException eee =
508                                   new EarlyExitException(7, input);
```

```
509                             throw eee;
510                         }
511                         cnt7++;
512                     } while (true);
513
514                     match(input,IDENT,FOLLOW_IDENT_in_typed_block239);
515                     pushFollow(FOLLOW_delimitorr_in_typed_block241);
516                     delimitorr();
517
518                     state._fsp--;
519
520                     match(input,NEWLINE,FOLLOW_NEWLINE_in_typed_block243);
521
522                     }
523                     break;
524                 case 4 :
525                     {
526                     match(input,18,FOLLOW_18_in_typed_block248);
527                     match(input,NEWLINE,FOLLOW_NEWLINE_in_typed_block250);
528                     match(input,IDENT,FOLLOW_IDENT_in_typed_block252);
529                     pushFollow(FOLLOW_delimitorl_in_typed_block254);
530                     delimitorl();
531
532                     state._fsp--;
533
534                     match(input,NEWLINE,FOLLOW_NEWLINE_in_typed_block256);
535                     int cnt8=0;
536                     loop8:
537                     do {
538                         int alt8=2;
539                         int LA8_0 = input.LA(1);
540
541                         if ( (LA8_0==IDENT) ) {
542                             int LA8_1 = input.LA(2);
543
544                             if ( (LA8_1==IDENT) ) {
545                                 alt8=1;
546                             }
547                         }
548
549                         switch (alt8) {
550                       case 1 :
551                           {
552                           pushFollow(FOLLOW_state_in_typed_block259);
553                           state();
554
555                           state._fsp--;
556
557                           }
558                           break;
559
560                       default :
561                           if ( cnt8 >= 1 ) break loop8;
562                                 EarlyExitException eee =
563                                     new EarlyExitException(8, input);
564                                 throw eee;
565                       }
566                       cnt8++;
567                     } while (true);
568
569                     match(input,IDENT,FOLLOW_IDENT_in_typed_block263);
570                     pushFollow(FOLLOW_delimitorr_in_typed_block265);
```

```
571                     delimitorr();
572
573                     state._fsp--;
574
575                     match(input,NEWLINE,FOLLOW_NEWLINE_in_typed_block267);
576
577                     }
578                     break;
579                 case 5 :
580                     {
581                     match(input,19,FOLLOW_19_in_typed_block272);
582                     match(input,NEWLINE,FOLLOW_NEWLINE_in_typed_block274);
583                     match(input,IDENT,FOLLOW_IDENT_in_typed_block276);
584                     pushFollow(FOLLOW_delimitorl_in_typed_block278);
585                     delimitorl();
586
587                     state._fsp--;
588
589                     match(input,NEWLINE,FOLLOW_NEWLINE_in_typed_block280);
590                     int cnt9=0;
591                     loop9:
592                     do {
593                         int alt9=2;
594                         int LA9_0 = input.LA(1);
595
596                         if ( (LA9_0==IDENT) ) {
597                             int LA9_1 = input.LA(2);
598
599                             if ( (LA9_1==IDENT) ) {
600                                 alt9=1;
601                             }
602                         }
603
604                         switch (alt9) {
605                       case 1 :
606                           {
607                           pushFollow(FOLLOW_plan_in_typed_block283);
608                           plan();
609
610                           state._fsp--;
611
612                           }
613                           break;
614
615                       default :
616                           if ( cnt9 >= 1 ) break loop9;
617                                 EarlyExitException eee =
618                                     new EarlyExitException(9, input);
619                                 throw eee;
620                       }
621                       cnt9++;
622                     } while (true);
623
624                     match(input,IDENT,FOLLOW_IDENT_in_typed_block287);
625                     pushFollow(FOLLOW_delimitorr_in_typed_block289);
626                     delimitorr();
627
628                     state._fsp--;
629
630                     match(input,NEWLINE,FOLLOW_NEWLINE_in_typed_block291);
631
632                     }
```

```
633                     break;
634                 }
635             }
636             catch (RecognitionException re) {
637                 reportError(re);
638                 recover(input,re);
639             }
640             finally {
641             }
642             return ;
643         }
644 
645         public final void identifier() throws RecognitionException {
646             try {
647                 {
648                 int cnt11=0;
649                 loop11:
650                 do {
651                     int alt11=2;
652                     int LA11_0 = input.LA(1);
653 
654                     if ( ((LA11_0>=STRING && LA11_0<=INTEGER)) ) {
655                         alt11=1;
656                     }
657 
658                     switch (alt11) {
659                   case 1 :
660                       {
661                       if ( (input.LA(1)>=STRING && input.LA(1)<=INTEGER) ) {
662                           input.consume();
663                           state.errorRecovery=false;
664                       }
665                       else {
666                           MismatchedSetException mse = new MismatchedSetException(null,input);
667                           throw mse;
668                       }
669                       }
670                       break;
671 
672                   default :
673                       if ( cnt11 >= 1 ) break loop11;
674                             EarlyExitException eee =
675                                 new EarlyExitException(11, input);
676                             throw eee;
677                     }
678                     cnt11++;
679                 } while (true);
680                 }
681             }
682             catch (RecognitionException re) {
683                 reportError(re);
684                 recover(input,re);
685             }
686             finally {
687             }
688             return ;
689         }
690 
691 
692         public final void action() throws RecognitionException {
693             try {
694                 {
```

```
695             match(input,IDENT,FOLLOW_IDENT_in_action319);
696             match(input,IDENT,FOLLOW_IDENT_in_action321);
697             pushFollow(FOLLOW_act_key_in_action323);
698             act_key();
699 
700             state._fsp--;
701 
702             match(input,WS,FOLLOW_WS_in_action325);
703             pushFollow(FOLLOW_colon_in_action327);
704             colon();
705 
706             state._fsp--;
707 
708             match(input,WS,FOLLOW_WS_in_action329);
709             match(input,STRING,FOLLOW_STRING_in_action331);
710             pushFollow(FOLLOW_semi_in_action333);
711             semi();
712 
713             state._fsp--;
714 
715             match(input,NEWLINE,FOLLOW_NEWLINE_in_action335);
716 
717             }
718 
719         }
720         catch (RecognitionException re) {
721             reportError(re);
722             recover(input,re);
723         }
724         finally {
725         }
726         return ;
727     }
728 
729     public final void act_key() throws RecognitionException {
730         try {
731             {
732             int cnt12=0;
733             loop12:
734             do {
735                 int alt12=2;
736                 int LA12_0 = input.LA(1);
737 
738                 if ( ((LA12_0>=20 && LA12_0<=23)) ) {
739                     alt12=1;
740                 }
741 
742                 switch (alt12) {
743               case 1 :
744                   {
745                   if ( (input.LA(1)>=20 && input.LA(1)<=23) ) {
746                       input.consume();
747                       state.errorRecovery=false;
748                   }
749                   else {
750                       MismatchedSetException mse = new MismatchedSetException(null,input)
751                       throw mse;
752                   }
753                   }
754                   break;
755 
756               default :
```

```
757                     if ( cnt12 >= 1 ) break loop12;
758                           EarlyExitException eee =
759                               new EarlyExitException(12, input);
760                           throw eee;
761                     }
762                     cnt12++;
763                 } while (true);
764                 }
765             }
766             catch (RecognitionException re) {
767                 reportError(re);
768                 recover(input,re);
769             }
770             finally {
771             }
772             return ;
773         }
774 
775         public final void method() throws RecognitionException {
776             try {
777                 {
778                 match(input,IDENT,FOLLOW_IDENT_in_method368);
779                 match(input,IDENT,FOLLOW_IDENT_in_method370);
780                 pushFollow(FOLLOW_mtd_key_in_method372);
781                 mtd_key();
782 
783                 state._fsp--;
784 
785                 match(input,WS,FOLLOW_WS_in_method374);
786                 pushFollow(FOLLOW_colon_in_method376);
787                 colon();
788 
789                 state._fsp--;
790 
791                 match(input,WS,FOLLOW_WS_in_method378);
792                 match(input,STRING,FOLLOW_STRING_in_method380);
793                 pushFollow(FOLLOW_semi_in_method382);
794                 semi();
795 
796                 state._fsp--;
797 
798                 match(input,NEWLINE,FOLLOW_NEWLINE_in_method384);
799 
800                 }
801 
802             }
803             catch (RecognitionException re) {
804                 reportError(re);
805                 recover(input,re);
806             }
807             finally {
808             }
809             return ;
810         }
811 
812 
813         public final void mtd_key() throws RecognitionException {
814             try {
815                 {
816                 int cnt13=0;
817                 loop13:
818                 do {
```

```
819                 int alt13=2;
820                 int LA13_0 = input.LA(1);
821 
822                 if ( (LA13_0==20||LA13_0==24) ) {
823                     alt13=1;
824                 }
825 
826                 switch (alt13) {
827             case 1 :
828                 {
829                 if ( input.LA(1)==20||input.LA(1)==24 ) {
830                     input.consume();
831                     state.errorRecovery=false;
832                 }
833                 else {
834                     MismatchedSetException mse = new MismatchedSetException(null,input)
835                     throw mse;
836                 }
837 
838                 }
839                 break;
840 
841             default :
842                 if ( cnt13 >= 1 ) break loop13;
843                       EarlyExitException eee =
844                           new EarlyExitException(13, input);
845                       throw eee;
846                 }
847                 cnt13++;
848             } while (true);
849             }
850         }
851         catch (RecognitionException re) {
852             reportError(re);
853             recover(input,re);
854         }
855         finally {
856         }
857         return ;
858     }
859 
860 
861     public final void state() throws RecognitionException {
862         try {
863             {
864             match(input,IDENT,FOLLOW_IDENT_in_state407);
865             match(input,IDENT,FOLLOW_IDENT_in_state409);
866             pushFollow(FOLLOW_state_key_in_state411);
867             state_key();
868 
869             state._fsp--;
870 
871             match(input,WS,FOLLOW_WS_in_state413);
872             pushFollow(FOLLOW_colon_in_state415);
873             colon();
874 
875             state._fsp--;
876 
877             match(input,WS,FOLLOW_WS_in_state417);
878             match(input,STRING,FOLLOW_STRING_in_state419);
879             pushFollow(FOLLOW_semi_in_state421);
880             semi();
```

```
881
882             state._fsp--;
883
884             match(input,NEWLINE,FOLLOW_NEWLINE_in_state423);
885
886             }
887
888         }
889         catch (RecognitionException re) {
890             reportError(re);
891             recover(input,re);
892         }
893         finally {
894         }
895         return ;
896     }
897
898
899     public final void state_key() throws RecognitionException {
900         try {
901             {
902             int cnt14=0;
903             loop14:
904             do {
905                 int alt14=2;
906                 int LA14_0 = input.LA(1);
907
908                 if ( (LA14_0==20||LA14_0==25) ) {
909                     alt14=1;
910                 }
911
912                 switch (alt14) {
913               case 1 :
914                   {
915                   if ( input.LA(1)==20||input.LA(1)==25 ) {
916                       input.consume();
917                       state.errorRecovery=false;
918                   }
919                   else {
920                       MismatchedSetException mse = new MismatchedSetException(null,input);
921                       throw mse;
922                   }
923
924                   }
925                   break;
926
927               default :
928                   if ( cnt14 >= 1 ) break loop14;
929                         EarlyExitException eee =
930                             new EarlyExitException(14, input);
931                         throw eee;
932               }
933               cnt14++;
934             } while (true);
935             }
936         }
937         catch (RecognitionException re) {
938             reportError(re);
939             recover(input,re);
940         }
941         finally {
942         }
```

```
943         return ;
944     }
945
946
947     public final void goal() throws RecognitionException {
948         try {
949             {
950             match(input,IDENT,FOLLOW_IDENT_in_goal452);
951             match(input,IDENT,FOLLOW_IDENT_in_goal454);
952             pushFollow(FOLLOW_goal_key_in_goal456);
953             goal_key();
954
955             state._fsp--;
956
957             match(input,WS,FOLLOW_WS_in_goal458);
958             pushFollow(FOLLOW_colon_in_goal460);
959             colon();
960
961             state._fsp--;
962
963             match(input,WS,FOLLOW_WS_in_goal462);
964             match(input,STRING,FOLLOW_STRING_in_goal464);
965             pushFollow(FOLLOW_semi_in_goal466);
966             semi();
967
968             state._fsp--;
969
970             match(input,NEWLINE,FOLLOW_NEWLINE_in_goal468);
971
972             }
973
974         }
975         catch (RecognitionException re) {
976             reportError(re);
977             recover(input,re);
978         }
979         finally {
980         }
981         return ;
982     }
983
984
985     public final void goal_key() throws RecognitionException {
986         try {
987             {
988             int cnt15=0;
989             loop15:
990             do {
991                 int alt15=2;
992                 int LA15_0 = input.LA(1);
993
994                 if ( ((LA15_0>=26 && LA15_0<=28)) ) {
995                     alt15=1;
996                 }
997
998                 switch (alt15) {
999               case 1 :
1000                  {
1001                  if ( (input.LA(1)>=26 && input.LA(1)<=28) ) {
1002                      input.consume();
1003                      state.errorRecovery=false;
1004                  }
```

```
1005                    else {
1006                        MismatchedSetException mse = new MismatchedSetException(null,input);
1007                        throw mse;
1008                    }
1009                    }
1010                    break;
1011
1012                default :
1013                    if ( cnt15 >= 1 ) break loop15;
1014                          EarlyExitException eee =
1015                              new EarlyExitException(15, input);
1016                          throw eee;
1017                  }
1018                  cnt15++;
1019              } while (true);
1020              }
1021          }
1022          catch (RecognitionException re) {
1023              reportError(re);
1024              recover(input,re);
1025          }
1026          finally {
1027          }
1028          return ;
1029      }
1030
1031
1032      public final void plan() throws RecognitionException {
1033          try {
1034              {
1035              match(input,IDENT,FOLLOW_IDENT_in_plan495);
1036              match(input,IDENT,FOLLOW_IDENT_in_plan497);
1037              pushFollow(FOLLOW_plan_key_in_plan499);
1038              plan_key();
1039
1040              state._fsp--;
1041
1042              match(input,WS,FOLLOW_WS_in_plan501);
1043              pushFollow(FOLLOW_colon_in_plan503);
1044              colon();
1045
1046              state._fsp--;
1047
1048              match(input,WS,FOLLOW_WS_in_plan505);
1049              match(input,STRING,FOLLOW_STRING_in_plan507);
1050              pushFollow(FOLLOW_semi_in_plan509);
1051              semi();
1052
1053              state._fsp--;
1054
1055              match(input,NEWLINE,FOLLOW_NEWLINE_in_plan511);
1056
1057              }
1058
1059          }
1060          catch (RecognitionException re) {
1061              reportError(re);
1062              recover(input,re);
1063          }
1064          finally {
1065          }
1066          return ;
```

```
1067    }
1068
1069    public final void plan_key() throws RecognitionException {
1070        try {
1071            {
1072            int cnt16=0;
1073            loop16:
1074            do {
1075                int alt16=2;
1076                int LA16_0 = input.LA(1);
1077
1078                if ( (LA16_0==15||(LA16_0>=29 && LA16_0<=32)) ) {
1079                    alt16=1;
1080                }
1081
1082                switch (alt16) {
1083              case 1 :
1084                  {
1085                  if ( input.LA(1)==15||(input.LA(1)>=29 && input.LA(1)<=32) ) {
1086                      input.consume();
1087                      state.errorRecovery=false;
1088                  }
1089                  else {
1090                      MismatchedSetException mse = new MismatchedSetException(null,input)
1091                      throw mse;
1092                  }
1093                  }
1094                    break;
1095
1096              default :
1097                  if ( cnt16 >= 1 ) break loop16;
1098                        EarlyExitException eee =
1099                            new EarlyExitException(16, input);
1100                        throw eee;
1101                }
1102                cnt16++;
1103            } while (true);
1104
1105            }
1106        }
1107        catch (RecognitionException re) {
1108            reportError(re);
1109            recover(input,re);
1110        }
1111        finally {
1112        }
1113        return ;
1114    }
1115
1116    public static final BitSet FOLLOW_domainID_in_pkd11 =
1117                                new BitSet(new long[]{0x0000000000000010L});
1118    public static final BitSet FOLLOW_NEWLINE_in_pkd13 =
1119                                new BitSet(new long[]{0x0000000000000200L});
1120    public static final BitSet FOLLOW_9_in_pkd15 =
1121                                new BitSet(new long[]{0x0000000000000010L});
1122    public static final BitSet FOLLOW_NEWLINE_in_pkd17 =
1123                                new BitSet(new long[]{0x0000000000000020L});
1124    public static final BitSet FOLLOW_IDENT_in_pkd19 =
1125                                new BitSet(new long[]{0x00000000000F8000L});
1126    public static final BitSet FOLLOW_blocks_in_pkd21 =
1127                                new BitSet(new long[]{0x0000000000000400L});
1128    public static final BitSet FOLLOW_10_in_pkd23 =
```

```
1129                                    new BitSet(new long[]{0x0000000000000002L});
1130    public static final BitSet FOLLOW_typed_block_in_blocks32 =
1131                                    new BitSet(new long[]{0x000000000000F8002L});
1132    public static final BitSet FOLLOW_11_in_delimitorl44 =
1133                                    new BitSet(new long[]{0x0000000000000002L});
1134    public static final BitSet FOLLOW_12_in_delimitorr53 =
1135                                    new BitSet(new long[]{0x0000000000000002L});
1136    public static final BitSet FOLLOW_13_in_semi62 =
1137                                    new BitSet(new long[]{0x0000000000000002L});
1138    public static final BitSet FOLLOW_14_in_colon70 =
1139                                    new BitSet(new long[]{0x0000000000000002L});
1140    public static final BitSet FOLLOW_set_in_keyword94 =
1141                                    new BitSet(new long[]{0x0000000000000182L});
1142    public static final BitSet FOLLOW_STRING_in_input107 =
1143                                    new BitSet(new long[]{0x0000000000000082L});
1144    public static final BitSet FOLLOW_STRING_in_domainID163 =
1145                                    new BitSet(new long[]{0x0000000000000082L});
1146    public static final BitSet FOLLOW_15_in_typed_block177 =
1147                                    new BitSet(new long[]{0x0000000000000010L});
1148    public static final BitSet FOLLOW_NEWLINE_in_typed_block179 =
1149                                    new BitSet(new long[]{0x0000000000000020L});
1150    public static final BitSet FOLLOW_IDENT_in_typed_block181 =
1151                                    new BitSet(new long[]{0x0000000000000800L});
1152    public static final BitSet FOLLOW_delimitorl_in_typed_block183 =
1153                                    new BitSet(new long[]{0x0000000000000010L});
1154    public static final BitSet FOLLOW_NEWLINE_in_typed_block185 =
1155                                    new BitSet(new long[]{0x0000000000000020L});
1156    public static final BitSet FOLLOW_action_in_typed_block188 =
1157                                    new BitSet(new long[]{0x0000000000000020L});
1158    public static final BitSet FOLLOW_IDENT_in_typed_block192 =
1159                                    new BitSet(new long[]{0x0000000000001000L});
1160    public static final BitSet FOLLOW_delimitorr_in_typed_block194 =
1161                                    new BitSet(new long[]{0x0000000000000010L});
1162    public static final BitSet FOLLOW_NEWLINE_in_typed_block196 =
1163                                    new BitSet(new long[]{0x0000000000000002L});
1164    public static final BitSet FOLLOW_16_in_typed_block201 =
1165                                    new BitSet(new long[]{0x0000000000000010L});
1166    public static final BitSet FOLLOW_NEWLINE_in_typed_block202 =
1167                                    new BitSet(new long[]{0x0000000000000020L});
1168    public static final BitSet FOLLOW_IDENT_in_typed_block204 =
1169                                    new BitSet(new long[]{0x0000000000000800L});
1170    public static final BitSet FOLLOW_delimitorl_in_typed_block206 =
1171                                    new BitSet(new long[]{0x0000000000000010L});
1172    public static final BitSet FOLLOW_NEWLINE_in_typed_block208 =
1173                                    new BitSet(new long[]{0x0000000000000020L});
1174    public static final BitSet FOLLOW_method_in_typed_block211 =
1175                                    new BitSet(new long[]{0x0000000000000020L});
1176    public static final BitSet FOLLOW_IDENT_in_typed_block215 =
1177                                    new BitSet(new long[]{0x0000000000001000L});
1178    public static final BitSet FOLLOW_delimitorr_in_typed_block217 =
1179                                    new BitSet(new long[]{0x0000000000000010L});
1180    public static final BitSet FOLLOW_NEWLINE_in_typed_block219 =
1181                                    new BitSet(new long[]{0x0000000000000002L});
1182    public static final BitSet FOLLOW_17_in_typed_block224 =
1183                                    new BitSet(new long[]{0x0000000000000010L});
1184    public static final BitSet FOLLOW_NEWLINE_in_typed_block226 =
1185                                    new BitSet(new long[]{0x0000000000000020L});
1186    public static final BitSet FOLLOW_IDENT_in_typed_block228 =
1187                                    new BitSet(new long[]{0x0000000000000800L});
1188    public static final BitSet FOLLOW_delimitorl_in_typed_block230 =
1189                                    new BitSet(new long[]{0x0000000000000010L});
1190    public static final BitSet FOLLOW_NEWLINE_in_typed_block232 =
```

```
1191                                    new BitSet(new long[]{0x0000000000000020L});
1192        public static final BitSet FOLLOW_goal_in_typed_block235 =
1193                                    new BitSet(new long[]{0x0000000000000020L});
1194        public static final BitSet FOLLOW_IDENT_in_typed_block239 =
1195                                    new BitSet(new long[]{0x0000000000001000L});
1196        public static final BitSet FOLLOW_delimitorr_in_typed_block241 =
1197                                    new BitSet(new long[]{0x0000000000000010L});
1198        public static final BitSet FOLLOW_NEWLINE_in_typed_block243 =
1199                                    new BitSet(new long[]{0x0000000000000002L});
1200        public static final BitSet FOLLOW_18_in_typed_block248 =
1201                                    new BitSet(new long[]{0x0000000000000010L});
1202        public static final BitSet FOLLOW_NEWLINE_in_typed_block250 =
1203                                    new BitSet(new long[]{0x0000000000000020L});
1204        public static final BitSet FOLLOW_IDENT_in_typed_block252 =
1205                                    new BitSet(new long[]{0x0000000000000800L});
1206        public static final BitSet FOLLOW_delimitorl_in_typed_block254 =
1207                                    new BitSet(new long[]{0x0000000000000010L});
1208        public static final BitSet FOLLOW_NEWLINE_in_typed_block256 =
1209                                    new BitSet(new long[]{0x0000000000000020L});
1210        public static final BitSet FOLLOW_state_in_typed_block259 =
1211                                    new BitSet(new long[]{0x0000000000000020L});
1212        public static final BitSet FOLLOW_IDENT_in_typed_block263 =
1213                                    new BitSet(new long[]{0x0000000000001000L});
1214        public static final BitSet FOLLOW_delimitorr_in_typed_block265 =
1215                                    new BitSet(new long[]{0x0000000000000010L});
1216        public static final BitSet FOLLOW_NEWLINE_in_typed_block267 =
1217                                    new BitSet(new long[]{0x0000000000000002L});
1218        public static final BitSet FOLLOW_19_in_typed_block272 =
1219                                    new BitSet(new long[]{0x0000000000000010L});
1220        public static final BitSet FOLLOW_NEWLINE_in_typed_block274 =
1221                                    new BitSet(new long[]{0x0000000000000020L});
1222        public static final BitSet FOLLOW_IDENT_in_typed_block276 =
1223                                    new BitSet(new long[]{0x0000000000000800L});
1224        public static final BitSet FOLLOW_delimitorl_in_typed_block278 =
1225                                    new BitSet(new long[]{0x0000000000000010L});
1226        public static final BitSet FOLLOW_NEWLINE_in_typed_block280 =
1227                                    new BitSet(new long[]{0x0000000000000020L});
1228        public static final BitSet FOLLOW_plan_in_typed_block283 =
1229                                    new BitSet(new long[]{0x0000000000000020L});
1230        public static final BitSet FOLLOW_IDENT_in_typed_block287 =
1231                                    new BitSet(new long[]{0x0000000000001000L});
1232        public static final BitSet FOLLOW_delimitorr_in_typed_block289 =
1233                                    new BitSet(new long[]{0x0000000000000010L});
1234        public static final BitSet FOLLOW_NEWLINE_in_typed_block291 =
1235                                    new BitSet(new long[]{0x0000000000000002L});
1236        public static final BitSet FOLLOW_set_in_identifier303 =
1237                                    new BitSet(new long[]{0x0000000000000182L});
1238        public static final BitSet FOLLOW_IDENT_in_action319 =
1239                                    new BitSet(new long[]{0x0000000000000020L});
1240        public static final BitSet FOLLOW_IDENT_in_action321 =
1241                                    new BitSet(new long[]{0x0000000000F00000L});
1242        public static final BitSet FOLLOW_act_key_in_action323 =
1243                                    new BitSet(new long[]{0x0000000000000040L});
1244        public static final BitSet FOLLOW_WS_in_action325 =
1245                                    new BitSet(new long[]{0x0000000000004000L});
1246        public static final BitSet FOLLOW_colon_in_action327 =
1247                                    new BitSet(new long[]{0x0000000000000040L});
1248        public static final BitSet FOLLOW_WS_in_action329 =
1249                                    new BitSet(new long[]{0x0000000000000080L});
1250        public static final BitSet FOLLOW_STRING_in_action331 =
1251                                    new BitSet(new long[]{0x0000000000002000L});
1252        public static final BitSet FOLLOW_semi_in_action333 =
```

```
1253                                    new BitSet(new long[]{0x0000000000000010L});
1254        public static final BitSet FOLLOW_NEWLINE_in_action335 =
1255                                    new BitSet(new long[]{0x0000000000000002L});
1256        public static final BitSet FOLLOW_set_in_act_key342 =
1257                                    new BitSet(new long[]{0x0000000000F00002L});
1258        public static final BitSet FOLLOW_IDENT_in_method368 =
1259                                    new BitSet(new long[]{0x0000000000000020L});
1260        public static final BitSet FOLLOW_IDENT_in_method370 =
1261                                    new BitSet(new long[]{0x0000000001100000L});
1262        public static final BitSet FOLLOW_mtd_key_in_method372 =
1263                                    new BitSet(new long[]{0x0000000000000040L});
1264        public static final BitSet FOLLOW_WS_in_method374 =
1265                                    new BitSet(new long[]{0x0000000000004000L});
1266        public static final BitSet FOLLOW_colon_in_method376 =
1267                                    new BitSet(new long[]{0x0000000000000040L});
1268        public static final BitSet FOLLOW_WS_in_method378 =
1269                                    new BitSet(new long[]{0x0000000000000080L});
1270        public static final BitSet FOLLOW_STRING_in_method380 =
1271                                    new BitSet(new long[]{0x0000000000002000L});
1272        public static final BitSet FOLLOW_semi_in_method382 =
1273                                    new BitSet(new long[]{0x0000000000000010L});
1274        public static final BitSet FOLLOW_NEWLINE_in_method384 =
1275                                    new BitSet(new long[]{0x0000000000000002L});
1276        public static final BitSet FOLLOW_set_in_mtd_key391 =
1277                                    new BitSet(new long[]{0x0000000001100002L});
1278        public static final BitSet FOLLOW_IDENT_in_state407 =
1279                                    new BitSet(new long[]{0x0000000000000020L});
1280        public static final BitSet FOLLOW_IDENT_in_state409 =
1281                                    new BitSet(new long[]{0x0000000002100000L});
1282        public static final BitSet FOLLOW_state_key_in_state411 =
1283                                    new BitSet(new long[]{0x0000000000000040L});
1284        public static final BitSet FOLLOW_WS_in_state413 =
1285                                    new BitSet(new long[]{0x0000000000004000L});
1286        public static final BitSet FOLLOW_colon_in_state415 =
1287                                    new BitSet(new long[]{0x0000000000000040L});
1288        public static final BitSet FOLLOW_WS_in_state417 =
1289                                    new BitSet(new long[]{0x0000000000000080L});
1290        public static final BitSet FOLLOW_STRING_in_state419 =
1291                                    new BitSet(new long[]{0x0000000000002000L});
1292        public static final BitSet FOLLOW_semi_in_state421 =
1293                                    new BitSet(new long[]{0x0000000000000010L});
1294        public static final BitSet FOLLOW_NEWLINE_in_state423 =
1295                                    new BitSet(new long[]{0x0000000000000002L});
1296        public static final BitSet FOLLOW_set_in_state_key431 =
1297                                    new BitSet(new long[]{0x0000000002100002L});
1298        public static final BitSet FOLLOW_IDENT_in_goal452 =
1299                                    new BitSet(new long[]{0x0000000000000020L});
1300        public static final BitSet FOLLOW_IDENT_in_goal454 =
1301                                    new BitSet(new long[]{0x000000001C000000L});
1302        public static final BitSet FOLLOW_goal_key_in_goal456 =
1303                                    new BitSet(new long[]{0x0000000000000040L});
1304        public static final BitSet FOLLOW_WS_in_goal458 =
1305                                    new BitSet(new long[]{0x0000000000004000L});
1306        public static final BitSet FOLLOW_colon_in_goal460 =
1307                                    new BitSet(new long[]{0x0000000000000040L});
1308        public static final BitSet FOLLOW_WS_in_goal462 =
1309                                    new BitSet(new long[]{0x0000000000000080L});
1310        public static final BitSet FOLLOW_STRING_in_goal464 =
1311                                    new BitSet(new long[]{0x0000000000002000L});
1312        public static final BitSet FOLLOW_semi_in_goal466 =
1313                                    new BitSet(new long[]{0x0000000000000010L});
1314        public static final BitSet FOLLOW_NEWLINE_in_goal468 =
```

```
1315                                new BitSet(new long[]{0x0000000000000002L});
1316    public static final BitSet FOLLOW_set_in_goal_key474 =
1317                                new BitSet(new long[]{0x000000001C000002L});
1318    public static final BitSet FOLLOW_IDENT_in_plan495 =
1319                                new BitSet(new long[]{0x0000000000000020L});
1320    public static final BitSet FOLLOW_IDENT_in_plan497 =
1321                                new BitSet(new long[]{0x00000001E0008000L});
1322    public static final BitSet FOLLOW_plan_key_in_plan499 =
1323                                new BitSet(new long[]{0x0000000000000040L});
1324    public static final BitSet FOLLOW_WS_in_plan501 =
1325                                new BitSet(new long[]{0x0000000000004000L});
1326    public static final BitSet FOLLOW_colon_in_plan503 =
1327                                new BitSet(new long[]{0x0000000000000040L});
1328    public static final BitSet FOLLOW_WS_in_plan505 =
1329                                new BitSet(new long[]{0x0000000000000080L});
1330    public static final BitSet FOLLOW_STRING_in_plan507 =
1331                                new BitSet(new long[]{0x0000000000002000L});
1332    public static final BitSet FOLLOW_semi_in_plan509 =
1333                                new BitSet(new long[]{0x0000000000000010L});
1334    public static final BitSet FOLLOW_NEWLINE_in_plan511 =
1335                                new BitSet(new long[]{0x0000000000000002L});
1336    public static final BitSet FOLLOW_set_in_plan_key517 =
1337                                new BitSet(new long[]{0x00000001E0008002L});
1338 }
```

B.3 Example Grammar of an Action Block

```
grammar actions;
options output=template;

INDENT : '
t';
DELIM_L : '';
DELIM_R : '';
COLON : ':';
WS : ' ';
SEMI : ';';
STRING : ('0'..'9'|'a'..'z'|'A'..'Z')+;
NEWLINE : '
n';

action_block:
'ACTION' NEWLINE INDENT DELIM_L NEWLINE (action)+ INDENT DELIM_R NEWLINE;

//------------------------------------------------------------------
//Defintion of an action block

action : (INDENT INDENT act_key);

varname : STRING;
```

```
act_key : (act_id|domain|param|descr);

descr : precond|effect;

domain : 'DOMAIN' WS COLON WS STRING+ SEMI NEWLINE;
var     : 'VAR' WS varname WS type WS;

act_id : 'ACTION_NAME' WS COLON WS STRING SEMI NEWLINE;

type : ('Server'|'Router'|'Apache'|'Datebase'|'Firewall'|'WebServer'|'AppServer');
state : ('Running'|'Halt'|'Stop'|'Starting');

precond :
    'PRECONDITIONS' WS COLON WS ('('type WS COLON WS state')' WS)+ SEMI NEWLINE;
effect :
    'EFFECTS' WS COLON WS ('('type WS COLON WS state')'WS)+ SEMI NEWLINE;
param :
    'PARAMETERS' WS COLON WS (var SEMI)+ NEWLINE;
```

C Example Output of the Applications

C.1 Fault Recovery Planning in Cloud Infrastructure

The following figure shows the Web-based representation of the planning problem and the computed plan solution.

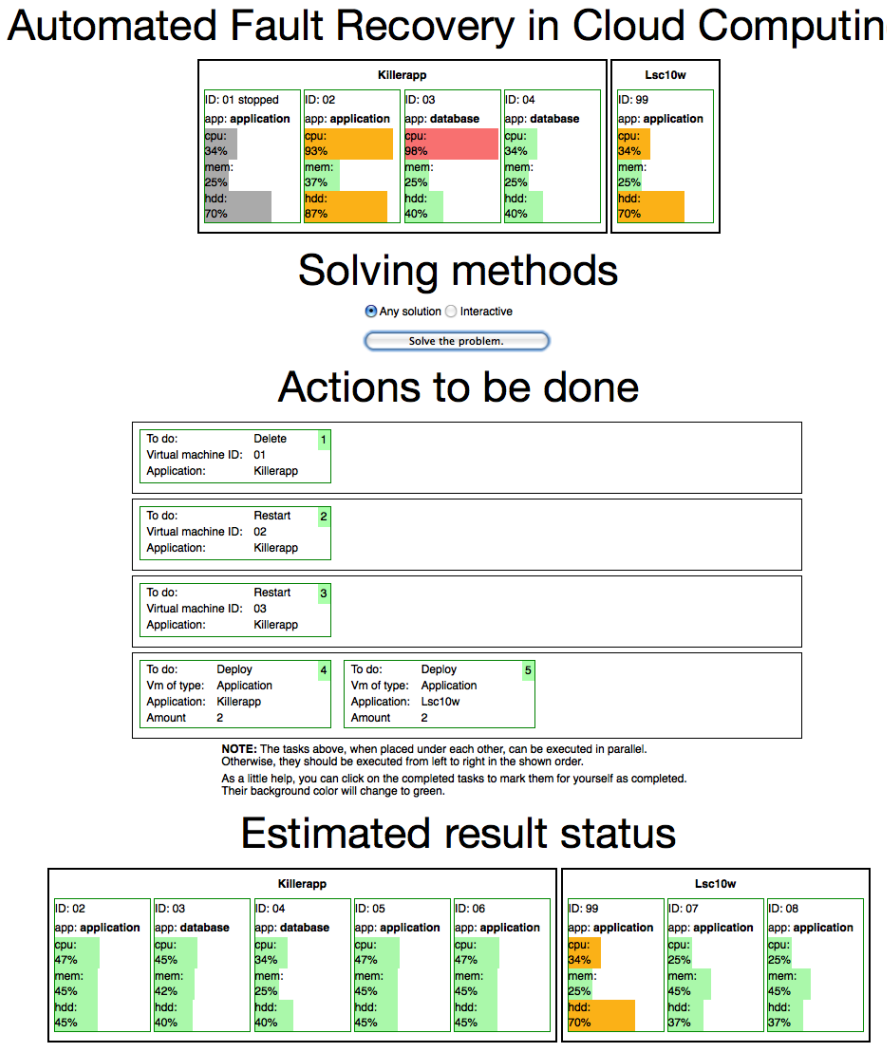


Figure C.1: Web-based representation of the recovery planning problem and its computed plan solution

C.2 Change Planner

Following is an example output of the change activity planner. The goal of the change plan is to refine the high-level task SpeedUpWebApplication into implementable change actions. In this application scenario, we want to speedup a Wiki web application container with load-balancing enabled. In the result computed by the planner,the J2EE container, which holds the Wiki web application, is replicated to a newly installed virtual server. Afterwards a load-balancer is installed and finally, the planner adds the replica of the web-application container to the load-balancer.

```
\Remaining task network:
  Preliminary plan:
  Remaining tasks: SpeedUpWebApplication,
Available REFINEMENT OPERATION:
[0] UpgradeDatabaseServerHardware
[1] UpgradeWebServerHardware
[2] MigrateDatabase
[3] EnableLoadbalancing
[4] UpgradeDatabaseServerAndWebServerHardware
Choose a REFINEMENT OPERATION: 3
Backtrack over remaining REFINEMENT OPERATION? (y/n)? n
Calculating bindings for task SpeedUpWebApplication and refinement
    EnableLoadbalancing
Available BINDING:
[0] J2eeApplication:application=8=Wiki Application
[1] ECommerceJ2eeApplication:application=11=ECommerce Application
Choose a BINDING: ? 0
Backtrack over remaining BINDING? (y/n)? n
Applying method EnableLoadbalancing to task SpeedUpWebApplication
Enter temporal deadline for sub task InstallLoadBalancer (-1 for no
    deadline): ? -1
Enter temporal deadline for sub task InstallJ2eeApplication (-1 for no
     deadline): ? -1
Enter temporal deadline for sub task AddContainerToLoadbalancer (-1
    for no deadline): ? -1
Enter temporal deadline for sub task InstallJ2eeServer (-1 for no
    deadline): ? -1
Remaining task network:
  Preliminary plan:
  Remaining tasks: InstallLoadBalancer,InstallJ2eeApplication,
      InstallJ2eeServer,AddContainerToLoadbalancer,
Available FIRST TASK:
[0] Task: InstallLoadBalancer
[1] Task: InstallJ2eeServer
Choose a FIRST TASK: ? 0
Available REFINEMENT OPERATION:
[0] InstallLoadBalancer-Script
[1] LoadbalancerOnServer
Choose a REFINEMENT OPERATION: ? 0
Backtrack over remaining REFINEMENT OPERATION? (y/n)? n
Calculating bindings for task InstallLoadBalancer and refinement
    InstallLoadBalancer-Script
Applying operator InstallLoadBalancer-Script to task
    InstallLoadBalancer
Remaining task network:
  Preliminary plan: InstallLoadBalancer-Script(lb <-- [LoadBalancer
      =16=nil])
```

```
39   Remaining tasks: InstallJ2eeApplication, InstallJ2eeServer,
         AddContainerToLoadbalancer,
40 Calculating bindings for task InstallJ2eeServer and refinement
       DefaultJ2eeServerInstallation
41 Applying method DefaultJ2eeServerInstallation to task
       InstallJ2eeServer
42 Enter temporal deadline for sub task SetupServer (-1 for no deadline):
        ? -1
43 Enter temporal deadline for sub task InstallJ2eeContainerSoftware (-1
       for no deadline): ? -1
44 Remaining task network:
45   Preliminary plan: InstallLoadBalancer-Script(lb <-- [LoadBalancer
         =16=nil])
46   Remaining tasks: SetupServer, InstallJ2eeApplication,
         InstallJ2eeContainerSoftware, AddContainerToLoadbalancer,
47 Available FIRST TASK:
48 [0] Task: SetupServer
49 [1] Task: InstallJ2eeApplication
50 Choose a FIRST TASK: ? 0
51 Available REFINEMENT OPERATION:
52 [0] UseExistingServer
53 [1] InstallVirtualServerByScript
54 Choose a REFINEMENT OPERATION: ? 1
55 Backtrack over remaining REFINEMENT OPERATION? (y/n)? n
56 Calculating bindings for task SetupServer and refinement
       InstallVirtualServerByScript
57 Applying operator InstallVirtualServerByScript to task SetupServer
58 Remaining task network:
59   Preliminary plan: InstallLoadBalancer-Script(lb <-- [LoadBalancer
         =16=nil]), InstallVirtualServerByScript(mem <-- [Integer=1000],
         server <-- [WebServer=17=nil])
60   Remaining tasks: InstallJ2eeContainerSoftware,
         AddContainerToLoadbalancer, InstallJ2eeApplication,
61 Available FIRST TASK:
62 [0] Task: InstallJ2eeContainerSoftware
63 [1] Task: InstallJ2eeApplication
64 Choose a FIRST TASK: ? 0
65 Calculating bindings for task InstallJ2eeContainerSoftware and
       refinement InstallJ2eeContainerSoftware-Script
66 Applying operator InstallJ2eeContainerSoftware-Script to task
       InstallJ2eeContainerSoftware
67 Remaining task network:
68   Preliminary plan: InstallLoadBalancer-Script(lb <-- [LoadBalancer
         =16=nil]), InstallVirtualServerByScript(mem <-- [Integer=1000],
         server <-- [WebServer=17=nil]), InstallJ2eeContainerSoftware-
         Script(webserver <-- [WebServer=17=nil], cont <-- [J2eeContainer
         =18=nil])
69   Remaining tasks: AddContainerToLoadbalancer, InstallJ2eeApplication,
70 Calculating bindings for task InstallJ2eeApplication and refinement
       InstallJ2eeApplication-Script
71 Applying operator InstallJ2eeApplication-Script to task
       InstallJ2eeApplication
72 Remaining task network:
73   Preliminary plan: InstallLoadBalancer-Script(lb <-- [LoadBalancer
         =16=nil]), InstallVirtualServerByScript(mem <-- [Integer=1000],
         server <-- [WebServer=17=nil]), InstallJ2eeContainerSoftware-
         Script(webserver <-- [WebServer=17=nil], cont <-- [J2eeContainer
         =18=nil]), InstallJ2eeApplication-Script(app <-- [J2eeApplication
         =8=Wiki Application], cont <-- [J2eeContainer=18=nil])
74   Remaining tasks: AddContainerToLoadbalancer,
75 Calculating bindings for task AddContainerToLoadbalancer and
       refinement AddContainerToLoadBalancer-Script
```

```
76 Applying operator AddContainerToLoadBalancer-Script to task
       AddContainerToLoadbalancer
77 Found new plan (1): InstallLoadBalancer-Script(lb <-- [LoadBalancer
       =16=nil]),InstallVirtualServerByScript(mem <-- [Integer=1000],
       server <-- [WebServer=17=nil]),InstallJ2eeContainerSoftware-Script
       (webserver <-- [WebServer=17=nil],cont <-- [J2eeContainer=18=nil])
       ,InstallJ2eeApplication-Script(app <-- [J2eeApplication=8=Wiki
       Application],cont <-- [J2eeContainer=18=nil]),
       AddContainerToLoadBalancer-Script(container <-- [J2eeContainer=18=
       nil],lb <-- [LoadBalancer=16=nil])
78 Backtracking
79 Backtracking
80 Backtracking
81 Backtracking
82 Backtracking
83 Backtracking
84 Backtracking
85
86 Found plans (total 1)
87 (0) InstallLoadBalancer-Script(lb <-- [LoadBalancer=16=nil]),
       InstallVirtualServerByScript(mem <-- [Integer=1000],server <-- [
       WebServer=17=nil]),InstallJ2eeContainerSoftware-Script(webserver
       <-- [WebServer=17=nil],cont <-- [J2eeContainer=18=nil]),
       InstallJ2eeApplication-Script(app <-- [J2eeApplication=8=Wiki
       Application],cont <-- [J2eeContainer=18=nil]),
       AddContainerToLoadBalancer-Script(container <-- [J2eeContainer=18=
       nil],lb <-- [LoadBalancer=16=nil])
88 Task InstallLoadBalancer: starts [0.0 , 35.0] end [10.0 , 45.0]
       duration [10.0 , 45.0]
89 Task InstallJ2eeServer: starts [0.0 , 15.0] end [20.0 , 35.0] duration
        [20.0 , 35.0]
90 Action InstallVirtualServerByScript: starts [0.0 , 15.0] end [10.0 ,
       25.0] duration [10.0 , 10.0]
91 Action InstallJ2eeContainerSoftware-Script: starts [10.0 , 25.0] end
       [20.0 , 35.0] duration [10.0 , 10.0]
92 Task InstallJ2eeApplication: starts [20.0 , 35.0] end [30.0 , 45.0]
       duration [10.0 , 25.0]
93 Task AddContainerToLoadbalancer: starts [30.0 , 45.0] end [35.0 ,
       50.0] duration [5.0 , 20.0]
94 Task SpeedUpWebApplication: starts [0.0 , 15.0] end [35.0 , 50.0]
       duration [35.0 , 50.0]
95 Task InstallJ2eeContainerSoftware: starts [10.0 , 25.0] end [20.0 ,
       35.0] duration [10.0 , 25.0]
96 Action InstallJ2eeApplication-Script: starts [20.0 , 35.0] end [30.0 ,
        45.0] duration [10.0 , 10.0]
97 Task SetupServer: starts [0.0 , 15.0] end [10.0 , 25.0] duration [10.0
        , 25.0]
98 Action AddContainerToLoadBalancer-Script: starts [30.0 , 45.0] end
       [35.0 , 50.0] duration [5.0 , 5.0]
99 Action InstallLoadBalancer-Script: starts [0.0 , 35.0] end [10.0 ,
       45.0] duration [10.0 , 10.0]
100
101 Total time consumed for HTN planning: 57796ms
102 Total time consumed for task network copying: 0ms (0%)
103 Total time consumed for knowledge base rollbacks: 735ms
104 Total time consumed for knowledge base assertions: 1172ms
105 Total time consumed for knowledge base queries: 281ms
106 Total time consumed for knowledge base operations: 2188ms
```

Bibliography

[AA96] J.R. Abrial and JR Abrial. *The B-book*, volume 146. Cambridge University Press Cambridge, UK, 1996.

[AA05] Akhil Sahai Arthur Andrzejak, Ulf Herman. Feedbackflow - an adaptive workflow generator for systems management, 2005.

[AAD+92] A. Abramovici, W.E. Althouse, R.W.P. Drever, Y. Gursel, S. Kawamura, F.J. Raab, D. Shoemaker, L. Sievers, R.E. Spero, K.S. Thorne, et al. LIGO: The laser interferometer gravitational-wave observatory. *Science*, 256(5055):325, 1992.

[AHA04] Wil Van Der Aalst, Kees Van Hee, and Wil Van Der Aalst. *Workflow Management: Models, Methods and Systems*. MIT Press, 2004.

[All83] J.F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[AMB+08] J.A. Allen, D. Mott, A. Bahrami, J. Yuan, C. Giammanco, and J. Patel. A Framework for Supporting Human Military Planning. In *Proceedings of the Second Annual Conference of the International Technology Alliance*, volume 1, 2008.

[AMF+08] A. Andrzejak, C. Mastroianni, P. Fragopoulou, D. Kondo, P. Malecot, A. Reinefeld, F. Schintke, T. Sch
"utt, G.C. Silaghi, L.M. Silva, et al. Grid Architectural Issues: State-of-the-art and Future Trends, 2008.

[AMP05] D.W. Aha, M. Molineaux, and M. Ponsen. Learning to win: Case-based plan selection in a real-time strategy game. *Lecture notes in computer science*, 3620:5, 2005.

[Ars06a] Naveed Arshad. *A Planning-Based Approach to Failure Recovery in Distributed Systems*. PhD thesis, University of Colorado at Boulder, 2006.

[Ars06b] Naveed Arshad. *A Planning-Based Approach to Failure Recovery in Distributed Systems*. PhD thesis, University of Coloardo at Boulder, USA, 2006.

[ARSS05] A. Andrzejak, A. Reinefeld, F. Schintke, and T. Schutt. On adaptability in grid systems. *Future Generation Grids, CoreGRID series. Springer-Verlag*, 2005.

[Bar10] Daniel Bartholomew. Sql vs. nosql. *Linux J.*, 2010, July 2010.

[BB05] B. Burns and O. Brock. Toward optimal configuration space sampling. In *Robotics: Science and Systems*, 2005.

[BBB+04] A. Boukhtouta, A. Bedrouni, J. Berger, F. Bouak, and A. Guitouni. A survey of military planning systems. In *The 9th ICCRTS Int. Command and Control Research and Technology Symposium*, 2004.

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10 20 states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[BCP+01] P. Bertoli, A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. MBP: a model based planner. In *Proc. of the IJCAI01901 Workshop on Planning under Uncertainty and Incomplete Information*, 2001.

[BDG+03] J. Blythe, E. Deelman, Y. Gil, C. Kesselman, A. Agarwal, G. Mehta, and K. Vahi. The role of planning in grid computing. In *Proceedings of 13th International Conference on Automated Planning and Scheduling (ICAPS)*, 2003.

[BDGK03] Jim Blythe, Ewa Deelman, Yolanda Gil, and Carl Kesselman. Transparent grid computing: a knowledge-based approach. In *15th Innovative Applications of Artificial Intelligence Conference*, 2003.

[Bel57] R. Bellman. A Markovian decision process. 1957.

[Bel03] Richard E. Bellman. *Dynamic Programming*. Dover Publications, 2003.

[BF97] A.L. Blum and M.L. Furst. Fast planning through planning graph analysis. *Artificial intelligence*, 90(1-2):281–300, 1997.

[BHL+04] M. Burstein, J. Hobbs, O. Lassila, D. Mcdermott, S. Mcilraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, et al. OWL-S: Semantic markup for web services. *W3C Member Submission*, 2004.

[BHM+04] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard. Web services architecture. *W3C Working Group Note*, 11:2005–1, 2004.

[BJMR05] J. Bresina, A. Jonsson, P. Morris, and K. Rajan. Activity planning for the mars exploration rovers. In *Fourteenth International Conference on Automated Planning and Scheduling*, pages 40–49, 2005.

[BK00] F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1-2):123–191, 2000.

[BKK02] A. Brown, G. Kar, and A. Keller. An active approach to characterizing dynamic dependencies for problem determination in a distributed environment. In *Integrated Network Management Proceedings, 2001 IEEE/IFIP International Symposium on*, pages 377–390. IEEE, 2002.

[Boa90] IEEE Standards Board. *IEEE 90: IEEE Standard Glossary of Software Engineering Terminology.* IEEE Std 610.12 - 1990. The Institute of Electrical and Electronics Engineers, 1990.

[Boo82] G. Booch. Object-oriented design. *ACM SIGAda Ada Letters*, 1(3):64–76, 1982.

[Bou02] C. Boutilier. A pomdp formulation of preference elicitation problems. In *Proceedings of the National Conference on Artificial Intelligence*, pages 239–246. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2002.

[BP01] A. Brown and D. Patterson. To err is human. In *Proceedings of the First Workshop on Evaluating and Architecting System dependabilitY (EASY '01)*, 2001.

[Bre01] Eric A. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 05(4):46–55, 2001.

[BS05] S. Kambhampati B. Srivastava. Challenges in adapting autoamted planning for autonomic computing. In *American Association for Artificial Intelligence*, 2005.

[BSP+02] J.P. Bigus, D.A. Schlosnagle, J.R. Pilgrim, W.N.M. III, and Y. Diao. ABLE: A toolkit for building multiagent autonomic systems. *IBM Systems Journal*, 41(3):350–371, 2002.

[BWE+94] Anthony Barrett, Daniel S. Weld, Oren Etzioni, Steve Hanks, James Hendler, Craig Knoblock, and Rao Kambhampati. Partial-order planning: Evaluating possible efficiency gains. *Artificial Intelligence*, 67:71–112, 1994.

[Byl91] T. Bylander. Complexity results for planning. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, volume 1, pages 274–279. Citeseer, 1991.

[CDG+08] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[CFR+07] H. Chan, A. Fern, S. Ray, N. Wilson, and C. Ventura. Online planning for resource production in real-time strategy games. In *Proc. of the In. Conference on Automated Planning and Scheduling*, 2007.

[CGGT97] R. Cimatti, E. Giunchiglia, F. Giunchiglia, and P. Traverso. Planning via model checking: A decision procedure for AR. *In Proceedings of ECP-97*, 1997.

[CGP99] E. M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*, chapter 2, page 314. MIT Press, 1999.

[CKL95] A.R. Cassandra, L.P. Kaelbling, and M.L. Littman. Acting optimally in partially observable stochastic domains. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1023–1023, 1995.

[CKR05] Benjamin Cheung, Gopal Kumar, and Sudarshan A. Rao. Statistical algorithms in fault detection and prediction: Toward a healthier network. *Bell Labs Technical Journal*, 9(4):171–185, 2005.

[CL85] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):75, 1985.

[CLRS01] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to algorithms*. The MIT press, 2001.

[CMAB05] M.T. Cox, H.É.C. MuÑOz-Avila, and R. Bergmann. Case-based planning. *The Knowledge Engineering Review*, 20(03):283–287, 2005.

[Com07] Office Of Government Commerce. *Information Technology Infrastructure Library V3 - Service Transition.* The Stationary Office, 2007.

[Cri91] F. Cristian. Understanding fault-tolerant distributed systems. 1991.

[CST03] M. Carman, L. Serafini, and P. Traverso. Web service composition as planning. In *proceedings of ICAPS03 International Conference on Automated Planning and Scheduling, Trento, Italy*, 2003.

[Dav02] Patterson David. A simple way to estimate the cost of downtime. USENIX 16th System Administrators Conference, 2002.

[DBGCK03] Ewa Deelman, James Blythe, Yolanda Gil, and Karan Vahi Carl Kesselman, Gaurang Mehta. Mapping abstract complex workflows onto grid environments. 1(1), March 2003.

[Dec03] R. Dechter. *Constraint processing.* Morgan Kaufmann, 2003.

[DGAV05] P. Doshi, R. Goodwin, R. Akkiraju, and K. Verma. Dynamic workflow composition: Using markov decision processes. *International Journal of Web Services Research*, 2(1):1–17, 2005.

[DKM+02] E. Deelman, C. Kesselman, G. Mehta, L. Meshkat, L. Pearlman, K. Blackburn, P. Ehrens, A. Lazzarini, R. Williams, and S. Koranda. GriPhyN and LIGO, building a virtual data grid for gravitational wave scientists. In *11th Intl Symposium on High Performance Distributed Computing*, volume 2002, 2002.

[DO02] David A. Patterson David Oppenheimer. Architecture and dependability of large-scale internet services. *IEEE Internet Computing*, 6(5):41–49, 2002.

[DO03] David A. Patterson David Oppenheimer, Archana Ganapathi. Why do internet services fail, and what can be done about it? In *Proceedings of USITS 03: 4th USENIX Symposium on Internet Technologies and Systems.* The USENIX Association, 2003.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Jounal of the ACM*, 7(3):201–215, 1960.

[EHN94] Kutluhan Erol, James Hendler, and Dana S. Nau. Umcp: A sound and complete procedure for hierarchical task-network planning. In *Proceedings of the 2nd International Conference on Artificial Intelligence Planning Systems (AIPS 94)*, pages 249–254, 1994.

[EHN95] K. Erol, J. Hendler, and D.S. Nau. HTN planning: Complexity and expressivity, 1995.

[EHN96] K. Erol, J. Hendler, and D.S. Nau. Complexity results for HTN planning. *Annals of Mathematics and Artificial Intelligence*, 18(1):69–93, 1996.

[EKK+06] T. Eilam, M.H. Kalantar, A.V. Konstantinou, G. Pacifici, J. Pershing, and A. Agrawal. Managing the configuration complexity of distributed applications in internet data centers. *Communications Magazine, IEEE*, 44(3):166–177, 2006.

[EMME+06] K. El Maghraoui, A. Meghranjani, T. Eilam, M. Kalantar, and A.V. Konstantinou. Model driven provisioning: Bridging the gap between declarative object models and procedural provisioning tools. In *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*, pages 404–423. Springer-Verlag New York, Inc., 2006.

[ENS95] K. Erol, D.S. Nau, and VS Subrahmanian. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence*, 76(1-2):75–88, 1995.

[FHT10] L. Field, J. Huang, and M. Tsai. GStat 2.0: Grid information system status monitoring. In *Journal of Physics: Conference Series*, volume 219, page 062045. IOP Publishing, 2010.

[Fis70] P.C. Fishburn. *Utility theory for decision making*. Storming Media, 1970.

[FJ03] J. Frank and A. Jónsson. Constraint-based attribute and interval planning. *Constraints*, 8(4):339–364, 2003.

[FL00] P. Frohlich and J. Link. Automated test case generation from dynamic models. *Lecture notes in computer science*, pages 472–492, 2000.

[FL02] M. Fox and D. Long. International planning competition, 2002.

[FLSDT10] J. Famaey, S. Latrea, J. Strassner, and F. De Turck. A hierarchical approach to autonomic network management. In *Network Operations and Management Symposium Workshops*

(NOMS Wksps), 2010 IEEE/IFIP, pages 225–232. IEEE, 2010.

[FN71] R. Fikes and N.J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3/4):189–208, 1971.

[FP02] Armando Fox and David Patterson. When does fast recovery trump hige reliability? In *Proceedings of the Second Workshop on Evaluating and Architecting System dependabilitY (EASY '02)*, San Jose, CA, USA, 2002.

[Gär99] F.C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys (CSUR)*, 31(1):1–26, 1999.

[GBJ+08] S. Gopisetty, E. Butler, S. Jaquet, M. Korupolu, TK Nayak, R. Routray, M. Seaman, A. Singh, C.H. Tan, S. Uttamchandani, et al. Automated planners for storage provisioning and disaster recovery. *IBM Journal of Research and Development*, 52(4):353–365, 2008.

[GC03] A.G. Ganek and T.A. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1):5–18, 2003.

[GDB+04] Y. Gil, E. Deelman, J. Blythe, C. Kesselman, and H. Tangmunarunkit. Artificial intelligence and grids: Workflow planning and beyond. *IEEE Intelligent Systems*, pages 26–33, 2004.

[GHH+02] M. Garschammer, R. Hauck, H.-G Hegering, B. Kempter, M. Langer, I. Radisic M. Nerb, H. Roeller, and H. Schmidt. Towards generic service management concepts - a service model based approach. In *The Seventh IFIP/IEEE International Symposium on Integrated Network Management*, 2002.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*, volume 206. Addison-wesley Reading, MA, 1995.

[GHK+98] M. Ghallab, A. Howe, C. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL– the planning domain definition language. *AIPS-98 planning committee*, 1998.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and intractability: A Guide to the theory of NP-Completeness*. Freeman San Francisco, 1979.

[GN92] N. Guptay and D.S. Nauz. On the complexity of blocks-world planning. *Artifical Intelligence*, 56(2-3):223–254, 1992.

[Gon08] Iker Gondra. Applying machine learning to software fault-proneness prediction. *Journal on System Software*, 81(2):186–195, 2008.

[GRD+07] Y. Gil, V. Ratnakar, E. Deelman, G. Mehta, and J. Kim. Wings for pegasus: Creating large-scale scientific applications using semantic representations of computational workflows. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, volume 22, page 1767. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2007.

[GS02] A. Gerevini and I. Serina. LPG: A planner based on local search for planning graphs with action costs. In *Proc. of the Sixth Int. Conf. on AI Planning and Scheduling*, pages 12–22, 2002.

[HAN99] Heinz-Gerd Hegering, Sebastian Abeck, and Bernhard Neumair. *Integrated Management of Networked Systems*. The Morgan Kaufmann Series in Networking. Morgan Kaufmann Publisher, 1999.

[Han07] Andreas Hanemann. *Automated IT Service Fault Diagnosis Based on Event Correlation Techniques*. PhD thesis, Ludwig-Maximilians-University Munich, 2007.

[Has01] P. Hasselmeyer. Managing dynamic service dependencies. In *12th International Workshop on Distributed Systems: Operations & Management (DSOM 2001), Nancy, France*, pages 141–150, 2001.

[Her02] George W. Herbert. Failure from the field: Complexity kills. In *Proceedings of the Second Workshop on Evaluating and Architecting System Dependability*, 2002.

[HJ00] P. Haslum and P. Jonsson. Some results on the complexity of planning with incomplete information. *Recent Advances in AI Planning*, pages 308–318, 2000.

[HZ07] E.A. Hansen and R. Zhou. Anytime heuristic search. *Journal of Artificial Intelligence Research*, 28(1):267–297, 2007.

[JDB] Oracle Corporation JDBC. The java database connectivity. `http://www.oracle.com/technetwork/java/overview-141217.html`.

[JJCD] Open Source Project JCOUCHDB Java5 CouchDB Driver. Java5 couchdb driver. `http://code.google.com/p/couchdb4j/`.

[KC03] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–52, 2003.

[Kep05] Jeffrey O. Kephart. Research challenges of autonomic computing. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 15–22, New York, NY, USA, 2005. ACM.

[KHW+04] A. Keller, J. Hellerstein, JL Wolf, K. Wu, and V. Krishnan. The CHAMPS system: Change management with planning and scheduling. In *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS 2004)*, 2004.

[KLC95] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Partially observable markov decision processes for artificial intelligence. In *KI*, pages 1–17, 1995.

[Kno91] C.A. Knoblock. Search reduction in hierarchical problem solving. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 686–691. Citeseer, 1991.

[Kor87] R.E. Korf. Planning as search: A quantitative approach. *Artificial Intelligence*, 33(1):65–88, 1987.

[KS85] H. Kautz and B. Selman. BLACKBOX: A new approach to the application of theorem proving to problem solving. In *In Workshop on Planning as Combinatorial Search, in conjunction with AIPS-98 (Conference on Artificial Intelligence Planning Systems*, 1985.

[Kuh97] D. Richard Kuhn. Source of failure in the public switched telephone network. *Computer*, 30(4):31–36, 1997.

[LASU06] M. LAM, A. AHO, R. SETHI, and J. ULLMAN. *Compilers, principles, techniques, and tools*. Addison-Wesley, 2006.

[Lat91] J.C. Latombe. *Robot motion planning*. Springer, 1991.

[LaV07] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, 2007.

[LDK10] F. Liu, V. Danciu, and P. Kerestey. A framework for automated fault recovery planning in large-scale virtualized infrastructures. *Modelling Autonomic Communication Environments*, pages 113–123, 2010.

[LGM98] M.L. Littman, J. Goldsmith, and M. Mundhenk. The computational complexity of probabilistic planning. *Journal of Artificial Intelligence Research*, 9(1):36, 1998.

[LGT04] M. Likhachev, G. Gordon, and S. Thrun. ARA*: Anytime A* with provable bounds on sub-optimality. *Advances in Neural Information Processing Systems*, 16, 2004.

[LL06] Yawei Li and Zhiling Lan. Exploit failure prediction for adaptive fault-tolerance in cluster computing. In *CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, pages 531–538, Washington, DC, USA, 2006. IEEE Computer Society.

[LM10] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[LR04] Jean-Claude Laprie and Brian Randell. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, 2004.

[LR09] K. Levanti and A. Ranganathan. Planning-based configuration and management of distributed systems. In *Integrated Network Management, 2009. IM '09. IFIP/IEEE International Symposium on*, pages 65–72, June 2009.

[LT04] L. Laibinis and E. Troubitsyna. Fault tolerance in a layered architecture: a general specification pattern in B. 2004.

[MAABN99] H. Muñoz-Avila, D.W. Aha, L. Breslow, and D. Nau. HICAP: An interactive case-based planning architecture and its application to noncombatant evacuation operations. In *PROC NATL CONF ARTIF INTELL.*, pages 870–875, 1999.

[MCBS03] R. Medeiros, W. Cirne, F. Brasileiro, and J. Sauve. Faults in Grids: Why are they so bad and What can be done about it? In *Grid Computing, 2003. Proceedings. Fourth International Workshop on*, pages 18–24, 2003.

[McD02] D. McDermott. Estimated-regression planning for interactions with web services. In *Proceeding of AIPS*, 2002.

[Mci94] S.A. Mcilraith. Towards a theory of diagnosis, testing and repair. In *In Proceedings of The Fifth International Workshop on Principles of Diagnosis*, 1994.

[MH86] R. Mohr and T.C. Henderson. Research Note Arc and Path Consistency Revisited. *Artificial intelligence*, 28:225–233, 1986.

[Min61] M. Minsky. Steps toward artificial intelligence. *Proceedings of the IRE*, 49(1):8–30, 1961.

[MMM04] F. Manola, E. Miller, and B. McBride. RDF primer. *W3C recommendation*, 10, 2004.

[MU08] F. Marquardt and A.M. Uhrmacher. Evaluating AI planning for service composition in smart environments. In *Proceedings of the 7th International Conference on Mobile and Ubiquitous Multimedia*, pages 48–55. ACM New York, NY, USA, 2008.

[MVH+04] D.L. McGuinness, F. Van Harmelen, et al. OWL web ontology language overview. *W3C recommendation*, 10:2004–03, 2004.

[NA05] Alexander L. Wolf Naveed Arshad, Dennis Heimbigner. Dealing with failures during failure recovery of distributed systems. In *Design and Evolution of Autonomic Application Software*, 2005.

[NA07] Alexander L. Wolf Naveed Arshad, Dennis Heimbigner. Deployment and dynamic reconfiguration planning for distributed software systems. *Software Quality Journal*, 15(3), 2007.

[NAI+03] D. Nau, T.C. Au, O. Ilghami, U. Kuter, J.W. Murdock, D. Wu, and F. Yaman. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research*, 20(1):379–404, 2003.

[NAI+05] D. Nau, T.C. Au, O. Ilghami, U. Kuter, D. Wu, F. Yaman, H. Muñoz-Avila, and J.W. Murdock. Applications of SHOP and SHOP2. *Intelligent Systems, IEEE*, 20(2):34–41, 2005.

[Nau95] D.S. Nau. AI planning versus manufacturing-operation planning: A case study. In *In Proceedings of the 14 th International Joint Conference on Artificial Intelligence*, 1995.

[Nau07] D.S. Nau. Current trends in automated planning. *AI magazine*, 28(4):43, 2007.

[NFF+05] A. Nareyek, E.C. Freuder, R. Fourer, E. Giunchiglia, R.P. Goldman, H. Kautz, J. Rintanen, and A. Tate. Constraints and AI planning. *IEEE Intelligent Systems*, 20(2):62–72, 2005.

[NGT04] Dana Nau, Malik Ghallab, and Paolo Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., 2004.

[NM02] S. Narayanan and S.A. McIlraith. Simulation, verification and automated composition of web services. In *Proceedings of the 11th international conference on World Wide Web*, pages 77–88. ACM New York, NY, USA, 2002.

[NPV+05] J. Norris, M. Powell, M. Vona, P. Backes, and J.V. Wick. Mars exploration rover operations with the science activity planner. In *IEEE International Conference on Robotics and Automation*, volume 4, page 4618, 2005.

[NSE+98] D.S. Nau, S.J.J. Smith, K. Erol, et al. Control strategies in HTN planning: Theory versus practice. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1127–1133. JOHN WILEY & SONS LTD, 1998.

[Par04] T.J. Parr. Enforcing strict model-view separation in template engines. In *Proceedings of the 13th international conference on World Wide Web*, pages 224–233. ACM, 2004.

[Par09] Terence Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. Pragmatic Bookshelf Series. Pragmatic Bookshelf, 2009.

[PB01] P. Poupart and C. Boutilier. Vector-space analysis of belief-state approximation for pomdps. In *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence*, pages 445–452, 2001.

[PBA+07] T. Patkos, A. Bikakis, G. Antoniou, M. Papadopouli, and D. Plexousakis. Distributed AI for Ambient Intelligence: Issues and Approaches. *Lecture Notes in Computer Science*, 4794:159, 2007.

[Ped87] E. Pednault. Formulating multi agent dynamic world problems in the classical planning framework. In M.P. Georgeff and A. Lansky, editors, *Reasoning about Actions and Plans*, pages 42–82. Morgan Kaufmann, 1987.

[PTD+08] M.P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, and B.J. Kr
"amer. Service-oriented computing: A research roadmap. *International Journal of Cooperative Information Systems*, 17(2):223–255, 2008.

[Put94] M.L. Puterman. *Markov decision processes: Discrete stochastic dynamic programming*. John Wiley & Sons, Inc. New York, NY, USA, 1994.

[PW92] J.S. Penberthy and D. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *proceedings of the third international conference on knowledge representation and reasoning*, pages 103–114, 1992.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, et al. *Object-oriented modeling and design*, volume 38. Prentice hall Englewood Cliffs (NJ), 1991.

[RC04] A. Ranganathan and R.H. Campbell. Autonomic pervasive computing based on planning. In *Proceedings of the First International Conference on Autonomic Computing (ICAC'04)*, pages 80–87, 2004.

[RDF05] W. Ruml, M.B. Do, and M. Fromherz. On-line planning and scheduling for high-speed manufacturing. In *Proc. of ICAPS*, volume 5, pages 30–39, 2005.

[RL05] A. Riabov and Z. Liu. Planning for stream processing systems. In *Proceedings of the National Conference on Artificial Intelligence*, volume 20, page 1205. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2005.

[RL06] A. Riabov and Z. Liu. Scalable planning for distributed stream processing systems. *Proceedings of International Conference on Automated Planning and Scheduling 2006*, 2006.

[Rod02] Gabrijela Dreo Rodosek. *A Framework for IT Service Management*. Ludwig-Maximilians-Uniersität München, 2002. Habilitationsschrift im Fach Informatik.

[RPPCD08] S. Ross, J. Pineau, S. Paquet, and B. Chaib-Draa. Online planning algorithms for POMDPs. *Journal of Artificial Intelligence Research*, 32(1):663–704, 2008.

[RRD04] Stefanie Rinderle, Manfred Reichert, and Peter Dadam. Correctness criteria for dynamic changes in workflow systems: a survey. *Data Knowl. Eng.*, 50(1):9–34, 2004.

[Sai07] Martin Sailer. *Konzeption einer Service-MIB – Analyse und Spezifikation dienstorientierter Managementinformation*. PhD thesis, Ludwig-Maximilians-University Munich, 2007.

[SBM10] S. Sohrabi, J.A. Baier, and S.A. McIlraith. Diagnosis as Planning Revisited. 2010.

[SBS04] B. Srivastava, J.P. Bigus, and D.A. Schlosnagle. Bringing planning to autonomic applications with ABLE. In *Proceedings of the First International Conference on Autonomic Computing*, pages 154–161. IEEE Computer Society, 2004.

[SC75] E.D. Sacerdoti and A.I. Center. The Non-Linear Nature of Plans. In *Advance papers of the fourth International Joint Conference on Artificial Intelligence: Tbilisi, Georgia, USSR, 3-8 September 1975*, page 206. Morgan Kaufmann, 1975.

[Sch01] S. Schneider. *The B-method: an introduction*. Palgrave Chippenham Wiltshire, 2001.

[Sch06] Klaus Schmidt. *High Availability and Disaster Recovery – Concept, Design, Implementation*. Springer, 2006.

[Sch08] David Schmitz. *Automated Service-Oriented Impact Analysis and Recovery Alternative Selection*. PhD thesis, Ludwig-Maximilians-University Munich, 2008.

[SG06] B. Schroeder and GA Gibson. A large-scale study of failures in high-performance computing systems. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pages 249–258, 2006.

[Ski08] S.S. Skiena. *The algorithm design manual*. Springer, 2008.

[SPW+04] E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. Nau. HTN planning for web service composition using SHOP2. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(4):377–396, 2004.

[SR02] Peter Norvig Stuart Russel. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2002.

[Sri04] B. Srivastava. A software framework for applying planning techniques. *Proc. Knowledge Based Computer Systems, Hyderabad*, 2004.

[SSD77] E.D. Sacerdoti, SRI International. Artificial Intelligence Center. Computer Science, and Technology Division. *A structure for plans and behavior*. Elsevier New York, 1977.

[ST01] J. Slaney and S. Thiébaux. Blocks world revisited. *Artificial Intelligence*, 125(1-2):119–153, 2001.

[Str00] I. Streinu. A combinatorial approach to planar non-colliding robot arm motion planning. In *ANNUAL SYMPOSIUM ON FOUNDATIONS OF COMPUTER SCIENCE*, volume 41, pages 443–453, 2000.

[Sus75] Gerald J. Sussman. A computer model of skill acquisition. 1975.

[TEHN96] R. Tsuneto, K. Erol, J. Hendler, and D. Nau. Commitment strategies in hierarchical task network planning. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, pages 536–542, 1996.

[TFL09] D. Trastour, R. Fink, and F. Liu. ChangeRefinery: Assisted Refinement of High-Level IT Change Requests. In *Proceedings of the 2009 IEEE International Symposium on Policies for Distributed Systems and Networks-Volume 00*, pages 68–75. IEEE Computer Society, 2009.

[TK00] W. Tan and B. Khoshnevis. Integration of process planning and scheduling review. *Journal of Intelligent Manufacturing*, 11(1):51–63, 2000.

[TLC+09] H.M. Tran, C. Lange, G. Chulkov, J. Schnwlder, and M. Kohlhase. Applying semantic techniques to search and analyze bug tracking data. *Journal of Network and Systems Management*, 17(3), 2009.

[TS07] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems - Principles and Paradigms*. Pearson Prentice Hall, 2007.

[VCP+95] M. Veloso, J. Carbonell, A. Perez, D. Borrajo, E. Fink, and J. Blythe. Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental & Theoretical Artificial Intelligence*, 7(1):81–120, 1995.

[Vil82] M.B. Vilain. A system for reasoning about time. In *Proc. AAAI*, volume 82, pages 197–201, 1982.

[VK86] M. Vilain and H. Kautz. Constraint propagation algorithms for temporal reasoning. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 377–382, 1986.

[VR05] M. Vukovic and P. Robinson. SHOP2 and TLPlan for proactive service composition. In *UK-Russia Workshop on Proactive Computing*, 2005.

[WAS98] D.S. Weld, C.R. Anderson, and D.E. Smith. Extending Graphplan to handle uncertainty & sensing actions. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, pages 897–904. JOHN WILEY & SONS LTD, 1998.

[XSTK04] H. Xiong, M. Steinbach, P.N. Tan, and V. Kumar. HICAP: Hierarchial clustering with pattern preservation. In *Proceedings of the 4th SIAM International Conference on Data Mining*, pages 279–290, 2004.

[YC94] Q. Yang and A. Chan. Delaying variable binding committments in planning. In *Proc. 2nd Intl. Conf. AI Planning Systems*, pages 182–187, 1994.

[YL04] H.L.S. Younes and M.L. Littman. PPDDL1. 0: The language for the probabilistic part of IPC-4. In *Proc. International Planning Competition*, 2004.

[YRB+04] R.M. Young, M.O. Riedl, M. Branly, A. Jhala, RJ Martin, and CJ Saretto. An architecture for integrating plan-based behavior generation with interactive game environments. *Journal of Game Development*, 1(1):51–70, 2004.

[ZMN05] F. Zhu, M.W. Mutka, and L.M. Ni. Service discovery in pervasive computing environments. *Pervasive Computing, IEEE*, 4(4):81–90, 2005.