INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Diplomarbeit

Modelling and Assisting the Design of IT Changes

Robert Fink

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Diplomarbeit

Modelling and Assisting the Design of IT Changes

Robert Fink

Aufgabensteller:	Prof. Dr. Heinz-Gerd Hegering
Betreuer:	Feng Liu David Trastour (Hewlett Packard Laboratories, Bristol)
Abgabetermin:	10. November 2008

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 7. November 2008

. . . . . .
(Unterschrift des Kandidaten)

Zusammenfassung

Die IT Infrastructure Library (ITIL) beschreibt Richtlinien, Prozesse und Empfehlungen für IT Service Management (ITSM). Der Change Management Prozess ist eine Empfehlung für die Behandlung von Änderungen an Bestandteilen der IT Infrastruktur und stellt sicher, dass Änderungsanträge (*Request for Change*, RFC) priorisiert und autorisiert werden, dass die Änderungen und ihre technischen und den Geschäftsbetrieb betreffenden Auswirkungen verstanden werden, sowie dass die für die Implementierung notwendingen Ressourcen verfügbar sind. Von Business-Usern in Textform eingereichte Änderungsanträge müssen dabei zunächst in Bezug auf gewünschte Funktionalität, die tatsächliche IT-Infrastruktur, sonstige Richtlinien und zeitliche Vorgaben untersucht werden, um dann in entsprechende konkrete Abfolgen von spezifischen Änderungs-Operationen übersetzt zu werden. Immer komplexere IT-Abteilungen sind hierbei eine zunehmend größere Herausforderung für die effektive und effiziente Bearbeitung einer stetig steigenden Anzahl von Änderungsanträgen: RFCs können in sich unklar oder fehlerhaft sein, die Einhaltung von Unternehmsrichtlinien und Best-Practices ist schwierig zu überprüfen und die manuelle Erstellung von Änderungsplänen ist zeitaufwendig und fehleranfällig.

Diese Arbeit untersucht einen auf automatischem Planen (*automated planning*) basierenden Ansatz für den computer-unterstützten Entwurf von Änderungsplänen. Wir zeigen, wie Best-Practices und sonstiges Implementierungswissen (*change recipes*) systematisiert gespeichert und zum Verfeinern von Änderungsanträgen in konkrete Implementierungspläne eingesetzt werden können. Die vorgestellte Architektur basiert auf einem HTN-Algorithmus (*Hierarchical Task Network*), welcher Benutzern assistiert, diverse Informationsquellen in konsistente Implementierungspläne zusammenzuführen. Statt auf die übliche, Logik-basierte Spezifikation der Planungs-Welt (*planning domain*) zurückzugreifen, realisieren wir die Integration eines objekt-orientierten Datenmodells für die IT Infrastruktur in den HTN-Formalismus. Ein Prototyp des Frameworks und des modifizierten HTN-Algorithmus demonstriert die Machbarkeit unseres Ansatzes.

Abstract

The IT Infrastructure Library (ITIL) is a set of best practices that are widely accepted for IT service management (ISTM). Change management is a core ITIL process that oversees the handling of IT changes and ensures that all change requests are carefully prioritised and authorised, that business and technical impacts are understood, and that required resources are available. During this process, IT operations teams first need to understand the change requests that are generated by business and IT personnel. They must then develop and execute concrete IT change plans for each request. The increasingly large and complex IT environment (people, technology and processes) presents a number of challenges to the efficient and effective design of the ever higher volume of IT changes: Change requests can be ill-defined, company policies and best practices are not systematically captured and enforced, manually designing changes is time consuming and error-prone. To overcome these issues we propose in this thesis an automated planning based approach to change design. We illustrate how change knowledge can be represented to encode best practices and how to refine high-level change requests into concrete plans.

The proposed architecture features in its core a Hierarchical Task Network (HTN) planner that assists the user in integrating information from various sources into consistent change plans. Instead of relying on the commonly used logic language formulation for the underlying planning domain, the presented approach introduces novel concepts to bring together the prevailing object-oriented modelling paradigms of Configuration Management Systems (CMS) with the logic programming based methods of automated planning algorithms. A prototypical implementation of the framework and the planning algorithm shows the feasibility of the approach.

Acknowledgements

To my supervisors David and Feng, for your continuous support and feedback, and fruitful and interesting discussions. David, thanks a lot for helping me out with your incredibly keen perception and breadth of knowledge, it was a great pleasure working with you!

To Andrew, Marianne, Eve, Stuart, Alan, Shane, Matt, and probably many others, I thank you for answering all my desperately ignorant questions on the English language.

To Benedikt, Weverton, Konstantin, Andrew B., Andrew F., Johannes, and Matthias for their valuable feedback on this work.

To all my colleagues at HP, especially Weverton, Guilherme, Andrew F., Andrew B., and Abdel for many interesting discussions and coffee or milky tea breaks.

To all the enthusiastic football players at HP and Easton, I had a great time, thank you! (My weaknesses are unpardonable. I will train my right foot, promised!)

Bristol, November 2008

Contents

1. Introduction and motivation

In response to the increasing size and complexity of IT organisations and systems, many companies started IT Service Management (ITSM) programs to increase the efficiency and effectiveness of IT service delivery and support. In this context, the IT Infrastructure Library (ITIL) [1] is a set of best practices for ITSM that defines common vocabularies and processes, and covers a wide spectrum of IT concerns, including service strategy, service design, service transition, service operation and continual service improvement. ITIL is becoming more and more widely adopted by IT organisations and supported by IT management software vendors [2].

In order to adequately support businesses that evolve at an accelerating pace, IT organisations, systems and processes must constantly be adjusted: IT must introduce new business services, and enhance or modify existing ones. Technology considerations are also a source of IT changes, for instance by adopting a new technology that is more cost efficient or by performing required maintenance on a hardware or software component. For these reasons, IT organisations must cope with an increasingly large number of changes. To ensure an efficient and prompt handling of all IT changes, ITIL recommends implementing a Change Management process to funnel all Requests for Change (RFCs) to a Change Advisory Board (CAB). The change management process ensures that RFCs are carefully evaluated, prioritised and authorised, that their business and technical impacts are understood, and that they are scheduled in order to meet human and technical resource requirements.

While implementing ITIL change management goes some way towards a more effective handling of IT change, a high volume of changes can still put a strain on IT operations teams. We have observed in some HP customer sites volumes of more than one thousand change requests per day. Handling such volumes of changes can become problematic, and without automation, decision-support and knowledge management tools, bottlenecks and inefficiencies are almost inevitable in the change management process.

In this thesis, we concentrate on the change design step, in which one or several IT practitioners develop a detailed plan of action to deploy the change onto the IT infrastructure. To the best of our knowledge, designing IT changes is still a manual process in practice, and no tool support exists that can assist an IT practitioner in deriving a detailed plan of action from the textual description provided in the RFC.

We have identified several shortcomings in this current situation. First, there can be an information impedance mismatch (i.e. lack of common vocabulary) between change requesters and IT practitioners. In a survey conducted to identify the main issues faced by change managers [3], the problem of ill-defined RFC was ranked among the three most important challenges. Indeed, RFC can be raised by technicians, for instance for the remediation of a security risk, but also by business users, for instance to change the behaviour of a business

service. While technical requesters may have enough knowledge and technical vocabulary to describe the RFC to the appropriate level of detail, this is often not the case for a business requester. The information provided by the business user tends to be incomplete and/or unclear for the IT technician. The second problem we have recognised is that best practices and IT policies are usually stored in unstructured formats (documents, web pages, presentations, or emails) and there is no systematic way of capturing this knowledge and making sure it is reused. The third problem we have identified is that manually designing changes is a time consuming activity that puts of lot of strain on the IT operations teams; this is exacerbated if the design process requires multiple iterations between the requester and the IT practitioners to clarify the meaning of the change request. Finally, manually designing change plans is error-prone since the validation needs to be done manually.

Although several aspects of change management have been addressed in the literature (see Section 1.4), much still needs to be done to address these four problems in IT change design. To the best of our knowledge, ChangeLedge [4] is the only work that explores this field, but, as we will see below, it does not address the first two challenges.

In this thesis, we present a solution for assisting the refinement of high-level change requests into concrete actionable change workflows. Our solution introduces the notion of a change catalogue which provides a clear interface between change requesters and IT practitioners and reduces the need to rely on textual descriptions. Best practices are captured and reused in the decomposition of change requests into change workflows. Finally, corporate IT policies can be modelled and enforced in change designs.

The main deliverables of this thesis are:

1. Development of a conceptual framework for IT change plan design, taking into account various involved information sources.

2. Investigation of the applicability of automated planning methods to IT change plan design and adoption of the Hierarchical Task Network (HTN) planning paradigm to object-oriented planning domains.

3. Evaluation of the aforementioned framework through application of our ChangeRefinery proof-of-concept implementation to the refinement of several exemplary high-level change tasks.

1.1. Structure of this thesis

This thesis is organised as follows. Section 1.2 embeds this work in the field of IT Service Management, and explains the typical challenges in IT change management that we try to solve. We give a rough overview and possible use cases of the proposed architecture for assisted change design in Section 1.3. Section 1.4 relates this thesis to existing work in ITSM and applications of automated planning techniques. Chapter 2 details the information sources we have identified, introduces the concept of a change catalogue to present to change requesters, and describes the information models to encode change knowledge. Chapter 3 presents the architecture of our proposed framework for assisted change design and details

the algorithms used in the refinement of high-level change tasks. Chapter 4 describes details and experimental results of the basic functionality of our prototypical implementation, ChangeRefinery, and is complemented by the presentation of advanced ChangeRefinery features and partially implemented or sketched ideas in Chapter 5. Finally, Chapter 6 names future work intentions and possible extensions to the framework and ChangeRefinery. Appendix A gives detailed and formal definitions of automated planning techniques which are illustrated by several examples, and Appendix B clarifies faulty definitions in the HTN characterisation from [5]. Appendix C contains listings and UML diagrams.

1.2. Context

1.2.1. IT service management and ITIL

IT service management is a discipline for managing information technology systems with emphasis on business and customers' perspectives. It aims to provide process driven frameworks to structure IT-related activities and interactions both within IT departments and between IT and external parties such as users or business customers. ITSM clearly focusses on the operational and management aspects of running IT systems while not dealing with software and hardware development or technical details and implementations.

Being one of the most commonly used ITSM frameworks, the IT Infrastructure Library [1] gives detailed but generic best practice inspired descriptions of numerous ITSM related tasks published in five integrated books that align with typical IT service life cycles (Figure 1.1): Service Strategy, Service Design, Service Transition, Service Operation, and Continual Service Improvement.

1.2.2. IT change management

Within the ITIL framework, the Service Transition publication [6] "provides guidance for the development and improvement of capabilities for transitioning new and changed services into operations", which includes as a major task the management of IT changes. In this context, the term *change* comprises any modification to entities (called *Configuration Items* or *CI*) involved in the operation of IT services. Consequently, change management covers minor tuning, such as granting access rights or rebooting devices, as well as high level tasks, such as design and implementation of new services due to emerging business needs.

The change management process includes several activities (Figure 1.2): In the initiation and review activities, a Request for Change (RFC) document describing the required change is created and reviewed. The change is then evaluated and the impact on IT infrastructure, IT services, and other affected entities is assessed. After the change is authorised, the required actions to perform the change are planned, implemented, and tested. Finally, the change is deployed, reviewed, and closed. As indicated in Figure 1.2 and explicitly stated in [6], all activities comprising the change management process should be executed in close conjunction with the configuration management, for which ITIL recommends the use of Configuration Management Systems (CMS) and Configuration Management Databases

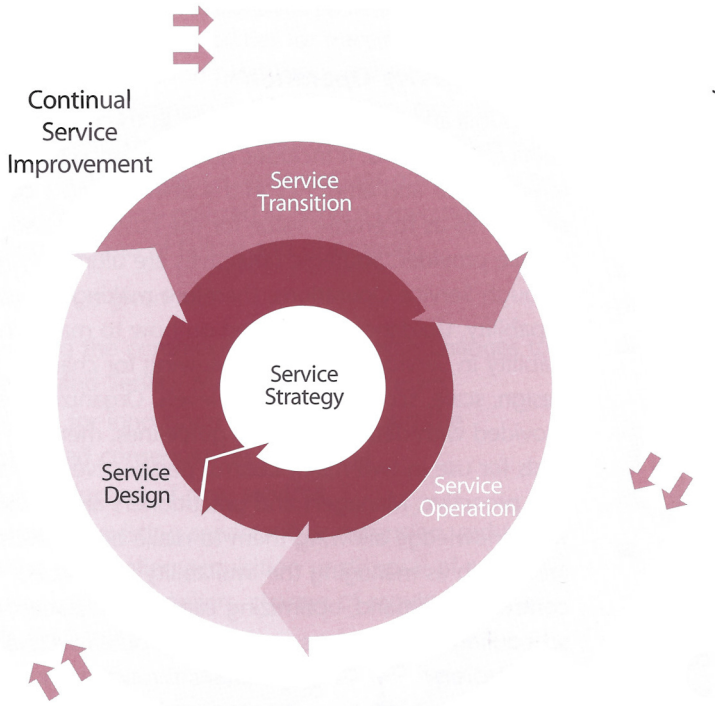


Figure 1.1: ITIL Core [6].

(CMDB) to manage information about all configuration items involved in ITSM processes. Significant or major changes go through the Change Advisory Board (CAB) for analysis and approval. The CAB is a group of people capable of analysing changes from a technical as well as from a business point of view. All aspects of the change management process are overseen by a change manager.

This work focusses on providing assistance to the change design and change planning activity in the change management process. Depending on the nature of a specific change, the planning task can be trivial as well as extremely complicated. Small changes, such as tweaking a configuration parameter, might even require no planning at all. A reasonable way of classifying changes is to group them by origination or demand; one may roughly differentiate between the following three types of changes:

- Changes due to **operational demand** are often short-term changes, usually emerging through user request. They have a very limited scope in terms of required resources and affected systems.

  Examples are minor bug fixing in response to incidents, performance tuning by adjusting application parameters, or access control.

- Changes due to **application demand** are medium-term changes with impact on existing services and typically come up as a consequence of changing user requirements or management decisions. The amount of required resources (personal, hardware, time, etc.) is considerably bigger than for operational-demand changes. Implementing the

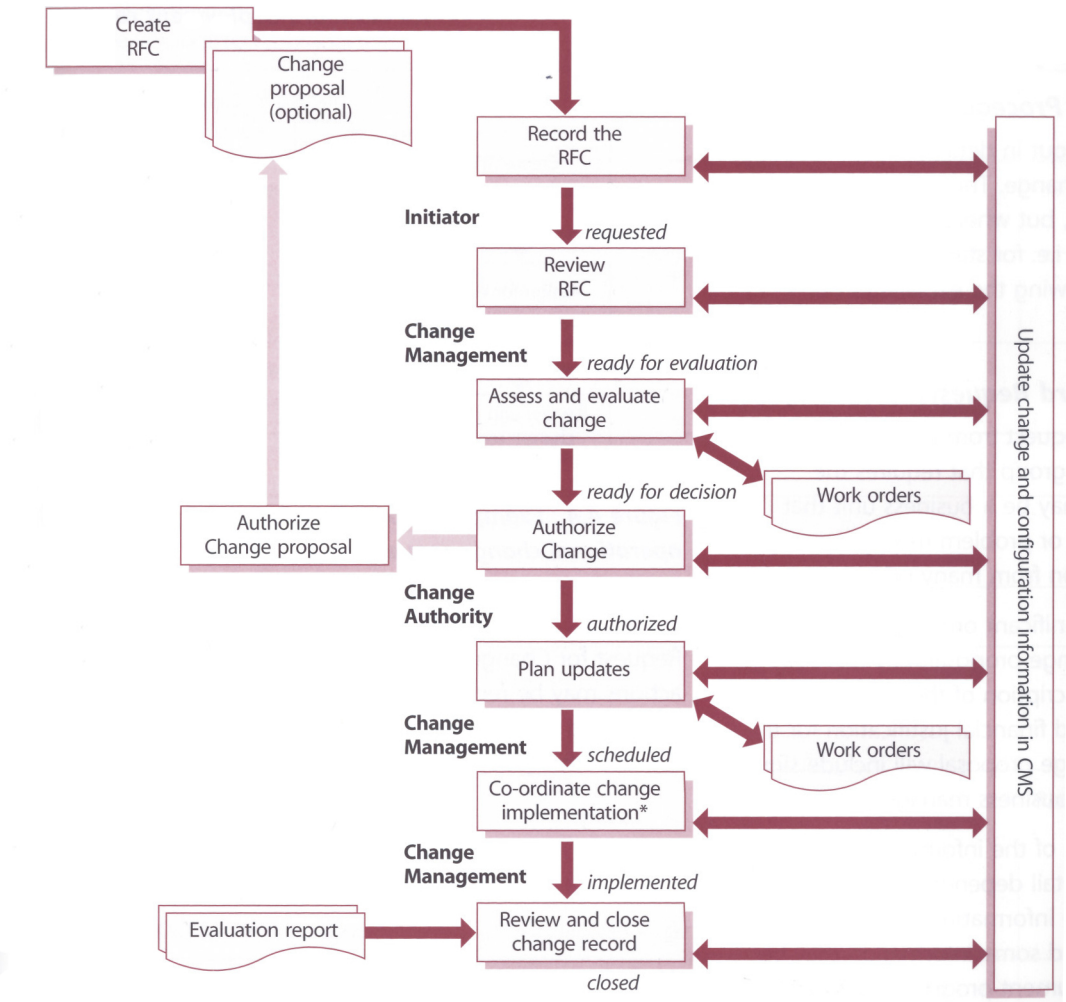


Figure 1.2: ITIL Change Management process [6].

change may include the addition of new functionality or modules to existing services, applying upgrades or patches, upgrading hardware, and transitioning services between different hardware systems.

The introduction of a new backup system or the migration to newer or different versions of operating systems, database servers or other software are examples for application-demand changes.

- Changes due to **strategic demand** are long-term and high-impact changes originating in strategic decisions by business units or the management. They usually require the introduction of new services, architectures or infrastructure, and might go beyond current knowledge and expertise of the IT department.

  As an example, imagine a web hosting company currently only offering web space that decides to also offer email services.

The impact of changes from operational demand is usually not too hard to understand and there are automation systems available[1] that assist IT technicians in implementing the RFC. Such systems usually assist users in specifying and deploying the change or even execute change activities completely autonomously. However, current approaches to configuration and installation management require the specification of concrete requirements at a very low level of abstraction and thus manual translation of high-level requirements is necessary. In this work, we investigate how to provide computer-aided assistance to the refinement of high-level change goals. We target operational demand and application demand changes, for which the change design task is already non-trivial, but for which on the other hand a certain amount of implementation knowledge and best practices are available. We capture these best practices in a systematic way (see Section 2.2) and use them to assist users in the refinement of high-level change tasks.

Having presented how change management and change design relates to the fields of IT service management and IT change management in particular, the following section summarises the major difficulties and challenges we are trying to address.

1.2.3. Challenges in IT change planning

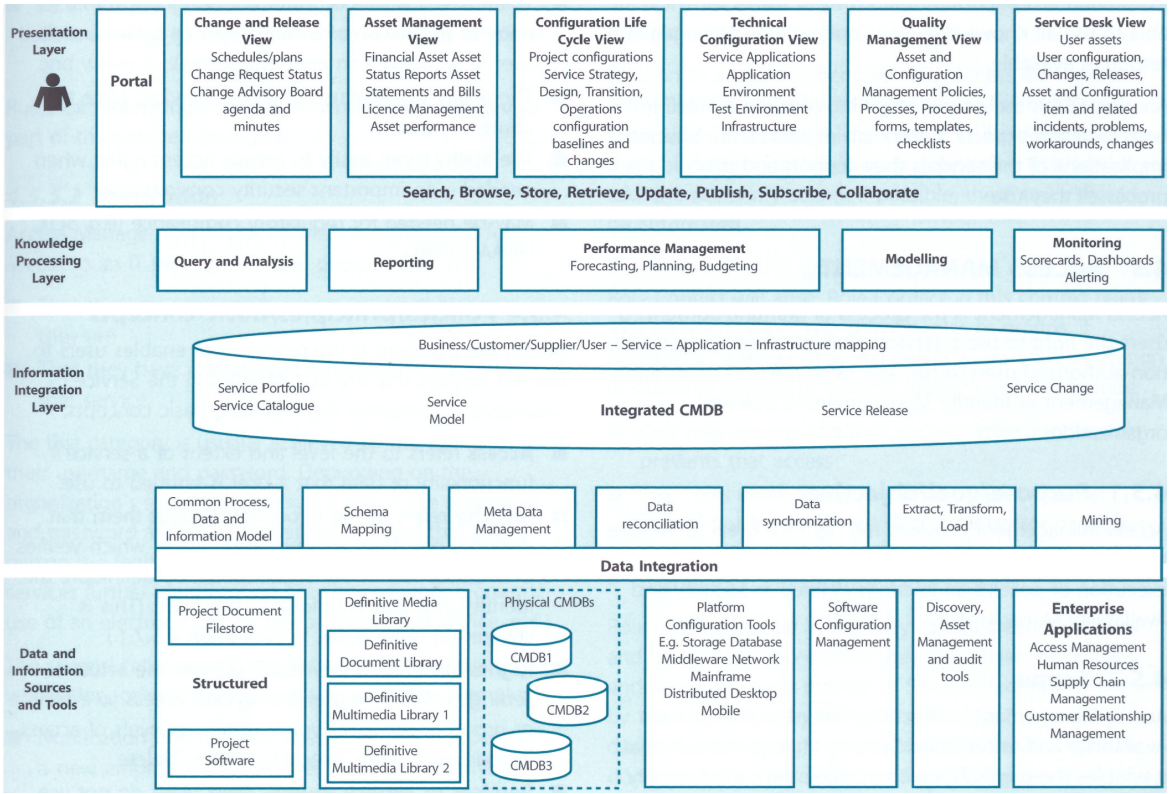


Figure 1.3: ITIL System Knowledge Management System [9].

[1]For example, LCFG: The Next Generation [7], HP Data Center Automation Center [8].

Although the change planning and co-ordination process is not extensively discussed in the ITIL Service Transition publication [6], it poses major challenges in practical realisations, as discovered by a 2006 survey [3] carried out by Universidade Federal de Campina Grande and Hewlett-Packard Brazil. The survey resulted in change scheduling and planning being the most critical task related to change management. One of the main issues making change planning a hard problem is the large number of non-uniform information sources contributing to the change management process. The ITILv3 Operation book [9] explicitly states the importance of an integrated *Service Knowledge Management System* and proposes a layered architecture for data storage, integration, processing, and presentation (Figure 1.3). The *Information Integration Layer* provides integrated and compiled views on diverse data sources from the underlying *Data and Information Sources and Tools* layer. In this work, we focus on the following types of information taking part in the change planning process:

- Current state of infrastructure and other managed entities: Configuration Management Databases (CMDB) can be used to model and store infrastructural information and interdependencies between all entities that potentially take part in the execution phase of a change.
- Implementation workflows, recipes, and best practices: Detailed knowledge on implementation workflows may be available from software documentation, written down in companies' own IT guidelines or implicitly memorised by managers or technicians.
- Constraints and policies: Planning and scheduling changes has to take into account a company's custom constraints and policies and should not violate Service-Level-Agreements (SLA) and global policies.

  Examples for constraints, policies, and SLAs are:

  - Database servers must never be shut down.
  - MySQL software may only be installed by trained personnel and must not run on Windows operating systems.
  - All changes require authorisation.
  - Every database must be backed up before its server is rebooted.
  - Allowed down times for web servers are 10 minutes per week. No downtime is allowed on Mondays.
  - No database server may serve more than four web applications.

Having to bear in mind these sources of information makes manual change plan generation a difficult, very time-consuming and error-prone task. Although there exist technologies and products that assist users in maintaining knowledge for the individual information sources[2], we are convinced that only an approach that integrates the different sources of information consistently into the change plan generation process provides reasonable assistance to users.

[2]Examples for those tools include: CMDB: Common Information Model (CIM) [10], HP uCMDB; Workflows: Business Process Execution Language (BPEL) [11], HP Data Center Automation Center [8], Workflow Management Coalition Workflow Standard (WfMC) [12]; Policies and constraints: RuleML [13], Drools [14].

Among others, non-integrated change planning is difficult and error-prone for the following reasons:

- As CMDB and workflow knowledge are decoupled, the effects of changes on the infrastructure cannot be assessed automatically.
- Unstructured storage of change knowledge exacerbates learning from previous experiences, which are usually a valuable resource to businesses and IT departments.
- Change plans are often authored by several people with different fields of expertise. Bringing together their individual contributions is difficult.
- As business change requesters and IT technicians usually do not share a common vocabulary (*impedance mismatch*), RFCs are likely to be interpreted mistakenly. Again, coordinating the requesters' needs and expectations with the actual capabilities of IT operations is time-consuming.
- Assessing and evaluating business metrics such as cost, risk, and time for change plans is difficult.

The next section sketches our approach for assisted refinement of high-level IT change requests to tackle the mentioned difficulties.

1.3. Assisted refinement of high-level IT change requests

We address the mentioned challenges in IT change management (Section 1.2.3) by investigating the assisted design of IT changes. With respect to the ITILv3 *Service Knowledge Management System* architecture (Figure 1.3), our solution fits in the *Knowledge Processing Layer*; it makes use of data from the underlying *Integrated Information Layer* and provides the computational means to assist the user in the change planning process.

Our vision is a graphical software tool that supports the change plan design process by guiding the user in refining change plans hierarchically. She can compose workflows from workflow templates and assign resources such as IT infrastructure components, personnel, and time. The software tool automatically enforces the consistency of the generated workflow with respect to policies and constraints on infrastructure states and change plan structure. The user is presented with alternative refinement options for workflow activities and every option is rated with respect to the estimated value of business metrics of the potential change plan. Wherever possible, the software tool automates the refinement process by choosing those refinements that promise to be most applicable.

As a simplified example of an exemplary change plan design process, consider Figures 1.4 and 1.5. Figure 1.4 depicts on the left side a set of workflows that a change manager can choose to refine change plans, and on the right side the simplified state of the IT infrastructure. The infrastructure comprises a web server (ws), and a database system (dbs) running on a database server (ds). The *InstallLoadBalancedJ2eeApplication* workflow is to be refined with help of the given sub task workflows for the tasks *InstallApplication* and *InstallLoadBalancer*. The result of the refinement process is illustrated in Figure 1.5. The integrated

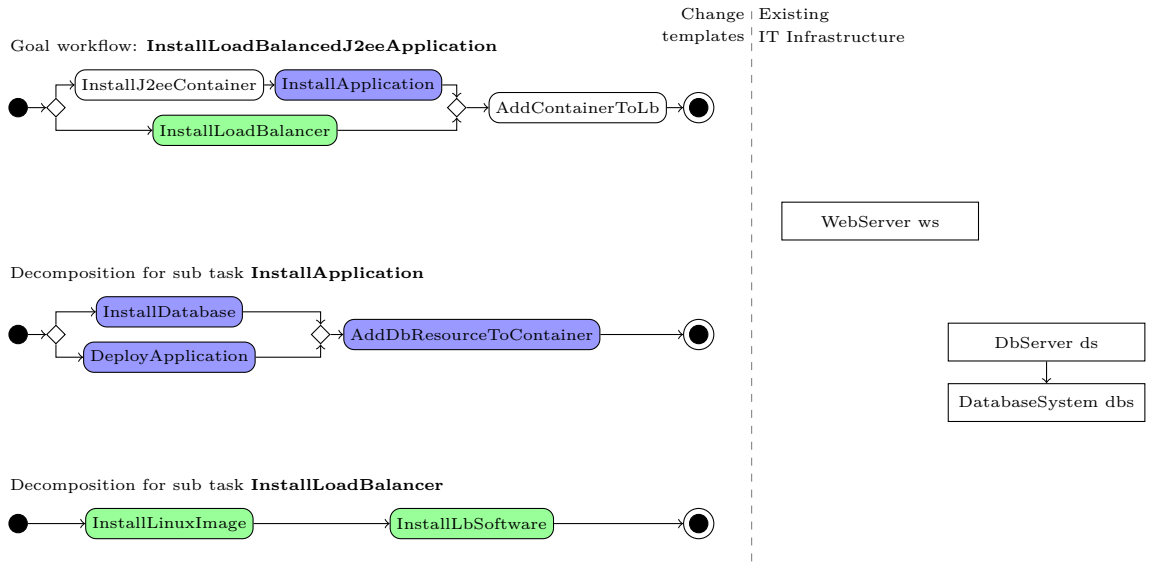


Figure 1.4: Simplified example for change templates and IT infrastructure before plan execution.

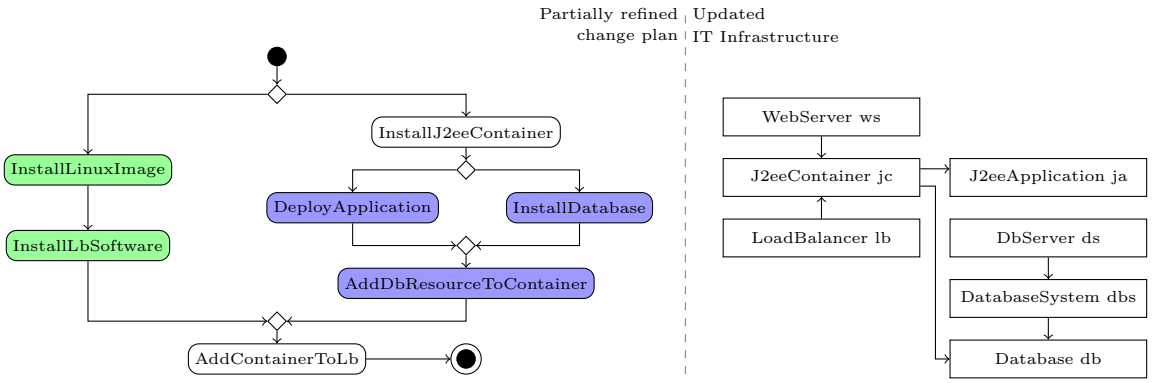


Figure 1.5: Simplified example for a partially refined change workflow and altered IT infrastructure after plan execution. The goal task *InstallLoadBalancedJ2eeApplication* (Figure 1.4) was refined by workflows for the tasks *InstallApplication* and *InstallLoadBalancer*. The integrated knowledge representation for workflows and IT infrastructure allows the computation of the effects on the infrastructure as shown on the right side: The existing configuration items ws, ds, and dbs are integrated with additional new CIs which entered the knowledge base by virtue of effects of the refined sub tasks.

knowledge representation for workflows and IT infrastructure allows to compute the effects of refined workflows on the infrastructure, as shown on the right side of Figure 1.5: The existing configuration items ws, ds, and dbs were chosen as variable bindings during the change plan design process, and are augmented with additional new configuration items which entered the IT infrastructure by virtue of effects of the refined sub tasks *InstallLbSoftware*, *AddContainerToLb*, *AddDbResourceToContainer*, *InstallDatabase* and *DeployApplication*.

In this thesis, we aim for the prototype of the aforementioned software tool. We follow a two step approach: The first, very naive version of the prototype incorporates the data models and information sources, but the complexity of the algorithms is reduced by a set of simplifying assumptions (detailed in Section 4.1). In a second step (Chapter 5), we relax some of the assumptions to make towards a more realistic and expressive tool. Our prototype makes use of automated planning algorithms joining the diverse information sources and synthesising them into a consistent plan.

This approach diminishes the afore-said challenges posed by change plan design because:

- The integrated approach allows change managers to foresee and simulate the outcome of a planned change. Infrastructure changes can be (semi-) automatically registered in the CMDB.
- The concept of hierarchically refinable workflows helps to mitigate the impedance mismatch between business requesters and IT technicians. Business managers can choose high-level change goals from a change catalogue. The controlled and structured change design process plays a part in contributing to the correctness of the change plan, i.e. it ensures that the generated plan has the desired outcome. Additionally, fully refined change plans can be used as step-by-step implementation instructions by less skilled workers.
- Business metrics provide decision support for alternative implementations of a given RFC. The business impact of a change can be assessed before it is actually implemented.
- Feedback from previous changes can be used to adapt and correct workflow templates and metric estimates.

1.3.1. Primary use case

Figure 1.6 depicts the main use case our solution is addressing. A business user, the *change requester*, has limited IT knowledge and needs to raise a request for change (RFC). Instead of relying solely on textual descriptions and being confronted with the problem of ill-defined RFCs mentioned in Section 1.2.3, the change requester browses the *change catalogue*, selects one of the high-level change tasks presented to her and fills in the required parameters. The RFC is later handled by one or several IT practitioners to be designed, documented, tested, and implemented. Using our solution, the RFC is continuously refined into smaller and smaller change tasks, until eventually an actionable workflow is generated. The output uses best practices from various IT work groups and complies with all corporate IT policies. A *change knowledge manager* role is responsible for maintaining the change catalogue and for interacting with IT practitioners to ensure that best practices for change design are captured and maintained.

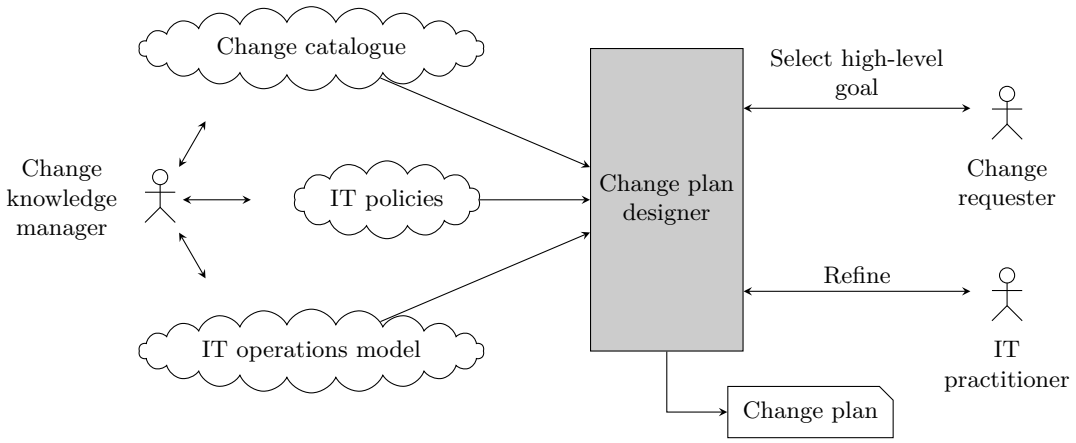


Figure 1.6: Assisted change design use case. A business user, the *change requester*, raises a request for change (RFC). The RFC is handled by IT practitioners to be designed, documented, tested, and implemented. A *change knowledge manager* is responsible for maintaining best practices, policies, and infrastructure knowledge, which all contribute to the change plan design tool.

1.3.2. Complementary user stories

The following user stories complement the above use case with examples of user roles and scenarios.

1.3.2.1. Stepwise refinement of high-level goals

Mark-Kevin, the service manager of a company offering hosting of web applications, discovers performance issues with a web shop application after having received a number of customer complaints. He creates a new RFC and chooses the high-level goal "Increase web application speed" from the service catalogue. He selects the affected web shop application from a menu and discusses possible measures with Jaqueline, the technology manager of his company.

They are presented with a list of available implementation solutions for the high-level goal "Increase web application speed", among them "Add new application server to load balancer", "Upgrade machine hardware" and "Change database software". They decide to choose the "Change database software" decomposition, knowing that the current database is not well suited for high-load environments. The affected database server machine is automatically selected and Jaqueline picks MySQL as the target database software. She can see the risk, cost, and time estimates of the current preliminary change plan.

Jaqueline browses through the policy repository and adds a new policy, stating that MySQL version 4 must not be installed on Microsoft Windows operating systems. Finally, she forwards the change plan to Mikkel, a database consultant, for further refinement.

Mikkel refines the change plan by choosing database versions and is confronted with warnings if he tries to choose MySQL version 4, as a Windows operating system is installed on

the target machine. Also, he is automatically recommended to add an "Upgrade machine RAM" task to the change plan when he chooses MySQL version 5.

The finished change plan is finally sent to a database technician and a hardware technician who can use it as a step-by-step manual for the change implementation. It includes detailed instructions that explicitly name the affected hardware (i.e. on which machine they are supposed to change something) and software versions (i.e. MySQL version 5), as well as temporal dependencies of their respective actions (i.e. upgrade machine RAM *before* installing MySQL 5).

1.3.2.2. Capturing change knowledge and implementation recipes

Jaqueline notices that several customers experience similar problems with the provided hosting services: An open source calendar application that is used by many customers recently changed the software requirements from PHP4 to PHP5. She decides to create a new change template called "Migrate web server from PHP4 to PHP5" and adds several sub tasks:

- Check other installed web applications for incompatibilities
- Backup web server data
- Install PHP5 module
- Reboot server

She also adds preconditions for this decomposition template to be applicable: The Apache HTTP server on which the PHP5 module is supposed to be installed must be at least of version 2.x and the machine must have at least 1GB of RAM.

Jaqueline talks to Mandy-Loreen, a technician with expertise in Apache server software to discuss the newly created decomposition template. Mandy-Loreen advises Jaqueline not to use Apache servers prior to version 2.x for new installations because Apache has announced to stop support for these version soon. Jaqueline adds a new company-wide policy to the policy repository stating that any installation of Apache version 1.x must be authorised by an IT manager.

1.3.2.3. Learning from past changes

In the *Review and close change record* activity in the company's ITIL change management process, Jaqueline's task is to evaluate the quality of implemented change plans. During the migration of a database to new server hardware, the technicians experienced loss of customer data because of incompatible DBMS versions and had to restore the data from a one week old backup. Jaqueline analyses the change plan and adds a backup task before the database migration task. She also adds a new policy, enforcing that "all changes to customer database systems must be authorised", and increases the estimated risk for the database migration branch of the workflow template.

1.4. Related work

Most IT management software vendors provide some support to implement the change management process, as laid out in the ITIL Service Transition book [6]. Process management tools such as HP Service Manager or BMC Remedy Change Management help to track the evolution of a change in all phases of its life cycle, but they provide no decision-support functionality for change design. In another class of products, orchestration capabilities for automated provisioning have been developed in solutions such as HP Operations Orchestration or Tivoli Intelligent Orchestrator. While these solutions provide graphical workflow editors, all decisions made during the design process are left to the technician using the editor, and no assistance is provided.

With CHAMPS (Change Management with Planning and Scheduling) [15], Keller *et al.* produced the seminal work for automation and optimisation in change management. In their work, change design was inferred from software and hardware dependencies, and was primarily targeted at software installation and de-installation. Knowledge reuse and the concept of high-level change requests were not the topic of their work.

FeedbackFlow [16] is a framework for automatic generation of workflows of system management actions. It uses a state-space planner to compute plans from a given set of available atomic actions, and an initial and a goal state of the IT environment. The plan is executed by a workflow engine which monitors its outcome and adjusts the planner's state appropriately. FeedbackFlow lacks the notion of refinable high-level tasks and has no mechanism for encoding best-practices and change recipes.

Cordeiro *et al.* [4] have proposed ChangeLedge, a conceptual solution for the automated refinement of preliminary change plans into actionable workflows. After defining a conceptual model, ChangeLedge uses the concept of change templates to capture and reuse change knowledge in the design of recurrent or similar IT changes. ChangeLedge and our solution address different issues in the problem of IT change design. First, ChangeLedge requires a skilled technician to understand the textual description found in the RFC and to draft a preliminary plan. It does not address the information impedance mismatch problem between change requesters and IT practitioners mentioned in Section 1.2.3. Also, ChangeLedge does not provide mechanisms to encode best practices, our second challenge; it generates any sequence of steps in which dependencies and constraints between activities are satisfied. There are two issues with this approach: On the one hand, it assumes complete and accurate knowledge of the dependencies in the IT model, and on the other hand, IT practitioners are usually only comfortable using tried and tested methods and would only trust a small subset of such automatically generated workflows. Finally, while our approach and ChangeLedge solve different problems, they can be seen as complementary: The former *assists* an IT practitioner in refining high-level change tasks, the latter *automates* the generation of low-level workflows. The integration of the two approaches is discussed in Section 5.4.

Other aspects of change management, such as change scheduling, have been proposed. Sauvé *et al.* [17] take the example of change scheduling to demonstrate linkage models between IT availability metrics and business objectives. Rebouças *et al.* [18] use the linkage model from [17] and formalise the problem of business-driven scheduling of changes, in which changes need to be assigned to maintenance windows. Trastour *et al.* [19] go one step further

by breaking down changes into the elementary activities that compose them and by providing a scalable solution to the change scheduling the problem. Finally, Setzer *et al.* [20] studied the impact of IT changes onto business processes and considered stochastic durations. The investigation of these business impact analysis techniques for the design of changes, as well as the integration of planning and scheduling of changes, is left for future work.

In the area of IT infrastructure design, Ramshaw *et al.* [21] propose a constraint programming approach to design IT infrastructure from SLA requirements. The main difference with what we propose here is that [21] focuses on generating IT system configurations that satisfy capacity and performance requirements, while we are interested in the processes for transforming existing IT systems.

The general field of automated planning [5], an area of artificial intelligence, provides techniques to generate plans of actions meeting certain objectives. This field has evolved from classical planning to more specialised forms of planning. In particular, Hierarchical Task Network (HTN) planning [5, 22] is well suited for domains in which plans are known to exhibit a hierarchic structure; the planning algorithm is then guided by predefined hierarchical decomposition templates. Although automated planning research and especially HTN planning have had successes in many domains[3], to our knowledge, it has not been applied to change design. Our proposed assisted design solution adopts HTN planning principles on knowledge representation and plan generation.

Finally, the mixed-initiative automated planning approach was applied to support user-centric plan design in various scenarios and frameworks (crisis intervention [24], Heracles framework for web-based information assistants [28], dialogue-based planning for transport and routing [29, 30]).

[3] HTN planning was, among others, deployed in Mars exploration [23], crisis intervention [24], workflow planning in Grid computing [25], analysis and life cycle management of plans [26], and template-based composition of web services [27]

2. Information model for change plan refinement

This chapter proposes data models for information contributing to the change design process: IT infrastructure knowledge, best practices and change recipes, and IT policies. To minimise IT service disruptions and the need for further corrective changes, IT practitioners favour to use procedures that are fully understood and tested. In our experience, even within large IT organisations, these procedures can be seen as *change recipes*, since they are rarely formalised; they may be documented in unstructured documents shared within workgroups, or implicitly memorised by managers or technicians. These recipes are typically scattered within the IT organisation, and a single change may involve recipes from different IT workgroups, such as a database workgroup or a Unix workgroup. Moreover, change recipes can be concrete (e.g. the steps to build a Windows 2003 web server to corporate standards) or abstract (e.g. the steps to consolidate a set of servers in a data centre), and would require further refinements to obtain an actionable workflow. We propose to formalise these change recipes in the form of *best practices* for change design. In addition to these, IT practitioners must also consider corporate *IT policies*. Such policies can impose constraints on the possible configuration of the IT infrastructure (e.g. all external facing web servers must have a firewall) and on the processes used to perform changes (for instance, all changes affecting the payroll system must be authorised by the financial controller). Finally, IT practitioners must take into account the state of IT operations, i.e. the state of the IT infrastructure and of the IT processes they need to interact with in order to carry on their work. A Configuration Management Database (CMDB) is typically used to model and store infrastructural information and interdependencies between all *configuration items* (*CI*s) that potentially take part in the execution phase of a change. Other tools, such as service management tools, track the state of IT processes.

Having presented the different sources of change knowledge and our assisted change refinement use case (Section 1.3), we now describe our approach to provide an integrated view on change knowledge. We first present the IT operations model, the change catalogue and best practices model, the main contribution of this information model, and finally the IT corporate policies model. Chapter 2 presents the static data models, while Chapter 3 focusses on the algorithms and interactions between the components of the framework that constitutes our solution.

2.1. IT operations model

Our solution requires an information model representing concepts from IT operations that are used in the definition of change activities. We assume that an object-oriented formulation

exists and that all classes have a root class, in order to have a generic way to refer to all such IT concepts. The model may represent hardware and software components, IT technicians and their skills, or steps in IT processes. For the purpose of this work, we have chosen to instantiate the IT operations model as a subset of the Common Information Model (CIM) [10]. Being an object-oriented framework, CIM defines hierarchies of configuration items through specialization and dependencies between items by associations, compositions, and aggregations. The root class of the CIM model is *ManagedElement*.

To get a first impression of how a concrete model of infrastructure components and their interdependencies may look like, and as a short preview on the exemplary J2EE domain used in the ChangeRefinery prototype (Section 4.3), consider Figure 4.4. The model features hierarchies of hardware components (web servers, load balancers, database servers), different kinds of databases, J2EE application containers, J2EE applications connected to databases, JDBC resources, and load balancer applications. Additionally, in compliance with the ITIL guidelines for configuration management systems, technicians and their respective skills in operating the above-mentioned entities are also included in the model. This J2EE scenario can be extended to support other domains or more complicated infrastructures by adding new classes and dependencies from CIM. Alternatively, the CIM model could be substituted by data models from commercial CMDB products.

2.2. Change catalogue and best practices model

We propose a model whose aims are, on the one hand, to provide a clear interface between change requesters and IT practitioners, and, on the other hand, to formalise the sets of best practices for IT changes. Figure 2.1 depicts a UML representation of this information model.

We introduce the notion of hierarchically decomposable change *tasks*. In our model, change tasks convey the meaning of *what* will be done, not *how* it will be done. For instance, deploying a web server is a task that could be implemented in many ways, depending on the expected traffic or on whether the server will be externally facing. A business user will typically not be concerned with the technical implementation details of a change, and should only have to deal with change tasks. Change tasks can represent activities at different levels of abstraction, from high-level tasks that a business user may ask for (e.g. altering a business process in an SAP system) to low-level technical tasks (e.g. installing an Oracle database).

To be able to represent workflows supporting parallel and sequential actions, our model includes *ordering constraint*s which provide qualitative binary temporal orderings between tasks. A *task network* is hence a partially ordered set of tasks.

The *change catalogue* is the subset of predefined task networks that an IT organisation chooses to expose to its users. We propose the change catalogue as an analogue to the ITIL service catalogue [6]: It contains a set of high-level IT changes that are available to change requesters. An RFC hence contains an instance of a task network to represent the goal workflow. Note that a goal change can be refined into different sub tasks and with diverse variable bindings. Hence, it does not represent or imply a particular solution to a change request, but it specifies *what* will be done.

A change task can have any number of *variable*s. Variables have a name, a type, which is either a data type (e.g. an integer representing a number of users) or a class from the IT operations model (the *WebServer* class), and an instance value. Values for all variables of the original task network (from the RFC) can be specified by the change requester. As we will see in the following section, the change design process consists of recursively refining change tasks into sub tasks until a concrete workflow is obtained. Variables are used to pass parameters between a task and its sub tasks.

We chose the Hierarchical Task Network (HTN) planning paradigm (Appendix A) to model both the static structure of change workflows and the dynamic refinement[1] of workflow tasks in sub tasks. Borrowing from the HTN planning paradigm, a change *task* (see Figure 2.1) can be realised by one of two *refinements*, an operator or a method. *Operators* are atomic activities that have known *effects* on the IT operations model. For instance, operators accomplishing the *InstallDatabase* task in Figure 1.5 have the effect of adding a database CI to the knowledge base. *Methods* encode the best practices or recipes mentioned earlier, and are the mechanism we use to refine a task into further lower-level tasks. In our model, several operators or methods could implement the same task, leading to different effects and task decompositions. Coming back to the introductory example of Figure 1.4, the *InstallApplication* task for instance could be decomposed by the shown simple sub workflow consisting of the sub tasks *DeployApplication*, *InstallDatabase*, and *AddDbResourceToContainer*, but one could also define an alternative method to refine the same task, for instance with additional authorisation and backup steps.

Because not all refinements (operators and methods) may be applicable to a given task in all IT infrastructure states, refinements are guarded by a *condition* on the IT operations model. The refinement can be used only when the condition is satisfied. For example, every refinement of the task *AddContainerToLb* in Figure 1.4 requires the existence of a *J2eeContainer* and a *LoadBalancer* CI in the knowledge base.

To see why the HTN model of partially ordered sets of tasks is particularly well-suited to represent hierarchically decomposable workflows, consider Figure 2.2. The Figure depicts on the left side a set of tasks $T = \{A, B, C, D\}$ with a partial order over T induced by $R = \{(A, B), (A, D), (B, D), (C, D)\} \subseteq T \times T$. The ordering constraints state that A and C have no predecessors, that D has no successor, that B is ordered after A, and that C is not related to A or B. The relation R intuitively induces the workflow shown on the right side of Figure 2.2.

2.3. IT policies

Change plans typically have to comply with corporate policies resulting either from business requirements, service level agreements (SLA) or technical considerations. We identify two different classes of policies which constrain the change plan design process. First, policies may qualify the allowed states of the IT infrastructure, for example by prescribing the use of certain software versions and by only allowing certain hardware and software combinations. Following the ITIL recommendation, this set of policies should be made available in the

[1] The reasons for HTN being algorithmically well-suited for our problem will be explained in Section 3.3.1.

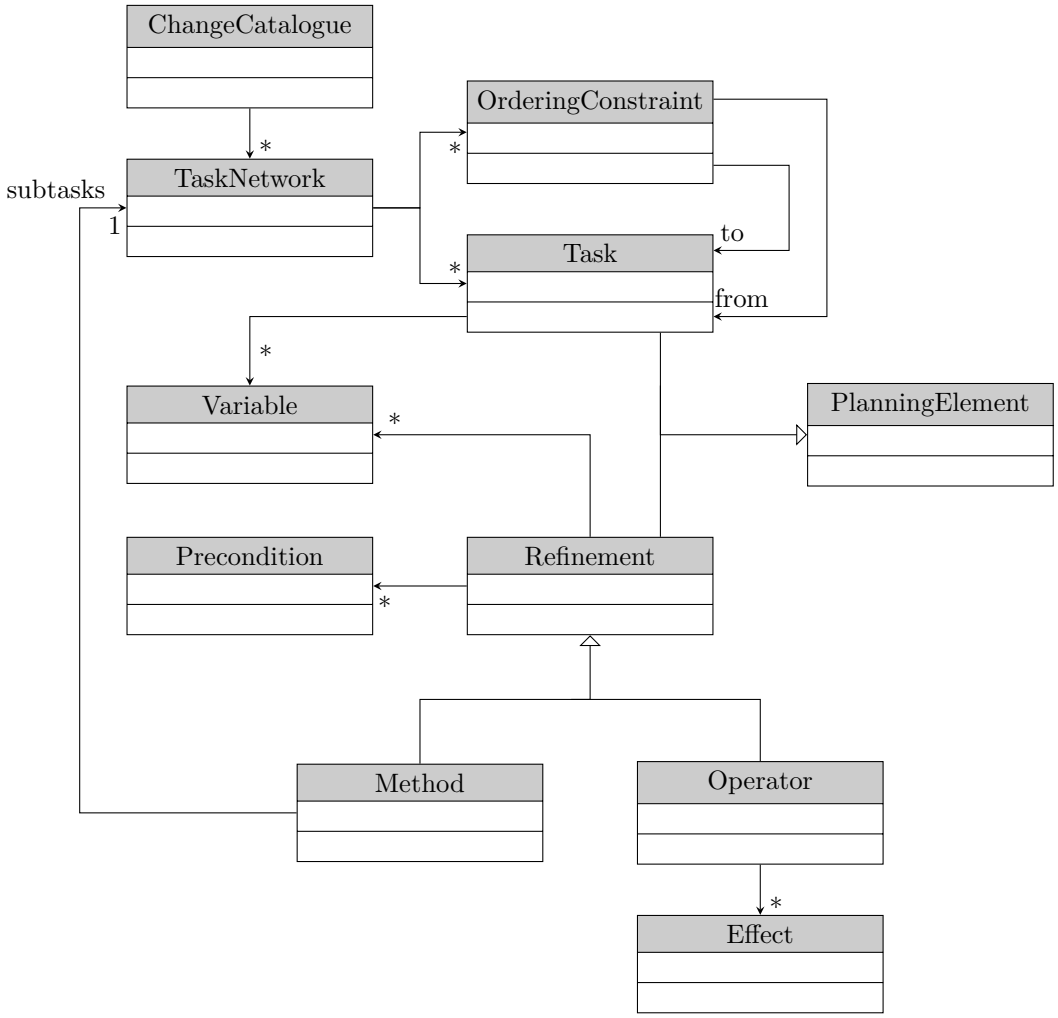


Figure 2.1: Change catalogue and best practices UML model. A task network is a partially ordered set of tasks (a workflow). Every task in the workflow can be refined by its methods or operators. A method decomposes a task in its sub tasks, an operator has effects that change the state of the world. A change catalogue is a collection of goal task networks.

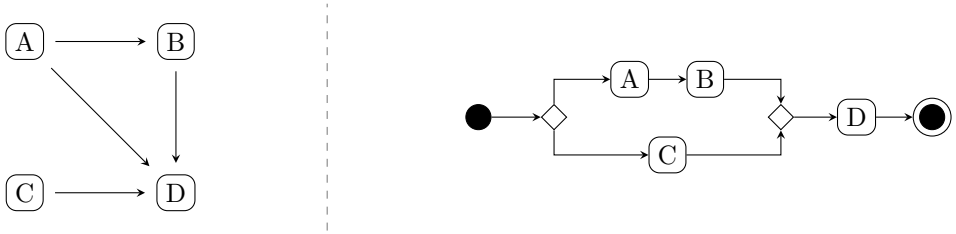


Figure 2.2: Relation between partially ordered sets of tasks (left side) and workflows (right side).

Definitive Media Library [1]. The second type of policies addresses the logical and temporal structure of change plans. Examples of such policies could be the obligation to perform systematic backups before certain classes of change tasks or the need to perform changes during maintenance windows specified in an SLA.

Our solution is independent of the choice of the policy language (e.g. RuleML [13], Drools [14]). The only requirement is the ability to express rules on the IT operations model and on the change catalogue and best practices model.

3. A framework for assisted refinement of change tasks

After having described the information view of our approach, we now present our conceptual architecture for assisted refinement of change tasks, give the requirements for the main components that compose the solution, and explain the algorithms used to refine high-level change tasks into actionable workflows.

Figure 3.1 depicts the architecture of our solution. The main goal of the architecture is to support the integration of various information sources into a consistent change design process. The *change planner* is the core component that compiles change plans from the connected information repositories. The *IT knowledge base* holds the IT operations model and acts as a simulation environment for the planner. The *change catalogue repository* holds the change catalogue and best practices model and includes tasks, methods, and operators. The *temporal reasoner* is used by the planner to enforce temporal consistency while building the plans. Finally, the *policy repository* and *policy engine* store and enforce the IT corporate policies.

Consider once again Figures 1.4 and 1.5 as a rough sketch of how we integrate IT infrastructure knowledge into the change plan. As explained in Section 2.2, every operator accomplishing a particular task in a workflow is guarded by a condition that specifies under which circumstances the operator is applicable. The condition typically imposes constraints on the existence and linkage of configuration items. A satisfied condition thus generates *variable bindings* that entail by which CIs the condition was satisfied. The variable bindings can then be used in the effects of an operator to itemise which CIs are to be altered. As an example, consider an operator for the sub task *AddDbResourceToContainer* in Figures 1.4 and 1.5: It requires as precondition the existence of a *Database* and a *J2eeContainer* CI in the knowledge base and could pose additional optional constraints, for example on the memory size of the *DbServer* CI on which the database system is installed. The natural language formulation of the knowledge base query reads "return all tuples [J2eeContainer, Database] such that Database is part of a DatabaseSystem and DatabaseSystem is installed on a DbServer and DbServer has at least 1GB RAM". A matching tuple (in this example just the one tuple [jc, db]) is then used to specify the operator's effect on the knowledge base: The db CI is added as a database resource to the J2eeContainer CI jc.

3.1. IT knowledge base

The IT knowledge base component stores the IT operations model and acts as abstract interface between the planner and different types of IT infrastructure and operational databases.

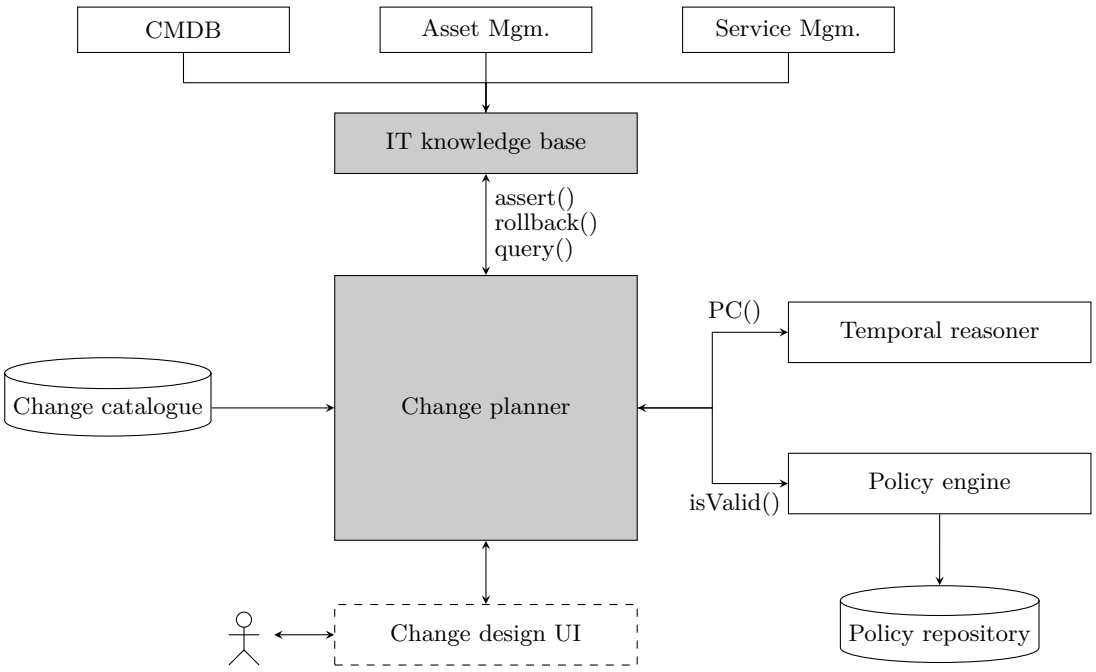


Figure 3.1: Architecture for assisted design of change tasks. The central change planner component integrates various sources of change knowledge into change plans. The IT knowledge base component is the interface to different kinds of configuration management repositories; the temporal reasoner and policy engine components provide and compute additional constraints on change plans.

In order to simulate the effects of operators, the change planner must be able to *assert* facts. In our object-oriented model, this means creating new objects, creating or changing dependencies between objects, and changing the values of attributes of objects. Because the change planner is implemented as a backtracking search (see Section 3.3), the IT knowledge base must be able to *snapshot* its current state and to *undo* assertions (*rollback*). In other words, it must provide versioning capabilities. At various steps in the design process, the planner must be able to verify if preconditions are satisfied in order to check what operators and methods are applicable. Hence, the knowledge base must also provide a *query* mechanism that return tuples of objects matching a given search criterion. Finally, it must *detect causal dependencies* between the effects of operators and the conditions of refinements to ensure that the generated plans are temporally consistent. The rationale for this last feature will be made evident in Section 3.4.

3.2. Change catalogue repository

The change catalogue repository stores the change template and best practices, or in HTN terms, the tasks, methods, and operators. This repository is accessed by change requesters when they need to author an RFCs. We do recognise that to make the change catalogue easily accessible to requesters, change tasks need to be annotated and categorised, but this falls outside the scope of this thesis. The change catalogue repository is also accessed by the planner during the refinement process, and this simply consists of navigating through the associations of the model, for instance to find suitable refinements for a given task.

3.3. Change planner

This section describes the change planner component which operates at the heart of our architecture and relies on an adapted HTN planning algorithm. We first justify the choice of HTN planning over state-space planning techniques in Section 3.3.1, and then detail our algorithm for HTN planning with object-oriented data models in Section 3.3.2.

3.3.1. Applicability of planning paradigms

In Section 2.2, we have already explained why the partial-order HTN planning paradigm is well-suited to model hierarchically decomposable workflows: A task network (i.e. a partially ordered set of tasks) represents a single workflow, where workflow activities correspond to tasks and parallel or sequential branching is specified by the partial order on the set of task nodes. Conditional activities are encoded in the preconditions of methods and operators; sub workflows are defined by decomposition methods. It remains to be asserted, why HTN planning is favourable against state-space based planning (see Section A.2) from an algorithmic point of view. In the latter, a state-space search algorithm aims at finding a sequence of operators that connects a given *initial state* with a given *goal state*; every operator has to be applicable with respect to its precondition and changes the state of the world according to its effects. Although HTN and state-space planning can in theory solve the same

problems (see Appendix A.3.2), they differ deeply in the style of the problem statement: In HTN, you specify a goal task, which is a high-level description of *what you want to do*, while in state-space planning you specify *which state is to be achieved*. The HTN notion of hierarchical decomposition by methods can be regarded a metric that assists the planning algorithm in finding a plan more easily, and the initial goal task network puts constraints on the composition of the final plan. In domains that feature hierarchic structure, HTN planning thus offers considerable performance advantages over state-space planning because huge parts of the planning space are pruned. Furthermore, the HTN paradigm offers plans of arbitrary degree of finesse: Operators making up the change plan can be either abstract high-level descriptions of how to accomplish a task[1], or highly specific low-level executable actions[2].

3.3.2. HTN planning with object-oriented data models

The change planner is the core component of the framework and uses a variation of the HTN planning algorithm. Its input is a task network specified by the change requester. By exploring the space of possible decompositions for the list of tasks in a given state, the algorithm continuously decomposes tasks by replacing them by their sub tasks as defined in the decomposition methods, until the initial set of goal tasks is transformed into a list of atomic operators (then called a list of actions or a plan), which is the algorithm's output. Depending on the application, one may be interested in computing all plans, an arbitrary plan, or the plan minimising a given metric. The planner is tightly coupled with the IT knowledge base which acts as the simulation environment for the planning algorithm: Effects of operators from partially refined workflows change the state of the knowledge base, queries validate the applicability of methods and operators and return variable bindings. Depending on the initial state of the world and on *preconditions* and *effects* of operators and methods, different decomposition methods might be applicable for a given task. A formal description of the HTN algorithm is given in Appendix A.3 and is covered to great extent in the automated planning literature [5]. We will concentrate on our contributions which are

(i) to provide clear separation of concerns between the various information repositories,

(ii) the adaptation to object-oriented models, and

(iii) the consideration of temporal constraints.

The classical HTN formulation (Appendix A) relies on first-order logic to define tasks, methods and operators; variable bindings are found and passed by unification, the state of the world is given by a set of atoms and preconditions, and effects are sets of literals. Because of the predominance of object-oriented models for IT infrastructure and operations, such as CIM, we do not express the infrastructure data model in terms of first-order logic. Instead, we rely on the IT knowledge base which can be queried to return all objects, such as configuration items, matching a method's or an operator's preconditions. The inverse adaption, namely using a conventional HTN planner that operates on states expressed as sets of atoms, and connecting it with a logic based knowledge base component which is enhanced to support object-oriented data models, is briefly discussed in Section 5.4.9.

Algorithm 1 Non-deterministic algorithm for partial-order forward decomposition HTN planning with object-oriented knowledge base and temporal reasoning integration.
Inputs: Knowledge base (kb), goal task network (tn).
Output: Plan π.
The algorithm assumes the existence of a global variable π that stores the solution plan, and an STP instance, stp.

```
1:  function HTN(kb, tn)
2:      if tn = ∅ then
3:          π ← empty plan
4:          return true
5:
6:      t ← tn.getFirstTask()                          ▷ pick one of the tasks without predecessors
7:      active ← t.getApplicableRefiners(kb)  ▷ find refinements with satisfied precondition
8:      if active = ∅ then
9:          return false                                    ▷ fail, if no refinement is applicable
10:
11:     performer ← active.choose()                          ▷ pick one of the refinements
12:     bindings ← performer.computeBindings()                    ▷ find its bindings
13:     b ← bindings.choose()                                ▷ pick one of the bindings
14:
15:
16:     if o ← performer is an operator then
17:         kb.assert(o.getEffects(b))                ▷ simulate effects of the chosen operator
18:         tn.removeTask(t)                  ▷ remove accomplished task from task network
19:         π.add(o)                                        ▷ add operator to solution plan
20:         stp.decompose(t, o)                ▷ impose temporal constraints on operator
21:
22:         if stp.PC() = true then                              ▷ ensure temporal consistency
23:             HTN(kb, tn)                    ▷ recursively refine remaining task network
24:         else
25:             return false
26:
27:
28:     else if m ← performer is a method then
29:         m.decompose(tn, t, b)              ▷ decompose task into method's sub tasks
30:         stp.decompose(t, m.subTasks)       ▷ impose temporal constraints on sub tasks
31:
32:         if stp.PC() = true then                              ▷ ensure temporal consistency
33:             HTN(kb, tn)                    ▷ recursively refine remaining task network
34:         else
35:             return false
```

Algorithm 2 Finding nodes without predecessors in a partially ordered set.

```
1: function findNodesWithoutPredecessor(Partially ordered set (U, E))
2:     candidates ← U
3:     for all (u1, u2) ∈ E do                          ▷ check all edges
4:         candidates ← candidates \ {u2}               ▷ remove nodes with incoming edge
5:     return candidates
```

Our adoption of the classical non-deterministic partial-order HTN algorithm is shown in Algorithm 1. Every recursive invocation of the algorithm takes as input a reference to the current state of the knowledge base (kb), and the task network (tn) to be refined. The algorithm picks a task without predecessors[3] in tn (*forward decomposition*, line 6) for further decomposition, chooses one of its applicable *refinements*[4] (lines 7,11), and computes its respective variable bindings (line 12); one variable binding is selected for the chosen refinement[5] (line 13). If the selected refinement is an operator, its effects are applied to the knowledge base (line 17), the refined task is removed from the task network (line 18), and the operator is added to the preliminary plan (line 19). Before recursively decomposing the remaining task network (line 23), the temporal reasoner component is invoked (lines 20, 22) to check the preliminary plan's temporal consistency (see Section 3.4). If, on the other, hand the chosen refinement is a method, the respective task is substituted with the method's sub tasks (line 29), and the resulting task network is recursively decomposed (line 33). Again, the temporal reasoner component is invoked (lines 30, 32) to check the preliminary plan's temporal consistency. A deterministic implementation of the algorithm (see Section 4.2.1) searches the whole space of possible decompositions by trying all refinements and bindings, and backtracking in the case of failure.

The knowledge base query is the central concept that links the HTN planning algorithm with the knowledge base component. On the framework level, the only requirement for the type of query mechanism is the support of queries against objects and their linkages, and the capability to return tuples of matching configuration items. The concrete implementation of the query mechanism in the ChangeRefinery prototype is detailed in Section 4.2.2.3.

3.4. Temporal reasoner

Real-world change plans consist of durative sequential and parallel change activities, both features that are not supported by the classical partial-order HTN formalism (Appendix A.3), where plans are sequences of atomic actions (Definition 14 in Appendix A.3). Total order plans may be acceptable for applications in which the plan is executed in a sequential fashion anyways (e.g. because all actions have the same resource requirements and therefore

[1] E.g., "ask a web application expert to install a Tomcat server."
[2] E.g., "execute the script /bin/myscript on server db.company.com with parameter x=Jenny."
[3] Algorithm 2 shows how to calculate all tasks that have no predecessors.
[4] A refinement is defined to be applicable in the current state of the knowledge base if its precondition query has a non-zero number of matching tuples in the knowledge base.
[5] Note that the concept of bindings does not increase the branching factor when compared with classical HTN, as different bindings for refinements correspond to different possible substitutions in classical HTN.

cannot be accomplished simultaneously), but certainly not for change plan design: we are seeking a change plan that conserves the parallel structure of the workflow templates and only contains additional ordering constraints where imposed by interactions between effects and preconditions of operators. Coming back to the initial example from Figure 1.5, operators accomplishing, for instance, the tasks *InstallDatabase* and *InstallJ2eeContainer* are usually independent and should remain temporally unconstrained if they affect different server CIs (ws and ds), as in the depicted case. On the other hand, if the database and J2EE container were chosen to be installed on the same server, additional ordering constraints are necessary to ensure that the installation activities do not conflict with required resources.

This section introduces how the Simple Temporal Problem (STP, Appendix A.5), a constraint satisfaction problem specifically tailored for efficient temporal reasoning, can be integrated to HTN planning in order to support both durative actions and parallel (i.e. partial-order) solution plans. An STP 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 the set of binary constraints between time points. For two time points, t_i and t_j, a constraint $[a_{ij}, b_{ij}] \in C$ restricts the relative distance of the two time points by requiring $a_{ij} \leq t_j - t_i \leq b_{ij}$. A qualitative ordering constraint between two time points, t_i and t_j can be expressed as $[a_{ij}, b_{ij}] = [0, \infty]$. To support durative actions and parallel plans, we associate start and end time points with every task and action in the HTN planning process. Temporal relations between those time points reflect quantitative and qualitative ordering constraints between tasks and actions. A detailed introduction to constraint satisfaction and temporal problems can be found in Appendix A.4.

3.4.1. HTN planning with parallel plans and quantitative temporal constraints

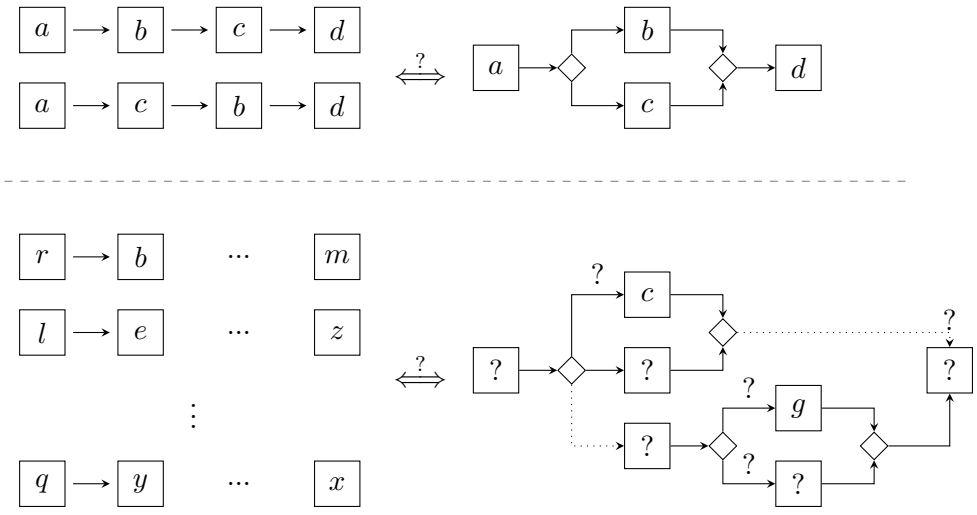


Figure 3.2: Total order versus partial-order plans. The top (bottom) figure illustrates a simple (more complex) example of a set of sequential plans (left side) and a corresponding workflow (right side).

A naive approach to attain parallel plans is to compute all possible sequential plans and

use them to reverse engineer a possible parallel plan (see Figure 3.2). This approach is not reasonably practicable for the following reasons:

1. Mapping a given set of sequential plans to a parallel plan is extremely complicated (see lower diagram in Figure 3.2).
2. Computing *all possible* sequential plans is in general intractable.
3. Even for extremely simple cases (upper diagram in Figure 3.2), if actions have durations and resource requirements, there is no way to decide whether a parallel plan is equivalent to a set of sequential plans. In the upper diagram of Figure 3.2, consider the durative actions b and c with conflicting resource requirements: Although both sequential plans might be perfectly valid, the corresponding parallel plan is not equivalent because it allows for simultaneous execution of b and c, which is impossible due to the conflicting resource requirements.

TimeLine [32] and SIADEX [33] are two different existing approaches to integrate HTN planning with temporal reasoning and to obtain parallel plans. In TimeLine, the classical HTN formalism is enhanced with durative actions, non-instant effects, and time-aware precondition handling; the proposed algorithm is provably sound and complete, but lacks the natural intuitive elegance of the pure HTN procedure. Our planner borrows from ideas implemented in the SIADEX planner [33], which utilises the combination of a classical partial-order HTN algorithm and an associated STP (Simple Temporal Problem, see chapter A.5) to perform temporal reasoning and to compute partial-order plans. The following description sketches out the SIADEX algorithm:

- Every action a_j in the change plan has a duration, $duration(a_j)$.
- Every literal l_j^k in the effect of an action a_j has a delay Δt_j^k by which they are achieved after the execution of its corresponding action.
- Given a sequence of states $s_0, s_1, s_2, ..., s_n$ induced by a partial plan $a_1, ..., a_n$, every state s_i is given by
$$s_i = \{\langle l_j^k, a_j, \Delta t_j^k \rangle\}$$
where every tuple $\langle l_j^k, a_j, \Delta t_j^k \rangle$ denotes a literal l_j^k that was introduced by the effect of action a_j with the delay Δt_j^k.
- Every partial plan $\pi = a_1, ..., a_n$ has an associated STP, T. The time points of T are the initial time point TR, and, for every $a_i \in \pi$, two time points $start(a_i)$ and $end(a_i)$.
- Whenever a new action a_i is added to the partial plan, the preconditions of a_i are checked against the state s_{i-1}. For every literal in the preconditions of a_i that matches a temporally annotated literal $\langle l_j^k, a_{j,j<i}, \Delta t_j^k \rangle \in s_{i-1}$, a temporal constraint[6] $[\Delta t_j^k, \infty)$ is posted between the producing start time point of the producing action, $start(a_j)$, and the start time point of the consuming action, $start(a_i)$. This means that the consumer action a_i can only be executed at least Δt_j^k time units after the action a_j which provides a_i's precondition l_j^k.

[6]The choice of using a half-open interval for an infinite interval is arbitrary. Throughout this work, closed intervals are used.

- The temporal links between tasks, as defined by the HTN methods, are passed to the STP whenever a task is decomposed by the HTN algorithm.
- An incremental path consistency algorithm (see Section A.4.2.2 and Algorithm A.4.2.2) enforces the consistency of the STP, and computes the equivalent minimal constraint network.
- When the HTN finds a total order plan π, the associated STP holds the corresponding partial-order plan. The latter complies with both the temporal links imposed by the task decomposition procedure, and those originating in mutual dependencies between preconditions and effects.

We adopt the basic SIADEX concept to enable partial-order plans in HTN planning. The idea can be sketched as follows:

1. Use partial-order forward decomposition to obtain a sequential plan. Note that in many planning domains most of the sequential ordering constraints have no semantic justification. They solely exist by virtue of the forward decomposition paradigm.
2. Drop all ordering constraints from the plan.
3. Add ordering constraints between actions where necessary:
   - Methods can define ordering constraints between sub tasks. These constraints must be enforced in the plan.
   - Let $\pi = \langle a_1, ..., a_n \rangle$ be an HTN plan. If the precondition of an action a_i depends on the effect of a previous action $a_{j,j<i}$, then we have to enforce the ordering constraint $[a_j \ \textit{before} \ a_i]$. This emulates the normal semantics of operator based planning.
   - Let $\pi = \langle a_1, ..., a_n \rangle$ be an HTN plan. If the precondition of an action a_j depends on the effect of a subsequent action $a_{i,i>j}$, then we have to enforce the ordering constraint $[a_j \ \textit{before} \ a_i]$. This is to ensure that a_i is executed *after* a_j and thus its effect cannot invalidate the precondition of a_j.

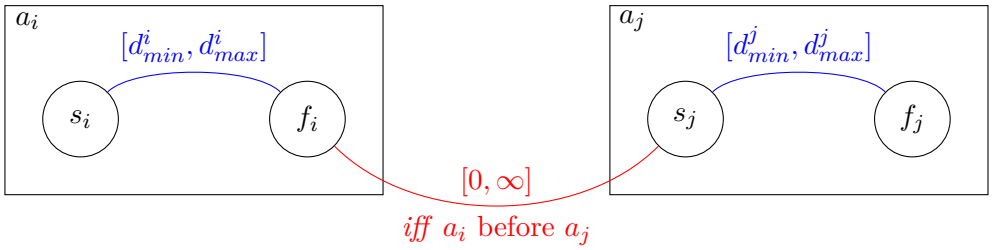


Figure 3.3: Detailed STP structure of HTN actions. Two actions a_i and a_j with their respective start and end time points, and a qualitative ordering constraint $[0, \infty]$ (red) that orders a_j after a_i. The duration of the actions is given by the temporal constraint $[d_{min}, d_{max}]$ (blue) between the start and end time points.

As mentioned in the introduction of this section, we use an STP data structure to capture not only qualitative ordering constraints, but also quantitative temporal relations. Figures

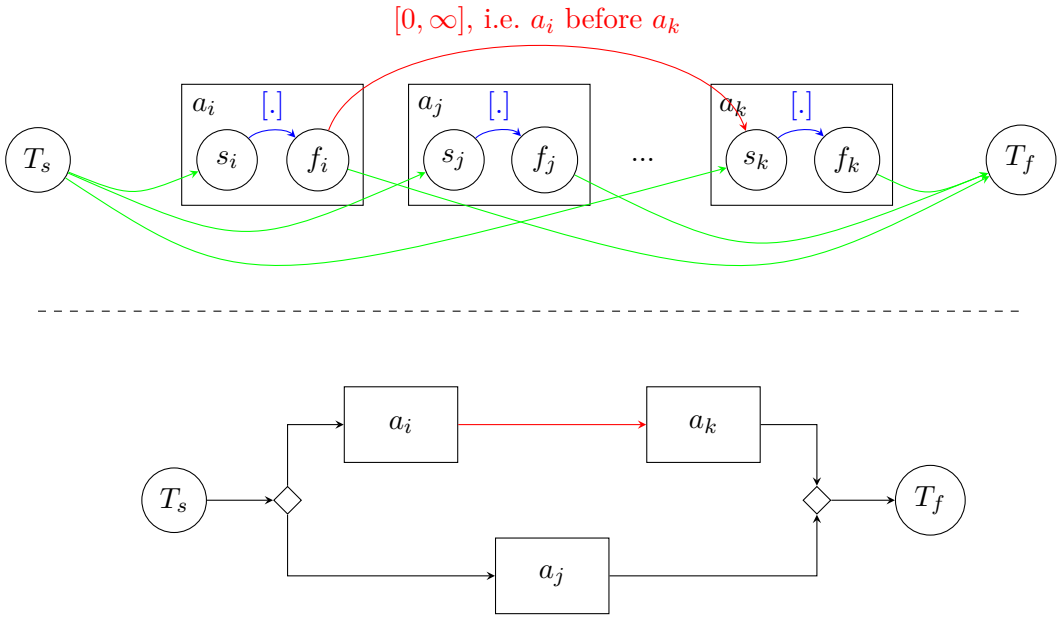


Figure 3.4: STP and workflow representations of HTN plans. The upper figure shows an STP with global start and end time points T_s and T_f, three actions $a_{i,j,k}$ and a qualitative ordering constraint (red) between a_i and a_k. The lower figure shows the corresponding workflow.

3.3 and 3.4 illustrate how a solution plan, the associated STP, and the induced workflow are related. Figure 3.3 depicts two actions, a_i and a_j together with their internal and mutual temporal structure. Every action a_k is represented by two time points, s_k and f_k, denoting the start and end time point of the action a_k, respectively. The duration of the every action a_k is constrained by $[d^k_{min}, d^k_{max}]$, which allows a_k to last between d^k_{min} and d^k_{max} time units. A qualitative ordering constraint $[a_i \ \textit{before} \ a_j]$ is stated by a temporal constraint $[0, \infty]$ between the end time point f_i of action a_i and the start time point s_j of action a_j. Similarly, a constraint $[a, b]$ between f_i and s_j allows action a_j to start between a and b time units after action a_i has finished. Figure 3.4 shows an STP with global start and end time points T_s and T_f, and three actions a_i, a_j and a_k. Action a_k is ordered *after* action a_i. The lower diagram displays the workflow induced by the STP; observe that action a_j is temporally unconstrained with respect to a_i and a_j, just as the STP suggests.

3.4.2. Integration of temporal reasoning and HTN planning

As the change plan is being refined by the HTN algorithm, temporal constraints are continuously added to the STP. Every plan π has an associated STP with start and end time points, T_s and T_f, and for every operator $o_i \in \pi$ there are two additional time points, s_i (*start*) and f_i (*finis*), representing the start and the end time points of the respective operator. Every time a task is decomposed by a method, temporal constraints between its sub tasks and the remaining tasks are posted (Algorithm 1, line 30) to enforce the ordering structure of the

sub workflow as encoded in the decomposition methods. Whenever a task is accomplished by an operator, the task's temporal properties are inherited by the operator (Algorithm 1, line 20). A durative action a_i with duration d is encoded by the constraint $[d, d]$ between the time points s_i and f_i; a task τ with a deadline t generates the constraint $[0, t]$ between the time points T_s and τ_s. Algorithm 3 details the decomposeTo() method used to pass temporal properties from tasks to sub planning elements (i.e. tasks or actions, see UML diagram in Figure 2.1 in Section 2.2).

Algorithm 3 The decomposeTo() method for task decomposition and forwarding of temporal constraints in an STP. In a given STP *stp*, decompose the task p into sub planning elements (actions or tasks) *into* and add additional ordering constraints c between elements of *into*. A qualitative ordering constraint $[0, \infty]$ between two time points t_i and t_j is displayed as $t_i \rightarrow t_j$, the subscripts s and f denote the start and end time points of planning elements.

```
1: function DECOMPOSETO(PlanningElement p, PlanningElement[] into, OrderingConstraint[] c)
2:
3:     for all PlanningElement e ∈ into do
4:         stp.add(e_s, e_f)                          ▷ add start and end time points
5:         stp.addConstraint( (p_s → e_s) )           ▷ e must start after the decomposed task p
6:         stp.addConstraint( (e_f → p_f) )           ▷ e must end before the decomposed task p
7:
8:     for all OrderingConstraint (p^1 → p^2) ∈ c do
9:         stp.addConstraint( (p^1_f → p^2_s) )       ▷ add remaining ordering constraints
```

Furthermore, to enforce the correct ordering of an action a_i added to the preliminary plan π, and all previous actions $o_j \in \pi, j < i$, one has to consider the mutual dependencies between the precondition and effect of a_i, and the preconditions and effects of all previous actions $a_j \in \pi, j < i$: A qualitative ordering constraint is posted between a_i and all those $a_j \in \pi, j < i$ whose effects contribute to the preconditions of a_i, or vice versa. This ensures that

(i) a_i is ordered *after* those actions whose effects contribute in making its precondition true, and that

(ii) the effects of a_i cannot invalidate the preconditions of previous actions.

To allow for this mechanism, the knowledge base component must be able to detect causal dependencies between preconditions and effects on configuration items: Every action must be annotated with the CIs that its effects alters, and with the CIs that occur in its precondition.

After updating the STP with new time points or temporal constraints, a path consistency algorithm (Section A.4.2.2, Algorithm A.4.2.2) can be used to check the consistency of the STP[7], and to compute its equivalent minimal network. Checking the consistency of the STP for every potentially new action or method decomposition in the preliminary plan can be

[7]For STPs the path-consistency (Section A.4.2.2) algorithm is a sound and complete method to check the consistency of the constraint problem and compute the minimal equivalent constraint network [34, 35]. Note that for general constraint satisfaction problems, path consistency is not complete.

accomplished in $O(n^3)$ time, and thus helps to efficiently prune large chunks of the HTN search space and ensures that all plans are temporally valid.

3.5. Policy engine

The policy engine can be integrated into the HTN algorithm in the same fashion as the temporal reasoner. Using the isValid() interface

boolean isValid(KnowledgeBase kb, Plan p)

the planner inquires the policy engine in every planning step, if the intermediate change plan p and the preliminary knowledge base state kb comply with policies from the policy repository. Failure in the validation of such policies causes the planner to backtrack. The benefit of this approach is that any off-the-shelf rule engine, such as the Drools system [14], can be used as the policy engine.

The policy engine component is currently not implemented in ChangeRefinery, and also not contained in Algorithm 1.

4. The ChangeRefinery prototype

In this section, we present our proof-of-concept implementation of the change plan design framework which comprises the change planner, knowledge base, and change catalogue components. We start by explaining the simplifying assumptions we made on the nature of both the participating information sources, and the resulting change plans. Then, Section 4.2 explains and documents implementation details of the prototype basing on those assumptions; Section 4.4 gives examples and evaluations of our prototype's basic functionality based on the scenario presented in Section 4.3. In a second step (Chapter 5), we relax some of the assumptions to make towards a more realistic and expressive tool.

4.1. Simplifying assumptions

The implementation of our first ChangeRefinery prototype is based on the following simplifying assumptions on information sources and change plans.

1. Resulting workflows are sequential and workflow activities are atomic, i.e. they have no duration or other temporal structure. This assumption will be relaxed by the integration of the temporal reasoner component, as described in Section 5.1.

2. Workflow activities do not fail. This especially implies that we do not cover remediation or rollback plans. In [36], Machado *et al.* propose an ITIL-compliant model to support rollback in IT change management systems. The integration of such systems is out of scope of this work.

3. Change templates are correct and complete. We present a hybrid HTN and state-space based planning concept for adaptive workflow reparation in Section 5.4.6.

4. We have exact and complete knowledge of the environment; the CMDB is a consistent and correct model of the actual IT infrastructure. Also, no concurrent changes modify the knowledge base simultaneously. We show how to find repair activities to cope with these kinds of inconsistencies in Section 5.4.6, and mention concurrent changes in Section 5.4.8.

5. Dependencies between configuration items are not taken into account. We assume, that all dependencies (e.g. *the installation of the Apache web server requires libraries foo.lib and bar.lib*) are explicitly encoded in change recipes (i.e. HTN methods). In Section 5.4.1, we briefly describe how the ChangeLedge system could be integrated with our approach in order to further refine change plans with respect to dependencies between affected configuration items.

4.2. Prototypical implementation of the change refinement architecture

4.2.1. Change planner

Listing C.1 shows the da.planner.htn.HTN class containing the ChangeRefinery HTN implementation. As the basic functioning of the (non-deterministic) algorithm was already explained in Section 3.3.2, this section focusses on the transition to a deterministic version of Algorithm 1, and highlights implementation details on variable handling and user choice points.

4.2.1.1. The deterministic ChangeRefinery HTN algorithm

Algorithm 4 Non-deterministic algorithm for a depth-first search on a tree data structure.

```
1: function TreeSearch-NonDet(t:Tree, v:Value)
2:     if t.value = v then
3:         return true
4:
5:     if t.subTree ≠ ∅ then
6:         subTree ← non-deterministically pick sub tree from t.subTrees
7:         return TreeSearch-NonDet(subTree, v)
8:
9:     return false
```

Algorithm 5 Deterministic algorithm for a depth-first search on a tree data structure.

```
1: function TreeSearch-Det(t:Tree, v:Value)
2:     if t.value = v then
3:         return true
4:
5:     for all subTree ∈ t.subTrees do
6:         if TreeSearch-Det(subTree, v) then
7:             return true
8:
9:     return false
```

Before digging into the details of our HTN implementation, consider Algorithm 4 as example for a non-deterministic algorithm for depth-first search on a tree data structure, and its deterministic equivalent in Algorithm 5. In both examples, *Tree* is a recursive data structure containing an arbitrary number of sub trees of type *Tree* in *Tree.subTrees*, and a value in *Tree.value*. Algorithm 4 is a non-deterministic method to decide whether a given tree t contains a value v in one of its nodes. The algorithm non-deterministically picks *one branch* of the tree and looks for the value v in all nodes of the chosen branch. One possible design of an equivalent deterministic algorithm resolves the non-deterministic choice (line 6 in Algorithm 4) line into a *loop over all possible sub trees* (line 5 in Algorithm 5). The

non-deterministic algorithm is defined to return true *iff* there exists an execution path that returns true. Similarly, the deterministic algorithm returns true *iff* the disjunction of all sub execution branches on different sub trees is true (lines 5–7). The general pattern for the design of a deterministic algorithm from its non-deterministic counterpart is thus to *loop over the possible choices and merge the results of the individual computation branches.*

The HTN algorithm depicted in Algorithm 1 has three non-deterministic choice points: In line 6 one of the possible tasks without predecessors is selected, and in lines 11 and 13 one of the applicable refinements for this task and one the available variable bindings are chosen. These choice points are resolved to *forall* loops in lines 74, 79 and 83 of Algorithm C.1. Inside these loops, we have to distinguish between operator and method refinements: In case of an operator, we assert its effects to the knowledge base (line 122), add the corresponding action to the current plan (line 145), and recursively invoke the HTN algorithm (line 165). In case of an operator, we decompose the refined task by the method's sub tasks (line 188) and recursively call the HTN algorithm on the decomposed task network (line 202). The integration of the temporal reasoning component (STP) is explained in Section 5.1.

Algorithm 6 Deterministic algorithm for a depth-first search on a tree data structure, without local tree variables. It assumes a global variable t holding the tree data structure.

```
1:  function TreeHasValue-GlobalTree-Det(v:value)
2:      if t.value = v then
3:          return true
4:
5:      i: int                                                  ▷ local variable
6:      for i ← 0; i ≤t.subTree.count(); i ← i+1 do
7:          t ← t.subTree[i]                  ▷ navigate "down" in one of the branches
8:          if TreeHasValue-GlobalTree-Det(v) then
9:              return true
10:
11:         t ← t.parentTree                                ▷ navigate "up", i.e. rollback
12:
13:     return false
```

What makes the HTN algorithm much more complicated than the previously explained tree search, is the integration of the knowledge base component, which acts as simulation environment for the planning process. The effect of a chosen operator is applied to the knowledge base and *must be undone before the next operator is tried* (line 178). Equivalently, the operator must be removed from the current plan before the next operator is tried (line 174). This rollback concept can again be compared to a tree search in the following way: Assume that the tree variable is not passed to the search algorithm as a local variable, but is accessible as a global variable. Furthermore let each tree point to its parent tree via *Tree.parentTree*. Algorithm 6 shows the depth-first search procedure for this case: In line 7, one of the sub trees is assigned to the global variable t, and the search is invoked recursively on one of the sub trees (lines 7,8). After the recursive call returns, we have to navigate back to its parent node in order to point t to the correct place in the tree.

Because the computations and particular task, refinement and binding choices in different branches of the HTN planning tree and independent and thus may not interfere, we *copy*

the entire task network and the selected variable bindings before recursively calling the HTN algorithm (lines 108–111).

4.2.1.2. User choice points

As it is often infeasible (and undesired) to calculate all plans for an HTN planning problem, we offer the user to choose one of the available tasks, refinements, and variable bindings at every branching point (lines 74, 79, and 83 in Algorithm C.1). Every choice point has its associated ChoicePointFactory object (HTN.taskCPF, HTN.refinementCPF, HTN.bindingCPF), which implements the

ChoicePoint<T> create(List<T> l, String name)

method to create and return a ChoicePoint object from the given list of tasks, refinements or bindings, respectively. Every ChoicePoint object implements the interface Iterable<T>, and can thus easily be used to loop over all elements of the given list. Different implementations of the Iterable interface allow different functionality: A SimpleChoicePoint lets the user select one particular choice and asks if she wants to backtrack over the remaining choices. The RandomisedChoicePoint randomly picks a choice and stops backtracking with a certain (configurable) probability. We use the RandomChoicePoint to run automated performance tests, and the SimpleChoicePoint to offer full flexibility and user control over the planning process.

4.2.1.3. Variable handling

The Binding class (Figure 4.3) is used to represent variable bindings for tasks and result tuples of precondition queries. It extends the HashMap<String, Variable> interface, and thus maps variable names to variable instances. Variables can be of type KbElementVariable or PropertyVariable, which represent knowledge base CIs and their properties, respectively. Because task definitions and task instances are decoupled (Figure 4.3), a task definition can only refer to variables by their name, not by Java objects. Every task object carries a binding object (the binding is passed in the constructor) with its current variable bindings. The methods and operators are defined in the TaskDefinition class and can access variables by their names.

4.2.2. IT knowledge base

The ChangeRefinery implementation of the knowledge base component comprises mainly two parts:

(i) a model of the IT infrastructure (CMDB), and

(ii) the implementation of the assert(), rollback(), and query() interfaces to access and manipulate data stored in the knowledge base.

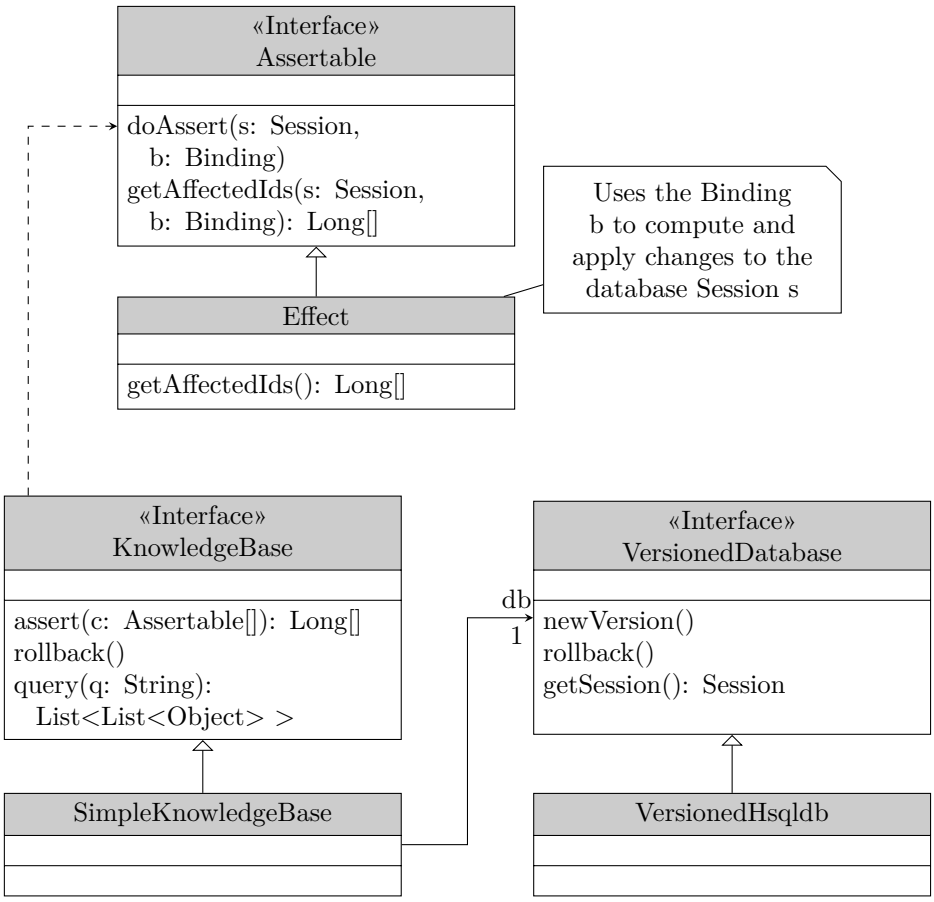


Figure 4.1: UML model of the knowledge base implementation. The SimpleKnowledgeBase class implements the assert(), rollback(), and query() interfaces, and uses a VersionedDatabase instance as database backend. Effects implement the Assertable interface and can thus be asserted the knowledge base via the assert() method.

While the implementation of the IT infrastructure model is described in Section 4.3.1, this section focusses on (ii). We chose to base our implementation on the Hibernate object-relational mapper [37]. Hibernate offers a transparent persistence service for Java objects; after annotating Java classes with mapping instructions, Hibernate takes care of synchronising instances of those classes with a relational database. The Hibernate Query Language (HQL) can be used to express queries against mapped objects, their properties, and interdependencies. HQL is fully object-oriented, understanding notions like inheritance and polymorphism. This aligns nicely with the strongly hierarchic nature of CIM, on which our IT infrastructure model is based. HQL is thus the main concept promoting the implementation of the query() interface.

We briefly examined alternative available frameworks as basis for our implementation. Table 4.1 compares those with respect to our requirements. As we intended to use an object-oriented implementation, and due to the lack of documentation for the EMF framework, we decided to use Hibernate.

	Versioning	Querying	Modelling	Planner integration	Documentation
EMF	•••	••	•••	•••	••
Hibernate	•	•••	•••	•••	•••
Prolog	••	••	••	•	•••
Jena	•••	••	••	•	••

Table 4.1: Requirements for the knowledge base technology. The Eclipse Modeling Framework Project (EMF) is an object-oriented framework for persistent data modelling and querying. Its main drawback is (as of today) the lack of usable documentation. Prolog is weak on object-oriented modelling. Jena [38] is a Java-based semantic web toolkit. Although having chosen to use Hibernate for our first prototype, we are convinced that semantic web technologies (especially OWL) are a promising approach to build a powerful knowledge base backend integrating object-oriented with logic concepts. This idea is further explained in Section 5.4.9.

Figure 4.1 shows a UML diagram of the main classes and methods constituting the knowledge base component. The KnowledgeBase interface defines the assert(), rollback(), and query() methods, which are implemented in the SimpleKnowledgeBase class. To support backtracking, the knowledge base makes use of a versioned database component, as abstracted by the VersionedDatabase class and implemented by the VersionedHsqldb class. The following sections detail the implementation of the rollback(), assert(), and query() interfaces.

4.2.2.1. The rollback() interface

The assert() and rollback() mechanisms of logic-based knowledge bases for common planners are quite straightforward to implement: Because a knowledge base state is given by a set s of logical atoms and asserting the effect of an operator comprises

(i) adding the atoms in the $^{+}$effects set to s, and

(ii) removing the atoms in the $^{-}$effects set from s (see Appendix A).

Consequently, memorising the $^{\pm}$effects is sufficient to support the rollback to a previous state, which can be accomplished by simply *removing* the atoms in the $^{+}$effects set, and *adding* the atoms in the $^{-}$effects set; assert() and rollback() are their mutual inverse functions.

The above is not applicable in our object-oriented and Java-based knowledge base implementation, as arbitrary Java code is not trivially invertible. Instead, the ChangeRefinery prototype *keeps a list of copies of every knowledge base state*. Asserting thus means to append a clone of the latest state to the list of copies and applying changes to that clone; rollback means to switch to an earlier copy in the list. Hibernate uses a relational database as storage backend, and to facility the copying of databases, we chose the HSQLDB [39] Java SQL database. HSQLDB can be configured to store the database in a single file. The implementation of the newVersion() method of the VersionedHsqldb class thus performs the following steps:

(i) shut down the old HSQLDB daemon

(ii) copy the current database file *file_i* to a new file *file_i+1*, and

(iii) start a new HSQLDB daemon on the new database file.

Equivalently, rollback() comprises

(i) shuting down the old HSQLDB running on database *file_i*, and

(ii) starting a new HSQLDB daemon on the previous database file *file_i-1*.

Although developing a versioned database is definitely out of scope of this thesis, we do consider the current implementation the central weak point of the ChangeRefinery prototype. Experimental results (see Section 5.4.3) show that every rollback() or assert() invocation takes approximately 100-200 milliseconds, and that more than 99% of the total planning time is spent on those operations. Most time is used for shutting down and starting a new instance of the HSQLDB server daemon, which has to be done whenever the database version is to be changed, i.e. in every rollback() or assert() call.

4.2.2.2. The assert() interface

Because the IT infrastructure model is implemented as plain Java code, the assert() interface is fairly easy to realise. Persistent Hibernate objects can be accessed through the load() method of an instance of the org.hibernate.Session class; new instances can be persisted by invoking the save() method. The assert() method of the KnowledgeBase interface takes as parameter a collection of Assertables, which have to implement the

```
Set<Long> doAssert(Session s, Binding b)
```

method. By accessing the Session parameter, the Assertable has full control over all Java objects in the knowledge base. The assert() method returns a set of Long objects, which are the IDs of the knowledge base objects affected by the assertion. The reason for this will be made clear in Section 5.2 on additional ordering constraints due to interdependencies between effects and preconditions of operators.

Every assertion creates a new version of the database backend by calling the newVersion() method of the VersionedDatabase object before accessing the Hibernate session.

4.2.2.3. The query() interface

By virtue of the Hibernate Query Language (HQL) support, the query() interface is trivially easy to implement. The query method

List<List<Object>> query(String q)

takes a query string in HQL syntax and returns tuples of matching knowledge base objects.

The BindingGenerator interface and its adjacent classes abstract from the HQL syntax, and are used to build knowledge base queries which act as preconditions for operators and methods (see Figures 4.2, 4.3 and Section 4.2.3). The BindingGenerator interface defines the getBindings() method, which is invoked by the HTN algorithm to compute and retrieve matching variable bindings for a refinement. The KnowledgeBase parameter specifies the knowledge base against which the query is to be computed, and the Binding[] parameter passes existing variable bindings for variables in the operator or method. The getBindings() method returns a list of found variable bindings.

The BindingGenerator interface is implemented by two classes, EmptyPrecondition and KbPrecondition. The EmptyPrecondition class generates a query which is always true and return one empty binding. It is used for refinements without preconditions and variables. The KbPrecondition class can be used to build knowledge base queries and bind variables. It maintains the following types of condition specifications:

1. ObjectBindingConstraint[] specifies query conditions on bound or unbound variables. If a variable is unbound, no constraint is added to the query. If a variable is bound, the variable identity is added to the query condition.

2. ObjectPropertyConstraint[] specifies constraints on properties of CIs.

3. unboundKbElements is a list of variables which *must be unbound*. This constraint is used to ensure in a precondition of operators creating new knowledge base CIs, that its effects do not change the assignments of already bound variables.

The Formula interface and its sub classes are used to actually build the query from the various constraints. The structure follows the usual recursive schema to construct propositional formula. The sub classes of the Atom class implement the concrete query translation into the Hibernate Query Language (HQL). The following enumeration details the semantics of the individual classes. At this, $*varName* loosely denotes the variable identifier of the variable named *varName*.

1. ObjectBindingConstraint: If variable vName is bound to a knowledge base object with ID=*id*, this binding is preserved by requiring *$vName.id=id*. If vName is unbound, the always true condition (*1=1*) is applied.

2. ObjectIdentityConstraint: Requires that v1 and v2 refer to the same objects. This translated to *$v1.id = $v2.id*.

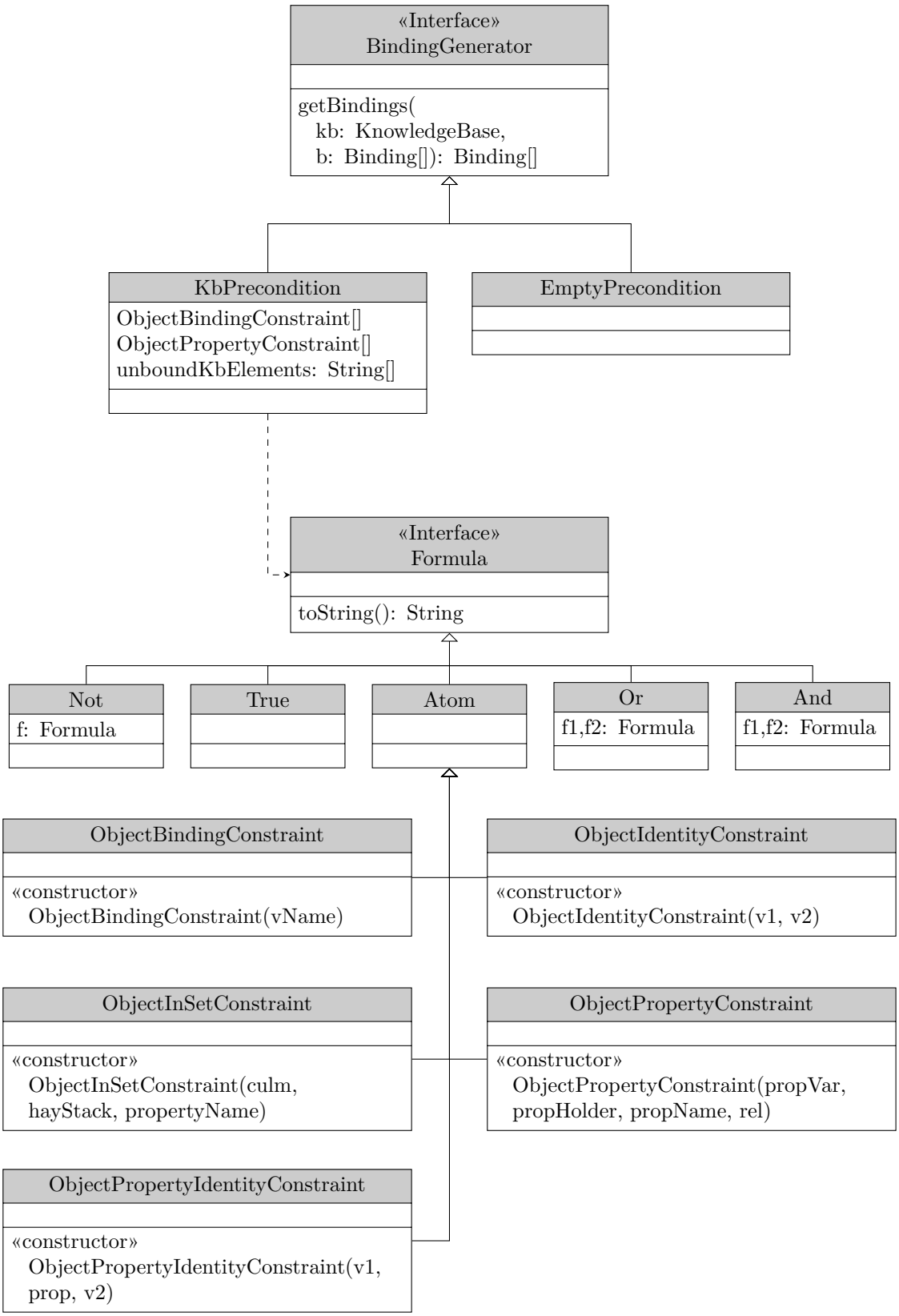


Figure 4.2: UML model of the BindingGenerator query building mechanism. Knowledge base queries are represented by the BindingGenerator interface; bindings are computed and returned in the getBindings() method. See text for a detailed description of the Formula concept.

3. ObjectInSetConstraint: True, if culm ∈ hayStack.propertyName, which translates to *$culm in elements($hayStack.$propertyName)*. This query is used to impose conditions on associations with cardinality * (e.g. the *databases* property of the *DbServer* class).

4. ObjectPropertyConstraint: True, if the propHolder.propName stands in relation *rel* to the variable propVar. Translates to *$propHolder.$propName $rel $propVar*.

5. ObjectPropertyIdentityConstraint: Requires that the prop property of variable v1 is identical to v2: *$v1$.prop = $v2*. This query is used to impose conditions on associations with cardinality 1 (e.g. the *database* property of the *J2eeJdbcResource* class).

The Java code for the ObjectInSetConstraint class is given in Listing 4.1.

Listing 4.1: Java code of the ObjectInSetConstraint class

```
1  package da.planner.htn.query;
2
3  import da.planner.htn.Binding;
4  import da.planner.htn.KbElementVariable;
5
6  public class ObjectInSetConstraint extends Atom {
7    private String culm;
8    public String getCulm(){return this.culm;}
9    private String hayStack;
10   public String getHayStack(){return this.hayStack;}
11   private String propertyName;
12   public String getPropertyName(){return this.propertyName;}
13
14   public ObjectInSetConstraint(String culm, String hayStack, String
       propertyName){
15     this.culm = culm;
16     this.hayStack = hayStack;
17     this.propertyName = propertyName;
18   }
19
20
21   public String getStringRepresentation(Binding b) {
22     KbElementVariable n = b.get(this.culm, KbElementVariable.class);
23     KbElementVariable h = b.get(this.hayStack, KbElementVariable.class);
24
25     return n.getQueryName()
26       + " in elements(" + h.getQueryName() + "." + this.propertyName + "
           )";
27   }
28
29 }
```

As an example, consider the precondition of the MigrateDatabase method of the SpeedUpWebApplication task (Listing 4.2). The query enforces existing bindings for three knowledge base variables (application, olddb, dbresource), and additionally imposes the constraint *dbresource ∈ application.j2eeJdbcResources AND dbresource.database = olddb*. The translated

HQL query shows the case of unbound olddb, and dbresource variables and the application variable bound to the CI with id 8.

Listing 4.2: Java code and HQL translation for the precondition specification of the MigrateDatabase method of the SpeedUpWebApplication task

```
1 KbPrecondition p = new KbPrecondition();
2
3 p.addKbElementConstraint("application");
4 p.addKbElementConstraint("olddb");
5 p.addKbElementConstraint("dbresource");
6 p.setAdditionalQueryConstraints(new And(
7   new ObjectInSetConstraint("dbresource", "application", "
        j2eeJdbcResources"),
8   new ObjectPropertyIdentityConstraint("dbresource", "database", "olddb"
        )
9 ));
10
11 // HQL translation: select new list(J2eeApplication0,Database4,
     J2eeJdbcResource5) from J2eeApplication as J2eeApplication0,Database
     as Database4,J2eeJdbcResource as J2eeJdbcResource5 where (
     J2eeApplication0.id=8 and 1=1 and 1=1 ) and (J2eeJdbcResource5 in
      elements(J2eeApplication0.j2eeJdbcResources) and J2eeJdbcResource5.
     database = Database4)
```

4.2.3. Change catalogue repository

The change catalogue repository stores change recipes and makes them available to the planner component (see Figure 3.1 and Section 3.2). As we model change tasks and their refinement to sub tasks in alignment with the HTN planning paradigm, the basic structure of the change catalogue data model (Figure 4.3) is built according to this scheme: The change catalogue is basically a collection of task definitions. Instances of the TaskDefinition class represent static definitions of tasks and their available refinements, methods, and operators. Instances of the Task class represent run-time task objects in the planning process. Every task object represents a task definition and owns a set of bindings for its variables. The Binding class is used to encapsulate a set of variable names together with their instance values. Variables can represent either knowledge base objects (CIs) or their properties, which is represented by the KbElementVariable and PropertyVariable class, respectively.

A method is defined by instantiating an anonymous sub class of the Method class, and implementing the abstract getSubNetwork() method. The method's sub task network is specified by instantiating a new TaskNetwork object and adding tasks and ordering constraints; variable bindings from the Binding parameter can be used to pass variable bindings to sub tasks. The returned task network is used by the perform() method in the HTN algorithm to decompose a task by a sub task network.

An operator is defined by instantiating an anonymous sub class of the Operator class. The constructor of this sub class typically adds Effect objects, which specify the operator's effect on the knowledge base. The sub classes of Effect accomplish different changes in the

knowledge base: AddManagedElement creates a new CI, AddToAssociation and RemoveFromAssociation add or remove links from one CI to another CI, IncrementProperty and SetProperty change the property values of CIs.

Every refinement (i.e. method or operator) can define its precondition query be setting an appropriate BindingGenerator (see Figure 4.2 and Section 4.2.2.3).

Java examples for task definitions can be found in Section 4.3.2 and Listing C.4.

4.3. The J2EE scenario

4.3.1. IT infrastructure components

Figure 4.4 shows a UML diagram of the J2EE scenario IT infrastructure. It is roughly based on the object hierarchy and naming conventions from the Common Information Model (CIM); ManagedElement is the root class for all entities in the infrastructure. Our scenario comprises WebServer, LoadBalancer, and DbServer classes, which are sub classes of Machine and share the cpuSpeed and memorySize properties. A DbServer can host a Database, which can be either a OracleDb or a MySqlDb. WebServers can host J2eeContainer applications, which run J2eeApplications and have J2eeModules. All Java entities have the JavaApplication super class which itself inherits from ManagedElement. A J2eeApplication can be connected to a Database by a J2eeJdbcResource. Additionally, we model IT technicians by the Human class and their skills by the Skill class.

Listing C.2 shows a very simple instantiation of this data model. It defines a DbServer, two WebServers, some J2eeApplications and one technician together with her skills. A bigger scenario is given in the da.scenarios.j2ee.kb.J2eeScenario class (Listing C.3).

4.3.2. Change catalogue

This section lists the task in the J2EE scenario together with their respective methods (with sub tasks) and operators (with preconditions and effects). A variable assignment $[var1 \leftarrow var2]$ in a sub task st of a method definition related to a task t means that the variable $var1$ of sub task st is assigned the value of variable $var2$ of task t.

Java code for the listed tasks can be found in the package da.scenarios.j2ee.domain. Listing C.4 shows the Java code for the SpeedUpWebApplication task definition and its refinements. Methods and operators are added to the task definition by invoking the addRefinement() method on an anonymous sub class of the Method or Operator class. Method sub classes have to implement the getTaskNetwork() method to define how it decomposes a given task into sub tasks. Operators define their effects by invoking their addEffect() method on instances of the Effect class. Both operators and methods configure their precondition query by setting an appropriate BindingGenerator.

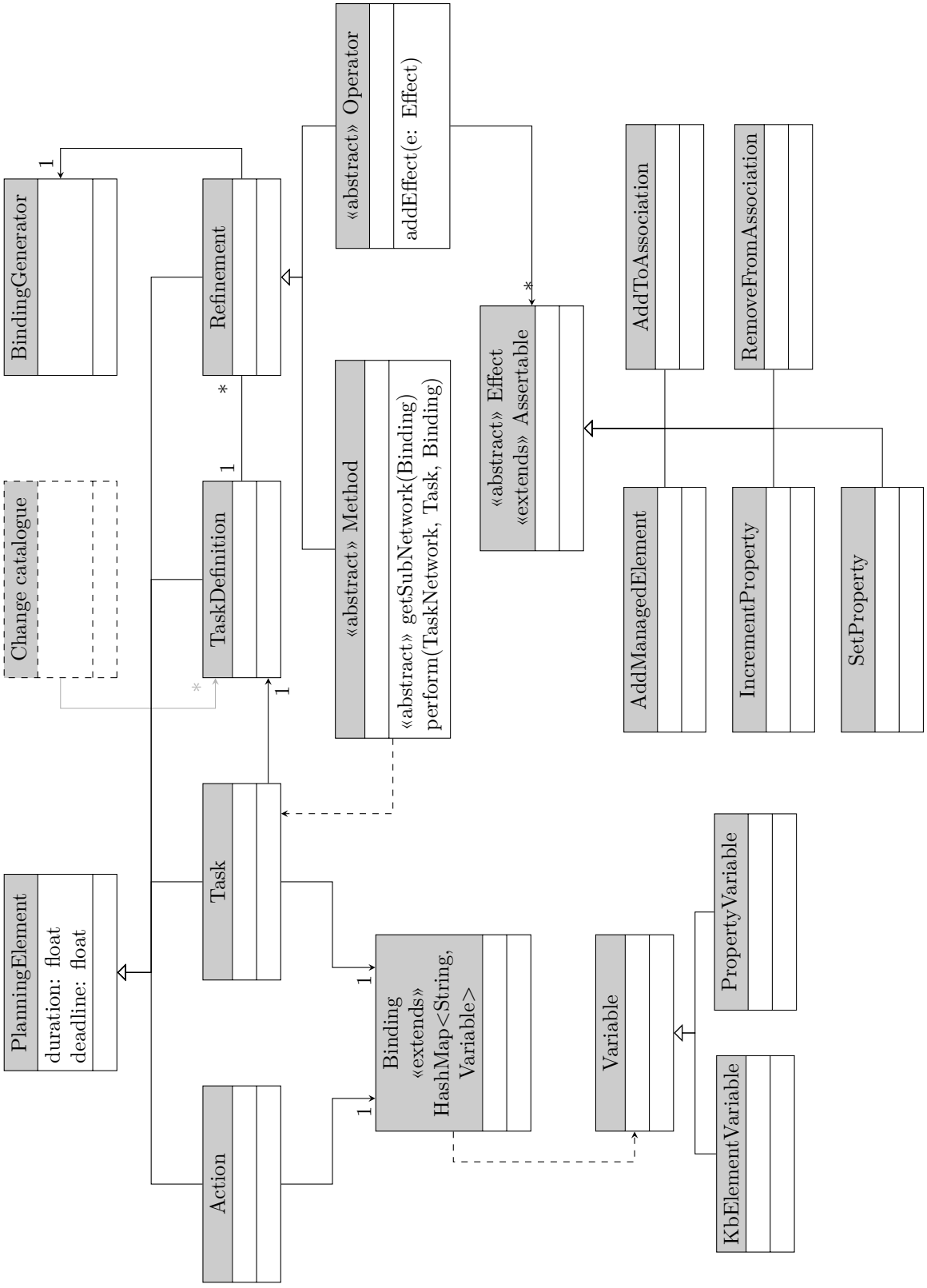


Figure 4.3: UML model of the change catalogue implementation.

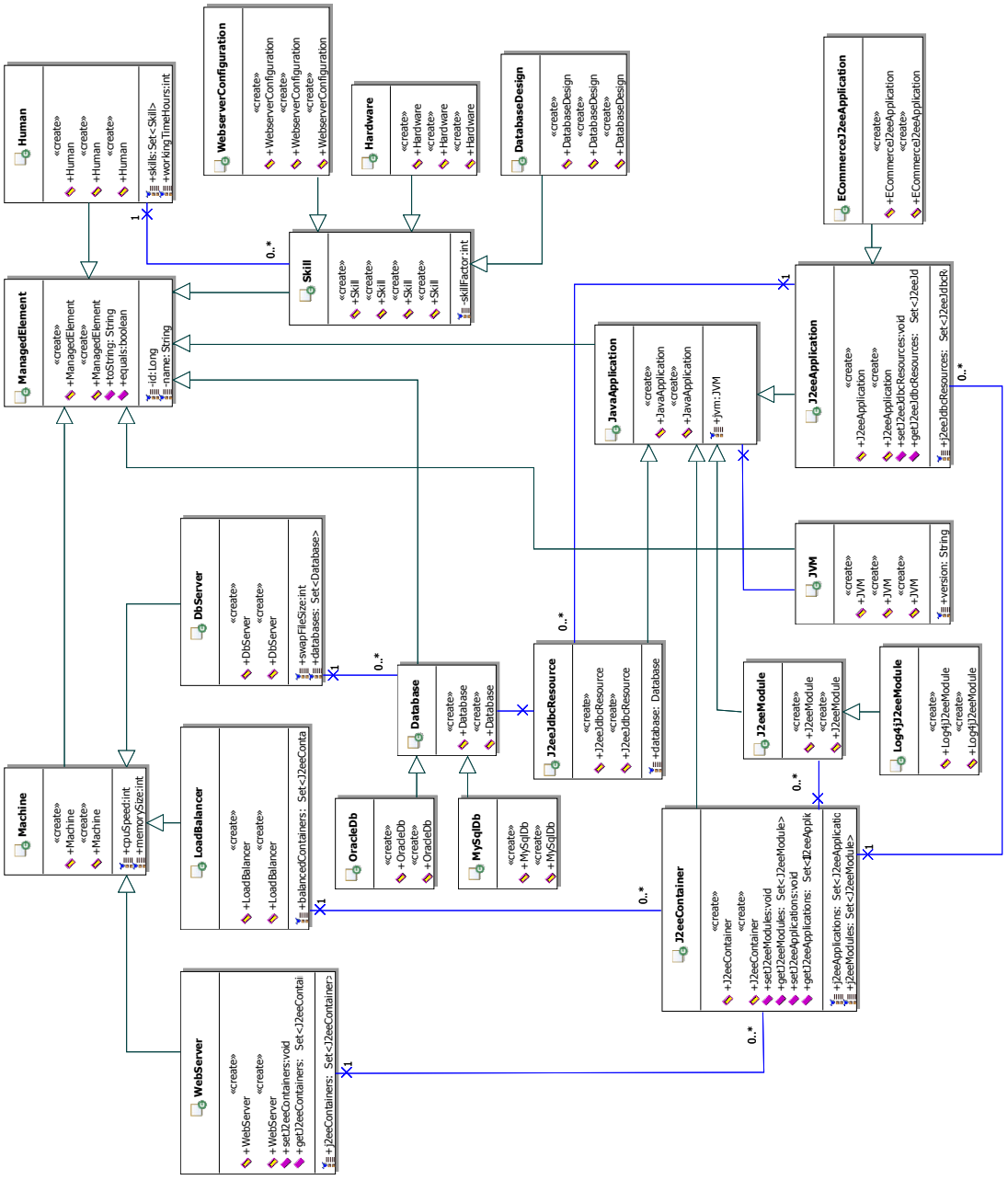


Figure 4.4: UML model of the IT infrastructure scenario. See Appendix C for a bigger version of the figure.

4.3.2.1. Task: SpeedUpWebApplication

Variables	J2eeApplication application
	LoadBalancer lb
	J2eeContainer jc
	WebServer ws, newws
	DbServer dbserver
	Database olddb, newdb
	J2eeJdbcResource dbresource
	Human expert
	Skill skill
Method	EnableLoadBalancing
Precondition	application is a CI
Sub tasks	InstallLoadBalancer(lb ← lb)
	InstallJ2eeServer(server ← newws, container ← jc)
	InstallJ2eeApplication(cont ← jc, app ← application)
	AddContainerToLoadbalancer(container ← jc, lb ← lb)
Method	MigrateDatabase
Precondition	application, dbserver, olddb, dbresource are CIs
	dbresource ∈ application.j2eeJdbcResources
	dbresource = database.olddb
Sub tasks	BackupDatabase(db ← olddb)
	InstallDatabase(dbServer ← dbserver, db ← newdb)
	CopyDatabaseContent(fromDb ← olddb, toDb ← newdb)
Operator	AskWebApplicationExpert
Duration	10
Precondition	application, expert, skill are CIs
	skill ∈ expert.skills
Effects	∅
Method	UpgradeWebServerHardware
Precondition	application, ws, jc are CIs
	jc ∈ ws.j2eeContainers
	application ∈ jc.j2eeApplications
Sub tasks	UpgradeHardware(machine ← ws)
Method	UpgradeDatabaseServerHardware
Precondition	application, dbserver, olddb, dbresource are CIs
	dbresource ∈ application.j2eeJdbcResources
	dbresource = database.olddb
	olddb ∈ dbserver.databases
Sub tasks	UpgradeHardware(machine ← dbserver)

Method	UpgradeDatabaseServerAndWebServerHardware
Precondition	application, dbserver, olddb, dbresource, ws, jc are CIs
	dbresource ∈ application.j2eeJdbcResources
	dbresource = database.olddb
	olddb ∈ dbserver.databases
	jc ∈ ws.j2eeContainers
	application ∈ jc.j2eeApplications
Sub tasks	UpgradeHardware(machine ← dbserver)
	UpgradeHardware(machine ← ws)

4.3.2.2. Task: UpgradeECommerceApplication

Variables	ECommerceJ2eeApplication application, newApplication
	WebServer ws
	J2eeContainer jc
	Log4jJ2eeModule mod
	Integer mem, cpu

Method	DefaultECommerceUpgrade
Precondition	application, ws, jc are CIs
	jc ∈ ws.j2eeContainers
	application ∈ jc.j2eeApplications
Sub tasks	UpgradeHardware(machine ← ws, mem ← mem, cpu ← cpu)
	InstallECommerceApplicationVersion5(app ← newApp, cont ← jc, log4jmodule ← mod)
	InstallJ2eeModule(module ← mod, container ← jc)

4.3.2.3. Task: InstallECommerceApplicationVersion5

Variables	J2eeContainer cont
	Log4jJ2eeModule log4jmodule
	ECommerceJ2eeApplication app

Operator	InstallECommerceApplicationVersion5-Script
Duration	10
Precondition	cont, log4jmodule are CIs
	app is unbound
	log4jmodule ∈ cont.j2eeModules
Effects	add new app CI
	add app to cont.j2eeApplications

4.3.2.4. Task: InstallJ2eeModule

Variables	J2eeContainer container J2eeModule module
Operator	InstallJ2eeModule-Script
Duration	5
Precondition	container is a CI module is unbound
Effects	add new module CI add module to container.j2eeModules

4.3.2.5. Task: AddContainerToLoadbalancer

Variables	J2eeContainer container LoadBalancer lb
Operator	AddContainerToLoadBalancer-Script
Duration	5
Precondition	container, lb are CIs container $\notin$ lb.balancedContainers
Effects	add container to lb.balancedContainers

4.3.2.6. Task: BackupDatabase

Variables	Database db
Operator	RunBackupScript
Duration	50
Precondition	db is a CI
Effects	$\emptyset$

4.3.2.7. Task: CopyDatabaseContent

Variables	Database fromDb, toDb
Operator	CopyDbScript
Duration	40
Precondition	fromDb, toDb are CIs fromDb $\neq$ toDb
Effects	$\emptyset$

4.3.2.8. Task: InstallDatabase

Variables	DbServer dbServer Database db DbServer dbServer Integer mem
Operator	InstallNewDatabaseOnNewDbServer-Script
Duration	5
Precondition	dbServer, db are unbound
Effects	create dbServer and db CIs add db to dbServer.databases
Operator	InstallNewDatabaseOnExistingDbServer
Duration	10
Precondition	dbServer is a CI db is unbound
Effects	create db CI add db to dbServer.databases
Method	InstallNewDatabaseOnSpecialDbServer
Precondition	
Sub tasks	SetupServer(server ← dbServer, mem ← mem) InstallDatabaseSoftware(dbserver ← dbserver, db ← db)

4.3.2.9. Task: InstallDatabaseSoftware

Variables	DbServer dbserver Database db
Operator	InstallMySqlDb-Script
Duration	15
Precondition	dbserver is a CI db is unbound
Effects	create db CI add db to server.databases
Operator	InstallOracleDb-Script
Duration	20
Precondition	dbserver is a CI db is unbound
Effects	create db CI add db to server.databases

4.3.2.10. Task: InstallJ2eeApplication

Variables	J2eeContainer cont J2eeApplication app
Operator	InstallJ2eeApplication-Script
Duration	10
Precondition	cont, app are CIs app $\notin$ cont.j2eeApplications
Effects	add app to cont.j2eeApplications

4.3.2.11. Task: InstallJ2eeContainerSoftware

Variables	DbServer webserver J2eeContainer cont
Operator	InstallJ2eeContainerSoftware-Script
Duration	10
Precondition	webserver is a CI cont is unbound
Effects	create cont CI add cont to webserver.j2eeContainers

4.3.2.12. Task: InstallJ2eeServer

Variables	WebServer server J2eeContainer container
Method	DefaultJ2eeServerInstallation
Precondition	$\emptyset$
Sub tasks	SetupServer(server $\leftarrow$ server, mem $\leftarrow$ 1000) InstallJ2eeContainerSoftware(webserver $\leftarrow$ server, cont $\leftarrow$ container)

4.3.2.13. Task: InstallLoadBalancer

Variables	LoadBalancer lb
Method	LoadbalancerOnServer
Precondition	
Sub tasks	SetupServer(server ← lb)
Operator	InstallLoadBalancer-Script
Duration	10
Precondition	lb is unbound
Effects	add lb CI
Operator	UseExistingLoadBalancer
Duration	1
Precondition	lb is CI
Effects	∅

4.3.2.14. Task: SetupServer

Variables	Machine server
	Integer mem
Operator	InstallVirtualServerByScript
Duration	10
Precondition	server is unbound
Effects	add server CI
Operator	UseExistingServer
Duration	1
Precondition	server is a CI
	server.memorySize $\geq$ mem
Effects	∅
Operator	UseLowEndServer
Duration	10
Precondition	server is a CI
	server.memorySize $\leq$ mem
Effects	∅

4.3.2.15. Task: UpgradeHardware

Variables	Machine machine Integer mem, cpu
Operator	UpgradeMem
Duration	15
Precondition	machine is a CI
Effects	machine.memorySize += mem
Operator	UpgradeCpu
Duration	15
Precondition	machine is a CI
Effects	machine.cpuSpeed += cpu
Operator	UpgradeMemAndCpu
Duration	25
Precondition	machine is a CI
Effects	machine.cpuSpeed += cpu machine.memorySize += mem

4.4. Evaluation of ChangeRefinery's basic capabilities

This section gives examples of the basic ChangeRefinery capabilities. The evaluation of more advanced features, such as temporal reasoning, and the illustration of the refinement of a more complex workflow can be found in Section 5.3.

4.4.1. Preconditions and variable bindings as CMDB queries

ChangeRefinery output: Listing C.5
Domain: J2eeScenario(5), Listing C.3
Goal task: SpeedUpWebApplication

The preconditions of the UpgradeWebServerHardware method for the SpeedUpWebApplication goal task constrain its applicability to "all J2eeApplications that are deployed on a J2eeContainer which is hosted on a WebServer". Lines 14-24 in Listing C.5 show the matching tuples of CMDB CIs. Variables in the remaining task network are bound according to the user's choice, which is reflected in the found change plan: In line 25, binding [0] with WebshopServer5 was chosen, and the final plan contains the action UpgradeMemAndCpu(machine ← WebshopServer5). As described earlier, preconditions of methods and operators are used to constrain their applicability to CMDB states and to bind variables to CMDB CIs.

4.4.2. Additional ordering constraints due to failed preconditions

ChangeRefinery output: Listing C.9
Domain: VerySimpleJ2eeScenario, Listing C.2
Goal task: UpgradeECommerceApplication

Listing C.9 shows all possible plans for the goal task UpgradeECommerceApplication in the scenario VerySimpleJ2eeScenario. The precondition *log4jmodule* $\in$ *cont.j2eeModules* (see Section 4.3.2) of the InstallECommerceApplicationVersion5-Script operator requires the Log4j module to be installed on the application container. As this is not the case in the initial state of the VerySimpleJ2eeScenario scenario, the InstallJ2eeModule-Script action is ordered before the InstallECommerceApplicationVersion5-Script action, because this is the only ordering that satisfies this preconditions.

4.4.3. Reuse of change recipe information and granularity of plan refinement

ChangeRefinery output: %
Domain: %
Goal tasks: SpeedUpWebApplication, UpgradeECommerceApplication

Task, methods, and operators can be reused to refine different goal tasks; the granularity of change plan refinement can be easily extend from rather abstract actions (e.g. the AskWebApplicationExpert operator for the SpeedUpWebApplication task) to specialised and very detailed actions (e.g. UpgradeMem and UpgradeCpu operators for the UpgradeHardware task).

As an example for change recipe reuse, consider the high-level goal tasks SpeedUpWebApplication and UpgradeECommerceApplication (see Section 4.3.2), which both use the UpgradeHardware sub task. The SpeedUpWebApplication goal task can be trivially accomplished by the single action AskWebApplicationExpert, or refined to a more complicated task network by the EnableLoadbalancing or MigrateDatabase methods.

5. Towards an advanced prototype

This chapter describes the temporal reasoning capabilities of ChangeRefinery , and explains them with examples (Section 5.3). Additionally, Section 5.4 briefly describes other possible but so far unimplemented extensions that we have been investigating (Section 5.4).

5.1. Quantitative temporal information: Durative actions and deadline tasks

As detailed in Section 3.4, the ChangeRefinery prototype uses the *Simple Temporal Problem* (STP) concept for temporal reasoning. Figure 5.1 depicts a UML diagram of the data model used to represent the STP. The classes STP, StpNode and TemporalRelation are a straightforward and independent implementation of the STP concept (Section A.5). Every time point is an instance of the StpNode class and a binary temporal constraint $[a', b']$ between two time points stpNode_1 and stpNode_2 is an instance of the TemporalRelation class with $[a \leftarrow a']$, $[b \leftarrow b']$, [from $\leftarrow \text{stpNode}_1$], and [to $\leftarrow \text{stpNode}_2$]. Nodes and constraints can be added to the STP via its add() and setConstraint() methods, respectively.

The HTN algorithm does not use the STP-related StpNode and TemporalRelation classes, but its own PlanningElement and OrderingConstraint objects: Instances of PlanningElement[1] are qualitatively ordered by instances of the OrderingConstraint class. The StpHtnConnector class functions as adapter between the HTN and the STP "world": Every PlanningElement has two associated STP time points, the start and the end time point. The global *init* and *finis* time points are created in the StpHtnConnector constructor. The add() method of the StpHtnConnector class can be used to add to the STP the set p of PlanningElements with a partial order as specified by the set c of OrderingConstraints. Every PlanningElement in p generates two StpNode instances in the STP and every OrderingConstraint between two PlanningElements in p generates the temporal constraint $[0, \infty]$ between the respective start and end time points.

Whenever a task is decomposed into sub tasks by the HTN algorithm, the decomposeTo() method (see Algorithm 3 in Section 3.4) is invoked. Intuitively, the decomposeTo() method translates the HTN δ-function for task decomposition to the STP data model and has the following effects:

- Add all PlanningElements in *into* to the STP (i.e. add start and end nodes for all of them).

[1] PlanningElement is the super class of tasks, operators and actions, see UML diagram in Figure 2.1 in Section 2.2.

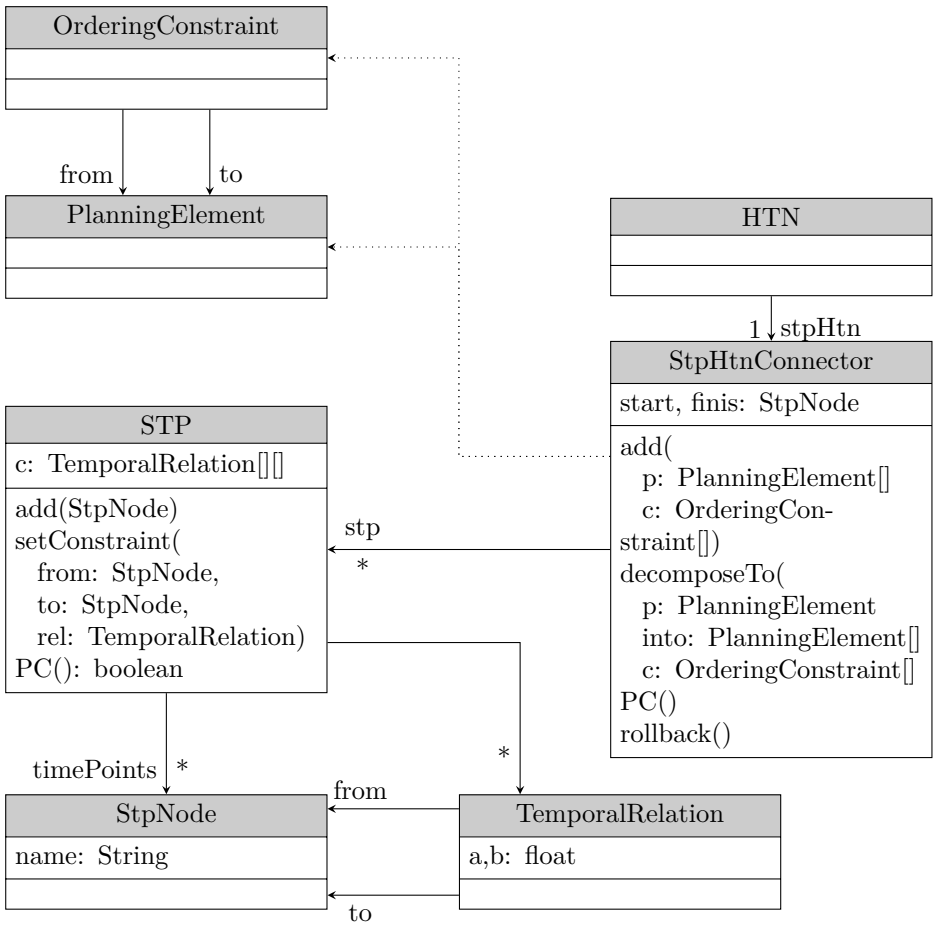


Figure 5.1: Simplified UML diagram for the integration of STP temporal reasoning with HTN planning. The StpHtnConnector is an adaptor between the HTN-related classes OrderingConstraint and PlanningElement, and their STP equivalents StpNode and TemporalRelation. See text for a detailed description.

- Enforce the temporal constraints of p to all PlanningElements in *into*. This enforces that all sub tasks have to be accomplished in the same time interval as their respective parent tasks.
- Add qualitative ordering constraints $[0, \infty]$ as specified by c.
- Add constraints $[0, d]$ between *init* time point and start time points of actions with deadline d.

The STP integration to the HTN algorithm (Listing C.1) is straightforward: In line 37, all tasks and ordering constraints are added to the initial STP. When a task is accomplished by an operator, we first compute the previous actions that interfere with the effects or preconditions of the accomplishing operator (lines 138–141); see Section 5.2 for details. The task is then decomposed using the decomposeTo() method with respect to the found additional ordering constraints (line 147). Similarly, in line 193, a task is decomposed into its sub tasks when refined by a method. After both decomposition activities, the path

consistency algorithm PC() is invoked to check the consistency of the resulting constraint network (lines 163 and 201, respectively). If it turns out to be inconsistent, the HTN branch is aborted; otherwise the HTN algorithm is called recursively. Finally, whenever a valid plan is found (lines 45–55) a copy of the current STP state is stored together with the plan. As HTN is implemented as a backtracking algorithm, the temporal constraints in the STP need to be rolled back similarly to the preliminary plan and the remaining task network. Unfortunately, all simple[2] path consistency algorithms are non-reversible and thus we have to store a copy of every STP version before it is changed by adding new nodes, constraints, or by invoking the path consistency algorithm. The StpHtnConnector class handles versioning of STPs transparently by saving the old state of the STP before applying changes. Furthermore, it exposes the rollback() function to return to the previous state of the STP. From the HTN algorithm's perspective, calling decomposeTo() and rollback() (lines 178 and 210) are the only interfaces required to communicate with the versioned STP.

Listing 5.1: ChangeRefinery implementation of the path consistency algorithm

```
1  public boolean PC() {
2    if(!this.isSymmetric)
3      throw new Error("PC is only defined for symmetric STPs");
4    for(int k=0; k<this.size(); k++){
5      for(int i=0; i<this.size(); i++){
6        if(i!=k)
7          for(int j=0; j<this.size(); j++){
8            if(j!=k && i!=j){
9              this.c.get(i).set(j, this.intersect(this.c.get(i).get(j),
                   this.compose(this.c.get(i).get(k), this.c.get(k).get(j)))
                   );
10
11             // the STP is inconsistent if a constraint is empty
12             // [a,b] <=> a <= t_j - t_i <= b
13             // a>b ==>  there are no t_j and t_i that satisfy the
                   constraint
14             if(this.c.get(i).get(j).a > this.c.get(i).get(j).b) return
                   false;
15           }
16         }
17     }
18   }
19   if(!this.checkSymmetry())
20     throw new Error("PC operation produced non-symmetric STP");
21   return true;
22 }
```

Our implementation of the path consistency algorithm (Algorithm 11) is depicted in Listing 5.1. The STP class uses a two-dimension LinkedList

```
private List<List<TemporalRelation>> c = new ArrayList<List<TemporalRelation>>();
```

to store the TemporalRelation objects in a $n \times n$ matrix where n is the number of time points in the STP. In the path consistency algorithm, the iteration of all triples is implemented as

[2] An approach for flexible constraint handling (including retractable constraints) in STPs is presented in [40].

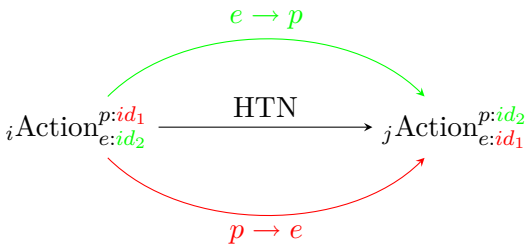


Figure 5.2: Additional ordering constraints due to CI dependencies.

three nested for-loops, which obviously result in $O(n^3)$ time complexity, where n is the number of time points in the STP.

5.2. Additional ordering constraints due to CI dependencies

As explained in Section 3.4, the knowledge base must support a dependency detection mechanism to decide if the preconditions and effects of two actions a_i and a_j interfere. The ChangeRefinery prototype supports the following simple dependency detection: Let p_i, e_i, p_j, e_j be the preconditions and effects of two actions a_i and a_j, and b_i, b_j their respective variable bindings. Define two functions queryDependencies(p_k, b_k) := "the set of IDs of knowledge base CIs in b_k that satisfy the precondition p_k", and effectDependencies(e_k, b_k) := "the set of IDs of knowledge base CIs in b_k that are altered by the effect e_k". We can know define a function depends() that decides whether the preconditions and effects of two actions interfere:

$$depends(a_i, a_j) = \begin{cases} true & if \text{queryDependencies}(p_i, b_i) \cap \text{effectDependencies}(e_j, b_j) \neq \emptyset \\ & \text{OR effectDependencies}(e_i, b_i) \cap \text{queryDependencies}(p_j, b_j) \neq \emptyset \\ false & \text{otherwise} \end{cases}$$

The dependency detection mechanism is illustrated in Figure 5.2: The two actions a_i and a_j are ordered sequentially by the HTN algorithm (edge labelled "HTN"). The effectDependencies and queryDependencies sets are found as subscript and superscript of the actions. The red ordering constraint (labelled "$p \rightarrow e$") is added because the effect of action a_j alters CI id_1 and can potentially invalidate the precondition of action a_i, which depends on CI id_1. Similarly, the green ordering constraint (labelled "$e \rightarrow p$") is added because the precondition of action a_j depends on CI id_2, which is altered by the effect of action a_i.

The computation of the effectDependencies set is supported by the method

```
public Set<Long> getAffectedIds(Session s, Binding b);
```

that is implemented by all sub classes of Assertable (and thus all sub classes of Effect, see Figures 4.1 and 4.3). The set is returned to the HTN algorithm by the assertToKb() method

of the knowledge base component (see Figure 4.1 and line 122 in Listing C.1). The queryDependecies set is directly calculated from the binding of an operator (lines 86–92 in Listing C.1). Every action in the plan stores the two sets as private variables queryDependencies and effectDependencies, they are passed to the action in its constructor (line 124).

The disjunction in the depends() function is calculated using an action's

```
public boolean preconditionDependsOnOtherActionsEffects(Action otherAction);
```

method (Listing 5.2), which computes if the intersection of the two sets if empty. Finally, for every interfering previous action, a qualitative ordering constraint is added to the STP (lines 137–141 and 147 in Listing C.1).

Listing 5.2: ChangeRefinery implementation of the preconditionDependsOnOtherActionsEffects() method of the Action class

```
1 public boolean preconditionDependsOnOtherActionsEffects(Action
      otherAction){
2   for(Long queryCi: this.queryDependencies)
3     for(Long effectCi: otherAction.getEffectDependencies())
4       if(queryCi.equals(effectCi))
5         return true;
6   return false;
7 }
```

5.3. Evaluation of ChangeRefinery's advanced capabilities

This section gives examples of temporal reasoning and dependency detection capabilities of the ChangeRefinery prototype. For the output of the ChangeRefinery sessions, please refer the listings as given in the beginning of every example. The resulting workflows and temporal constraints are illustrated in Tables 5.1 to 5.5 and the corresponding figures.

5.3.1. Reasoning on quantitative temporal information

ChangeRefinery output: Listings C.6, C.7
Domain: VerySimpleJ2eeScenario, Listing C.2
Goal task: SpeedUpWebApplication

This example shows how quantitative temporal constraints are propagated during the planning process and how they enforce the temporal validity of plans. Using the UpgradeDatabaseServerAndWebServerHardware method, the goal task SpeedUpWebApplication with temporal deadline 50 is refined into two UpgradeHardware sub tasks with deadlines 30 and 35, respectively. The UpgradeHardware sub tasks are supposed to upgrade the hardware of a WebServer and a DbServer to which the J2eeApplication is connected. For the first (second) sub tasks, the UpgradeMemAndCpu (UpgradeMem) operator with duration 25 (15) is chosen (lines 27 and 38, respectively). The change plan (line 49) shows the

two actions UpgradeMemAndCpu(machine ← MiscDbServer) and UpgradeMem(machine ← MiscServer) and the resulting Simple Temporal Problem (STP). Note how the durations and temporal deadlines are propagated as tasks are decomposed: The first UpgradeHardware task (line 50) with deadline 30 is decomposed into the UpgradeMemAndCpu action (line 53) with duration 25. The action therefore has to start in the interval $[0, 5]$ and and in the interval $[25, 30]$. The corresponding observation holds for the UpgradeMem action.

Now, consider the same change plan design process, but with an additional qualitative ordering constraint between the two UpgradeHardware sup tasks. The sequential execution of the UpgradeMemAndCpu and the UpgradeMem actions with a total duration of 40 is incompatible with the deadline 35 of the second sub task. Therefore the STP and thus the change plan are inconsistent[3] (see Listing C.7).

5.3.2. Additional ordering constraints due to CI dependencies

ChangeRefinery output: Listings C.8
Domain: VerySimpleJ2eeScenario, Listing C.2
Goal tasks: UpgradeHardware(machine ← id1, mem ← 1000) and SetupServer(server ← id1, mem ≥ 2000) without ordering constraint

This example shows how mutual CI dependencies between preconditions and effects of operators enforce additional qualitative ordering constraints in the resulting plan (see Sections 3.4 and 5.2 for more detailed explanations why this is necessary). The HTN algorithm is called with an initial task network comprising the tasks UpgradeHardware and SetupServer without any ordering constraint. Furthermore, the server and machine variables are initially bound to the MiscDbServer CI with ID 1. The MiscDbServer is initially equipped with memorySize=1000. In the planning process (Listing C.8), first the UpgradeHardware task is accomplished by the UpgradeMem operator with mem ← 1000. Its effect is to increase the memorySize property of the MiscDbServer CI from 1000 to 2000. Then, the UseExistingServer operator with the condition mem *geq* 2000 was chosen to refine the SetupServer task; it is applicable because the precondition MiscDbServer.memorySize ≥ 2000 is satisfied by the MiscDBServer CI.

Note that the HTN plan [UpgradeMem → UseLowEndServer] is valid, while the inverse plan [UseExistingServer → UpgradeMem] is invalid because the precondition of the UseExistingServer operator is not satisfied by the MiscDbServer CI with memorySize=1000. Without the special dependency detection mechanism explained in Sections 3.4 and 5.2 the STP would not include a qualitative ordering constraint between the two actions and would thus temporally allow a plan which is invalid with respect to the HTN semantics. Table 5.1 lists resulting temporal constraints, which are illustrated by Figure 5.3.

5.3.3. A complex example

This section exemplifies two different more complex change plan designs for the goal task SpeedUpWebApplication. The first plan introduces load balancing for the J2EE container,

[3] A constraint satisfaction problem is inconsistent, *iff* it is has an empty constraint. Here, the duration constraint $[15, 10]$ of the UpgradeMemory action (line 5 in Listing C.7) is empty, as $15 > 10$.

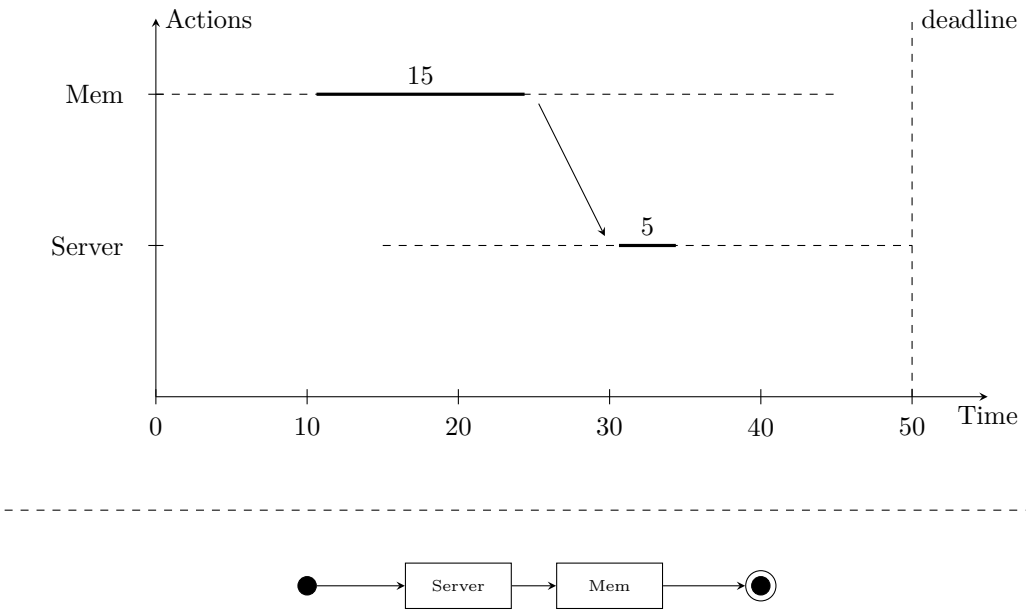


Figure 5.3: Graphical representation of the temporal relations between actions in the "Additional ordering constraints due to CI dependencies" change plan (Section 5.3.2). The top figure shows the temporal constraints from the STP: Every action has to be accomplished within its allowed time window (dashed lines) and has a duration (solid lines). Arrows between actions indicate qualitative ordering constraints, i.e. the UpgradeMem and UseExistingServer actions are sequentially ordered in this example. The bottom figure shows the corresponding workflow representation of the actions and their qualitative ordering constraints.

Table 5.1: STP constraints for the "Additional ordering constraints due to CI dependencies" change plan.

Action	Start	End	Duration
UpgradeMem (Mem)	[0, 30]	[15, 45]	15
UseExistingServer (Server)	[15, 45]	[20, 50]	5

and the second plan migrates the database server. For both plans, the resulting STP is shown as a table and visualised in a graphical representation highlighting the temporal constraints between actions. The graphical representation is then translated to a workflow diagram of the change plan.

5.3.3.1. Solution plan 1: Migrate database server

ChangeRefinery output: Listings C.11
Domain: VerySimpleJ2eeScenario, Listing C.2
Goal task: SpeedUpWebApplication

In this change plan, the SpeedUpWebApplication goal task is refined into a change plan that migrates the Wiki web application's database to a new database server using an Oracle database. The final plan comprises backing up the old database, setting up a new database server as a virtual server, installing the Oracle database software, and finally copying the content of the old database to the new database. Table 5.2 lists the involved actions and their respective variable bindings to knowledge base CIs. Table 5.3 lists the resulting temporal constraints for the change plan, which are visualised in Figure 5.4. It turns out that all actions have to be accomplished in a sequential fashion. The STP also contains information about the estimated total implementation time of the change, which is between 120 and 150.

Table 5.2: Actions in the "migrate database" change plan.

Action	Variable bindings
RunBackupScript (Backup)	db ← Database (id=5)
InstallVirtualServerByScript (VS)	server ← DbServer (id=16)
InstallOracleDbScript (DB)	db ← Database (id=17) dbserver ← DbServer (id=16)
CopyDbScript (Copy)	fromDb ← Database (id=5) toDb ← Database (id=17)

Table 5.3: STP constraints for the "migrate database" change plan.

Action	Start	End	Duration
RunBackupScript (Backup)	[0, 30]	[50, 80]	50
InstallVirtualServerByScript (VS)	[50, 80]	[60, 90]	10
InstallOracleDbScript (DB)	[60, 90]	[80, 110]	20
CopyDbScript (Copy)	[80, 110]	[120, 150]	40
Task SpeedUpWebApplication	[0, 30]	[120, 150]	[120, 150]

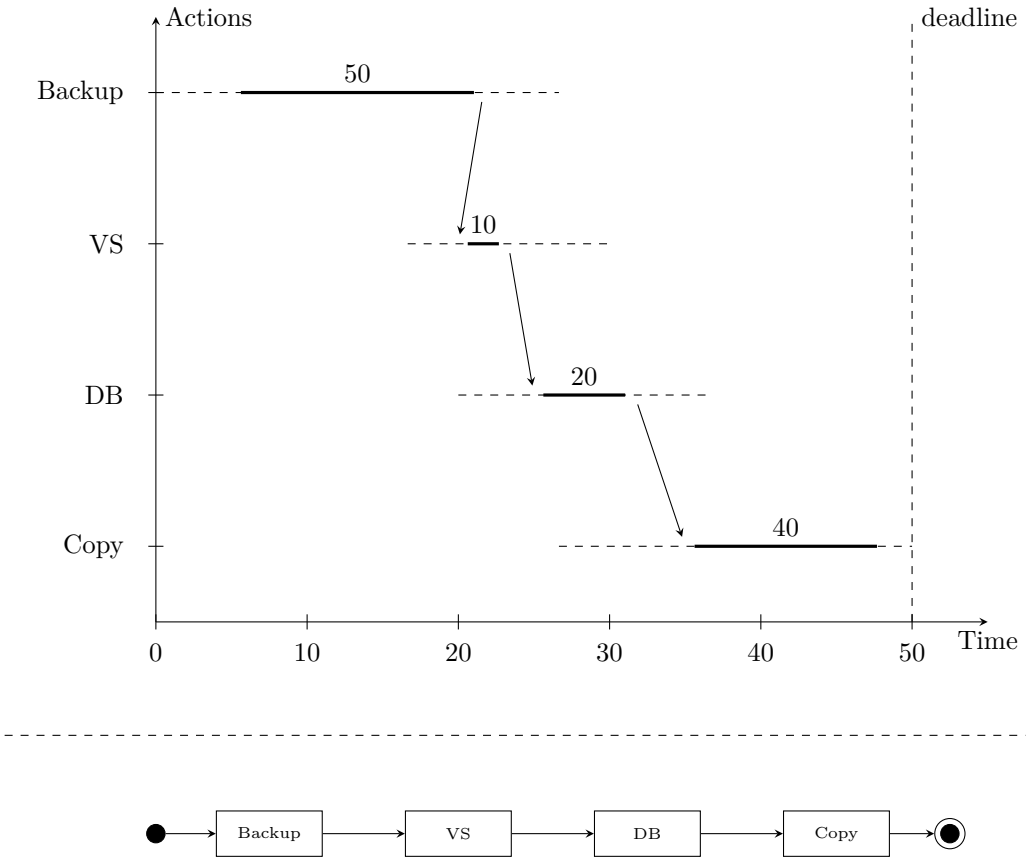


Figure 5.4: Graphical representation of the temporal relations between actions in the "migrate database" change plan (Section 5.3.3.1). The top figure shows the temporal constraints from the STP: Every action has to be accomplished within its allowed time window (dashed lines) and has a duration (solid lines). Arrows between actions indicate qualitative ordering constraints, i.e. all actions are sequentially ordered in this example. The bottom figure shows the corresponding workflow representation of the actions and their qualitative ordering constraints.

5.3.3.2. Solution plan 2: Enable load balancing

ChangeRefinery output: Listings C.10
Domain: VerySimpleJ2eeScenario, Listing C.2
Goal task: SpeedUpWebApplication

In this change plan, the SpeedUpWebApplication goal task is refined into a change plan that enables load balancing for the Wiki web application container. To accomplish this, the J2EE container holding the Wiki application is replicated to a freshly installed virtual server, a load balancer system is installed, and finally the J2EE container is added to the load balancer. Table 5.4 lists the involved actions and their respective variable bindings to knowledge base CIs. Table 5.5 lists the resulting temporal constraints for the change plan, which are visualised in Figure 5.5. The resulting change plan has a mixed sequential and parallel structure.

Figure 5.6 shows a graphical representation of the stepwise change plan refinement process. The upper figures depict the decomposition methods, the lower figures show the four decomposition steps from the initial goal task (step 1) to the final workflow (step 4). The planning process starts with the workflow from step 1, which consists of the single goal task, SpeedUpWebApplication. Applying the EnableLoadbalancing method decomposes the goal task into the workflow shown in step 2. The InstallJ2eeServer sub task is then further refined by the DefaultJ2eeServerInstallation method, which results in the workflow shown in step 3. Finally, in step 4 all tasks can by accomplished by their respective operators.

Table 5.4: Actions in the "enable load balancing" change plan.

Action	Variable bindings
InstallVirtualServerByScript (VS)	server ← WebServer (id=17)
InstallJ2eeContainerSoftware-Script (Cont)	webserver ← WebServer (id=17) cont ← J2eeContainer (id=18)
InstallJ2eeApplication-Script (App)	app ← J2eeApplication (id=8) cont ← J2eeContainer (id=18)
AddContainerToLoadBalancer-Script (AddContLb)	container ← J2eeContainer (id=18) lb ← LoadBalancer (id=16)
InstallLoadBalancer-Script (Lb)	lb ← LoadBalancer (id=16)

Table 5.5: STP constraints for the "enable load balancing" change plan.

Action	Start	End	Duration
InstallVirtualServerByScript (VS)	[0, 15]	[10, 25]	10
InstallJ2eeContainerSoftware-Script (Cont)	[10, 25]	[20, 35]	10
InstallJ2eeApplication-Script (App)	[20, 35]	[30, 45]	10
AddContainerToLoadBalancer-Script (AddContLb)	[30, 45]	[35, 50]	5
InstallLoadBalancer-Script (Lb)	[0, 35]	[10, 45]	10
Task SpeedUpWebApplication	[0, 15]	[35, 50]	[35, 50]

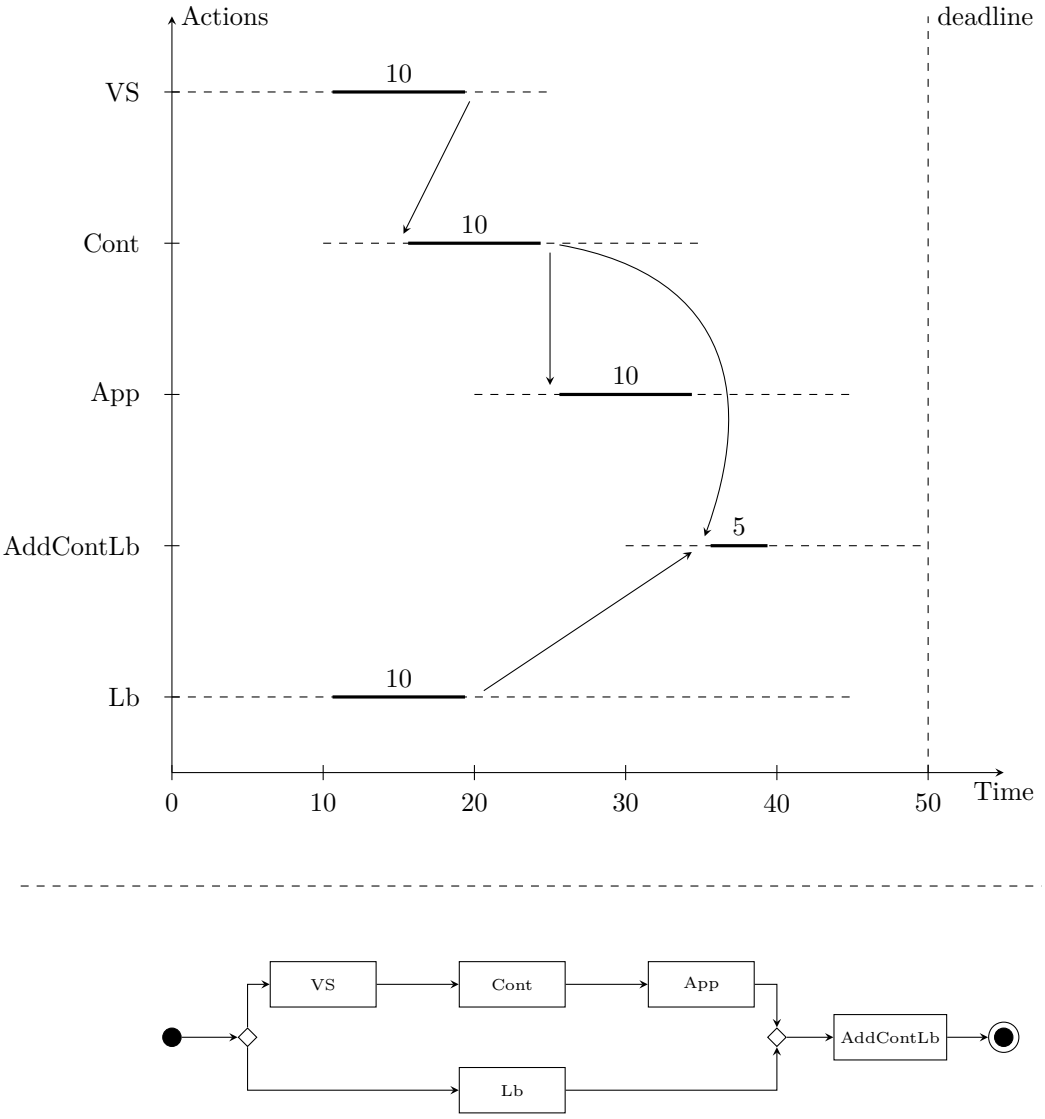


Figure 5.5: Graphical representation of the temporal relations between actions in the "enable load balancing" change plan (Section 5.3.3.2). The top figure shows the temporal constraints from the STP: Every action has to be accomplished within its allowed time window (dashed lines) and has a duration (solid lines). Arrows between actions indicate qualitative ordering constraints, e.g. the Lb action has to finish *before* the AddContLb action starts. The bottom figure shows the corresponding workflow representation of the actions and their qualitative ordering constraints.

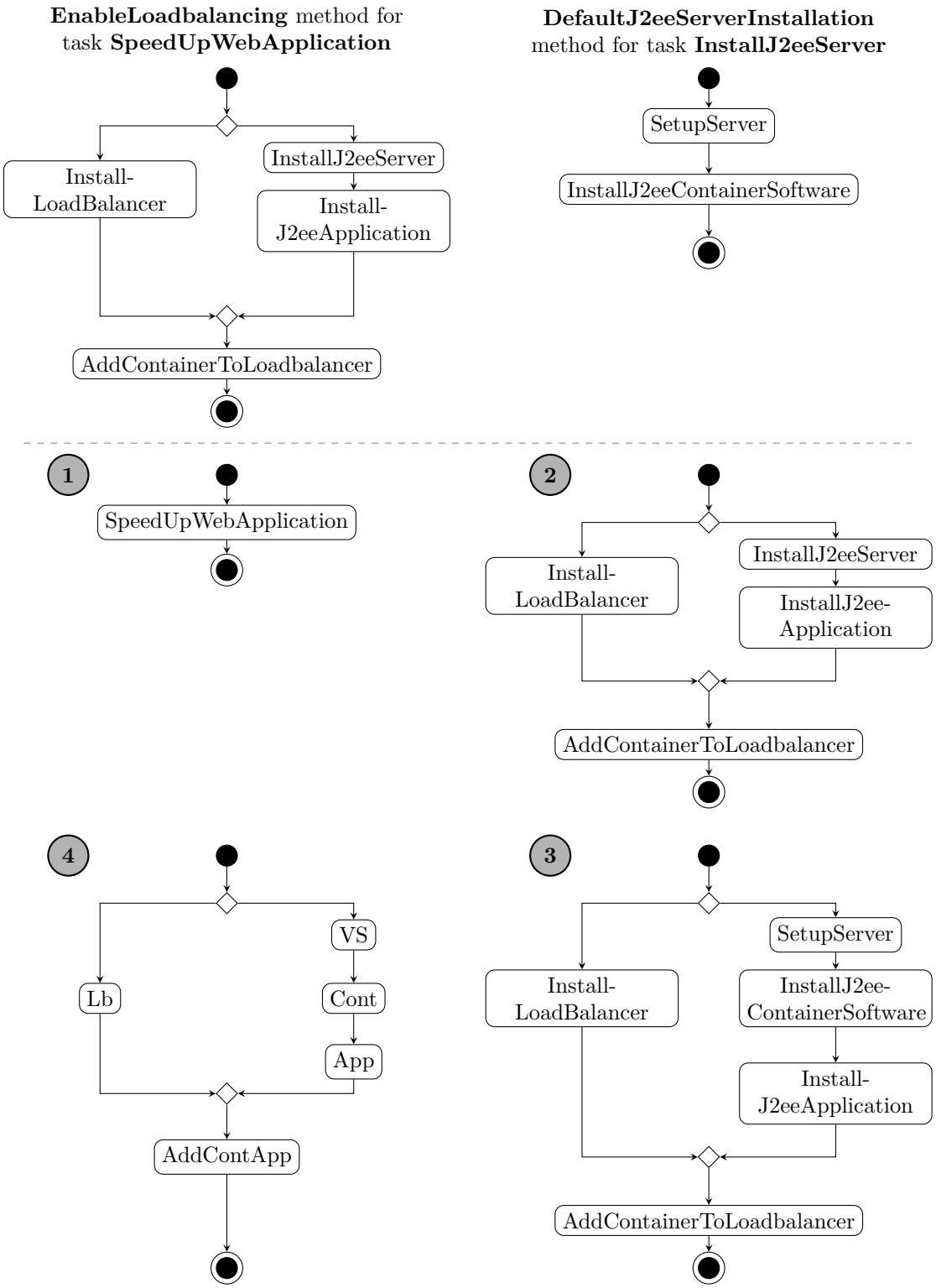


Figure 5.6: Graphical representation of the stepwise change plan refinement process. The upper and lower figures depict the decomposition methods, and the four decomposition steps, respectively.

5.4. Additional ideas for future extensions

5.4.1. Dependency resolution: The ChangeLedge approach to change planning

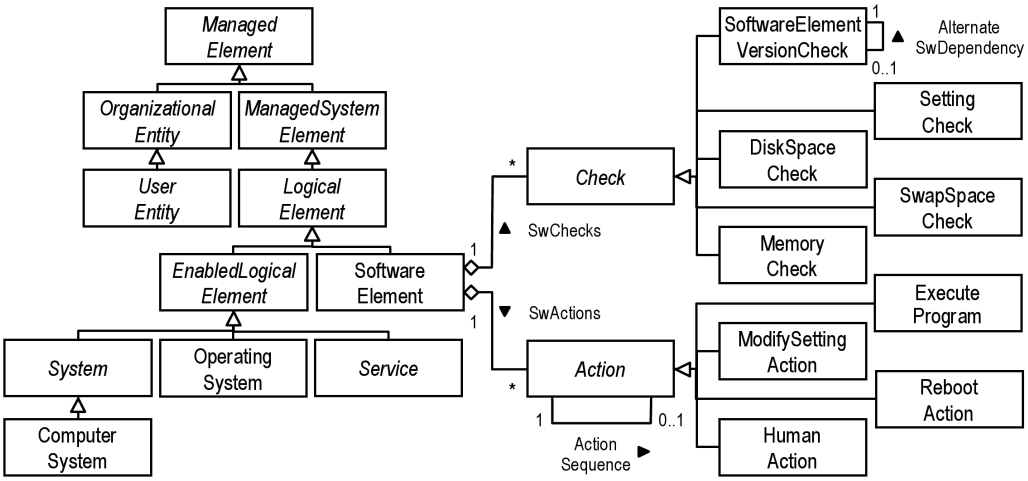


Figure 5.7: CIM-based IT infrastructure model of the ChangeLedge system [4].

The ChangeLedge [4,41] system for change plan refinement is based on a CIM-compliant IT infrastructure model (see Figure 5.7) that includes dependencies between configuration items. Every CI can exist in different states, and transitions between states are guarded by *checks*. Every failed check generates an *action* that is to be executed in order to satisfy the check. This model allows to express dependencies between configuration items and to refine a given workflow of CI transitions by chasing the depending actions of failed checks and including them in the workflow.

One of the drawbacks of ChangeLedge is that the design of the preliminary change plan which is input to the refinement system requires a highly skilled IT technician. A possible integration would use the ChangeRefinery approach to design such a preliminary change plan, and pass it on to ChangeLedge in order to further resolve configuration item dependencies.

5.4.2. Advanced decision support through change plan metrics

While we still want to leave decisions on different options for change plan refinement to the human operator, we would like to assist her in making the best choices by presenting her metrics, such as time, cost or risk, and to help her understand the trade-offs of various possibly very different change designs. These metrics might be displayed along with possible options at every user choice point. For example, choosing decomposition method a to refine a given task could be expensive but well-studied and proven to work by past changes in which it was used, whereas method b can be quickly and cheaply implementated but was never deployed before and thus seems to be quite risky.

One possible way of integrating metrics into the mixed-initiative planning concept is to annotate the available choices at every choice point (i.e. choice of bindings or refinements). The algorithm would *look ahead* (see Section 5.4.3) in the HTN planning space to be able to anticipate the future effect of every choice and to compute the metrics.

5.4.3. Advanced decision support through pre-compiled HTN planning trees

The current ChangeRefinery implementation offers user interaction at every choice point (i.e. at every non-deterministic choice of a task, refinement or binding). The user is only presented those choices that make sense in the current state of the planning process. For example, only methods and operations that are applicable in the current state of the knowledge base are presented. However, the continued planning process might reveal that a certain refinement can never lead to a valid solution plan and should thus not be offered to a user. Such a filter requires the *look-ahead* evaluation of the planning space, which is in general computationally infeasible. In our case however, early binding of variables to configuration items significantly prunes the search space and thus makes pre-compilation or background-compilation of HTN plans a considerable option.

In an extreme case, the whole HTN search space with respect to existing variable bindings in high-level tasks can be pre-compiled, and the user effectively browses the tree of known HTN decompositions. This concept would also allow for powerful decision support mechanism, as change plan metrics used to evaluate the quality of different change plans can easily be computed for different alternatives, once all possible HTN plans are known.

As mentioned in Section 4.2.2 an the knowledge base component of the ChangeRefinery prototype, the current implementation suffers from poor rollback() and assert() performance. For example, the computation of all 132 HTN plans for the goal task SpeedUpWebApplication on the scenario VerySimpleJ2eeScenario takes 174 seconds, out of which 167 seconds are used for assert() and rollback() operations, 4 seconds for query() computation, and 3 seconds the remaining parts of the algorithm.

5.4.4. Policies on infrastructure and change plans

In Section 2.3, we identified two classes of IT policies which impose constraints on change plans: The first type of policies constrains the allowed states of the IT infrastructure, and the second type prescribes conditions on the logical structure of the change plan itself. A straightforward integration of a policy engine component can be accomplished analogously to that of the temporal reasoner component: At every HTN planning step (i.e. choosing a method or an operator), the policy engine is asked to validate the preliminary plan and the intermediate state of the infrastructure with respect to all policies in the policy repository. A failed policy signalises that the chosen refinement is invalid and triggers backtracking.

It would of course be desirable to integrate an existing policy engine such as RuleML [13], Drools [14]. However, the feasibility of this concept can be shown without them: In the ChangeRefinery prototype, IT infrastructure policies can be specified in the same HQL formalism as refinement preconditions. The policy stating that “all databases that are

used by web applications need to run on MySQL database management systems" can be expressed as *SELECT db FROM OracleDb as db, J2eeJdbcResource as res, J2eeApplication as app WHERE res in elements(app.j2eeJdbcResources) AND res.database=db*. If this query has a non-empty number of result tuples, the policy is violated.

5.4.5. Richer temporal information

Timed initial literals were introduced in the Planning Domain Definition Language 2.2 (PDDL) [42] and "are a way of expressing a certain restricted form of exogenous events: Facts that will become TRUE or FALSE at time points that are known to the planner in advance, independently of the actions that the planner chooses to execute. Timed initial literals are thus deterministic unconditional exogenous events" [42]. In principle, the STP temporal reasoner component is able to reason over timed initial literals [33].

They might be useful to model certain states of the infrastructure, for example regular known server downtimes or backup cycles.

5.4.6. Hybrid state-space and HTN planning

The assumption of correct change template information (see Section 4.1) is very strong and also quite likely to be unsatisfied in real-world scenarios. Our idea is to use interleaved (*hybrid*) state-space and HTN planning in situations where the available methods and operators are incomplete and thus cannot be assembled to a valid plan. This might for example be the case when the designer of a change template forgot to include a sub task that provides some effect which is required by a precondition of a subsequent operator. Whenever such a problem is encountered, a state-space planner is invoked to find a sequence of operators that fix the deficient method.

This hybrid approach is also applicable to a second use case: Let *planning time* and *execution time* be the time points in which a change plan is designed and executed, respectively. If other change plans are implemented between planning time and execution time, the actual state of the IT infrastructure at execution time differs from the simulation environment at planning time, which can render the change plan invalid. If, for example, a change plan relies on a certain library to be installed at some specific version (which was the case at planning time), and another change upgrades the library to a newer version between planning and execution time, the former change plan becomes invalid. Invoking a state-space planner in this situation can help to fix the change plan by finding a sequence of states that repairs the preconditions of subsequent operators in the plan.

For both use cases, the problem statement is: *Given a (failed) precondition p and a current IT infrastructure state s, find a sequence of operators, $\pi = \langle o_1, ..., o_n \rangle$, such that p is satisfied in the state $\gamma(s, \pi)$*. Every naive, undirected state-space search is of course infeasible due to the large branching factor. Existing heuristic search algorithms like hill-climbing and A* [43, 44] rely on a heuristic function that assigns cost values to states. A domain independent heuristic usually compares the proximity of a given state with the goal state by counting the number of logical atoms that are missing to reach the goal state; the

smaller this number is, the closer the inspected state is to the goal state. Assessing the proximity of states in this way is not possible in our framework for a number of reasons[4], so that we propose the use of a heuristic that rates operators instead of states: Given a set of operators O, a state s and a precondition p, we define an *operator heuristic* as a function

$$H : p \times s \times O \rightarrow [0, 1],$$

that assigns a cost value between 0 and 1 to every operator in O. Intuitively, the cost value of an operator describes the inverse probability for this operator to be helpful in reaching a state that satisfies p.

Using an arbitrary operator heuristic, Algorithm 7 is a depth-first, heuristically guided algorithm for state-space search. It tries to find a repair plan by selecting at every branching point the cheapest applicable operator. The search is cut off by a maximum search depth and a maximum cost.

Algorithm 7 State-space forward search controlled by a heuristic on operators.

```
1:  function OH-FORWARD(State s, Goal g, Float maxCost, Int maxDepth)
2:      if s satisfies g then
3:          return true
4:      if maxDepth= 0 then
5:          return false
6:
7:      active ← {Operator o | o applicable in s and o.cost<maxCost}
8:      queue ← Min-Heap(active)
9:      while (queue ≠ ∅) ∧ (Operator q ← queue.next) do
10:         s.apply(q.effects)
11:         π.add(o)
12:         OH-FORWARD(s, g, maxCost − o.cost, maxDepth − 1)
13:         π.remove(o)
14:         s.rollback()
```

The following two sections describe alternative approaches for calculating the operator heuristic[5].

5.4.6.1. Property-based operator heuristic

Preconditions usually express properties and relations between CIs, e.g. *dbServer.memory ≥ 10 AND dbServer.cpu = 1 AND dbServer.hasDatabases ≤ 2 AND wikiDb ∈ dbServer.databases*. Similarly, effects of operators tweak such properties or connect different CIs. A simple heuristic might collect all properties (in this case memory, cpu, hasDatabases, databases) of a precondition, and assign the cost value 1 to every operator that modifies one of these

[4] One of them being that we do not use logical atoms and thus cannot count them.

[5] Note that both concepts do not make use of the state s. This obviously leaves much space for further improvement.

properties in its effects, and 0 otherwise.

$$H_{PB}(p = \{prop_1, ..., prop_n\}, s, o) = \begin{cases} 1 & \text{if any } prop_i \text{ is modified by the effects of } o \\ 0 & \text{otherwise} \end{cases}$$

The ChangeRefinery prototype is prepared for this kind of computation, as the classes for query building and effects in da.planner.htn.query and da.planner.htn.effects offer access to the names of properties explicitly.

5.4.6.2. Neural network approach to operator heuristics

The second approach uses a neural network to compute the heuristic function. The input to the neural network is a binary pattern that indicates which properties are present in the failed precondition and its output is the heuristic H_{NN} that assigns a value in $[0, 1]$ to every operator. Let P be the set of all properties of the IT infrastructure that may be part of precondition queries and O the set of all defined operators; then the neural net computes the function

$$H_{NN} : \{0, 1\}^{|P|} \rightarrow [0, 1]^{|O|},$$

which is a variation of the above-mentioned general operator heuristic. The neural network can be trained in different ways:

- Using test sets from the property-based heuristic (Section 5.4.6.1)
- By training sets explicitly provided by IT operators
- Automatically, i.e. by using successful past repair plans as training sets

5.4.7. Advanced detection of precondition and effect interference

The dependency detection mechanism presented in Section 5.2 is quite simplistic and orders actions by a pessimistic locking scheme: Whenever preconditions and effects depend on or modify the same CI, the actions are ordered sequentially. A more sophisticated concept could follow one of the following two approaches:

1. A more fined-grained model of configuration items and their properties would allow to make smarter decisions on the interplay of preconditions and effects.
2. In order to find out whether preconditions and effects interfere or not, one could simulate different orderings and check which are forbidden, i.e. which ordering constraints have to be imposed.

5.4.8. Managing multiple and concurrent changes

So far we assumed that there is only one change that is being planned for. In reality though, hundreds of RFCs must be dealt with concurrently. An incoming RFC runs through the change management process and after it was refined into an implementable workflow it is scheduled in some change window. However, many other changes will be implemented in the same change window. These changes also alter the IT infrastructure which thus differs from the knowledge base infrastructure simulation environment at planning time. Handling these concurrent changes turns out to be a highly non-trivial problem to solve.

5.4.9. Semantic web technology based knowledge base component

One of the primary goals of this thesis was to investigate the applicability of HTN planning to the prevailing object-oriented data models used in IT configuration management systems. While our framework and its prototypical implementation clearly shows that this is indeed a feasible approach, we discovered some drawbacks of our CIM/Hibernate based knowledge base implementation: Because the knowledge base state is not defined by a set of logical atoms, the implementation of the rollback functionality is non-trivial. Furthermore, reasoning on policies and state-based planning would be vastly facilitated by a set based knowledge representation.

Semantic web languages and technologies such as RDF [45], OWL [46] and Jena [38] appear to be a promising basis for such a knowledge base component: They offer a set based knowledge representation together with powerful query languages such as SPARQL, and provide expressive object-oriented modeling concepts [47]. Listing 5.3 gives a trivial example of a RDF/OWL definition of a sub class relationship between two classes, Server and DatabaseServer.

Listing 5.3: Sub class modeling in OWL

```
1 <owl:Class rdf:about="Server">
2   <rdfs:subClassOf rdf:resource="DatabaseServer" />
3 </owl:Class>
```

6. Conclusion and outlook

We have proposed a generic architecture for integrating various information sources necessary for IT change design. The Hierarchical Task Network planning paradigm augmented with temporal constraints is used to model and hierarchically refine change tasks into consistent change plans, considering the involved IT infrastructure and policies or constraints on such plans.

By building a prototype and evaluating it with real-life examples, we have demonstrated the viability of the approach. To substitute the IT operations model of our prototype with real IT operations data sources (CMDB, service management, or asset management products) would require to solve several information management problems, such as model mapping and integration, and the ability to have a knowledge base that scales to large data sets. Because of the mixed-initiative nature of the solution, measuring the *efficiency* of the tool (i.e. its speed for generating plans) is not the most significant metric. Further validation would require to deploy the tool in a live environment, and to measure its *effectiveness*, i.e. the improvement of productivity of IT practitioners in designing IT changes and the improvement of the quality of IT plans.

Some ideas for possible future work objectives were already sketched in Section 5.4. Immediate next steps could be to introduce decision-support features in our mixed-initiative scenario. In this thesis, an IT practitioner guides the design of the change plans by choosing refinements and variable bindings. While we still want to leave the decision to the human operator, we would like to assist her in making the best choices by presenting her metrics, such as time, cost or risk, and to help her understand the trade-offs of various change designs.

Straightforward extensions to the presented framework include the implementation of richer temporal information like timed initial literals (compare PDDL 2.2 [42]), which can be handled by the already introduced STP data structure [33].

The integration of established CMDB products is supported by our architecture and would offer the user a powerful tool for infrastructure knowledge editing and template based query building. On the other hand, evaluating the use of logic programming and semantic web techniques for CMDB knowledge representation promises advanced inference capabilities for the interdependencies of configuration items, as previously targeted by ChangeLedge [4]. The applicability of semantic web technology to IT configuration management systems is part of ongoing research at HP Labs.

Finally, we also intend to expand the proposed solution in two ways. First we will investigate the feasibility of the aforementioned hybrid planning approach (Section 5.4.6), i.e. interleaved state-space and HTN planning, to be able to reason on incomplete domain knowledge and repair ill-defined change plans. We also plan to study how our solution can be combined with proposed change scheduling solutions (e.g. [18,19] and Section 1.4), and how to resolve temporal and resource constraints into implementation schedules.

A. Automated planning and constraint satisfaction problems

This chapter depicts the basic definitions, notations and algorithms used in automated planning in order to introduce the reader to the planning concepts concepts used in this thesis. The presentation generally follows the syntax and definitions introduced in [5] and [35], but also borrows from original publications. Several examples illustrate the definitions of every section.

Automated planning as an area of artificial intelligence (AP) studies reasoning processes that find and organise actions by anticipating their expected results. Several everyday activities (such as routing, logistics, business decisions, land use planning, urban planning, etc.) imply such planning processes, although they are rarely experienced explicitly.

With its (recent) extensions like for example temporal planning [32, 33], interleaved planning/scheduling/resource allocation, etc., automated planning overlaps with related fields such as scheduling, constraint programming, and graph algorithms.

The remainder of this chapter is organised as follows: Section A.1 defines basic notions used in automated planning and presents the commonly used Dock-Worker-Robots planning domain. Section A.2 and A.3 cover state-space and HTN planning and illustrate examples for both concepts. Section A.4 on constraint satisfaction problems sets up the vocabulary for temporal constraints which are introduced in Section A.5.

A.1. Representations for classical planning

Before one can use automated planning algorithms to solve real life problems, these problem statements need to be encoded into machine-readable languages, so called *planning domains*. As it is obviously impossible to encode the entire world and as the computational complexity of planning problems increases with increasing richness of detail of the problem statement, the task of formalising the planning domain focusses on finding an abstraction that is both detailed enough to support expressing problems adequately, and sufficiently simple to be computably feasible. Although there exist different representations for classical planning (e.g. set-theoretic (based on propositional logic), classical representation (based on first-order logic without function symbols) and state variable representation (based on first-order logic with function symbols)), this thesis will only make use of the classical representation.

A.1.1. Classical representation

The classical representation comprises *states* as the description of the status of the world, and *operators* inducing transitions between states. In the following let $\mathcal{L}$ be a first-order language with finitely many predicate symbols and constant symbols and no function symbols. In the definitions, the explicit reference to $\mathcal{L}$ is usually omitted. The choice of the language $\mathcal{L}$ obviously depends on the domain, and, as above-mentioned, influences the class of expressible planning problems and the computational complexity of these problems.

Definition 1 (State)**.** *A state is a set of ground atoms of* $\mathcal{L}$*.*

Definition 2 (Operator, Action)**.** *An operator as a triple* $o = (name(o), precond(o), effects(o))$ *where* $name(o)$ *is the name of the operator and* $precond(o)$ *and* $effects(o)$ *are sets of literals (i.e. atoms or negated of atoms). The semantics of preconditions and effects are:*

- *For any set of literals* L*, let* L^+ *be the set of all atoms in* L *and* L^- *be the set of all atoms whose negations are in* L*. An action is a ground instance of an operator. An action* a *is applicable to a state* s*, iff* $precond^+(a) \subseteq s$ *and* $precond^-(a) \cap s = \emptyset$*. Preconditions restrict the applicability of operators to specific properties of states.*
- *The result of applying an action* a *to a state* s *is the state*

  $$\gamma(s, a) = (s - effects^-(a)) \cup effects^+(a).$$

  Effects describe how operators change the state of the world.

Definition 3 (Planning Domain)**.** *A classical planning domain in* $\mathcal{L}$ *is a restricted state-transition system* $\Sigma = (S, A, \gamma)$ *with*

- $S \subseteq \mathcal{P}$*(all ground atoms of* $\mathcal{L}$*)*
- A *is the set of ground instances of a set* O *of operators*
- $\gamma(s, a) = (s - effects^-(a)) \cup effects^+(a)$ *if* $a \in A$ *is applicable to* s *and otherwise* $\gamma(s, a)$ *is undefined*
- S *is closed under* γ

Definition 4 (Planning Problem)**.** *A classical planning problem is a triple* $\mathcal{P} = (\Sigma, s_0, g)$*, where*

- s_0*, the initial state, is any state in* S
- g*, the goal, is any set of ground literals*

The statement P *is the syntactic specification of a planning problem,* $P = (O, s_0, g)$*.*

Definition 5 (Plan)**.** *A plan is sequence of actions* $\pi = \langle a_1, ..., a_k \rangle$ *where* $k = |\pi| >= 0$ *is the length of* π*. The effect of applying a plan* π *to a state* s *is recursively defined as an extension of the state-transition function* γ*:*

$$\gamma(s, \pi) = \begin{cases} s & \text{if } k = 0 \\ \gamma(\gamma(s, a_1), \langle a_2, ...a_k \rangle) & \text{if } k > 0 \text{ and } a_1 \text{ is applicable to } s \\ \text{undefined} & \text{otherwise} \end{cases}$$

Definition 6 (Solution)**.** *A plan* π *is a solution to a planning problem* $\mathcal{P} = (\Sigma, s_0, g)$ *iff* $g \subseteq \gamma(s_0, \pi)$*.*

A.1.2. Example: The Dock-Worker-Robots domain

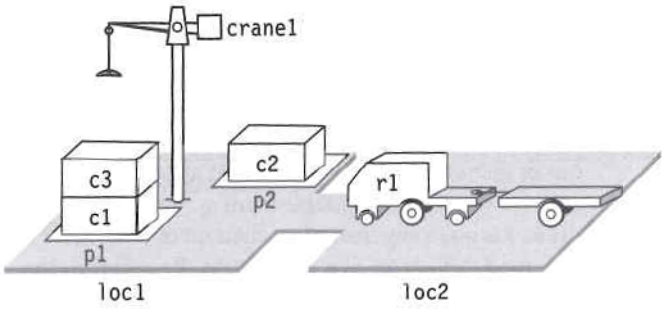


Figure A.1: The Dock-Worker-Robots planning domain [5]

As an example, consider the *Dock-Worker-Robots* (DWR) domain, see Figure A.1. The world consists of locations, robots, cranes, piles and containers. Planning problems are about moving containers between different piles and/or locations by using cranes and robots. Operators capture atomic actions on the objects in the domain: A crane can *take* or *put* containers and *load* or *unload* containers on/from robots. Robots can *move* between locations. The state in Figure A.1 could be described by the set of ground atoms s_1 = {in(c1,p1), in(c3, p1), top(c3,p1), on(c3,p1), on(c1,pallet), in(c2,p2), top(c2,p2), on(c2,pallet), empty(crane1), at(r1,loc2), occupied(loc2), unloaded(r1)}.

The operator take(k, l, c, d, p) makes the crane k in location l take container c off container d in pile p. The preconditions for *take* are {empty(k), top(c,p), on(c,d)} and the effects are {holding(k,c) ¬ empty(k), ¬in(c,p), ¬top(c,p), ¬on(c,d), top(d,p)}. Figure A.2 illustrates the action take(crane1, loc1, c3, c1, p1).

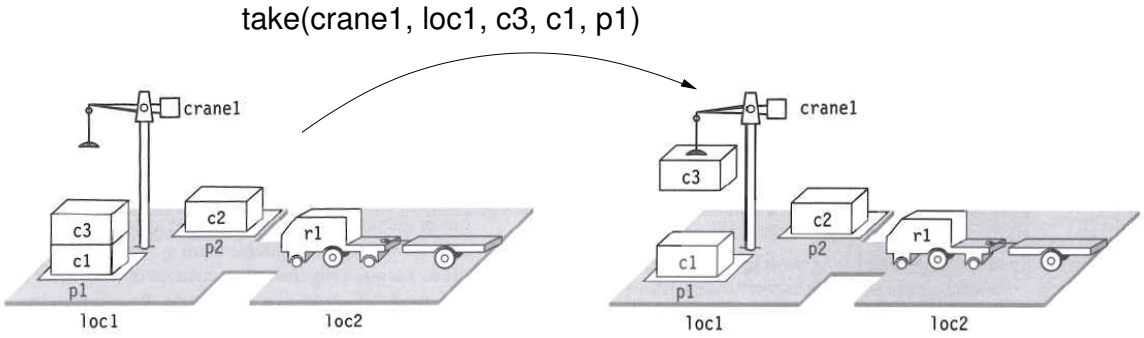


Figure A.2: Actions in the DWR domain [5]

A.2. State-space planning

State-space planning is the problem of finding a solution plan for a given planning problem statement $P = (O, s_0, g)$. State-space planning is the simplest classical planning algorithm

Algorithm 8 Non-deterministic state-space forward-search

```
1:  function FORWARD-SEARCH(O, s_0, g)
2:      s ← s_0
3:      π ← the empty plan
4:      while true do
5:          if s satisfies g then
6:              return π
7:          applicable ← {a | a is a ground instance of an operator in O
8:                           and precond(a) is true in s}
9:          if applicable = ∅ then
10:             return failure
11:         non-deterministically choose an action a ∈ applicable
12:         s ← γ(s, a)
13:         π ← π.a
```

and is usually implemented as forward or backward search in the finite space of all reachable states.

A.2.1. State-space planning algorithms

A sound and complete forward-search method for state-space planning is given by Algorithm 8. The non-deterministic algorithm starts with the initial state s_0 and picks applicable actions until eventually a state satisfying the goal formula g is found. A deterministic implementation usually uses backtracking to perform a depth-first exploration of all possible paths in the state space.

Similarly, a backwards-search algorithm starts with the goal state and constructs a solution plan by applying reverse state transitions γ^{-1} until the initial state is reached. The naive forward and backward-search methods struggle when opposed to real-world problems, because the search space is growing exponentially. The STRIPS algorithm [48] was an early attempt to reduce the search space, but is incomplete (i.e. does not guarantee to find a solution if there is one).

A.2.2. Guiding the planning algorithm

The naive forward-search can be regarded the brute-force method to find a list of actions that connect an initial state with a goal state. With increasing domain size this method quickly becomes unfeasible as the branching factor for every non-deterministic choice explodes and one has to consider strategies to guide the planning algorithm in order to find solutions more quickly. One such strategy is to use *Heuristics*, "a way of ranking a set of nodes in order of their relative desirability" [5]. Instead of choosing the next action and hence the next state randomly (Algorithm 8, line 13), the planner picks from the list of candidate states the state that minimises the heuristic function h. Designing good heuristics is a very complicated task and usually requires precise knowledge on the structure of both the domain and possible solutions to common planning problems. As an example, imagine the planning

task of finding inner-city public transport connections between two locations. Instead of searching the whole space of possible connections, the heuristics might guide the search by favouring those buses and trains that are directed towards the goal location. Note that the use of heuristics does not guarantee to find solutions more quickly or even to find good solutions at all. In this example, the planner might disregard the possibility of taking a bus in opposite direction and therefore being able to use a quick train for the rest of the trip.

A.3. HTN planning

An alternative strategy to guide the planning algorithms is deployed in Hierarchical Task Network (HTN) planning. The idea is to provide the planner as additional input with a set of hierarchical refinement recipes. The HTN formalism is well suited to scenarios in which solutions are known to display hierarchical structures and in which hierarchical refinement recipes are available, or easily constructable.

Although [5] distinguishes between Simple Task Network (STN) and Hierarchical Task Network (HTN) planning and as we only need the STN approach in this work, we are going to speak of HTN planning when we actually mean STN planning. This is to avoid confusion with the notion of Simple Temporal Problems (STP) and temporal networks which will be introduced later in this chapter.

The following definition introduce the fundamental concepts of HTN planning. The previous definitions of operators, actions, plan, the state-transition function $\gamma(s, a)$ in classical planning remain the same.

Definition 7 (Task)**.** *Every operator symbol is a task symbol and we introduce non-primitive task symbols as task symbols. A task is an expression of the form $t(r_1, ..., r_n)$ where t is a task symbol and all r_i are terms. A task is called a primitive task if its task symbol is an operator symbol, and is called a non-primitive task otherwise. A task is ground if all terms are ground. An action $a = (name(a), precond(a), effects(a))$ accomplishes a ground primitive task t in a state s iff $name(a) = t$ and a is applicable to s.*

Definition 8 (Task Network)**.** *A task network is an acyclic digraph $w = (U, E)$ in which each node $u \in U$ contains a task t_u. w is ground (primitive) if all tasks $\{t_u \,|\, u \in U\}$ are ground (primitive).*

The edges in E define a partial order in the tasks t_u in w. If w is totally ordered, we also write w as the sequence of its tasks, $\langle t_1, ..., t_k \rangle$.

By exploring the space of possible decompositions for the list of tasks in a given state, an HTN algorithm continuously decomposes tasks by replacing them by their sub tasks as defined in the decomposition methods, until the initial set of goal tasks is transformed into a list of atomic operators. Primitive tasks are accomplished by *actions* and non-primitive tasks are decomposed by so-called *methods*.

Definition 9 (HTN method)**.** *An HTN method is 4-tuple*

$$m = (name(m), task(m), precond(m), network(m))$$

in which the elements are described as follows.

- *name*(m)*, the name of the method, is a syntactic expression of the form* $n(x_1, ..., x_k)$*, where* n *is a unique method symbol and* x_i *are all of the variable symbols that occur anywhere in* m*.*
- *task*(m) *is a non-primitive task.*
- *precond*(m) *is a set of literals.*
- *network*(m) *is a task network whose tasks are called the sub tasks of* m*,* *subtasks*(m)*.*

A method m *is totally ordered, iff network*(m) *is totally ordered. A method has the purpose to decompose a non-primitive task into sub tasks given by network*(m)*; task*(m) *specifies the kind of task that is decomposable by* m*, and precond*(m) *restricts the applicability of* m *to certain states.*

Definition 10 (Applicability and relevance of methods)**.** *A method instance* m *is applicable to a state* s*, if* $precond^+(m) \subseteq s$ *and* $precond^-(m) \cap s = \emptyset$*.*

A method instance m *is relevant for a task* t*, if there is a substitution* σ *such that* $\sigma(t) =$*task*(m)*. The decomposition of* t *under* m *is network*(m)*.*

The following Definition 11 is the key concept for the semantics of HTN planning problems. We define how the decomposition of a task t_u in a task network w under a method m performed. Intuitively, merging w with the sub task network comprises replacing the decomposed task node u by the nodes of network(m) and adding constraints between nodes in the original network w and the new nodes in subtasks(m). The HTN formalism presented here (called Simple Task Network (STN) planning in [5]) is easy in that it incrementally builds linear plans in positive time: The algorithm chooses a node without predecessors and decomposes it. This ensures that the plan is build incrementally, starting from the empty plan and linearly adding action in positive time.

Definition 11 (Decomposition of task networks)**.** *Let* $w = (U, E)$ *be a task network,* u *be a node in* w *that has no predecessors in* w*, and* m *be a method that is relevant for* t_u *under some substitution* σ*. Let* $succ(u)$ *be the set of all immediate successors of* u*, i.e.* $succ(u) = \{u' \in U \mid (u, u') \in E\}$*. Let* (U', E') *bet the result of removing from* w *the node* u *and all edges that contain* u *and let* $U'_1 \subseteq U'$ *be the set of nodes in without predecessors in* (U', E')*. Let* (U_m, E_m) *be a copy of network(*m*) and* $U_{m,1} \subseteq U_m$ *be the set of nodes without predecessors in* (U_m, E_m)*. If* (U_m, E_m) *is nonempty, then the result of decomposing* u *in* w *by* m *under* σ *is this set of task networks:*

$$\delta(w, u, m, \sigma) = \{(\sigma(U' \cup U_m), \sigma(E_v)) \mid v \in U_{m,1}\}$$

where

$$E_v = E_m \cup (U_m \times succ(u)) \cup \{(v, u'_1) \mid u'_1 \in U'_1\}$$

Otherwise, $\delta(w, u, m, \sigma) = \{(\sigma(U'), \sigma(E'))\}$*.*

Note that $\delta(w, u, m\sigma)$ is a set of task networks, one for each node without predecessors in network(m). The new task network contains 3 kinds of edges:

1. E_m are the edges of the sub task network, network(m)

2. $U_m \times succ(u)$ are edges from every node in network(m) to every node without predecessor in the original task network. These edges ensure that all task in the sub network will occur before all remaining tasks in the original task network. (Remember that plans are built incrementally and linearly from the beginning to the end.)
3. $\{(v, u'_1) \mid u'_1 \in U'_1\}$ for some $v \in U_{m,1}$ are edges from the node v without predecessor in the sub task network U_m to all nodes without predecessor in the original network, $u'_1 \in U'_1$. These edges enforce that at least one task of network(m) will enter the plan before any of the remaining possible first tasks in the original network which ensures that the method's precondition is true in the correct place in the resulting task network, namely before the first task in network(m). Without these edges, the decomposition of a task in U'_1 might enter the plan before the first task in network(m) and corrupt m's preconditions.

Note: This definition consciously differs from the respective Definition 11.5 in [5] which is faulty for the following reason: With $v \in U_m$ and $\text{succ}_1(u) \subseteq \text{succ}(u)$ it follows $\{(v, u') \mid u' \in \text{succ}_1(u)\} \subseteq (U_m \times \text{succ}(u))$ and thus all E_v are identical. Our revised Definition 11 captures the intent described in the two paragraphs before definition 11.5 in [5] and in the lecture slides [49]. Furthermore, variable passing to sub tasks is not defined properly in [5] and Definition 11. See Appendices B for further explanations on both issues.

Definition 12 (HTN planning domain)**.** *An HTN planning domain is a pair $\mathcal{D} = (O, M)$ where O (M) is set of operators (methods).*

Definition 13 (HTN planning problem)**.** *An HTN planning problem is a 4-tuple $\mathcal{P} = (s_0, w, O, M)$ where s_0 is the initial state, w a task network called initial task network or goal task network and $\mathcal{D} = (O, M)$ is an HTN planning domain.*

We now define what it means for a plan $\pi = \langle a_1, ..., a_k \rangle$ to be a solution for a planning problem $\mathcal{P} = (s_0, w, O, M)$. Intuitively, it means that there is a way to decompose w into π such that π is executable (i.e. every action $a_{i+1} \in \pi$ is applicable in the respective state s_i) and each decomposition is applicable in the appropriate state of the world.

Definition 14 (HTN plans and solutions)**.** *Let $\mathcal{P} = (s_0, w, O, M)$ be a planning problem. Here are the cases in which a plan $\pi = \langle a_1, ..., a_k \rangle$ is a solution for $\mathcal{P}$.*

- *Case 1: w is empty. Then π is a solution for $\mathcal{P}$ iff π is empty.*
- *Case 2: There is a primitive task node $u \in w$ that has no predecessors in w. Then π is a solution for $\mathcal{P}$ iff a_1 is applicable to t_u in s_0 and the plan $\pi = \langle a_2, ..., a_k \rangle$ is a solution for this planning problem:*

  $$\mathcal{P}' = (\gamma(s_0, a_1), w \setminus \{u\}, O, M).$$

  Intuitively, $\mathcal{P}'$ is the planning problem produced by executing the first action in π and removing the corresponding task node from w.
- *Case 3: There is a non-primitive task node $u \in w$ that has no predecessors in w and an instance of m of some method in M that is relevant for t_u under a substitution σ and applicable in s_0. Then π is a solution for $\mathcal{P}$ if there is a task network $w' \in \delta(w, u, m, \sigma)$ such that π is a solution for the planning problem (s_0, w', O, M).*

A.3.1. Partial-order HTN planning

Algorithm 9 Partial-order HTN planning

```
1:  function PFD(s, w, O, M)
2:      if w = ∅ then
3:          return the empty plan
4:      non-deterministically choose any task u ∈ w that has no predecessors in w
5:      if t_u is a primitive task then
6:          active ← {(a, σ) | a is a ground instance of an operator in O,
7:                             σ is a substitution such that name(a) = σ(t_u),
8:                             and a is applicable to s}
9:          if active = ∅ then
10:             return failure
11:         non-deterministically choose any (a, σ) ∈ active
12:         π ← PFD(γ(s, a), σ(w − {u}), O, M)
13:         if π = failure then
14:             return failure
15:         else
16:             return a.π
17:     else
18:         active ← {(m, σ) | m is a ground instance of a method in M,
19:                            σ is a substitution such that name(m) = σ(t_u),
20:                            and m is applicable to s}
21:         if active = ∅ then
22:             return failure
23:         non-deterministically choose any (m, σ) ∈ active
24:         non-deterministically choose any task network w' ∈ δ(w, u, m, σ)
25:         return PFD(s, w', O, M)
```

Algorithm 9 is a direct non-deterministic, provably sound and complete implementation of Definition 14. Lines 5-16 handle case 2 from Definition 14, where a primitive task is accomplished by an action. The action is added to the plan and the algorithm is recursively called on the remaining task network. Lines 17-25 cover case 3 from Definition 14: An applicable and relevant method decomposes the chosen task and the algorithm is called on the decomposed new task network. A deterministic implementation of Algorithm 9 has to make sure that all possible choices are considered (lines 4, 11, 23, 24). This is usually accomplished by a backtracking implementation.

A.3.2. Comparison of HTN and state-space planning

The possibility to call methods recursively allows to encode planning problems whose solution set is a context-free language, whereas state-space planning domains and problems are by definition finite and can therefore encode only regular languages. From the above it follows that HTN planning is more expressive than state-space planning [50]. However,

when restricting the formally undecidable HTN model to make it decidable and hence useful for practical applications by e.g. limiting the plan length or excluding cyclic methods, HTN and state-space planning feature equivalent expressiveness [51]. Example 2 in section A.3.3.2 illustrates the encoding of state-space planning problems in HTN planning as proposed by [50]. The converse encoding is described in [51].

For many domains and problems, obviously especially those with a hierarchic structure, HTN planning offers a user-friendly way of providing heuristics to guide the planning algorithm. Additionally, HTN planning allows for mixed-initiative planning and can be extended with temporal reasoning capabilities (see Section 3.3.1).

A.3.3. HTN planning examples

The following two examples are HTN problem statements written for the SHOP2 planner [52]. The format of the files is defined in [53] and is similar to the Planning Domain Definition Language (PDDL, [42]). The SHOP2 planner implements the partial-order algorithm presented in section A.3.1 and features a couple of extensions to the classical HTN model (e.g. axioms, mathematical calculations, call-expressions, eval expressions, implications, quantifiers, ...).

The planning problem is described by two files, the domain definition file and the problem definition file. The domain file contains definitions for operators, methods and axioms in a straightforward fashion: Every operator is specified by its name including the variables and lists of preconditions, delete effects (effect$^-$) and add effects (effect$^+$). Every method is specified by its name including the variables, a list of preconditions and a sub task network. The network is not specified as a partially ordered set, but by nested lists of ordered and unordered sub tasks (for an example, consider the `drive-kids-around` method in listing A.1).

The problem file defines a HTN planning problem by the associated domain, a set of ground atoms describing the initial state, and a goal task network.

A.3.3.1. How to spend a day

This example defines two planning problems: `spend-a-lazy-day` and `spend-a-busy-day`. Although `spend-a-lazy-day` is an extremely easy example without variables, preconditions and effects, the solutions offer a surprise: The SHOP2 algorithm finds all four solution plans and the second solution shows that unordered (i.e. partial-order) task specification enables interleaved plans.

```
Problem SPEND-A-LAZY-DAY with :WHICH = :ALL, :VERBOSE = :LONG-PLANS
Totals: Plans Mincost Maxcost Expansions Inferences  CPU time  Real time
           4     3.0     4.0         20          7     0.112      0.351
Plans:
((((!BRUSH-TEETH) 1.0 (!BRUSH-TEETH) 1.0 (!LIE-DOWN) 1.0)
 ((!BRUSH-TEETH) 1.0 (!WATCH-TV) 1.0 (!BRUSH-TEETH) 1.0 (!LIE-DOWN) 1.0)
```

```
 ((!BRUSH-TEETH) 1.0 (!BRUSH-TEETH) 1.0 (!WATCH-TV) 1.0 (!LIE-DOWN) 1.0)
 ((!BRUSH-TEETH) 1.0 (!BRUSH-TEETH) 1.0 (!LIE-DOWN) 1.0 (!WATCH-TV) 1.0))
```

The `spend-a-busy-day` example is much more complicated and solutions are not obvious at first glance. The problem is to organise a day with duties such as driving kids to different locations and eating lunch. Resource allocation (choosing different cars with different fuel consumptions for different ways) as well as routing decisions have to be made. The kids have different requirements on the car, Peggy for example wants the `audi` in the morning and the `ferrari` in the afternoon, Kevin wants the `truck` in the morning but has no preference in the afternoon. The solutions include an estimate on the cost (fuel consumption $\times$ distance) of every solution plan.

SHOP2 returns the following four solutions, the cheapest one costs 13904 units, the most expensive one 19304 units.

```
------------------------------------------------------------------------
Problem SPEND-A-BUSY-DAY with :WHICH = :ALL, :VERBOSE = :LONG-PLANS
Totals: Plans Mincost Maxcost Expansions Inferences  CPU time  Real time
           4 13904.0 19304.0        963       4037     0.204      0.816
Plans:
(((!BRUSH-TEETH) 1.0 (!DRIVE MYSELF FERRARI HOME CITY) 1400
  (!DRIVE KEVIN TRUCK CITY SPORT) 4800 (!DRIVE MYSELF TRUCK SPORT HOME) 4200
  (!DRIVE PEGGY AUDI HOME CITY) 600 (!EAT-LUNCH) 1.0
  (!DRIVE MYSELF AUDI CITY SPORT) 480 (!DRIVE MYSELF AUDI SPORT SPORT) 0
  (!DRIVE KEVIN AUDI SPORT HOME) 420 (!DRIVE MYSELF TRUCK HOME CITY) 6000
  (!DRIVE PEGGY FERRARI CITY HOME) 1400 (!BRUSH-TEETH) 1.0 (!LIE-DOWN) 1.0)
 ((!BRUSH-TEETH) 1.0 (!DRIVE MYSELF FERRARI HOME CITY) 1400
  (!DRIVE KEVIN TRUCK CITY SPORT) 4800 (!DRIVE MYSELF TRUCK SPORT HOME) 4200
  (!DRIVE PEGGY AUDI HOME CITY) 600 (!EAT-LUNCH) 1.0
  (!DRIVE MYSELF AUDI CITY SPORT) 480 (!DRIVE MYSELF AUDI SPORT SPORT) 0
  (!DRIVE KEVIN AUDI SPORT HOME) 420 (!DRIVE MYSELF AUDI HOME CITY) 600
  (!DRIVE PEGGY FERRARI CITY HOME) 1400 (!BRUSH-TEETH) 1.0 (!LIE-DOWN) 1.0)
 ((!BRUSH-TEETH) 1.0 (!DRIVE MYSELF FERRARI HOME CITY) 1400
  (!DRIVE KEVIN TRUCK CITY SPORT) 4800 (!DRIVE MYSELF TRUCK SPORT HOME) 4200
  (!DRIVE PEGGY AUDI HOME CITY) 600 (!DRIVE MYSELF AUDI CITY SPORT) 480
  (!EAT-LUNCH) 1.0 (!DRIVE MYSELF AUDI SPORT SPORT) 0
  (!DRIVE KEVIN AUDI SPORT HOME) 420 (!DRIVE MYSELF TRUCK HOME CITY) 6000
  (!DRIVE PEGGY FERRARI CITY HOME) 1400 (!BRUSH-TEETH) 1.0 (!LIE-DOWN) 1.0)
 ((!BRUSH-TEETH) 1.0 (!DRIVE MYSELF FERRARI HOME CITY) 1400
  (!DRIVE KEVIN TRUCK CITY SPORT) 4800 (!DRIVE MYSELF TRUCK SPORT HOME) 4200
  (!DRIVE PEGGY AUDI HOME CITY) 600 (!DRIVE MYSELF AUDI CITY SPORT) 480
  (!EAT-LUNCH) 1.0 (!DRIVE MYSELF AUDI SPORT SPORT) 0
  (!DRIVE KEVIN AUDI SPORT HOME) 420 (!DRIVE MYSELF AUDI HOME CITY) 600
  (!DRIVE PEGGY FERRARI CITY HOME) 1400 (!BRUSH-TEETH) 1.0 (!LIE-DOWN) 1.0))
```

Listing A.1: "How to spend a day" domain definition

```
(in-package :shop2-user)

(defdomain spend-a-day
  (

    (:operator (!brush-teeth)
               ()
               ())

    (:operator (!lie-down)
          ()
          ())

    (:operator (!watch-tv)
          ()
          ())

    (:operator (!eat-lunch)
          ()
          ())

    (:operator (!drive ?person ?car ?from ?to)
          ( (at ?from) (carAt ?car ?from) (fuel-cost ?car ?cost) (
             distance ?from ?to ?dist) )
          ( (at ?from) (carAt ?car ?from) )
          ( (at ?to) (carAt ?car ?to) )
          (call * ?cost ?dist)
          )

    (:method (spend-a-lazy-day)
       (())
       ( (morning) (evening) )
       )

    (:method (spend-a-busy-day)
      (())
  ( (morning) (drive-kids-around) (go-to-bed) )
  )

    (:method (drive-kids-around)
  ((car ?car))
  (:ordered
    (:unordered
      (bring peggy audi home city)
      (bring kevin truck city sport)
    )

    (lunch)

    (:unordered
      (bring kevin ?car sport home)
```

```
51        (bring peggy ferrari city home)
52      )
53    )
54      )
55
56      (:method (bring ?kid ?car ?from ?to)
57    ( (at ?currentPos) (carAt ?firstCar ?currentPos) (carAt ?car ?from) )
58    (:unordered  (!drive myself ?firstCar ?currentPos ?from)
59       (!drive ?kid ?car ?from ?to) )
60      )
61
62      (:method (lunch)
63    ( (at ?currentPos) (carAt ?car ?currentPos) (place ?someWhere) )
64    (:unordered (!eat-lunch) (!drive myself ?car ?currentPos ?someWhere) )
65      )
66
67
68
69
70      (:method (morning)
71         (())
72         ( (!brush-teeth) )
73         )
74
75      (:method (evening)
76         (())
77         ( (go-to-bed) )
78         )
79
80      (:method (evening)
81         (())
82         (:unordered (!watch-tv) (go-to-bed) )
83         )
84
85      (:method (go-to-bed)
86         (())
87         ( (!brush-teeth) (!lie-down))
88         )
89
90
91      ;axioms
92      (:- (same ?x ?x) nil)
93      (:- (distance ?x ?x 0) nil)
94      (:- (distance ?x ?y ?d) (distance ?y ?x ?d) )
95 ))
```

Listing A.2: “How to spend a day” problem statement

```
1 (in-package :shop2-user)
2
3 (make-problem 'spend-a-lazy-day 'spend-a-day
4   '()
5   '( (spend-a-lazy-day) )
```

```
6 )
7
8 (make-problem 'spend-a-busy-day 'spend-a-day
9   '( (place home) (place city) (place sport) (car audi) (car truck) (car
        ferrari)
10      (at home)
11      (carAt audi home) (carAt truck city) (carAt ferrari home)
12      (fuel-cost audi 6) (fuel-cost ferrari 14)  (fuel-cost truck 60)
13      (distance home city 100) (distance home sport 70) (distance sport
           city 80) )
14   '( (spend-a-busy-day) )
15 )
16
17
18
19 (make-problem-set 'spend-a-day-problems '(
20   spend-a-lazy-day spend-a-busy-day
21 ))
```

A.3.3.2. HTN-encoded state-space planning problem Tower of Hanoi

The second example (Listings A.3 and A.4) illustrates the HTN encoding of the renowned state-space planning problem Tower of Hanoi. The state-space formulation is straightforward: The operator `(!move ?from ?to )` moves a disk from one pile to another and has preconditions which enforce the rules of the game.

SHOP2 returns the following solution to the 3-disk Tower of Hanoi planning problem:

```
Problem HANOI-PROBLEM with :WHICH = :ID-FIRST, :VERBOSE = :LONG-PLANS
  Depth Plans Mincost Maxcost Expansions Inferences  CPU time  Real time
     0     0       -       -          1          0     0.012      0.047
     1     0       -       -         11          5     0.056      0.225
     2     0       -       -         14         37     0.052      0.210
     3     0       -       -         35         48     0.004      0.013
     4     0       -       -         44        121     0.000      0.002
     5     0       -       -        106        154     0.004      0.016
     6     0       -       -        128        370     0.008      0.031
     7     0       -       -        292        456     0.012      0.047
     8     0       -       -        356       1034     0.020      0.082
     9     0       -       -        826       1282     0.032      0.129
    10     0       -       -       1004       2952     0.080      0.319
    11     0       -       -       2326       3650     0.108      0.432
    12     0       -       -       2840       8374     0.244      0.975
    13     0       -       -       6656      10392     0.344      1.377
    14     0       -       -       8132      24086     0.696      2.780
    15     0       -       -      19130      29906     1.072      4.289
    16     0       -       -      23420      69488     2.176      8.705
```

```
    17     0       -       -      55406      86426     3.405     13.616
    18     0       -       -      67880     201782     6.760     27.042
    19     0       -       -     161017     251120    10.657     42.617
    20     0       -       -     197446     587487    21.749     86.992
    21     1    13.0    13.0      42704      66417     3.072     12.283
Totals: Plans Mincost Maxcost Expansions Inferences  CPU time  Real time
           1    13.0    13.0     589774    1345587    50.587    202.332
Plans:
(((!MOVE PILE1 PILE2) 1.0 (!MOVE PILE1 PILE3) 1.0 (!MOVE PILE2 PILE3) 1.0
  (!MOVE PILE1 PILE2) 1.0 (!MOVE PILE3 PILE1) 1.0 (!MOVE PILE3 PILE2) 1.0
  (!MOVE PILE1 PILE2) 1.0 (!MOVE-DUMMY) 1.0
    (!ACHIEVE (ON BLOCK2 BLOCK3)) 1.0
  (!ACHIEVE (ON BLOCK1 BLOCK2)) 1.0 (!ACHIEVE (TOP BLOCK1 PILE2)) 1.0
  (!ACHIEVE (TOP TABLE PILE1)) 1.0 (!ACHIEVE (TOP TABLE PILE3)) 1.0))
```

Listing A.3: Tower of Hanoi domain definition

```
(in-package :shop2-user)


(defdomain hanoi (
  (:operator (!move ?from ?to)
  ;precond
  (
    (top ?b1 ?from)
    (top ?b2 ?to)
    (on ?b1 ?underB1)
    (smaller ?b1 ?b2)
  )

  ;delete
  (
    (top ?b2 ?to)
    (top ?b1 ?from)
    (on ?b1 ?underB1)
  )
  ;add
  (
    (top ?b1 ?to)
    (top ?underB1 ?from)
    (on ?b1 ?b2)
  )
  )

  (:operator (!move-dummy)
  ()
  ()
  ()
  )

  (:operator (!achieve ?goal)
```

```
35          (?goal)
36          ()
37          ()
38      )
39
40      (:method (move-method)
41      ((pile ?from) (pile ?to))
42      ((!move ?from ?to) (move-method))
43      )
44
45      (:method (move-method)
46      (())
47      ((!move-dummy))
48      )
49
50
51
52      ; axioms
53      (:- (smaller block1 block2) ())
54      (:- (smaller block1 block3) ())
55      (:- (smaller block1 block4) ())
56      (:- (smaller block1 block5) ())
57      (:- (smaller block2 block3) ())
58      (:- (smaller block2 block4) ())
59      (:- (smaller block2 block5) ())
60      (:- (smaller block3 block4) ())
61      (:- (smaller block3 block5) ())
62      (:- (smaller block4 block5) ())
63      (:- (smaller ?b table) ())
64
65      )
66  )
```

Listing A.4: Tower of Hanoi problem statement

```
1  (in-package :shop2-user)
2
3  (make-problem 'hanoi-problem 'hanoi
4    ; initial state
5    '(
6      (block block1) (block block2) (block block3) (block block4)
7    (pile pile1) (pile pile2) (pile pile3)
8
9    (on block4 table) (on block3 block4) (on block2 block3) (on block1
         block2) (top block1 pile1)
10   (top table pile2)
11   (top table pile3)
12   )
13
14   ; task network
15   '(
16     (move-method)
17     (!achieve(on block4 table))
```

```
18      (!achieve(on block3 block4))
19      (!achieve(on block2 block3))
20      (!achieve(on block1 block2))
21      (!achieve(top block1 pile2))
22      (!achieve(top table pile1))
23      (!achieve(top table pile3))
24    )
25 )
```

A.4. Constraint satisfaction problems

Given a set of variables and their respective domains and a set of constraints specifying the compatible values that the variables may take, constraint satisfaction is the problem of finding a value for each variable such that all values are consistent with the constraints. It is a very general and powerful problem-solving paradigm that is applicable to a broad variety of computational problems. The following sections introduce the basic definitions, a general but naive and in most cases infeasible algorithm to solve arbitrary constraint networks, and the concept of constraint propagation.

A.4.1. Basic definitions

Definition 15 (Constraint satisfaction problem)**.** *A Constraint satisfaction problem (CSP) is a triple* $\mathcal{P} = (X, \mathcal{D}, \mathcal{C})$ *where:*

- $X = \{x_1, ..., x_n\}$ *is a finite set of n variables.*
- $\mathcal{D} = \{D_1, ..., D_n\}$ *is a set of finite domains of the variables,* $x_i \in D_i$*.*
- $\mathcal{C} = \{c_1, ..., c_m\}$ *is a set of constraints. A constraint* c_i *is a relation* R_i *defined on a subset of variables* S_i *with* $S_i \subseteq X$ *and denotes the variables' legal simultaneous value assignments. If the scope* S_i *of a constraint is clear, we will identify the constraint* c_i *with its relation* R_i *or (if the the scope is for example* $S_i = \{x, y, z\}$*) with* R_{xyz} *or* c_{xyz}*.*

Definition 16 (Solution of a CSP)**.** *A solution of a CSP* $\mathcal{P} = (X, \mathcal{D}, \mathcal{C})$ *is an instantiation* $\sigma = (v_1, ..., v_n)$ *of all its variables such that all constraints are satisfied.* $\mathcal{P}$ *is consistent, iff it has a solution.*

Definition 17 (Binary constraint network)**.** *A CSP* $\mathcal{P} = (X, \mathcal{D}, \mathcal{C})$ *is binary iff all constraints* C_i *are binary relations. Binary CSP can be represented as a graph in which every node is a CSP variable* x_i *labelled by its domain* D_i *and each edge* (x_i, x_j) *is labelled by the corresponding constraint* c_{ij}*. A binary CSP is for this reason also called constraint network.*

The symmetrical relation to a binary relation c is defined as $\bar{c} = \{(a, b) \,|\, (b, a) \in c\}$*. A constraint network is symmetrical if for every constraint* c_{ij} *the symmetrical relation is also in* $\mathcal{C}$*.*

Algorithm 10 Backtracking search algorithm for binary CSPs

```
1: function BacktrackCsp(σ, P = (X, D, C))
2:     if X = ∅ then
3:         return σ
4:     select any x_i ∈ X
5:     for all v_j ∈ σ do
6:         D_i ← D_i ∩ {v ∈ D_i | (v, v_j) ∈ c_ij}
7:     if D_i = ∅ then
8:         return failure
9:     non-deterministically choose v_i ∈ D_i
10:    BacktrackCsp(σ.(v_i), (X \ {x_i}, D, C))
```

The complete and sound non-deterministic Algorithm 10 solves arbitrary binary constraint networks and returns all solutions.

Definition 18 (Redundant values and constraints)**.** *A value v in a domain D_i is redundant iff does not appear in any solution, a tuple in a constraint c_j is redundant iff it is not an element of any solution.*

A.4.2. Constraint propagation

Algorithm 10 runs in $O(n^d)$ for $n = |X|$ and $d = max_i\{|D_i|\}$ and – being a large combinatorial problem – is thus generally not suitable to solve real world problems with large domains. Constraint propagation is a filtering technique to reduce the size of the search space by removing redundant values from domains and redundant constraints. Propagating a constraint on a variable x_i consists of accounting for the local effects of constraints between x_i and adjacent nodes in the constraint network by removing redundant values and tuples.

A.4.2.1. Arc-consistency

Definition 19 (Arc-consistency)**.** *Given a constraint network $\mathcal{P} = (X, \mathcal{D}, \mathcal{C})$ with $R_{ij} \in \mathcal{C}$, a variable x_i is arc-consistent to x_j iff for every value $v_i \in D_i$ there exists a value $v_j \in D_j$ such that $(v_i, v_j) \in R_{ij}$. An arc $\{x_i, x_j\}$ is arc-consistent iff x_i is arc-consistent to x_j, and x_j is arc-consistent to x_i. $\mathcal{P}$ is arc-consistent iff all of its arcs are arc-consistent.*

A simple algorithm for arc-consistency is to perform an iteration over all pairs (x_i, x_j) of a constraint network and filter the domains D_i and D_j by:

$$D_i \leftarrow \{v_i \in D_i \,|\, \exists v_j \in D_j \text{ such that } (v_i, v_j) \in R_{ij}\}$$

$$D_j \leftarrow \{v_j \in D_j \,|\, \exists v_i \in D_i \text{ such that } (v_i, v_j) \in R_{ij}\}$$

Arc-consistent constraint networks are not necessarily consistent.

The task networks in Figure A.3 consists of three variables A, B, C with two values each. The arc-consistency algorithm filters the domains to $D_A = \{a_1\}, D_B = \{b_1\}, D_C = \{c_1, c_2\}$.

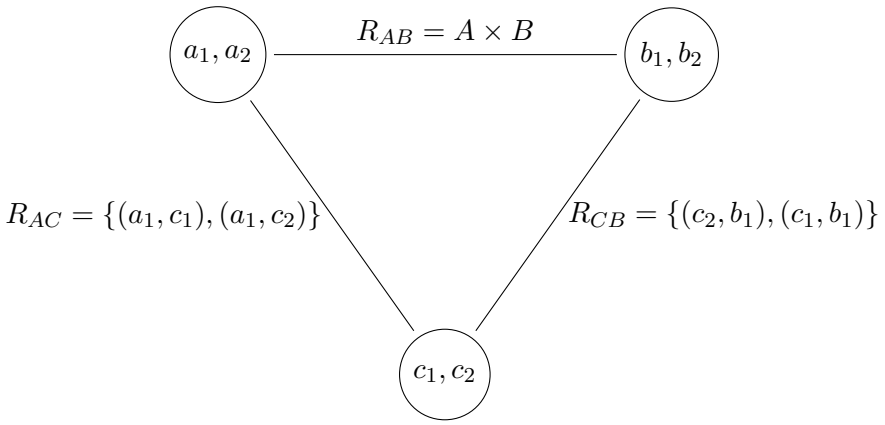


Figure A.3: Illustration of a binary constraint network

A.4.2.2. Path-consistency

The path-consistency filter tests all triples of variables x_i, x_j, x_k by checking that they have values that meet the three constraints c_{ij}, c_{ik}, c_{ik}. A pair of values (v_i, v_j) can only be part of a solution, if it meets the constraint c_{ij} and if there is a value v_k for x_k such that (v_i, v_k) meets c_{ik} and (v_k, v_j) meets c_{kj}. The two constraints c_{ik} and c_{kj} entail by transitivity a constraint on x_i and x_j, called the composition $c_{ik} \bullet c_{kj}$:

$$c_{ik} \bullet c_{kj} = \{(v_i, v_j), v_i \in D_i, v_j \in D_j \mid \exists v_k \in D_k : (v_i, v_k) \in c_{ik} \text{ and } (v_k, v_j) \in c_{kj}\}$$

A pair (v_i, v_j) has to meet c_{ij} as well as the composition $c_{ik} \bullet c_{kj}$ for every k, otherwise it is redundant. Hence the filtering operation is:

$$c_{ij} \leftarrow c_{ij} \cap [c_{ik} \bullet c_{kj}], \text{ for every } k \neq i, j$$

Algorithm 11 is the straightforward algorithm for path-consistency.

In the example of Figure A.3, the composition $c_{AC} \bullet c_{CB}$ is $\{(a_1, b_1)\}$ and the constraint $c_{AB} = A \times B$ can be replaced by $c_{AB} \leftarrow [A \times B] \cap \{(a_1, b_1)\} = \{(a_1, b_1)\}$ without changing the solutions of the constraint network.

Algorithm 11 Path-consistency in a constraint network

```
1: function PathConsistency(C)
2:     while not all constraints in C are stabilized do
3:         for all k: 1 ≤ k ≤ n do
4:             for all pairs (i, j): 1 ≤ i < j ≤ n; i, j ≠ k do
5:                 c_ij ← c_ij ∩ [c_ik • c_kj]
6:                 if c_ij = ∅ then
7:                     fail(inconsistent)
```

The notion of path-consistency can be extended to checking the consistency of all k-tuples in the constraint network. For $k = n$, k-consistency is obviously equivalent to consistency of the entire network.

A.5. Quantitative temporal constraints

Constraint satisfaction techniques can be applied to temporal reasoning which comprises temporal knowledge bases, consistency checking and inference capabilities to discover new temporal information from existing. For qualitative temporal reasoning there exist two major approaches: Interval algebra [54] and point algebra [55] use time intervals and time points as temporal objects, respectively. The temporal constraint satisfaction problem (TCSP) [34,35] is a special constraint satisfaction problem where variables are time points and temporal information is represented by a set of unary and binary constraints.

Definition 20 (Temporal constraint satisfaction problem [34, 35])**.** *A temporal constraint satisfaction problem (TCSP) involves a set of variables* $\{x_1, ..., x_n\}$ *having continuous domains; each variable represents a time point. Each constraint is represented by a set of intervals* $\{I_1, ..., I_k\} = \{[a_1, b_1], ..., [a_k, b_k]\}$*. An unary constraint* T_i *restricts the domain of variable* x_i *to the given set of intervals; that is, it represents the disjunction*

$$(a_1 \leq x_i \leq b_1) \vee ... \vee (a_k \leq x_i \leq b_k)$$

A binary constraint T_{ij} *constraints the permissible values for the distance* $x_j - x_i$*; it represents the disjunction*

$$(a_1 \leq x_j - x_i \leq b_1) \vee ... \vee (a_k \leq x_j - x_i \leq b_k)$$

A network of binary constraint (a binary TCSP) consists of a set of variables $\{x_1, ..., x_n\}$ and a set of unary and binary constraints. Such a network can be represented by a directed constraint graph G, where nodes represent variables and an edge $i \rightarrow j$ indicates that a constraint T_{ij} is specified between x_i and x_j; it is labelled by the interval set. Each constraint T_{ij} implies an equivalent symmetrical constraint T_{ji}. However, only one of these is usually shown in the constraint graph. A special time point, x_0, is introduced to represent the "beginning of the world". All times are relative to x_0; thus, we may treat each unary constraint T_i as a binary constraint T_{0i}.

Definition 21 (Minimal and binary decomposable networks)**.** *Given a TCSP, a value* v *is a feasible value for variable* x_i *if there exists a solution in which* $x_i = v$*. The set of all feasible values of a variable is called the minimal domain. A minimal constraint* T_{ij} *between* x_i *and* x_j *is the set of all feasible values for* $x_j - x_i$*. A network is minimal iff its domains and constraints are minimal. A network is binary decomposable if every consistent assignment of values to a set of variables* S *can be extended to a solution.*

Definition 22 (Simple temporal problem)**.** *A TCSP in which all constraints specify a single interval is called a simple temporal problem (STP). In such a network, each edge* $i \rightarrow j$ *is labelled by a single interval* $[a_{ij}, b_{ij}]$ *that represents the constraint*

$$a_{ij} \leq x_j - x_i \leq b_{ij}.$$

An STP can be associated with a directed edge-weighted graph $G_d = (V, E_d)$, called a distance graph. It has the same node set as the constraint graph G, and each edge $i \rightarrow j \in E_d$ is labelled by a weight a_{ij} representing the linear inequality $x_j - x_i \leq a_{ij}$. The d-Graph is

similar to the distance graph in that is has the same node set, but each edge $i \to j$ is defined to be labelled by the shortest path length, d_{ij}, from node i to node j in G_d.

Simple temporal problems feature especially nice algorithmic properties, as formalised by the following theorems [34, 35, 56].

Theorem 1 (Consistency of STPs [35])**.** *An STP is consistent iff its distance graph G_d has no negative cycles.*

Theorem 2 (Decomposability)**.** *Any consistent STP is decomposable relative to the constraints in its d-graph.*

Theorem 3 (Minimal representation of STPs)**.** *Given a consistent STP, T, and its d-graph, the equivalent STP, M, defined by the edges d_{ij} in T's d-graph,*

$$\forall i, j : \; M_{ij} = \{[-d_{ji}, d_{ij}]\}$$

is the minimal network representation of T.

Algorithm 12 Floyd-Warshall's all-pairs-shortest-path algorithm

```
1: function FLOYD-WARSHALL((V, E))
2:     for all i, j ∈ V do
3:         if (i, j) ∈ E then
4:             d(i, j) ← l_ij                    ▷ l_ij is the label of the edge i → j
5:         else
6:             d(i, j) ← ∞
7:         d(i, i) ← 0
8:     for all i, j, k ∈ V do                    ▷ Time complexity O(|V|^3)
9:         d(i, j) ← min{d(i, j), d(i, k) + d(k, j)}
```

The d-graph of an STP can be constructed by applying Floyd-Warshall's all-pairs-shortest-path algorithm (see Algorithm 12) to the distance graph. The algorithm provides a consistency check of the STP (it is consistent *iff* all diagonal distances d_{ii} are non-negative) and computes the minimal domains and constraints in $O(n^3)$ time. Due to Theorem 2, assembling a solution from the d-graph doesn't not require backtracking and can be accomplished in $O(n^2)$.

An alternative and equivalent method for checking consistency and finding the minimal network makes use of the path-consistency algorithm. It turns out [34, 56] that path-consistency is equivalent to consistency for STPs and that the path-consistency Algorithm 11 reaches a fixed point after a single iteration and that the fixed point corresponds to the minimal network. The composition and intersection operations for STPs are defined as follows:

- Composition: $r_{ij} \bullet r_{jk} = [a_{ij} + a_{jk}, b_{ij} + b_{jk}]$
- Intersection: $r_{ij} \cap r'_{ij} = [\max\{a_{ij}, a'_{ij}\}, \max\{b_{ij}, b'_{ij}\}]$

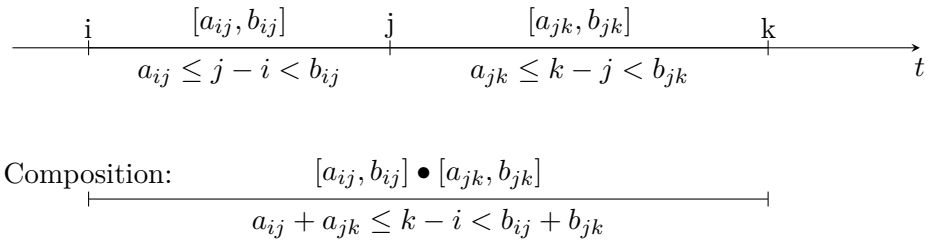


Figure A.4: The composition operation for STPs

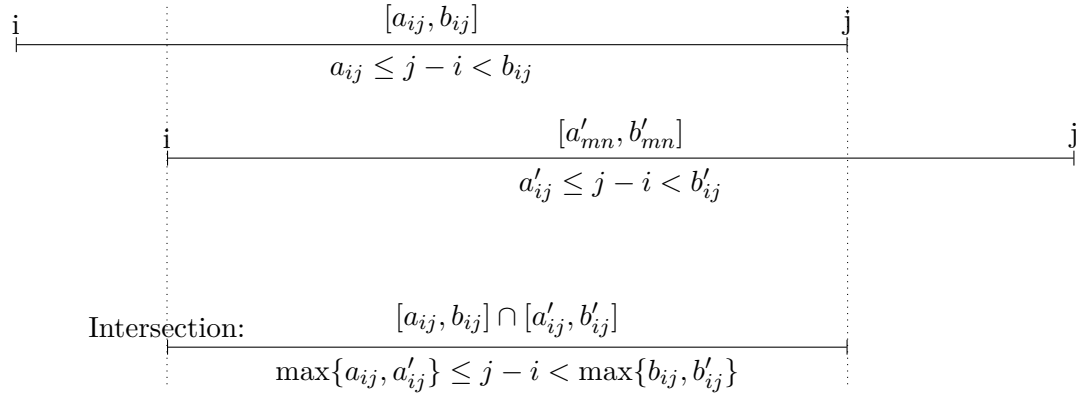


Figure A.5: The intersection operation for STPs

In analogy to the general CSP case, composition corresponds to entailed constraints between three time points and intersection corresponds to enforcing two different constraints on a pair of time points, see Figures A.4 and A.5.

Theorem 4 (Path-consistency for STPs)**.** *Given an STP T, the path-consistency algorithm is sound and complete for deciding the consistency problem and reaches a fixed point in one iteration. The fixed point corresponds to the minimal network of T. Thus, for STPs, path-consistency and the all-pairs-shortest-path algorithm have equivalent properties with respect to consistency checking and minimal network computation.*

In contrast to the network-based approach, the path-consistency algorithm allows for more efficient consistency checking if temporal information is added incrementally to the STP. The incremental version of Algorithm 11 can be used to efficiently propagate new temporal information, check consistency and compute the minimal STP.

B. Revised semantics of partial-order forward decomposition HTN planning

B.1. Ordering of sub tasks

In [5], the semantics of HTN planning are specified by the recursive Definition 11.8 of a *solution plan* for an HTN planning problem. Case 1 defines an empty plan to be the solution for the empty task network, and cases 2 and 3 define the semantics of actions and methods, respectively. A method decomposes a given task by its sub tasks and includes ordering constraints to ensure the consistency of the resulting task network with respect to qualitative orderings. The decomposition is formalised by Definition 11.5:

Original Definition 11.5
Let $w = (U, E)$ be a task network, u be a node in w that has no predecessors in w, and m be a method that is relevant for t_u under some substitution σ. Let $\text{succ}(u)$ be the set of all immediate successors of u, i.e. $\text{succ}(u) = \{u' \in U \mid (u, u') \in E\}$. Let $\text{succ}_1(u)$ be the set of all immediate successors of u for which u is the *only* predecessor. Let (U', E') be the result of removing u and all edges that contain u. Let (U_m, E_m) be a copy of network(m). If (U_m, E_m) is nonempty, then the result of decomposing u in w by m under σ is this set of task networks:

$$\delta(w, u, m, \sigma) = \{(\sigma(U' \cup U_m), \sigma(E_v)) \mid v \in \text{subtasks}(m)\}$$

where

$$E_v = E_m \cup (U_m \times \text{succ}(u)) \cup \{(v, u') \mid u' \in \text{succ}_1(u)\}$$

Otherwise, $\delta(w, u, m, \sigma) = \{(\sigma(U'), \sigma(E'))\}$.

I would like to argue that Definition 11.5 does not capture the intent of the explanations in the first two paragraphs on page 235 in [5] and in slide 25 in [49]. The set of edges in δ, E_v consists of three different types of edges:

1. E_m is the partial order on the sub tasks, as defined by the method m.

2. The edges $U_m \times \text{succ}(u)$ order every sub task *before* all successors $\text{succ}(u)$ of the decomposed task u. By definition, there are no predecessors of u.

3. The third class of edges has the following justification: In the resulting plan, the preconditions of method m shall be satisfied right before the first action which accomplishes a sub tasks of m. If network(m) is partially ordered, there can be more than one such possible first task node v in network(m) and the algorithm has to decide for one of them which consequently must be ordered before all possible first tasks of the original task network, w. Thus, the decomposition results in different task networks, one for every possible first task in network(m). However, Definition 11.5 does not provide these ordering constraints. Furthermore, Definition 11.5 looks faulty for the following formal reason:

$$v \in U_m \wedge \ \mathrm{succ}_1(u) \subseteq \mathrm{succ}(u) \Longrightarrow \{(v, u') \mid u' \in \mathrm{succ}_1(u)\} \subseteq (U_m \times \mathrm{succ}(u))$$

The above is true for all $v \in \mathrm{subtasks}(m)$ and therefore all sets E_v are identical.

While the edges of type 1 and 2 are intuitively clear, those of type 3 need more justification. One could even argue that the semantic of preconditions of methods is irrelevant in solution plans and that the correct ordering is enforced by the preconditions of actions.

Trying to capture the semantics intended by the first two paragraphs on page 235 and by the above-mentioned slide 25, I propose the following definition instead:

Revised Definition 11.5
Let $w = (U, E)$ be a task network, u be a node in w that has no predecessors in w, and m be a method that is relevant for t_u under some substitution σ. Let $\mathrm{succ}(u)$ be the set of all immediate successors of u, i.e. $\mathrm{succ}(u) = \{u' \in U \mid (u, u') \in E\}$. Let (U', E') bet the result of removing u and all edges that contain u and $U'_1 \subseteq U'$ be the set of nodes without predecessors in (U', E'). Let (U_m, E_m) be a copy of network(m) and $U_{m,1} \subseteq U_m$ be the set of nodes without predecessors in (U_m, E_m). If (U_m, E_m) is nonempty, then the result of decomposing u in w be m under σ is this set of task networks:

$$\delta(w, u, m, \sigma) = \{(\sigma(U' \cup U_m), \sigma(E_v)) \mid v \in U_{m,1}\}$$

where

$$E_v = E_m \cup (U_m \times \mathrm{succ}(u)) \cup \{(v, u'_1) \mid u'_1 \in U'_1\}$$

Otherwise, $\delta(w, u, m, \sigma) = \{(\sigma(U'), \sigma(E'))\}$.

The definition ensures that δ is a set of task networks, one for each possible first task in network(m). E_v contains ordering constraints from the possible first node v in network(m) to a all possible first task nodes in the original task network, ensuring that task v is chosen before them and that precond(m) is valid before the action accomplishing the first task in network(m).

B.2. Variable backwards passing

Listing B.1: Variable passing in HTN: exemplarily domain definition

```
(defdomain binding (
  (:operator (!op1 ?x) (exist ?x) () ())
  (:operator (!op2 ?x) (exist ?x) () ())

  (:method (do)
    ; precondition
    ()

    ; sub tasks
    ((!op1 ?x) (!op2 ?y))
  )
))
```

Listing B.2: Variable passing in HTN: exemplarily problem definition

```
(defproblem problem binding
  ; initial state
  ((exist a) (exist b))

  ; goal task
  ((do))
)
```

Consider the HTN planning problem defined by Listings B.1 and B.2 in SHOP syntax. The method `do` decomposes into two sub tasks, `!op1 ?x` and `!op2 ?y`. The initial state consists of two atoms, `(exist a)` and `(exist b)` and the JSHOP HTN planner [53,57], not surprisingly, finds the four solution plans

1. `!op1 a, !op2 a`
2. `!op1 a, !op2 b`
3. `!op1 b, !op2 a`
4. `!op1 b, !op2 b`

Listing B.3: Variable passing in HTN: exemplarily domain definition

```
(defdomain binding (
  (:operator (!op1 ?x) (exist ?x) () ())
  (:operator (!op2 ?x) (exist ?x) () ())

  (:method (do)
    ; precondition 1
    ()

    ; precondition 2
    ; ((exist ?x))

    ; sub tasks
    ((!op1 ?x) (!op2 ?x))
  )
))
```

Next, consider the same problem B.2 with the domain specified by B.3. With the empty precondition 1 this is exactly the same planning problem, except for the variable specification of the method's sub tasks: Now the same variable identifier is used for both sub tasks. Intuitively, one expects all plans to reflect this choice in that the variable bindings for `?x` should be the same for `!op1 ?x` and `!op2 ?x`. However, both the HTN formulation given in [5][1], and the JSHOP implementation treat variables differently: As `?x` is unbound in the `do` method, the method is applicable with the empty substitution, $\sigma = \emptyset$. After the decomposition, the variables `x` in `op1` and `op2` are *local* to their respective operators and thus independent; the substitutions are not passed backwards to the method. The counter-intuitive solution plans are thus

1. `!op1 a, !op2 a`
2. `!op1 a, !op2 b`
3. `!op1 b, !op2 a`
4. `!op1 b, !op2 b`

This problem vanishes if all variables occurring in sub tasks are previously bound by a precondition, as in case 2 of Listing B.3. Here, the solution plans are

1. `!op1 a, !op2 a`
2. `!op1 b, !op2 b`

To demonstrate the different semantic of variables in HTN and classical logic programming, consider Listing B.4 which shows a Prolog program which is analogous to the previous HTN examples. The solutions to `?method(X)` are as expected:

```
?- method(X).
X = a ;
X = b.
```

The solutions correspond to the proof

1. `op1(a)` ← `exist(a)`, `op2(a)` ← `exist(a)`
2. `op1(a)` ← `exist(a)`, `op2(a)` ← `exist(a)`

Changing the definition of the `method` predicate to `method(X,Y) :- op1(X), op2(Y)` adds the two other solutions corresponding to

1. `op1(a)` ← `exist(a)`, `op2(b)` ← `exist(b)`
2. `op1(b)` ← `exist(b)`, `op2(a)` ← `exist(a)`

[1] The exact semantics of variable passing in [5] depends on the notion of *substitutions*, which is not clearly formalised in the book. Since *relevancy* of a method m for a task m under some substitution σ is also defined for unground methods (see Definition 11.4 in [5]), a task may be decomposed into sub tasks containing variables with the same name which can be assigned different values in the ongoing planning process. This is exactly the counter-intuitive handling of variables with the same identifier.

```
?- method(X,Y).
X = a,
Y = a ;
X = a,
Y = b ;
X = b,
Y = a ;
X = b,
Y = b.
```

Listing B.4: Variable passing in Prolog

```
% Database
exist(a). exist(b).

% Definition of method and operators
method(X) :- op1(X), op2(X).
op1(X) :- exist(X).
op2(X) :- exist(X).
```

C. Additional listings and UML diagrams

C.1. ChangeRefinery HTN algorithm

Listing C.1: Java code of the ChangeRefinery HTN implementation

```
1  package da.planner.htn;
2
3  import java.util.ArrayList;
4  import java.util.Collection;
5  import java.util.Date;
6  import java.util.HashMap;
7  import java.util.HashSet;
8  import java.util.List;
9  import java.util.Map;
10 import java.util.Set;
11
12 import da.knowledgebase.core.KnowledgeBase;
13 import da.planner.stp.StpHtnConnector;
14 import da.util.Log;
15
16 public class Htn {
17   private KnowledgeBase kb;
18   private Plan currentPlan;
19   private StpHtnConnector stpHtn = new StpHtnConnector();
20   private HtnStatistics stats = new HtnStatistics();
21   private ChoicePointFactory<Task> taskCPF = new
       SimpleChoicePointFactory<Task>();
22   private ChoicePointFactory<Refinement> refinementCPF = new
       SimpleChoicePointFactory<Refinement>();
23   private ChoicePointFactory<Binding> bindingCPF = new
       SimpleChoicePointFactory<Binding>();
24   public void setTaskCPF(ChoicePointFactory<Task> c){this.taskCPF = c;}
25   public void setRefinementCPF(ChoicePointFactory<Refinement> c){this.
       refinementCPF = c;}
26   public void setBindingCPF(ChoicePointFactory<Binding> c){this.
       bindingCPF = c;}
27   private TaskModifier taskModifier = new TaskModifier(false);
28   public void setTaskModifier(TaskModifier t){this.taskModifier = t;}
29
30   public Htn(KnowledgeBase kb) {
31     this.kb = kb;
32   }
33
34   public HtnStatistics findPlans(TaskNetwork tn) {
```

```
35      Long t = new Date().getTime();
36      this.currentPlan = new Plan();
37      this.stpHtn.add(tn.getNodes(), tn.getEdges());
38      this.pfd(tn);
39      this.stats.totalTime = new Date().getTime() - t;
40      return this.stats;
41    }
42
43
44    public boolean pfd(TaskNetwork tn) {
45      if (tn.getNodes().size() == 0) {
46        /*
47         * the base case of the recursive pfd invocations:
48         * the empty plan accomplished an empty task network.
49         */
50        Plan foundplan = this.currentPlan.clone();
51        foundplan.setStpHtn(this.stpHtn.clone());
52        this.stats.addPlan(foundplan);
53        Log.l("Found new plan (" + this.stats.plans.size() + "): " +
              foundplan.toString(), 1);
54        return true;
55      }
56
57      Log.l("Remaining task network: \n"
58          + "\tPreliminary plan: " + this.currentPlan + "\n"
59          + "\tRemaining tasks: " + tn.getTaskListString()
60          , 16);
61
62        /*
63         * find all tasks that have no predecessor in the task network.
64         * all of those are candidates for the next task to be refined.
65         */
66        List<Task> firstNodes = new ArrayList<Task>();
67        for(Task t: tn.getNodesWithoutPredecessor())
68          firstNodes.add(t);
69
70        String msg = "Nodes without predecessor: "; for(Task task:
              firstNodes){msg+=task.getTaskDefinition().getName();msg+=",";}
71        Log.l(msg, 32);
72
73        // choose one of the tasks without predecessors to be refined
74        for(Task task: this.taskCPF.get(firstNodes, "FIRST TASK")){
75
76
77          // find all refinements for <task> that have a satisfied
                precondition
78          List<Refinement> active = task.getTaskDefinition().
                getActiveRefinements(kb, task.getBinding());
79          for(Refinement performer: this.refinementCPF.get(active, "
                REFINEMENT OPERATION")) {
80
81            Log.l("Calculating bindings for task " + task.getName() + "
                  and refinement " + performer.getName(), 16);
```

```
82          List<Binding> bindings = performer.computeBindings(this.kb,
                task.getBinding());
83          for(Binding b: this.bindingCPF.get(bindings, "BINDING")){
84
85            // save the knowledge base ids which are part of the query
86            Set<Long> queryDependencies = new HashSet<Long>();
87            for(Variable v: b.values())
88              if(v.getClass() == KbElementVariable.class){
89                KbElementVariable k = (KbElementVariable) v;
90                if(k.getValue() != null)
91                  queryDependencies.add(k.getValue());
92              }
93
94
95
96            /*
97             * add task's binding to the bindings returned by the
                    refinement query.
98             * this is necessary because the refinement query might not
                    contain
99             * all variables that may be used in a decomposition method.
                    ie, a method
100            * can use the task's variables as well as those variable
                    from its
101            * precondition.
102            */
103           b.add(task.getBinding());
104
105
106           // deep clone the task network, all the tasks and variables.
107           Long t = new Date().getTime();
108           Map<Task, Task> taskMapping = new HashMap<Task, Task>();
109           Map<Variable, Variable> variableMapping = new HashMap<
                Variable, Variable>();
110           TaskNetwork tnCopy = tn.clone(taskMapping, variableMapping);
111           Binding bCopy = b.clone(variableMapping);
112           this.stats.taskNetworkCopyTime += new Date().getTime() - t;
113
114
115
116           if(performer.isOperator()) {
117             Operator o = (Operator) performer;
118             Log.l("Applying␣operator␣" + o.getName() + "␣to␣task␣" +
                    task.getTaskDefinition().getName(), 16);
119
120
121             // apply operator with selected binding
122             Set<Long> effectDependencies = this.kb.assertToKb(o.
                    getEffects(), bCopy);
123             // create an action object from the operator o
124             Action a = new Action(o, queryDependencies,
                    effectDependencies, bCopy);
125
```

```
126
127             /*
128              * find additional ordering constraints between the
129              * current action a and previous actions ap. if the
130              * precondition or effect of a previous action ap depend
                   on
131              * a CI which is modified by the effect of the current
132              * action a or part of the query of the current action,
133              * we add the ordering constraint ap -> a
134              * to ensure that effects do not invalidate other
135              * actions preconditions
136              */
137             List<OrderingConstraint> additionalOrderingConstraints =
                    new ArrayList<OrderingConstraint>();
138             for(Action previousAction: this.currentPlan)
139               if(a.preconditionDependsOnOtherActionsEffects(
                      previousAction)
140                   || previousAction.
                          preconditionDependsOnOtherActionsEffects(a))
141                 additionalOrderingConstraints.add(new
                        OrderingConstraint(previousAction, a));
142
143
144             // add a to the current HTN plan
145             this.currentPlan.add(a);
146             // maintain temporal constraints in the HTN by decomposing
                    task t into action a
147             this.stpHtn.decomposeTo(taskMapping.get(task), a,
                    additionalOrderingConstraints);
148
149
150             // copy chosen variable bindings to the accomplished task
151             Task newTask = taskMapping.get(task);
152             for(String vName: newTask.getBinding().keySet()){
153               if(b.containsKey(vName)){
154                 newTask.getBinding().get(vName).assignFrom(bCopy.get(
                        vName));
155               }
156             }
157
158
159             /* if the current plan is temporally consistent,
160              * recursively call pfd with a copy of the
161              * task network minus the achieved task
162              */
163             if(this.stpHtn.PC()){
164               tnCopy.removeNodeWithEdges(newTask);
165               this.pfd(tnCopy);
166             } else
167               Log.l("Partial plan is temporally inconsistent, STP:\n"
                      + this.stpHtn, 16);
168
169
```

```
170
171             // backtracking comprises...
172             Log.l("Backtracking", 16);
173             // ... removing o from the plan ...
174             this.currentPlan.removeLast();
175             // ... rolling back the knowledge base ...
176             this.kb.rollback();
177             // ... and the STP.
178             this.stpHtn.rollback();
179
180
181           } else {
182             // method handling
183             Method m = (Method) performer;
184             Log.l("Applying method " + m.getName() + " to task " +
                    task.getTaskDefinition().getName(), 16);
185
186
187             // apply the method to the task network.
188             Collection<OrderingConstraint> newEdges = m.perform(tnCopy
                    , taskMapping.get(task), bCopy);
189             TaskNetwork subNetwork = m.getSubNetwork(bCopy);
190             this.taskModifier.modify(subNetwork.getNodes());
191
192             // decompose the task into its sub tasks in the STP
193             this.stpHtn.decomposeTo(taskMapping.get(task), subNetwork.
                    getNodes() , newEdges);
194
195
196
197             /* if the current plan is temporally consistent,
198              * recursively call pfd on a copy of the decomposed
199              * task network
200              */
201             if(this.stpHtn.PC())
202               this.pfd(tnCopy);
203             else
204               Log.l("Partial plan is temporally inconsistent, STP:\n"
                      + this.stpHtn, 16);
205             Log.l("Backtracking", 16);
206
207
208             // no rollback required on HTN side because the new branch
209             // operators on a copy of the task net work. only rollback
                    STP.
210             this.stpHtn.rollback();
211           }
212         }
213       }
214
215     }
216
217   return false;
```

```
218   }
219 }
```

C.2. Example scenarios

Listing C.2: JAVA code for the example domain VerySimpleJ2eeScenario

```
1  package da.scenarios.j2ee.kb;
2
3  import org.hibernate.Session;
4
5  import da.knowledgebase.core.Assertable;
6  import da.knowledgebase.model.Database;
7  import da.knowledgebase.model.DatabaseDesign;
8  import da.knowledgebase.model.DbServer;
9  import da.knowledgebase.model.ECommerceJ2eeApplication;
10 import da.knowledgebase.model.Hardware;
11 import da.knowledgebase.model.Human;
12 import da.knowledgebase.model.J2eeApplication;
13 import da.knowledgebase.model.J2eeContainer;
14 import da.knowledgebase.model.J2eeJdbcResource;
15 import da.knowledgebase.model.Skill;
16 import da.knowledgebase.model.WebServer;
17 import da.knowledgebase.model.WebserverConfiguration;
18 import da.planner.htn.Binding;
19
20 public class VerySimpleJ2eeScenario implements Assertable{
21   public void doAssert(Session d, Binding b) {
22
23
24
25     // DbServer for all applications
26     DbServer dbserver = new DbServer("MiscDbServer");
27     dbserver.setCpuSpeed(1000);
28     dbserver.setMemorySize(1000);
29     d.save(dbserver);
30
31
32     // Setup Server for Wiki
33     {
34       WebServer ws = new WebServer("WikiServer");
35       ws.setCpuSpeed(1000);
36       ws.setMemorySize(1000);
37       d.save(ws);
38       Database wikidb = new Database("WikiDb");
39       dbserver.getDatabases().add(wikidb);
40       J2eeContainer jc = new J2eeContainer("WikiContainer");
41       ws.getJ2eeContainers().add(jc);
42
43       J2eeApplication ja = new J2eeApplication("Wiki Application");
44       jc.getJ2eeApplications().add(ja);
```

```
45        J2eeJdbcResource jdbcr = new J2eeJdbcResource("WikiJdbcResource");
46        ja.getJ2eeJdbcResources().add(jdbcr);
47        jdbcr.setDatabase(wikidb);
48      }
49
50      // Setup Server for E-Commerce application
51      {
52        WebServer ws = new WebServer("ECommerceServer");
53        ws.setCpuSpeed(1000);
54        ws.setMemorySize(1000);
55        d.save(ws);
56        Database wikidb = new Database("ECommerceDb");
57        dbserver.getDatabases().add(wikidb);
58        J2eeContainer jc = new J2eeContainer("ECommerceContainer");
59        ws.getJ2eeContainers().add(jc);
60
61        J2eeApplication ja = new ECommerceJ2eeApplication("ECommerce␣
              Application");
62        jc.getJ2eeApplications().add(ja);
63        J2eeJdbcResource jdbcr = new J2eeJdbcResource("
              ECommerceJdbcResource");
64        ja.getJ2eeJdbcResources().add(jdbcr);
65        jdbcr.setDatabase(wikidb);
66      }
67
68
69
70
71      // People and skills
72      {
73        Human mandyloreen = new Human("Mandy-Loreen", 10); d.save(
              mandyloreen);
74        Skill s41 = new DatabaseDesign(100);
75        Skill s42 = new Hardware(95);
76        Skill s43 = new WebserverConfiguration(90);
77        mandyloreen.getSkills().add(s41);
78        mandyloreen.getSkills().add(s42);
79        mandyloreen.getSkills().add(s43);
80      }
81
82    }
83  }
```

Listing C.3: JAVA code for the example domain J2eeScenario

```
1  package da.scenarios.j2ee.kb;
2
3  import org.hibernate.Session;
4
5  import da.knowledgebase.core.Assertable;
6  import da.knowledgebase.model.Database;
7  import da.knowledgebase.model.DatabaseDesign;
8  import da.knowledgebase.model.DbServer;
```

```
9  import da.knowledgebase.model.Hardware;
10 import da.knowledgebase.model.Human;
11 import da.knowledgebase.model.J2eeApplication;
12 import da.knowledgebase.model.J2eeContainer;
13 import da.knowledgebase.model.J2eeJdbcResource;
14 import da.knowledgebase.model.J2eeModule;
15 import da.knowledgebase.model.JVM;
16 import da.knowledgebase.model.LoadBalancer;
17 import da.knowledgebase.model.MySqlDb;
18 import da.knowledgebase.model.Skill;
19 import da.knowledgebase.model.WebServer;
20 import da.knowledgebase.model.WebserverConfiguration;
21 import da.planner.htn.Binding;
22
23 public class J2eeScenario implements Assertable{
24   private int num;
25   public J2eeScenario(int num){
26     this.num = num;
27   }
28
29   public void doAssert(Session d, Binding b) {
30
31     // Webshop Infrastructure
32     {
33       LoadBalancer webshoplb = new LoadBalancer("WebshopLoadBalancer");
34       d.save(webshoplb);
35       Database webshopdb = new MySqlDb("WebShopDb");
36       DbServer dbserver = new DbServer("WebshopDbServer");
37       dbserver.setCpuSpeed(1000);
38       dbserver.setMemorySize(2000);
39       dbserver.getDatabases().add(webshopdb);
40       d.save(dbserver);
41
42
43       for(int i=1; i<=num; i++) {
44         WebServer ws = new WebServer("WebshopServer" + i);
45         ws.setCpuSpeed(1000);
46         ws.setMemorySize(2000);
47         d.save(ws);
48
49         J2eeContainer jc = new J2eeContainer("WebshopContainer" + i);
50         ws.getJ2eeContainers().add(jc);
51         webshoplb.getBalancedContainers().add(jc);
52         JVM jvm = new JVM("jvm1.5-" + i, "1.5");
53         jc.setJvm(jvm);
54         J2eeModule jma = new J2eeModule("dom4j Module" + i);
55         jc.getJ2eeModules().add(jma);
56         J2eeModule jmb = new J2eeModule("log4j Module" + i);
57         jc.getJ2eeModules().add(jmb);
58         J2eeApplication ja = new J2eeApplication("Webshop Application" +
               i);
59         jc.getJ2eeApplications().add(ja);
```

```
60          J2eeJdbcResource jdbcr1 = new J2eeJdbcResource("
                WebshopJdbcResource" + i);
61          ja.getJ2eeJdbcResources().add(jdbcr1);
62          jdbcr1.setDatabase(webshopdb);
63
64
65        }
66      }
67
68
69
70      // DbServer for all other application
71      DbServer dbserver = new DbServer("MiscDbServer");
72      dbserver.setCpuSpeed(1000);
73      dbserver.setMemorySize(1000);
74      d.save(dbserver);
75
76
77
78
79      // Setup Server for Wiki, Expense Management and Calendar
80      {
81        WebServer ws = new WebServer("MiscServer");
82        ws.setCpuSpeed(1000);
83        ws.setMemorySize(1000);
84        d.save(ws);
85        Database wikidb = new Database("WikiDb");
86        Database expensedb = new Database("ExpenseDb");
87        Database calendardb = new Database("CalendarDb");
88        dbserver.getDatabases().add(wikidb);
89        dbserver.getDatabases().add(expensedb);
90        dbserver.getDatabases().add(calendardb);
91        J2eeContainer jc = new J2eeContainer("WebshopContainer");
92        ws.getJ2eeContainers().add(jc);
93        JVM jvm = new JVM("jvm1.5", "1.4");
94        jc.setJvm(jvm);
95        J2eeModule jma = new J2eeModule("Ajax Module");
96        jc.getJ2eeModules().add(jma);
97
98
99        J2eeApplication ja = new J2eeApplication("Calendar Application");
100       jc.getJ2eeApplications().add(ja);
101       J2eeJdbcResource jdbcr = new J2eeJdbcResource("
              CalendarJdbcResource");
102       ja.getJ2eeJdbcResources().add(jdbcr);
103       jdbcr.setDatabase(calendardb);
104
105       ja = new J2eeApplication("Wiki Application");
106       jc.getJ2eeApplications().add(ja);
107       jdbcr = new J2eeJdbcResource("WikiJdbcResource");
108       ja.getJ2eeJdbcResources().add(jdbcr);
109       jdbcr.setDatabase(wikidb);
110       jdbcr = new J2eeJdbcResource("Wiki-Foreign-Cal-JdbcResource");
```

```
111         ja.getJ2eeJdbcResources().add(jdbcr);
112         jdbcr.setDatabase(calendardb);
113 
114         ja = new J2eeApplication("Expense Management Application");
115         jc.getJ2eeApplications().add(ja);
116         jdbcr = new J2eeJdbcResource("ExpenseJdbcResource");
117         ja.getJ2eeJdbcResources().add(jdbcr);
118         jdbcr.setDatabase(expensedb);
119       }
120 
121 
122       // Setup Server for Webmail
123       {
124         WebServer ws = new WebServer("WebmailServer");
125         ws.setCpuSpeed(1000);
126         ws.setMemorySize(1000);
127         d.save(ws);
128         J2eeContainer jc = new J2eeContainer("WebmailContainer");
129         ws.getJ2eeContainers().add(jc);
130         JVM jvm = new JVM("jvm1.5", "1.4");
131         jc.setJvm(jvm);
132         J2eeModule jma = new J2eeModule("IMAP Module");
133         jc.getJ2eeModules().add(jma);
134         jma = new J2eeModule("POP3 Module");
135         jc.getJ2eeModules().add(jma);
136         jma = new J2eeModule("Ajax Module");
137         jc.getJ2eeModules().add(jma);
138 
139 
140         J2eeApplication ja = new J2eeApplication("Webmail Application");
141         jc.getJ2eeApplications().add(ja);
142       }
143 
144 
145       // Setup Server for Ticket System and Document Management
146       {
147         WebServer ws = new WebServer("Doc+TicketServer");
148         ws.setCpuSpeed(1000);
149         ws.setMemorySize(1000);
150         d.save(ws);
151         J2eeContainer jc1 = new J2eeContainer("DocumentManagementContainer
              ");
152         ws.getJ2eeContainers().add(jc1);
153         J2eeContainer jc2 = new J2eeContainer("TicketSystemContainer");
154         ws.getJ2eeContainers().add(jc2);
155 
156         JVM jvm = new JVM("jvm1.5", "1.4"); jc1.setJvm(jvm);
157         jvm = new JVM("jvm1.5", "1.5"); jc2.setJvm(jvm);
158         J2eeModule jma = new J2eeModule("SMB Module");
159         jc1.getJ2eeModules().add(jma);
160         jma = new J2eeModule("WebFTP Module");
161         jc1.getJ2eeModules().add(jma);
162 
```

```
163       jma = new J2eeModule("POP3 Module");
164       jc1.getJ2eeModules().add(jma);
165       jma = new J2eeModule("Ajax Module");
166       jc1.getJ2eeModules().add(jma);
167
168       J2eeApplication ja1 = new J2eeApplication("Document Management
              Application");
169       jc1.getJ2eeApplications().add(ja1);
170       J2eeApplication ja2 = new J2eeApplication("Ticket System
              Application");
171       jc1.getJ2eeApplications().add(ja2);
172
173       Database db = new MySqlDb("TicketSystemDatabase");
174       J2eeJdbcResource jdbcr = new J2eeJdbcResource("
              TicketSystemJdbcResource");
175       ja2.getJ2eeJdbcResources().add(jdbcr);
176       jdbcr.setDatabase(db);
177       dbserver.getDatabases().add(db);
178     }
179
180
181     // People and skills
182     {
183       Human uschi = new Human("Uschi", 10); d.save(uschi);
184       Skill s11 = new DatabaseDesign(10);
185       Skill s12 = new Hardware(9);
186       Skill s13 = new WebserverConfiguration(3);
187       uschi.getSkills().add(s11);
188       uschi.getSkills().add(s12);
189       uschi.getSkills().add(s13);
190
191       Human charlene = new Human("Charlene", 10); d.save(charlene);
192       Skill s21 = new DatabaseDesign(60);
193       Skill s22 = new Hardware(90);
194       charlene.getSkills().add(s21);
195       charlene.getSkills().add(s22);
196
197       Human markkevin = new Human("Mark-Kevin", 10); d.save(markkevin);
198       Skill s31 = new DatabaseDesign(100);
199       Skill s32 = new Hardware(15);
200       Skill s33 = new WebserverConfiguration(5);
201       markkevin.getSkills().add(s31);
202       markkevin.getSkills().add(s32);
203       markkevin.getSkills().add(s33);
204
205
206       Human mandyloreen = new Human("Mandy-Loreen", 10); d.save(
              mandyloreen);
207       Skill s41 = new DatabaseDesign(100);
208       Skill s42 = new Hardware(95);
209       Skill s43 = new WebserverConfiguration(90);
210       mandyloreen.getSkills().add(s41);
211       mandyloreen.getSkills().add(s42);
```

```
212         mandyloreen.getSkills().add(s43);
213
214       }
215     }
216 }
```

C.3. Example task, method and operator definitions

Listing C.4: JAVA code the SpeedUpWebApplication task definition

```
1  package da.scenarios.j2ee.domain;
2
3  import da.knowledgebase.model.Database;
4  import da.knowledgebase.model.DbServer;
5  import da.planner.htn.Binding;
6  import da.planner.htn.KbElementVariable;
7  import da.planner.htn.Method;
8  import da.planner.htn.Operator;
9  import da.planner.htn.OrderingConstraint;
10 import da.planner.htn.PropertyVariable;
11 import da.planner.htn.Task;
12 import da.planner.htn.TaskDefinition;
13 import da.planner.htn.TaskNetwork;
14 import da.planner.htn.query.And;
15 import da.planner.htn.query.KbPrecondition;
16 import da.planner.htn.query.Not;
17 import da.planner.htn.query.ObjectInSetConstraint;
18 import da.planner.htn.query.ObjectPropertyIdentityConstraint;
19
20 public class SpeedUpWebApplication extends TaskDefinition {
21   public SpeedUpWebApplication(){
22
23     this.addPerformer(
24         new Method("EnableLoadbalancing", this)
25         {{
26           KbPrecondition p = new KbPrecondition();
27           this.setBindingGenerator(p);
28           p.addKbElementConstraint("application");
29         }
30
31
32         protected TaskNetwork getSubNetwork(Binding b) {
33           TaskNetwork tn = new TaskNetwork();
34
35           Task lbNode = new Task(new InstallLoadBalancer(), new Binding
                 ()
36           ._put("lb", b.get("lb"))
37           );
38           Task serverNode = new Task( new InstallJ2eeServer(), new
                 Binding()
39           ._put("server", b.get("newws"))
```

```
40          ._put("container", b.get("jc"))
41          );
42
43          Task appNode = new Task(new InstallJ2eeApplication(), new
                Binding()
44          ._put("cont", b.get("jc"))
45          ._put("app", b.get("application"))
46          );
47
48          Task addthemNode = new Task(new AddContainerToLoadbalancer(),
                new Binding()
49          ._put("container", b.get("jc"))
50          ._put("lb", b.get("lb"))
51          );
52
53          tn.addNode(lbNode);
54          tn.addNode(serverNode);
55          tn.addNode(appNode);
56          tn.addNode(addthemNode);
57          tn.addOrderingConstraint(new OrderingConstraint(serverNode,
                appNode));
58          tn.addOrderingConstraint(new OrderingConstraint(appNode,
                addthemNode));
59          tn.addOrderingConstraint(new OrderingConstraint(lbNode,
                addthemNode));
60          return tn;
61        }
62        });
63
64
65
66      this.addPerformer(
67          new Method("MigrateDatabase", this)
68          {{
69            KbPrecondition p = new KbPrecondition();
70            this.setBindingGenerator(p);
71            p.addKbElementConstraint("application");
72            p.addKbElementConstraint("olddb");
73            p.addKbElementConstraint("dbresource");
74            p.setAdditionalQueryConstraints(new And(
75                new ObjectInSetConstraint("dbresource", "application", "
                    j2eeJdbcResources"),
76                new ObjectPropertyIdentityConstraint("dbresource", "
                    database", "olddb")
77            ));
78          }
79
80          protected TaskNetwork getSubNetwork(Binding b) {
81            TaskNetwork tn = new TaskNetwork();
82            KbElementVariable newdb = new KbElementVariable(Database.class
                , null, null);
83            KbElementVariable newdbserver = new KbElementVariable(DbServer
                .class, null, null);
```

```
84
85              Task backupNode = new Task(new BackupDatabase(), new Binding()
86              ._put("db", b.get("olddb"))
87              );
88              Task installdbNode = new Task( new InstallDatabase(), new
                    Binding()
89              ._put("dbServer", newdbserver)
90              ._put("db", newdb)
91              );
92              Task copyNode = new Task(new CopyDatabaseContent(), new
                    Binding()
93              ._put("fromDb", b.get("olddb"))
94              ._put("toDb", newdb)
95              );
96
97              //TODO: connect newdb and application via resource
98
99
100             tn.addNode(installdbNode);
101             tn.addNode(backupNode);
102             tn.addNode(copyNode);
103             tn.addOrderingConstraint(new OrderingConstraint(backupNode,
                    installdbNode));
104             tn.addOrderingConstraint(new OrderingConstraint(installdbNode,
                     copyNode));
105             return tn;
106         }
107
108         }
109
110
111     );
112
113
114     this.addPerformer(
115         new Operator("AskWebApplicationExpert", this, 10)
116         {{
117             KbPrecondition p = new KbPrecondition();
118             this.setBindingGenerator(p);
119             p.addKbElementConstraint("application");
120             p.addKbElementConstraint("expert");
121             p.addKbElementConstraint("skill");
122             p.setAdditionalQueryConstraints(new Not(new
                    ObjectInSetConstraint("skill", "expert", "skills")));
123         }
124         }
125     );
126
127
128
129     this.addPerformer(
130         new Method("UpgradeWebServerHardware", this)
131         {{
```

```
132           KbPrecondition p = new KbPrecondition();
133           this.setBindingGenerator(p);
134           p.addKbElementConstraint("application");
135           p.addKbElementConstraint("ws");
136           p.addKbElementConstraint("jc");
137           p.setAdditionalQueryConstraints(new And(
138               new ObjectInSetConstraint("jc", "ws", "j2eeContainers"),
139               new ObjectInSetConstraint("application", "jc", "
                      j2eeApplications")
140           ));
141
142       }
143       protected TaskNetwork getSubNetwork(Binding b) {
144           TaskNetwork tn = new TaskNetwork();
145           PropertyVariable mem = new PropertyVariable(Integer.class,
                  1000);
146           PropertyVariable cpu = new PropertyVariable(Integer.class, 1);
147           Task upgrade = new Task(new UpgradeHardware(), new Binding()
148           ._put("machine", b.get("ws"))
149           ._put("mem", mem)
150           ._put("cpu", cpu)
151           );
152           tn.addNode(upgrade);
153
154           return tn;
155       }
156       }
157   );
158
159
160   this.addPerformer(
161       new Method("UpgradeDatabaseServerHardware", this)
162       {{
163           KbPrecondition p = new KbPrecondition();
164           this.setBindingGenerator(p);
165           p.addKbElementConstraint("application");
166           p.addKbElementConstraint("dbserver");
167           p.addKbElementConstraint("olddb");
168           p.addKbElementConstraint("dbresource");
169           p.setAdditionalQueryConstraints(new And(new And(
170               new ObjectInSetConstraint("dbresource", "application", "
                      j2eeJdbcResources"),
171               new ObjectPropertyIdentityConstraint("dbresource", "
                      database", "olddb")),
172               new ObjectInSetConstraint("olddb", "dbserver", "databases"
                      )
173           ));
174
175       }
176       protected TaskNetwork getSubNetwork(Binding b) {
177           TaskNetwork tn = new TaskNetwork();
178           PropertyVariable mem = new PropertyVariable(Integer.class,
                  1000);
```

```
179             PropertyVariable cpu = new PropertyVariable(Integer.class, 1);
180
181             Task upgrade = new Task(new UpgradeHardware(), new Binding()
182             ._put("machine", b.get("dbserver"))
183             ._put("mem", mem)
184             ._put("cpu", cpu)
185             );
186             tn.addNode(upgrade);
187
188             return tn;
189         }
190         }
191     );
192
193
194     this.addPerformer(
195         new Method("UpgradeDatabaseServerAndWebServerHardware", this)
196         {{
197             KbPrecondition p = new KbPrecondition();
198             this.setBindingGenerator(p);
199             p.addKbElementConstraint("application");
200             p.addKbElementConstraint("ws");
201             p.addKbElementConstraint("jc");
202
203             p.addKbElementConstraint("dbserver");
204             p.addKbElementConstraint("olddb");
205             p.addKbElementConstraint("dbresource");
206             p.setAdditionalQueryConstraints(new And(
207                 new And(new And(
208                     new ObjectInSetConstraint("dbresource", "application",
                            "j2eeJdbcResources"),
209                     new ObjectPropertyIdentityConstraint("dbresource", "
                        database", "olddb")),
210                     new ObjectInSetConstraint("olddb", "dbserver", "
                        databases")),
211
212                     new And(
213                         new ObjectInSetConstraint("jc", "ws", "
                            j2eeContainers"),
214                         new ObjectInSetConstraint("application", "jc", "
                            j2eeApplications"))
215             ));
216         }
217         protected TaskNetwork getSubNetwork(Binding b) {
218             TaskNetwork tn = new TaskNetwork();
219             PropertyVariable mem = new PropertyVariable(Integer.class,
                    1000);
220             PropertyVariable cpu = new PropertyVariable(Integer.class, 1);
221
222             Task upgradeDb = new Task(new UpgradeHardware(), new Binding()
223             ._put("machine", b.get("dbserver"))
224             ._put("mem", mem)
225             ._put("cpu", cpu)
```

```
226             );
227             tn.addNode(upgradeDb);
228
229             Task upgradeWs = new Task(new UpgradeHardware(), new Binding()
230             ._put("machine", b.get("ws"))
231             ._put("mem", mem)
232             ._put("cpu", cpu)
233             );
234             tn.addNode(upgradeWs);
235             //tn.addOrderingConstraint(new OrderingConstraint(upgradeDb,
                    upgradeWs));
236             return tn;
237           }
238           }
239       );
240   }
241 }
```

C.4. ChangeRefinery outputs for examples in Sections 4.4 and 5.3

Listing C.5: ChangeRefinery output for Section 4.4.1

```
1 Remaining task network:
2   Preliminary plan:
3   Remaining tasks: SpeedUpWebApplication,
4 Available REFINEMENT OPERATION:
5 [0] EnableLoadbalancing
6 [1] MigrateDatabase
7 [2] UpgradeWebServerHardware
8 [3] AskWebApplicationExpert
9 [4] UpgradeDatabaseServerHardware
10 Choose a REFINEMENT OPERATION: 2
11 Backtrack over remaining REFINEMENT OPERATION? (y/n)? n
12 Calculating bindings for task SpeedUpWebApplication and refinement
       UpgradeWebServerHardware
13 Available BINDING:
14 [0] J2eeContainer:jc=18=WebshopContainer5, J2eeApplication:application
       =21=Webshop Application5, WebServer:ws=8=WebshopServer5
15 [1] J2eeContainer:jc=24=WebshopContainer3, J2eeApplication:application
       =27=Webshop Application3, WebServer:ws=6=WebshopServer3
16 [2] J2eeContainer:jc=30=WebshopContainer4, J2eeApplication:application
       =33=Webshop Application4, WebServer:ws=7=WebshopServer4
17 [3] J2eeContainer:jc=36=WebshopContainer2, J2eeApplication:application
       =39=Webshop Application2, WebServer:ws=5=WebshopServer2
18 [4] J2eeContainer:jc=42=WebshopContainer1, J2eeApplication:application
       =45=Webshop Application1, WebServer:ws=4=WebshopServer1
19 [5] J2eeContainer:jc=52=WebshopContainer, J2eeApplication:application
       =54=Wiki Application, WebServer:ws=10=MiscServer
20 [6] J2eeContainer:jc=52=WebshopContainer, J2eeApplication:application
       =57=Expense Management Application, WebServer:ws=10=MiscServer
```

```
21 [7] J2eeContainer:jc=52=WebshopContainer, J2eeApplication:application
       =59=Calendar Application, WebServer:ws=10=MiscServer
22 [8] J2eeContainer:jc=62=WebmailContainer, J2eeApplication:application
       =66=Webmail Application, WebServer:ws=11=WebmailServer
23 [9] J2eeContainer:jc=68=DocumentManagementContainer, J2eeApplication:
       application=73=Ticket System Application, WebServer:ws=12=Doc+
       TicketServer
24 [10] J2eeContainer:jc=68=DocumentManagementContainer, J2eeApplication:
       application=75=Document Management Application, WebServer:ws=12=Doc+
       TicketServer
25 Choose a BINDING: ? 0
26 Backtrack over remaining BINDING? (y/n)? n
27 Applying method UpgradeWebServerHardware to task SpeedUpWebApplication
28 Enter temporal deadline for sub task UpgradeHardware (-1 for no deadline
       ): ? 40
29 Remaining task network:
30   Preliminary plan:
31   Remaining tasks: UpgradeHardware,
32 Available REFINEMENT OPERATION:
33 [0] UpgradeCpu
34 [1] UpgradeMem
35 [2] UpgradeMemAndCpu
36 Choose a REFINEMENT OPERATION: ? 2
37 Backtrack over remaining REFINEMENT OPERATION? (y/n)? n
38 Calculating bindings for task UpgradeHardware and refinement
       UpgradeMemAndCpu
39 Applying operator UpgradeMemAndCpu to task UpgradeHardware
40 Found new plan (1): UpgradeMemAndCpu(cpu <-- [Integer=1],mem <-- [
       Integer=1000],machine <-- [WebServer=8=WebshopServer5])
41 Backtracking
42 Backtracking
43
44
45 Found plans (total 1)
46 (0) UpgradeMemAndCpu(cpu <-- [Integer=1],mem <-- [Integer=1000],machine
       <-- [WebServer=8=WebshopServer5])
47 Action UpgradeMemAndCpu: starts [0.0 , 15.0] end [25.0 , 40.0] duration
       [25.0 , 25.0]
48 Task SpeedUpWebApplication: starts [0.0 , 15.0] end [25.0 , 50.0]
       duration [25.0 , 50.0]
49 Task UpgradeHardware: starts [0.0 , 15.0] end [25.0 , 40.0] duration
       [25.0 , 40.0]
```

Listing C.6: ChangeRefinery output for Section 5.3.1

```
1 Remaining task network:
2   Preliminary plan:
3   Remaining tasks: SpeedUpWebApplication,
4 Available REFINEMENT OPERATION:
5 [0] UpgradeDatabaseServerHardware
6 [1] UpgradeDatabaseServerAndWebServerHardware
7 [2] EnableLoadbalancing
8 [3] MigrateDatabase
```

```
[4] UpgradeWebServerHardware
Choose a REFINEMENT OPERATION: 1
Backtrack over remaining REFINEMENT OPERATION? (y/n)? n
Calculating bindings for task SpeedUpWebApplication and refinement
    UpgradeDatabaseServerAndWebServerHardware
Applying method UpgradeDatabaseServerAndWebServerHardware to task
    SpeedUpWebApplication
Enter temporal deadline for sub task UpgradeHardware (-1 for no deadline
    ): ? 30
Enter temporal deadline for sub task UpgradeHardware (-1 for no deadline
    ): ? 35
Remaining task network:
  Preliminary plan:
  Remaining tasks: UpgradeHardware,UpgradeHardware,
Available FIRST TASK:
[0] Task: UpgradeHardware
[1] Task: UpgradeHardware
Choose a FIRST TASK: ? 0
Available REFINEMENT OPERATION:
[0] UpgradeMemAndCpu
[1] UpgradeCpu
[2] UpgradeMem
Choose a REFINEMENT OPERATION: ? 0
Backtrack over remaining REFINEMENT OPERATION? (y/n)? n
Calculating bindings for task UpgradeHardware and refinement
    UpgradeMemAndCpu
Applying operator UpgradeMemAndCpu to task UpgradeHardware
Remaining task network:
  Preliminary plan: UpgradeMemAndCpu(cpu <-- [Integer=1],mem <-- [
      Integer=1000],machine <-- [DbServer=1=MiscDbServer])
  Remaining tasks: UpgradeHardware,
Available REFINEMENT OPERATION:
[0] UpgradeMem
[1] UpgradeMemAndCpu
[2] UpgradeCpu
Choose a REFINEMENT OPERATION: ? 0
Backtrack over remaining REFINEMENT OPERATION? (y/n)? n
Calculating bindings for task UpgradeHardware and refinement UpgradeMem
Applying operator UpgradeMem to task UpgradeHardware
Found new plan (1): UpgradeMemAndCpu(cpu <-- [Integer=1],mem <-- [
    Integer=1000],machine <-- [DbServer=1=MiscDbServer]),UpgradeMem(mem
    <-- [Integer=1000],cpu <-- [Integer=1],machine <-- [WebServer=2=
    MiscServer])
Backtracking
Backtracking
Backtracking


Found plans (total 1)
(0) UpgradeMemAndCpu(cpu <-- [Integer=1],mem <-- [Integer=1000],machine
    <-- [DbServer=1=MiscDbServer]),UpgradeMem(mem <-- [Integer=1000],cpu
    <-- [Integer=1],machine <-- [WebServer=2=MiscServer])
```

```
50 Task UpgradeHardware: starts [0.0 , 5.0] end [25.0 , 30.0] duration
       [25.0 , 30.0]
51 Task UpgradeHardware: starts [0.0 , 20.0] end [15.0 , 35.0] duration
       [15.0 , 35.0]
52 Task SpeedUpWebApplication: starts [0.0 , 5.0] end [25.0 , 50.0]
       duration [25.0 , 50.0]
53 Action UpgradeMemAndCpu: starts [0.0 , 5.0] end [25.0 , 30.0] duration
       [25.0 , 25.0]
54 Action UpgradeMem: starts [0.0 , 20.0] end [15.0 , 35.0] duration [15.0
       , 15.0]
```

Listing C.7: Example of an inconsistent STP

```
1 Task UpgradeHardware: starts [0.0 , 5.0] end [25.0 , 30.0] duration
      [25.0 , 30.0]
2 Task UpgradeHardware: starts [25.0 , 35.0] end [25.0 , 35.0] duration
      [0.0 , 10.0]
3 Task SpeedUpWebApplication: starts [0.0 , 5.0] end [25.0 , 50.0]
      duration [25.0 , 50.0]
4 Action UpgradeMemAndCpu: starts [0.0 , 5.0] end [25.0 , 30.0] duration
      [25.0 , 25.0]
5 Action UpgradeMem: starts [25.0 , 1000.0] end [0.0 , 35.0] duration
      [15.0 , 10.0]
```

Listing C.8: Example of additional ordering constraints due to mutual dependencies of effects and preconditions

```
1 Remaining task network:
2   Preliminary plan:
3   Remaining tasks: UpgradeHardware,SetupServer,
4 Available FIRST TASK:
5 [0] Task: UpgradeHardware
6 [1] Task: SetupServer
7 Choose a FIRST TASK: 0
8 Available REFINEMENT OPERATION:
9 [0] UpgradeCpu
10 [1] UpgradeMem
11 [2] UpgradeMemAndCpu
12 Choose a REFINEMENT OPERATION: ? 1
13 Backtrack over remaining REFINEMENT OPERATION? (y/n)? n
14 Calculating bindings for task UpgradeHardware and refinement UpgradeMem
15 Applying operator UpgradeMem to task UpgradeHardware
16 Remaining task network:
17   Preliminary plan: UpgradeMem(cpu <-- [Integer=1000],mem <-- [Integer
       =1000],machine <-- [DbServer=1=MiscDbServer]; QueryDependencies
       ={1})
18   Remaining tasks: SetupServer,
19 Available REFINEMENT OPERATION:
20 [0] UseLowEndServer
21 [1] UseExistingServer
22 Choose a REFINEMENT OPERATION: ? 1
23 Backtrack over remaining REFINEMENT OPERATION? (y/n)? n
24 Calculating bindings for task SetupServer and refinement
       UseExistingServer
```

```
25 Applying operator UseExistingServer to task SetupServer
26 Found new plan (1): UpgradeMem(cpu <-- [Integer=1000],mem <-- [Integer
       =1000],machine <-- [DbServer=1=MiscDbServer]; QueryDependencies={1}),
       UseExistingServer(mem <-- [Integer=2000],server <-- [DbServer=1=
       MiscDbServer]; QueryDependencies={1})
27 Backtracking
28 Backtracking
29
30
31 Found plans (total 1)
32 (0) UpgradeMem(cpu <-- [Integer=1000],mem <-- [Integer=1000],machine <--
        [DbServer=1=MiscDbServer]; QueryDependencies={1}),UseExistingServer(
       mem <-- [Integer=2000],server <-- [DbServer=1=MiscDbServer];
       QueryDependencies={1})
33 Action UseExistingServer: starts [15.0, 45.0] ends [20.0, 50.0] duration
        [5.0, 5.0]
34 Task SetupServer: starts [0.0, 45.0] ends [20.0, 50.0] duration [5.0,
       50.0]
35 Task UpgradeHardware: starts [0.0, 30.0] ends [15.0, 50.0] duration
       [15.0, 50.0]
36 Action UpgradeMem: starts [0.0, 30.0] ends [15.0, 45.0] duration [15.0,
       15.0]
```

Listing C.9: Example of additional ordering constraints due to failed preconditions

```
1 (0) UpgradeMemAndCpu(cpu <-- [Integer=1],mem <-- [Integer=1000],machine
      <-- [WebServer=3=ECommerceServer]),InstallJ2eeModule-Script(module
      <-- [Log4jJ2eeModule=16=nil],container <-- [J2eeContainer=10=
      ECommerceContainer]),InstallECommerceApplicationVersion5-Script(app
      <-- [ECommerceJ2eeApplication=17=nil],log4jmodule <-- [
      Log4jJ2eeModule=16=nil],cont <-- [J2eeContainer=10=ECommerceContainer
      ])
2 (1) UpgradeCpu(cpu <-- [Integer=1],mem <-- [Integer=1000],machine <-- [
      WebServer=3=ECommerceServer]),InstallJ2eeModule-Script(module <-- [
      Log4jJ2eeModule=16=nil],container <-- [J2eeContainer=10=
      ECommerceContainer]),InstallECommerceApplicationVersion5-Script(app
      <-- [ECommerceJ2eeApplication=17=nil],log4jmodule <-- [
      Log4jJ2eeModule=16=nil],cont <-- [J2eeContainer=10=ECommerceContainer
      ])
3 (2) UpgradeMem(cpu <-- [Integer=1],mem <-- [Integer=1000],machine <-- [
      WebServer=3=ECommerceServer]),InstallJ2eeModule-Script(module <-- [
      Log4jJ2eeModule=16=nil],container <-- [J2eeContainer=10=
      ECommerceContainer]),InstallECommerceApplicationVersion5-Script(app
      <-- [ECommerceJ2eeApplication=17=nil],log4jmodule <-- [
      Log4jJ2eeModule=16=nil],cont <-- [J2eeContainer=10=ECommerceContainer
      ])
4 (3) InstallJ2eeModule-Script(module <-- [Log4jJ2eeModule=16=nil],
      container <-- [J2eeContainer=10=ECommerceContainer]),
      InstallECommerceApplicationVersion5-Script(app <-- [
      ECommerceJ2eeApplication=17=nil],log4jmodule <-- [Log4jJ2eeModule=16=
      nil],cont <-- [J2eeContainer=10=ECommerceContainer]),UpgradeMemAndCpu
      (cpu <-- [Integer=1],mem <-- [Integer=1000],machine <-- [WebServer=3=
      ECommerceServer])
```

```
5 (4) InstallJ2eeModule-Script(module <-- [Log4jJ2eeModule=16=nil],
      container <-- [J2eeContainer=10=ECommerceContainer]),
      InstallECommerceApplicationVersion5-Script(app <-- [
      ECommerceJ2eeApplication=17=nil],log4jmodule <-- [Log4jJ2eeModule=16=
      nil],cont <-- [J2eeContainer=10=ECommerceContainer]),UpgradeCpu(cpu
      <-- [Integer=1],mem <-- [Integer=1000],machine <-- [WebServer=3=
      ECommerceServer])
6 (5) InstallJ2eeModule-Script(module <-- [Log4jJ2eeModule=16=nil],
      container <-- [J2eeContainer=10=ECommerceContainer]),
      InstallECommerceApplicationVersion5-Script(app <-- [
      ECommerceJ2eeApplication=17=nil],log4jmodule <-- [Log4jJ2eeModule=16=
      nil],cont <-- [J2eeContainer=10=ECommerceContainer]),UpgradeMem(cpu
      <-- [Integer=1],mem <-- [Integer=1000],machine <-- [WebServer=3=
      ECommerceServer])
7 (6) InstallJ2eeModule-Script(module <-- [Log4jJ2eeModule=16=nil],
      container <-- [J2eeContainer=10=ECommerceContainer]),UpgradeMemAndCpu
      (mem <-- [Integer=1000],cpu <-- [Integer=1],machine <-- [WebServer=3=
      ECommerceServer]),InstallECommerceApplicationVersion5-Script(app <--
      [ECommerceJ2eeApplication=17=nil],log4jmodule <-- [Log4jJ2eeModule
      =16=nil],cont <-- [J2eeContainer=10=ECommerceContainer])
8 (7) InstallJ2eeModule-Script(module <-- [Log4jJ2eeModule=16=nil],
      container <-- [J2eeContainer=10=ECommerceContainer]),UpgradeCpu(mem
      <-- [Integer=1000],cpu <-- [Integer=1],machine <-- [WebServer=3=
      ECommerceServer]),InstallECommerceApplicationVersion5-Script(app <--
      [ECommerceJ2eeApplication=17=nil],log4jmodule <-- [Log4jJ2eeModule
      =16=nil],cont <-- [J2eeContainer=10=ECommerceContainer])
9 (8) InstallJ2eeModule-Script(module <-- [Log4jJ2eeModule=16=nil],
      container <-- [J2eeContainer=10=ECommerceContainer]),UpgradeMem(mem
      <-- [Integer=1000],cpu <-- [Integer=1],machine <-- [WebServer=3=
      ECommerceServer]),InstallECommerceApplicationVersion5-Script(app <--
      [ECommerceJ2eeApplication=17=nil],log4jmodule <-- [Log4jJ2eeModule
      =16=nil],cont <-- [J2eeContainer=10=ECommerceContainer])
```

Listing C.10: "Enable load balancing" change plan for complex example (for Section 5.3.3)

```
1 Remaining task network:
2   Preliminary plan:
3   Remaining tasks: SpeedUpWebApplication,
4 Available REFINEMENT OPERATION:
5 [0] UpgradeDatabaseServerHardware
6 [1] UpgradeWebServerHardware
7 [2] MigrateDatabase
8 [3] EnableLoadbalancing
9 [4] UpgradeDatabaseServerAndWebServerHardware
10 Choose a REFINEMENT OPERATION: 3
11 Backtrack over remaining REFINEMENT OPERATION? (y/n)? n
12 Calculating bindings for task SpeedUpWebApplication and refinement
       EnableLoadbalancing
13 Available BINDING:
14 [0] J2eeApplication:application=8=Wiki Application
15 [1] ECommerceJ2eeApplication:application=11=ECommerce Application
16 Choose a BINDING: ? 0
17 Backtrack over remaining BINDING? (y/n)? n
```

```
18 Applying method EnableLoadbalancing to task SpeedUpWebApplication
19 Enter temporal deadline for sub task InstallLoadBalancer (-1 for no
     deadline): ? -1
20 Enter temporal deadline for sub task InstallJ2eeApplication (-1 for no
     deadline): ? -1
21 Enter temporal deadline for sub task AddContainerToLoadbalancer (-1 for
     no deadline): ? -1
22 Enter temporal deadline for sub task InstallJ2eeServer (-1 for no
     deadline): ? -1
23 Remaining task network:
24   Preliminary plan:
25   Remaining tasks: InstallLoadBalancer,InstallJ2eeApplication,
       InstallJ2eeServer,AddContainerToLoadbalancer,
26 Available FIRST TASK:
27 [0] Task: InstallLoadBalancer
28 [1] Task: InstallJ2eeServer
29 Choose a FIRST TASK: ? 0
30 Available REFINEMENT OPERATION:
31 [0] InstallLoadBalancer-Script
32 [1] LoadbalancerOnServer
33 Choose a REFINEMENT OPERATION: ? 0
34 Backtrack over remaining REFINEMENT OPERATION? (y/n)? n
35 Calculating bindings for task InstallLoadBalancer and refinement
     InstallLoadBalancer-Script
36 Applying operator InstallLoadBalancer-Script to task InstallLoadBalancer
37 Remaining task network:
38   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
87 Found plans (total 1)
88 (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])
89 Task InstallLoadBalancer: starts [0.0 , 35.0] end [10.0 , 45.0] duration
       [10.0 , 45.0]
90 Task InstallJ2eeServer: starts [0.0 , 15.0] end [20.0 , 35.0] duration
       [20.0 , 35.0]
91 Action InstallVirtualServerByScript: starts [0.0 , 15.0] end [10.0 ,
       25.0] duration [10.0 , 10.0]
92 Action InstallJ2eeContainerSoftware-Script: starts [10.0 , 25.0] end
       [20.0 , 35.0] duration [10.0 , 10.0]
93 Task InstallJ2eeApplication: starts [20.0 , 35.0] end [30.0 , 45.0]
       duration [10.0 , 25.0]
94 Task AddContainerToLoadbalancer: starts [30.0 , 45.0] end [35.0 , 50.0]
       duration [5.0 , 20.0]
95 Task SpeedUpWebApplication: starts [0.0 , 15.0] end [35.0 , 50.0]
       duration [35.0 , 50.0]
96 Task InstallJ2eeContainerSoftware: starts [10.0 , 25.0] end [20.0 ,
       35.0] duration [10.0 , 25.0]
97 Action InstallJ2eeApplication-Script: starts [20.0 , 35.0] end [30.0 ,
       45.0] duration [10.0 , 10.0]
98 Task SetupServer: starts [0.0 , 15.0] end [10.0 , 25.0] duration [10.0 ,
       25.0]
99 Action AddContainerToLoadBalancer-Script: starts [30.0 , 45.0] end [35.0
       , 50.0] duration [5.0 , 5.0]
100 Action InstallLoadBalancer-Script: starts [0.0 , 35.0] end [10.0 , 45.0]
       duration [10.0 , 10.0]
101
102
103
104
105 Total time consumed for HTN planning: 57796ms
106 Total time consumed for task network copying: 0ms (0%)
107
108
109
110 Total time consumed for knowledge base rollbacks: 735ms
111 Total time consumed for knowledge base assertions: 1172ms
112 Total time consumed for knowledge base queries: 281ms
113 Total time consumed for knowledge base operations: 2188ms
```

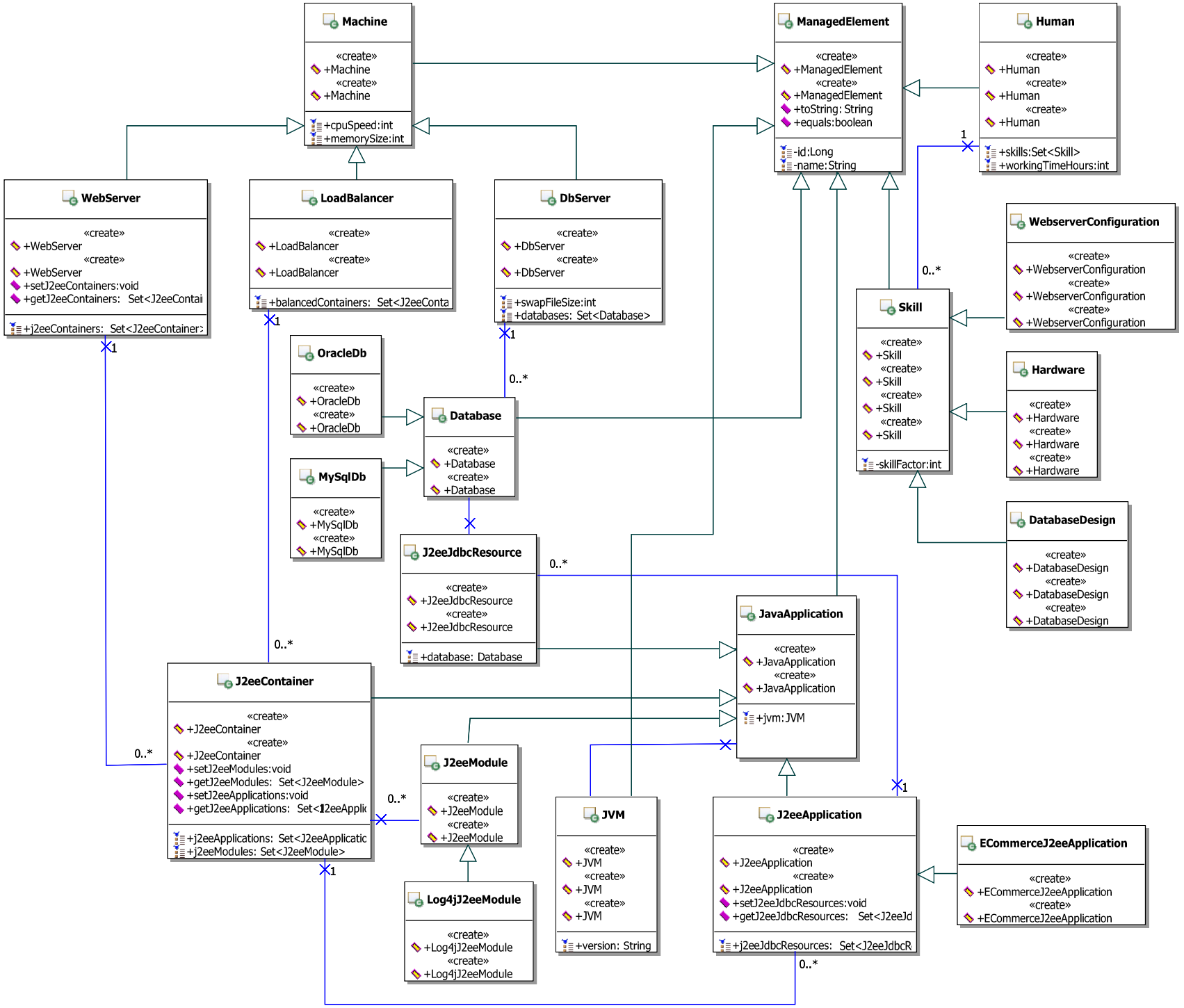
Listing C.11: “Migrate database” change plan for complex example (for Section 5.3.3)

```
1 Remaining task network:
2   Preliminary plan:
3   Remaining tasks: SpeedUpWebApplication,
```

```
4  Available REFINEMENT OPERATION:
5  [0] UpgradeDatabaseServerHardware
6  [1] EnableLoadbalancing
7  [2] UpgradeWebServerHardware
8  [3] UpgradeDatabaseServerAndWebServerHardware
9  [4] MigrateDatabase
10 Choose a REFINEMENT OPERATION: 4
11 Backtrack over remaining REFINEMENT OPERATION? (y/n)? n
12 Calculating bindings for task SpeedUpWebApplication and refinement
       MigrateDatabase
13 Available BINDING:
14 [0] J2eeApplication:application=8=Wiki Application, J2eeJdbcResource:
       dbresource=9=WikiJdbcResource, Database:olddb=5=WikiDb
15 [1] ECommerceJ2eeApplication:application=11=ECommerce Application,
       J2eeJdbcResource:dbresource=12=ECommerceJdbcResource, Database:olddb
       =6=ECommerceDb
16 Choose a BINDING: ? 0
17 Backtrack over remaining BINDING? (y/n)? n
18 Applying method MigrateDatabase to task SpeedUpWebApplication
19 Enter temporal deadline for sub task CopyDatabaseContent (-1 for no
       deadline): ? -1
20 Enter temporal deadline for sub task BackupDatabase (-1 for no deadline)
       : ? -1
21 Enter temporal deadline for sub task InstallDatabase (-1 for no deadline
       ): ? -1
22 Remaining task network:
23   Preliminary plan:
24   Remaining tasks: CopyDatabaseContent,BackupDatabase,InstallDatabase,
25 Calculating bindings for task BackupDatabase and refinement
       RunBackupScript
26 Applying operator RunBackupScript to task BackupDatabase
27 Remaining task network:
28   Preliminary plan: RunBackupScript(db <-- [Database=5=WikiDb])
29   Remaining tasks: CopyDatabaseContent,InstallDatabase,
30 Available REFINEMENT OPERATION:
31 [0] InstallNewDatabaseOnNewDbServer-Script
32 [1] InstallNewDatabaseOnExistingDbServer
33 [2] InstallNewDatabaseOnSpecialDbServer
34 Choose a REFINEMENT OPERATION: ? 2
35 Backtrack over remaining REFINEMENT OPERATION? (y/n)? n
36 Calculating bindings for task InstallDatabase and refinement
       InstallNewDatabaseOnSpecialDbServer
37 Applying method InstallNewDatabaseOnSpecialDbServer to task
       InstallDatabase
38 Enter temporal deadline for sub task SetupServer (-1 for no deadline): ?
        -1
39 Enter temporal deadline for sub task InstallDatabaseSoftware (-1 for no
       deadline): ? -1
40 Remaining task network:
41   Preliminary plan: RunBackupScript(db <-- [Database=5=WikiDb])
42   Remaining tasks: SetupServer,CopyDatabaseContent,
         InstallDatabaseSoftware,
43 Available FIRST TASK:
```

```
44 [0] Task: SetupServer
45 [1] Task: CopyDatabaseContent
46 Choose a FIRST TASK: ? 0
47 Available REFINEMENT OPERATION:
48 [0] UseExistingServer
49 [1] InstallVirtualServerByScript
50 Choose a REFINEMENT OPERATION: ? 1
51 Backtrack over remaining REFINEMENT OPERATION? (y/n)? n
52 Calculating bindings for task SetupServer and refinement
       InstallVirtualServerByScript
53 Applying operator InstallVirtualServerByScript to task SetupServer
54 Remaining task network:
55   Preliminary plan: RunBackupScript(db <-- [Database=5=WikiDb]),
         InstallVirtualServerByScript(mem <-- [Integer=null],server <-- [
         DbServer=16=nil])
56   Remaining tasks: CopyDatabaseContent,InstallDatabaseSoftware,
57 Available FIRST TASK:
58 [0] Task: CopyDatabaseContent
59 [1] Task: InstallDatabaseSoftware
60 Choose a FIRST TASK: ? 1
61 Available REFINEMENT OPERATION:
62 [0] InstallOracleDbScript
63 [1] InstallMysqlDbScript
64 Choose a REFINEMENT OPERATION: ? 0
65 Backtrack over remaining REFINEMENT OPERATION? (y/n)? n
66 Calculating bindings for task InstallDatabaseSoftware and refinement
       InstallOracleDbScript
67 Applying operator InstallOracleDbScript to task InstallDatabaseSoftware
68 Remaining task network:
69   Preliminary plan: RunBackupScript(db <-- [Database=5=WikiDb]),
         InstallVirtualServerByScript(mem <-- [Integer=null],server <-- [
         DbServer=16=nil]),InstallOracleDbScript(db <-- [Database=17=nil],
         dbserver <-- [DbServer=16=nil])
70   Remaining tasks: CopyDatabaseContent,
71 Calculating bindings for task CopyDatabaseContent and refinement
       CopyDbScript
72 Applying operator CopyDbScript to task CopyDatabaseContent
73 Found new plan (1): RunBackupScript(db <-- [Database=5=WikiDb]),
       InstallVirtualServerByScript(mem <-- [Integer=null],server <-- [
       DbServer=16=nil]),InstallOracleDbScript(db <-- [Database=17=nil],
       dbserver <-- [DbServer=16=nil]),CopyDbScript(fromDb <-- [Database=5=
       WikiDb],toDb <-- [Database=17=nil])
74 Backtracking
75 Backtracking
76 Backtracking
77 Backtracking
78 Backtracking
79 Backtracking
80
81
82 Found plans (total 1)
83 (0) RunBackupScript(db <-- [Database=5=WikiDb]),
       InstallVirtualServerByScript(mem <-- [Integer=null],server <-- [
```

```
    DbServer=16=nil]),InstallOracleDbScript(db <-- [Database=17=nil],
    dbserver <-- [DbServer=16=nil]),CopyDbScript(fromDb <-- [Database=5=
    WikiDb],toDb <-- [Database=17=nil])
84 Action CopyDbScript: starts [80.0, 110.0] ends [120.0, 150.0] duration
    [40.0, 40.0]
85 Task SetupServer: starts [50.0, 80.0] ends [60.0, 90.0] duration [10.0,
    40.0]
86 Task InstallDatabase: starts [50.0, 80.0] ends [80.0, 110.0] duration
    [30.0, 60.0]
87 Action InstallOracleDbScript: starts [60.0, 90.0] ends [80.0, 110.0]
    duration [20.0, 20.0]
88 Task InstallDatabaseSoftware: starts [60.0, 90.0] ends [80.0, 110.0]
    duration [20.0, 50.0]
89 Task CopyDatabaseContent: starts [80.0, 110.0] ends [120.0, 150.0]
    duration [40.0, 70.0]
90 Task SpeedUpWebApplication: starts [0.0, 30.0] ends [120.0, 150.0]
    duration [120.0, 150.0]
91 Action RunBackupScript: starts [0.0, 30.0] ends [50.0, 80.0] duration
    [50.0, 50.0]
92 Action InstallVirtualServerByScript: starts [50.0, 80.0] ends [60.0,
    90.0] duration [10.0, 10.0]
93 Task BackupDatabase: starts [0.0, 30.0] ends [50.0, 80.0] duration
    [50.0, 80.0]
94
95
96
97
98 Total time consumed for HTN planning: 77405ms
99 Total time consumed for task network copying: 15ms (0%)
100
101
102
103 Total time consumed for knowledge base rollbacks: 624ms
104 Total time consumed for knowledge base assertions: 859ms
105 Total time consumed for knowledge base queries: 329ms
106 Total time consumed for knowledge base operations: 1812ms
```


Machine
«create»
+Machine
«create»
+Machine
+cpuSpeed:int
+memorySize:int
ManagedElement
«create»
+ManagedElement
«create»
+ManagedElement
+toString: String
+equals:boolean
-id:Long
-name:String
Human
«create»
+Human
«create»
+Human
«create»
+Human
+skills:Set<Skill>
+workingTimeHours:int
WebServer
«create»
+WebServer
«create»
+WebServer
+setJ2eeContainers:void
+getJ2eeContainers: Set<J2eeContai
+j2eeContainers: Set<J2eeContainer>
LoadBalancer
«create»
+LoadBalancer
«create»
+LoadBalancer
+balancedContainers: Set<J2eeConta
DbServer
«create»
+DbServer
«create»
+DbServer
+swapFileSize:int
+databases: Set<Database>
WebserverConfiguration
«create»
+WebserverConfiguration
«create»
+WebserverConfiguration
«create»
+WebserverConfiguration
Skill
«create»
+Skill
«create»
+Skill
«create»
+Skill
«create»
+Skill
-skillFactor:int
Hardware
«create»
+Hardware
«create»
+Hardware
«create»
+Hardware
OracleDb
«create»
+OracleDb
«create»
+OracleDb
Database
«create»
+Database
«create»
+Database
MySqlDb
«create»
+MySqlDb
«create»
+MySqlDb
DatabaseDesign
«create»
+DatabaseDesign
«create»
+DatabaseDesign
«create»
+DatabaseDesign
J2eeJdbcResource
«create»
+J2eeJdbcResource
«create»
+J2eeJdbcResource
+database: Database
JavaApplication
«create»
+JavaApplication
«create»
+JavaApplication
+jvm: JVM
J2eeContainer
«create»
+J2eeContainer
«create»
+J2eeContainer
+setJ2eeModules:void
+getJ2eeModules: Set<J2eeModule>
+setJ2eeApplications:void
+getJ2eeApplications: Set<J2eeAppli
+j2eeApplications: Set<J2eeApplicatic
+j2eeModules: Set<J2eeModule>
J2eeModule
«create»
+J2eeModule
«create»
+J2eeModule
JVM
«create»
+JVM
«create»
+JVM
«create»
+JVM
+version: String
J2eeApplication
«create»
+J2eeApplication
«create»
+J2eeApplication
+setJ2eeJdbcResources:void
+getJ2eeJdbcResources: Set<J2eeJd
+j2eeJdbcResources: Set<J2eeJdbcR
ECommerceJ2eeApplication
«create»
+ECommerceJ2eeApplication
«create»
+ECommerceJ2eeApplication
Log4jJ2eeModule
«create»
+Log4jJ2eeModule
«create»
+Log4jJ2eeModule
1
0..*

List of Abbreviations

AI ..... Artificial Intelligence
CI ..... Configuration Item [6]
CIM ..... Common Information Model [10]
CMDB ..... Configuration Management Database [6]
HTN ..... Hierarchical Task Network
ITIL ..... IT Infrastructure Library
ITSM ..... IT Service Management
RFC ..... Request for Change
STP ..... Simple Temporal Problem [34]
UML ..... Unified Modeling Language

List of Figures

List of Algorithms

Listings

Bibliography

[1] *IT Infrastructure Library.* Office of Government Commerce, UK, 2003.

[2] D. Dubie, "Itil adoption increases in u.s., proficiency still lacking." [Online]. Available: http://www.networkworld.com/news/2008/022908-itil-adoption.html

[3] R. Rebouças, R. Santos, J. Sauvé, and A. Moura, "The HP-Bottom Line Project, IT Change Management Challenges - Results of 2006 Web Survey, Technical Report DSC005-06," Computing Systems Department, Federal University of Campina Grande, Brazil, Tech. Rep., 2006.

[4] W. Cordeiro, G. Machado, F. Daitx, C. Both, L. Gaspary, L. Granville, A. Sahai, C. Bartolini, D. Trastour, and K. Saikoski, "A Template-based Solution to Support Knowledge Reuse in IT Change Design," in *11th IEEE/IFIP Network Operations and Management Symposium (NOMS).* IEEE, 2008, pp. 355–362.

[5] D. Nau, M. Ghallab, and P. Traverso, *Automated Planning: Theory & Practice.* Morgan Kaufmann Publishers Inc., 2004.

[6] *IT Infrastructure Library, "ITIL Service Transition".* Office of Government Commerce, UK, 2003.

[7] P. Anderson and A. Scobie, "LCFG: The Next Generation," University of Edinburgh, 2002. [Online]. Available: http://www.lcfg.org/doc/ukuug2002.pdf

[8] "HP Data Center Automation Center." [Online]. Available: https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-271-273_4000_100___

[9] *IT Infrastructure Library, "ITIL Service Operation".* Office of Government Commerce, UK, 2003.

[10] "Common Information Model (CIM)," Distributed Management Task Force, 2007. [Online]. Available: http://www.dmtf.org/standards/cim/

[11] "Business Process Execution Language." [Online]. Available: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf

[12] "Workflow Management Coalition Workflow Standard (WFMC)," Workflow Management Coalition, 2002. [Online]. Available: http://www.wfmc.org/standards/docs/TC-1025_10_xpdl_102502.pdf

[13] "RuleML." [Online]. Available: http://www.ruleml.org/

[14] "Drools Business Rule Management System." [Online]. Available: http://www.jboss.org/drools/

[15] A. Keller, J. Hellerstein, J. Wolf, K. Wu, and V. Krishnan, "The CHAMPS System: Change Management with Planning and Scheduling," in *9th IEEE/IFIP Network Operations and Management Symposium (NOMS)*. IEEE, 2004, pp. 395–408.

[16] A. Andrzejak, U. Hermann, and A. Sahai, "Feedbackflow-an adaptive workflow generator for systems management," in *ICAC*, 2005, pp. 335–336.

[17] J. P. Sauvé, R. Rebouças, A. Moura, C. Bartolini, A. Boulmakoul, and D. Trastour, "Business-driven decision support for change management: Planning and scheduling of changes," in *17th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM)*. Springer, 2006, pp. 173–184.

[18] R. Rebouças, J. P. Sauvé, A. Moura, C. Bartolini, and D. Trastour, "A decision support tool to optimize scheduling of it changes," in *10th IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE, 2007, pp. 343–352.

[19] D. Trastour, M. Rahmouni, and C. Bartolini, "Activity-based scheduling of it changes," in *International Conference on Autonomous Infrastructure, Management and Security (AIMS)*. Springer, 2007, pp. 73–84.

[20] T. Setzer, K. Bhattacharya, and H. Ludwig, "Decision support for service transition management," in *IEEE Network Operations and Management Symposium (NOMS)*, 2008.

[21] L. Ramshaw, A. Sahai, J. Saxe, and S. Singhal, "Cauldron: A policy-based design tool," in *7th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2006)*. IEEE Computer Society, 2006, pp. 113–122.

[22] K. Erol, J. Hendler, and D. S. Nau, "Htn planning: Complexity and expressivity," in *Twelfth National Conference on Artificial Intelligence (AAAI-94)*. AAAI Press/MIT Press, 1994, pp. 1123–1128.

[23] T. Estlin, R. Castano, R. Anderson, D. Gaines, F. Fisher, and M. Judd, "Learning and Planning for Mars Rover Science," in *IJCAI 2003 workshop notes on Issues in Designing Physical Agents for Dynamic Real-Time Environments*, 2003.

[24] J. Fernández-Olivares, L. A. Castillo, Ó. García-Pérez, and F. Palao, "Bringing users and planning technology together. experiences in siadex," in *Sixteenth International Conference on Automated Planning and Scheduling (ICAPS)*. AAAI, 2006, pp. 11–20.

[25] Y. Gil, E. Deelman, J. Blythe, C. Kesselman, and H. Tangmunarunkit, "Artificial intelligence and grids: Workflow planning and beyond," *IEEE Intelligent Systems*, vol. 19, no. 1, pp. 26–33, 2004.

[26] B. Srivastava, J. Vanhatalo, and J. Koehler, "Managing the life cycle of plans," in *AAAI*, M. M. Veloso and S. Kambhampati, Eds. AAAI Press / The MIT Press, 2005, pp. 1569–1575.

[27] E. Sirin, B. Parsia, and J. Hendler, "Template-based composition of semantic web services," *AAAI Fall Symposium on Agents and the Semantic Web*, 2005.

[28] C. Knoblock, S. Minton, J. Ambite, M. Muslea, J. Oh, and M. Frank, "Mixed-initiative, multi-source information assistants," *Proceedings of the 10th international conference on World Wide Web*, pp. 697–707, 2001.

[29] G. Ferguson and J. Allen, "TRIPS: An Integrated Intelligent Problem-Solving Assistant," *Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, pp. 567–572, 1998.

[30] G. Ferguson, J. Allen, and B. Miller, "TRAINS-95: Towards a mixed-initiative planning assistant," *Proceedings of the Third Conference on Artificial Intelligence Planning Systems (AIPS-96)*, pp. 70–77, 1996.

[31] James South, "If you put a penis on Britannia she's just like Brad Pit in Fight Club," 2008.

[32] F. Yaman and D. Nau, "TimeLine: An HTN Planner That can Reason About Time," *AIPS 2002 Workshop on Planning for Temporal Domains*, 2002.

[33] L. A. Castillo, J. Fernández-Olivares, Ó. García-Pérez, and F. Palao, "Efficiently handling temporal knowledge in an htn planner," in *Sixteenth International Conference on Automated Planning and Scheduling (ICAPS)*. AAAI, 2006, pp. 63–72.

[34] R. Dechter, I. Meiri, and J. Pearl, "Temporal Constraint Networks," *Artificial Intelligence*, vol. 49, no. 1-3, pp. 61–95, 1991.

[35] R. Dechter, *Constraint Processing*. Morgan Kaufmann, 2003.

[36] G. Machado, F. Daitx, W. Cordeiro, C. Both, L. Gaspary, L. Granville, C. Bartolini, A. Sahai, D. Trastour, and K. Saikoski, "Enabling rollback support in IT change management systems," in *Network Operations and Management Symposium, 2008. NOMS 2008. IEEE*, 2008, pp. 347–354.

[37] "Hibernate." [Online]. Available: http://www.hibernate.org/

[38] "Jena - A Semantic Web Framework for Java." [Online]. Available: http://jena.sourceforge.net/

[39] "HSQLDB." [Online]. Available: http://hsqldb.org/

[40] A. Cesta and A. Oddi, "Gaining efficiency and flexibility in the simple temporal problem," in *TIME '96: Proceedings of the 3rd Workshop on Temporal Representation and Reasoning (TIME'96)*. Washington, DC, USA: IEEE Computer Society, 1996, p. 45.

[41] W. Cordeiro, G. Machado, F. Andreis, A. Santos, C. Both, L. Gaspary, L. Granville, C. Bartolini, and D. Trastour, "A Runtime Constraint-Aware Solution for Automated Refinement of IT Change Plans," *Lecture notes in computer science*, vol. 5273, pp. 69–82, 2008.

[42] S. Edelkamp and J. Hoffmann, "PDDL2.2: The language for the classical part of the 4th international planning competition," in *4th International Planning Competition (IPC'04), at ICAPS'04*, 2004.

[43] R. Dechter and J. Pearl, "Generalized best-first search strategies and the optimality of a*," *J. ACM*, vol. 32, no. 3, pp. 505–536, 1985.

[44] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2002.

[45] F. Manola, E. Miller, and B. McBride, "RDF Primer - W3C Recommendation," 2004. [Online]. Available: http://www.w3.org/TR/2004/REC-rdf-primer-20040210

[46] S. Bechhoffer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuiness, P. F. Patel-Schneider, and L. A. Stein, "OWL Web Ontology Language Reference - W3C Recommendation," 2004. [Online]. Available: http://www.w3.org/TR/2004/REC-owl-ref-20040210/

[47] D. Reynolds, C. Thompson, J. Mukerji, and D. Coleman, "An assessment of RDF/OWL modelling," HP technical report, HP Laboratories Bristol, Hewlett-Packard, HPL-2005-189, 2005, Tech. Rep.

[48] R. Fikes and N. Nilsson, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," *Artificial Intelligence*, vol. 2, no. 3/4, pp. 189–208, 1971.

[49] D. Nau, "Lecture slides for Automated Planning: Theory and Practise." [Online]. Available: http://www.cs.umd.edu/~nau/planning/slides/

[50] K. Erol, J. Hendler, and D. Nau, "Complexity results for hierarchical task-network planning," *Annals of Mathematics and Artificial Intelligence*, vol. 18, no. 1, pp. 69–93, 1996.

[51] M. Lekavy and P. Navrat, "Expressivity of STRIPS-Like and HTN-Like Planning," *Lecture notes in computer science*, vol. 4496, p. 121, 2007.

[52] D. Nau, H. Munoz-Avila, Y. Cao, A. Lotem, and S. Mitchell, "Total-order planning with partially ordered subtasks," *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, pp. 425–430, 2001.

[53] O. Ilghami, "Documentation for JSHOP2," 2005.

[54] J. Allen, "Maintaining knowledge about temporal intervals," *Communications of the ACM*, vol. 26, no. 11, pp. 832–843, 1983.

[55] M. Vilain, H. Kautz, and P. van Beek, "Constraint propagation algorithms for temporal reasoning: A revised report," *Readings in Qualitative Reasoning about Physical Systems*, pp. 373–381, 1989.

[56] U. Montanari, "Networks of Constraints: Fundamental Properties and Applications to Picture Processing," 1971.

[57] "SHOP and JSHOP." [Online]. Available: http://www.cs.umd.edu/projects/shop/description.html