Technische Universität München
Fakultät für Informatik
Lehrstuhl für Datenbanksysteme

One is not enough: A hybrid HTN/Classical Planning approach for IT Change Management

Sebastian Hagen

Masterarbeit im Elitestudiengang Software Engineering

Autor:	Sebastian Hagen
Erstgutachter:	Prof. Alfons Kemper, Phd. (TUM)
Zweitgutachter:	Prof. Dr. Bernhard Bauer (Universität Augsburg)
Betreuer:	Daniel Gmach (TUM)
	Martina Albutiu (TUM)

Erklärung:

Ich versichere, dass ich diese Masterarbeit selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

(Ort, Datum)	(Unterschrift)

Acknowledgement:

This work would not have been possible without the support of various people. First of all, I want to thank Prof. Alfons Kemper from TUM for giving me the possibility to get to know what it is like to work in an industrial research environment at HP Labs. In addition to that, I want to thank my thesis advisors Martina Albutiu and Daniel Gmach from TUM for their constant support and mentoring regarding this work. Their words of advice, especially regarding scientific writing, have intensively contributed to the quality of this work.

There are also various people at HP Labs I want to thank. I especially want to thank Nigel Edwards for giving me the opportunity to work within the SLiM project at HP Labs, for giving me the freedom and the trust to choose the direction of my work, and for his valuable constant input to my work. I also like to thank Lawrence Wilcock for his support and advices regarding this work as well as for the deeper insights to the SLiM project. In addition to that, I thank Johannes Kirschnick for sharing so many details about the SLiM project. Furthermore, I like to thank Andrew Farrel for his support and guidance regarding AI Planning techniques. Finally, I like to thank John Manley and all other colleagues in the Automated Infrastructure Lab for creating an enjoyable working environment.

In the end I like to thank all professors form the Elite Graduate Program Software Engineering. They did not only give me the possibility to further develop my technical and research, but also my personal skills over the last two years. In addition to that, I like to thank my parents for always supporting me with my studies.

Abstract. Change design is an important part of IT Change Management. It is concerned with the creation of a plan in order to achieve a high-level goal. Recent work has been focusing on planning by refinement of tasks, e. g., using HTN approaches. These approaches do not consider that domain objects have states, leading to various drawbacks.
First, hierarchical problem solving behavior is mixed with domain object behavior. Second, dependencies are not made explicit and they can be better described by referring to the states of other domain objects.
To overcome these drawbacks we design a hybrid HTN/Classical Planning approach that copes with task refinement and planning according to states of domain objects. Using the proposed algorithm we demonstrate how to write knowledge bases that clearly separate hierarchical problem solving behavior, behavior of domain objects, and dependencies. This increases maintainability, extendability, and reuseability of our knowledge bases.
We demonstrate the applicability of our algorithm and proposed domain description by applying it to change request planning of SAP systems.

Contents

1. Introduction 1
1.1. Motivation . . . 1
1.2. Research problem . . . 2
1.3. Proposed solution . . . 4
1.4. Outline of the work . . . 5

2. The SAP case study 6
2.1. The Model Information Flow . . . 6
2.1.1. Idea of the Model Information Flow . . . 6
2.1.2. States of the Model Information Flow . . . 7
2.1.3. Characteristics of the Model Information Flow . . . 9
2.2. Domain objects of the planning domain . . . 10
2.2.1. View on SAP systems in the Deployed state . . . 10
2.2.2. A general behavioral model for domain objects . . . 13
2.3. Dependencies in the SAP case study . . . 14
2.3.1. Dependencies of a database Grounded Execution Service . . . 14
2.3.2. Dependencies of a central-instance Grounded Execution Service . . 15
2.3.3. Dependencies of a dialog-instance Grounded Execution Service . . 16
2.3.4. Additional dependencies regarding other model components . . . 16
2.3.5. A general model for dependencies between domain objects . . . 17
2.4. Workflows within the SAP case study . . . 17
2.5. Model of the planning domain . . . 18
2.6. Requirements for a planning approach . . . 20

3. Evaluation of existing planning approaches 23
3.1. Classical Planning . . . 23
3.1.1. A conceptual model for planning . . . 23
3.1.2. A restricted model for planning . . . 24
3.1.3. Classical Planning . . . 26
3.2. Hierarchical Task Network Planning . . . 27
3.3. Limitations of the planning approaches . . . 30
3.3.1. Planning over states of objects . . . 30
3.3.2. Changes to the lifecycle of domain objects . . . 32
3.3.3. Description of workflows . . . 33
3.3.4. Description of dependencies . . . 33
3.3.5. Notion of hierarchical decomposition . . . 36

3.4. Need for a Hybrid Planning approach 37

4. Overview of the hybrid approach 38

4.1. Basic ideas behind the hybrid approach 38

4.1.1. Hierarchical task refinement 38

4.1.2. State-transition systems 38

4.1.3. Describing dependencies in the hybrid approach 40

4.2. High-level example of the hybrid approach 41

4.3. The conceptual model of the hybrid approach 44

5. A Domain Specific Language describing the Hybrid Planning domain 51

5.1. The Groovy language for DSL specification 51

5.1.1. Short overview of Groovy 51

5.1.2. Groovy closures 52

5.1.3. Dynamic object orientation 56

5.1.4. Advantages of Groovy as a language for planning 57

5.2. The DSL to describe state-transition systems 58

5.2.1. The frame to describe state-transition systems 59

5.2.2. Describing transitions 62

5.2.3. Describing dependencies 64

5.3. A DSL to describe HTN methods 67

5.4. A DSL to describe HTN operators 70

6. The hybrid HTN/Classical Planning approach 72

6.1. Model of the knowledge base defined by the DSL 72

6.2. The algorithm of the hybrid approach 74

6.2.1. Hybrid approach top-level algorithm 74

6.2.2. HTN Planning algorithm 75

6.2.3. Classical Planning algorithm 78

6.3. Example of the hybrid approach algorithm 81

6.4. Temporal constraints and plan consistency 86

6.4.1. Maintaining temporal constraints during task decomposition 87

6.4.2. Plan extension and plan consistency 91

6.4.3. Dependencies for reserved change requests 94

7. Related work 99

7.1. Related Hybrid HTN Planning approaches 99

7.2. Related Change Management approaches 102

8. Conclusion and future work 106

A. Appendix 108

List of Figures

2.1. States of the *Model Information Flow* described by a model, the *Service Lifecycle Model* 7
2.2. A *decentralized SAP* system 11
2.3. State-transition systems for virtual machines, physical machines, services, and volumes 12
2.4. An UML class diagram describing the planning domain and the characteristics of the class “AI_ChangeRequest” 19

3.1. A conceptual model for *offline planning* [22] 25
3.2. An hierarchical task network for decomposing the task “install_software” in the context of an SAP system. 29
3.3. HTN Planning considering states 31
3.4. HTN Planning taking dependencies and states of domain objects into account 34

4.1. Example of a non-atomic change request linked to a transition of a state-transition system 39
4.2. Example of a non-atomic change request and a dependency linked to a transition 40
4.3. A high-level view of the hybrid approach algorithm 41

5.1. EBNF for the frame of state-transition systems 59
5.2. EBNF for transitions and subtasks 62
5.3. EBNF for dependencies 64
5.4. EBNF description for HTN methods 67
5.5. EBNF describing HTN operators 70

6.1. UML class diagram describing the knowledge base of the planner 73
6.2. Decomposition tree after expansion of change request a 88
6.3. Decomposition tree after expansion of change request b 88
6.4. Decomposition tree after expansion of change request d 89
6.5. Decomposition tree after expansion of change request g 90
6.6. Final plan consisting of atomic tasks and $after$ relation 90
6.7. Unspecified temporal relationships between parallel subtasks 92
6.8. Dependencies between the subtasks of a reserved change request 94

List of Tables

2.1. Dependencies of database transitions 15
2.2. Dependencies of central-instance transitions 15
2.3. Dependencies of dialog-instance transitions 16

Listings

5.1. Defining and calling a closure 52
5.2. Defining and passing a closure to a method 53
5.3. Different possibilities how to pass a closure to a closure 54
5.4. A simple DSL for state-transition systems 55
5.5. Code to define the closures from Listing 5.4 56
5.6. A DSL using calls to unknown methods 57
5.7. Method to catch calls to unknown methods 58
5.8. A DSL describing the frame of state-transition systems 60
5.9. Possible "valid" closures 60
5.10. The part of the DSL describing transitions 62
5.11. DSL describing dependencies for the "install" transition 65
5.12. DSL describing an HTN method 68
5.13. "try_all" closure describing alternatives in the decomposition 69
5.14. DSL describing an HTN operator 71

A.1. Pseudo code to plan for the "resolve_dependency" change request 108

1. Introduction

This chapter motivates and introduces the research done in this work. Particularly we start by motivating our research in the area of change request planning in Section 1.1. Although there has been done lots of research in this area over the last years, we identify a research problem that has not yet satisfactory been solved in Section 1.2. After that, we highlight in Section 1.3 how the solution proposed in this work solves the identified research problem. Finally, we provide an outline for the remainder of this work in Section 1.4.

1.1. Motivation

Data centers have been massively grown in size and numbers over the last years. In September 2002 Google reported to have 15,000 servers in six data centers [27]. By 2005 Stephen Arnold [5] reported that Google has grown to over 100,000 systems in 15 data centers. Today's estimates assume about 500,000 servers in over 35 data centers [25]. The growth of data centers in general is very much driven by the proliferation of the *Software as a Service* (*SaaS*) paradigm, the need for commercial high-performance computing, and the success of advertisement based services and online consumer services [25]. The increasing popularity of the *Cloud Computing* paradigm and the data centers needed by Cloud Computing providers emphasize this development. Cloud Computing offers SaaS in a large, scalable, distributed computing environment, the *Cloud*. With the increase in the density of data centers, there comes more software being hosted in the "Cloud". But not only the amount of software is increasing but also its complexity because it has become affordable to build large scalable distributed software systems hosted in data centers. For example, Animoto [4], [41] is a service that takes a music file and a set of pictures and renders both to a video clip showing the pictures based on characteristics of the music. Animoto's rendering service is based on Amazon EC2 [2] and S3 [3], Amazon's Cloud Computing services. A few days after Animoto has been launched on Facebook, it was gradually scaled up from 50 EC2 computing nodes, i. e., 50 virtual machines doing the rendering, to 3,500 EC2 nodes. Depending on the load, the Animoto application automatically increases or decreases the number of rendering units. Another project used about 100 EC2 nodes for the generation of about 11 million pdf articles making up the archive of the New York Times from 1851 until 1922 [24]. These large scale services put additional pressure on the IT Change Management that is done behind the scenes at the Cloud Computing providers. For example virtual machines have to be migrated to other physical servers due to workload or maintenance constraints. The pure size of data centers and massively distributed applications like Animoto make it impossible to oversee the current state of the data center without assistance by a computer.

According to [25] large scale data centers can only be operated cost-efficiently if the standard processes in the data center are automated. One of these processes is IT *Change Management* [29], which defines how changes are dealt with in a data center environment. Changes to the IT environment, i. e., software systems and hardware systems, are formalized by a *Request for Change* (*RFC*). The RFC is mapped to a first high-level change request describing the high-level goal in order to achieve the RFC. One important step in this process is the planning of the change request. Given a high-level task, e. g., to bring up a new Cloud service, a detailed executable plan is generated. The actions that make up the plan change the software and virtual infrastructure running in the data center in order to achieve the high-level goal. Due to the size of todays data centers and complexity of software hosted in them, automated change planning becomes a key technology to automate the maintenance of data centers. Plans for such complex systems cannot be generated by hand any more because

- operators are not capable of overseeing the current state of a massively distributed application like Animoto any more.
- the effects of changes are difficult to assess in an environment that is heavily driven by constraints induced by software components. For example, it might be difficult to assess in which order applications can be shutdown because they might depend on each other.
- policies accounting in the data center, e. g., security policies, are not allowed to be violated while performing the changes. In addition to that, the data center has to be changed constantly to conform to policies. For example, virtual machines with less load might be aggregated on a server to free resources for additional virtual machines.
- generating plans by hand becomes an highly error-prone task due to the complexity of distributed software systems and the dependencies accounting between different software components. In addition to that, plans can contain hundreds of steps growing too big in order to be created by hand.

All of these aspects motivate the research done in change request planning.

1.2. Research problem

Because of the importance of change request planning as outlined in Section 1.1, different approaches for change request planning have been studied over the last years:
Early work was done by Keller et al. [31] on the *IBM CHAMPS* system. The work focuses on planning and scheduling for non-hierarchical tasks. It tries to exploit a high degree of parallelism. Dependencies between software components are defined by data structures. Compared to our work findings about planning from the area of *Artificial Intelligence* (*AI*) *Planning*[1] are not considered, leading to the creation of corrupt plans. In

[1]AI Planning is a branch of Artificial Intelligence that deals with algorithms and strategies to construct plans of actions to achieve a certain goal.

addition to that, the hierarchy inherent to the domain of IT Change Management planning is not taken into account. Furthermore, CHAMPS does planning exclusively due to dependencies and does not consider planning according to best practices in Change Management.
Cordeiro et al. [10] overcome the disadvantages of CHAMPS in the hierarchical domain by proposing *ChangeLedge*, a system for the hierarchical refinement of change plans. To the best of our knowledge it is the first work that introduces a planning algorithm that takes the hierarchical refinement of change requests into account. High-level plans are gradually refined according to dependencies until a workflow is reached that can be directly executed. The drawbacks of Cordeiro's approach are that refinement based on best practices and workflow specification cannot be expressed on lower refinement levels. Furthermore, plans are not generated taking best practices of IT Change Management into account. Although the approach takes preconditions and effects of operators into account, it can generate unsound plans.
Recently, Trastour et al. have introduced *ChangeRefinery* [46], a system for the automated refinement of change requests. Compared to ChangeLedge, ChangeRefinery applies *Hierarchical Task Network* (*HTN*) Planning [16], a well defined AI Planning technique, to the domain of IT change request planning for the first time. HTN refines abstract high-level tasks into a set of atomic tasks that cannot be decomposed any further. This tasks are then executed to implement the plan. Although HTN is suited for change request refinement, it has some drawbacks when applied to IT change request planning because the domain of IT change management is not entirely hierarchical.
When looking at the domain objects of an IT change request planning domain, it becomes apparent that lots of these objects have a state. For example a database can be "running" or "stopped". The same accounts for an application server. Virtual machines can be in states like "undeployed", "deployed", "running", or "paused" depending on the underlying virtualization technique. Using HTN in a domain where domain objects do have a state and where refinement of tasks is necessary, leads to the mixture of hierarchical problem solving knowledge and the description of behavior of domain objects. The behavior of domain objects is better expressed by state-transition systems. Mixing both in one data structure decreases the comprehensibility of the knowledge base. In addition to that, dependencies in the IT change request planning domain are best described by referring to the state of domain objects. For example, a database can only be stopped if all other services accessing the database are already in state "stopped". Nevertheless, stopping a database might be a more complex task with need for further refinement. Describing the dependencies in a pure HTN domain mixes up tasks that solve dependencies and tasks changing the state of the domain object the dependency is solved for. In addition to that, the states defining a domain object are implicitly coded into then HTN methods. All the mentioned issues lead to decreased readability, maintainability, and adaptability of the knowledge base. This makes it hard to cope with changes of the knowledge base, e. g., when new virtualization techniques or new kinds of services with different dependencies are added to the planning domain. Our work aims at overcoming these problems

by introducing a hybrid HTN/Classical Planning[2] approach for IT change request planning.

1.3. Proposed solution

In this work we introduce a hybrid HTN/Classical Planning approach for IT change request planning to overcome the problems of a pure HTN approach as described in Section 1.2. Our approach combines HTN with an extended version of Classical Planning enabling us to seamlessly switch between both planning techniques. The hierarchical refinement of change requests is described by HTN methods. Domain objects have a state-transition system associated describing their behavior. Dependencies are linked to transitions in the state-transition systems. They describe Classical Planning problems that need to be solved in order to execute the transition. Transitions of a state-transition system are linked to a task that can be subject to further refinement by HTN methods. This task needs to be solved when taking the transition.
The hybrid approach has the following advantages compared to pure HTN approaches like [46] in the domain of IT change request planning:

- Clear separation between data structures describing the behavior of domain objects, e. g., a service, virtual machine, or physical machine, and hierarchical problem solving behavior.
- Clear expression of dependencies between domain objects by Classical Planning problems.
- Increased maintainability, adaptability, and extendability of the planning domain due to separated descriptions for object behavior, hierarchical problem solving, and dependencies.
- No indirect coding of domain object behavior into HTN methods. The behavior of domain objects is made explicit.
- An algorithm that has a well defined semantics for dependencies in the domain of IT Change Management.
- Computation of HTN decompositions by solving Classical Planning problems.

We derive the idea of the hybrid approach by conducting a case study. The case study examines change requests in the context of SAP systems, a software product frequently hosted in corporate data centers. Upon this case study we derive a general model to describe dependencies and finally the hybrid approach.

[2]Classical Planning is a planning approach that plans by searching through the state-space of a restricted state-transition system [22].

1.4. Outline of the work

The remainder of this work is organized as follows.

Chapter 2 introduces the SAP case study upon which the development of the hybrid approach is based. We introduce the planning domain and explain why hierarchical task decomposition as well as the notion of state-transition systems is needed. In addition to that, we explain the dependencies that account in the domain and finally derive eight requirements for the hybrid approach.

Chapter 3 provides an introduction to HTN and Classical Planning. By giving some examples from the SAP case study we argue that neither a pure HTN approach nor a Classical Planning approach fulfills all of our needs. In addition to that, we highlight how the hybrid approach overcomes the weaknesses of the two pure approaches.

Chapter 4 introduces the main ideas of the hybrid approach. We describe how HTN and Classical Planning are linked together by the hybrid approach. Furthermore, we explain a basic high-level run of the hybrid approach algorithm and introduce the theoretic model underlying the hybrid approach.

Chapter 5 Proposes a Domain Specific Language which is used to describe the planning domain of the hybrid approach. We explain how the DSL facilitates reuseability, extendability, and the separation of concepts like domain object behavior, hierarchical refinement rules, and dependencies. An EBNF description of the DSL is given in order to define the precise syntax of the DSL. Furthermore, we provide some examples showing how the planning domain of the SAP case study is described in the DSL.

Chapter 6 explains the algorithm of the hybrid approach. We provide separate algorithms for the HTN and the Classical planner that call each other to do the planning. Having introduced the algorithm, we show how it solves an example from the SAP case study. Furthermore, we explain how the algorithm keeps track of temporal dependencies in order to produce sound plans.

Chapter 7 takes a closer look at related work regarding hybrid HTN Planning approaches in general as well as related planning approaches in the field of IT change request planning.

Chapter 8 concludes the work and explains our plans for future work.

2. The SAP case study

This chapter introduces the SAP case study. It explains the challenges a planning approach for the SAP case study has to face and solve. We explain the domain objects, their characteristics, and the constraints and dependencies a planner needs to plan for in the SAP case study. Thus, this chapter builds an important foundation for the solution proposed later on in this work.
First of all, we explain the characteristics of the Model Information Flow. The planning approach developed in this work needs to be able to plan for the changes of the Model Information Flow. After that, Section 2.2 explains the domain objects of the SAP case. In addition to that, we describe how our approach captures the behavior of domain objects and how it differs compared to previous work. We continue with a precise description of dependencies that account for the SAP case study in Section 2.3. After that, Section 2.4 introduces some workflows, i. e., best practices how to manage software and hardware in the SAP case study. A planner needs to be able to plan according to workflows. Furthermore, we introduce an UML class diagram in Section 2.5 which describes the domain objects of the planning domain and their relationship to change requests. The hybrid approach planner uses this object oriented model in order to do the planning. Finally, Section 2.6 derives requirements from the analysis conducted in this chapter. They should be satisfied by the Hybrid Planning approach developed in this work.

2.1. The Model Information Flow

The Model Information Flow (MIF) is an important concept of the SLiM project. Because the MIF changes the state of the world, a planner needs to be able to plan for its changes. This section provides an introduction to the Model Information Flow. The big picture behind the Model Information Flow is introduced in Subsection 2.1.1, followed by an overview of the states of the MIF in Subsection 2.1.2. Finally, we extract the main characteristics of the Model Information Flow having an influence on the design of our planner in Subsection 2.1.3.

2.1.1. Idea of the Model Information Flow

This subsection provides an introduction to the Model Information Flow. The *Model Information Flow* provides a generic, model based view on components of a data center. Hardware, virtual machines (VMs), and software components are administered over their lifecycle by a well defined model, the *Model Information Flow*.
In order to deploy an SAP system, there are lots of different actions necessary, that need to be done step by step. Such actions are, for example, the collection of non-functional or

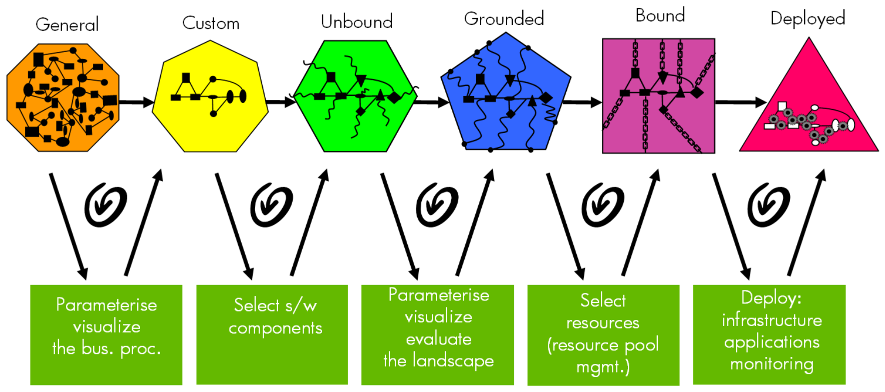

General
Custom
Unbound
Grounded
Bound
Deployed
Parameterise visualize the bus. proc.
Select s/w components
Parameterise visualize evaluate the landscape
Select resources (resource pool mgmt.)
Deploy: infrastructure applications monitoring

Figure 2.1.: States of the *Model Information Flow* described by a model, the *Service Lifecycle Model*

functional requirements and the deployment of the actual system on the physical hardware. No matter which service, e. g., an SAP system or any other Cloud service, is to be brought to life, the *Model Information Flow* describes best practice steps and the necessary tools in order to automate this process. The *Model Information Flow* starts with a very abstract model and refines this model until a model is reached which formalizes software components running on virtual machines and physical machines. Each of these refinement steps takes the model, which is an object oriented EMF model, and changes attributes of objects or adds / deletes new objects including associations. These changes to the model are done by tools, e. g., stand alone applications or just software packages of the SLiM project. Thus, the model always reflects the current state of the system. A planning approach within the project needs to be able to plan for the changes of the model through the Model Information Flow. Changing the state of the model involves the invocation of a set of tools in sequence or in parallel. Thus, a planner needs to be able to compute a set of sequential or parallel tool invocations in order to reach a well defined state of the Model Information Flow.

2.1.2. States of the Model Information Flow

This subsection gives an overview of the different states of the Model Information Flow and the change requests necessary to change the state of the model. Change requests can be directly matched to tool invocations in order to accomplish a transition to another state of a model. An in depth introduction to the states of the Model Information Flow can be found in [42], which also builds the basis for this subsection.

General state

The *General* state (see Figure 2.1) is the initial state of the *Model Information Flow*. It describes abstract requirements on a business process level. After the model has been instantiated in General state, its state can be changed to the Custom state.

A change request changing the state to the Custom state would be "change state from General to Custom".

Custom state

The *Custom* state adds functional and non-functional requirements to the model. Functional requirements can further refine the business processes described by the model. Non-functional characteristics can be, e. g., security, performance, reliability, and scalability measures.

In order to transform the model from General to Custom state, there needs to be a change request to collect the non-functional requirements like "collect non-functional requirements". Another change request in this context is the change request "start Template Browser" which starts the *Template Browser*, a tool enabling the customer to navigate through the model. Further requirements need to be collected through additional change requests as needed. As soon as all the functional requirements are collected from the customer, the state of the model can be changed to the Unbound state. In order to get back to the General state a generic change request called "restore model to General state" can be used.

Unbound state

The *Unbound* state adds information from the software vendors to the Custom model. The information from the software vendors describe the components that are necessary in order to support the chosen business processes. Components could be, e. g., application servers, databases, or external services.

The transition from the Custom to Unbound state can be performed using the Template Browser. In order to rollback the changes made to the model, the change request "restore model to Custom state" needs to be applied.

Grounded state

The *Grounded* state describes a complete design of the Cloud service. It contains details about the infrastructure design, the software components of the service, and their mapping to virtual machines.

In order to perform the transition from Unbound to Grounded, a design has to be chosen from different design alternatives. Thus, a change request "choose design template" needs to be planned for. In addition to that performance parameters need to be set by the "set performance parameters" change request. Finally, a new model, the System Model, needs to be created by the change request "create System Model".

Bound state

The *Bound* state extends the Grounded state by adding information about resource bindings. Resources like hosts, storage, and networking are acquired from a shared virtualized resource pool.

The transition from Grounded to Bound state involves calling a tool, the *Resource Acquisition Service*. It maps the virtual resources like virtual machines to physical resources. Thus, a change request "acquire resources" becomes necessary to transform the model to the Bound state. In order to get back to the Grounded state, we need to free the resources, which is done by the change request "release resources".

Deployed state

The *Deployed* state includes information about the deployed and running components that belong to the created service instance, e. g., the SAP system. In order to deploy the infrastructure, resources need to be configured by the *Resource Deployment Service*, software needs to be installed by the *Software Deployment Service*, and software is configured by the *Software Configuration Service*. Furthermore, application monitoring is added.

This leads to the change requests "create resources", "deploy software", "configure software", and "deploy monitoring". In order to get back to the Bound state, the changes mentioned above need to be reversed. Thus, the software needs to be stopped, monitoring needs to be removed, and resources need to be unbound. This leads to the change requests "terminate software", "stop monitoring", and "unbind resources".

2.1.3. Characteristics of the Model Information Flow

This subsection summarizes the main characteristics of the Model Information Flow and their effects on the planning algorithm. First of all, the Model Information Flow describes a set of best practice states in which a SLiM model can be. It is thus wise to have a notion of states in order to plan for the changes of the MIF. In addition to that, transitions between the states are accomplished by solving a set of partially[1] ordered change requests. These change requests might need to be further refined. This demands an hierarchical refinement of tasks linked to a transition. None of the previous works, e. g., [10] and [46], on IT change planning take an explicit notion of states together with the concept of task refinement into account. Previous work has only been focusing on hierarchical refinement of tasks.
Furthermore, change requests regarding the MIF can be generally described as "bring the model into state x", where x is a state of the Model Information Flow. Be aware that change requests of this pattern describe a Classical Planning problem [22], because we define the state a transition system needs to be brought in. All in all, the Model

[1]a partial order $<$ is a binary relation which is antisymmetric (i. e., $\forall a \forall b : a < b \wedge b < a \rightarrow a = b$), transitive (i. e., $\forall a \forall b \forall c : a < b \wedge b < c \rightarrow a < c$) and reflexive (i. e., $\forall a : a < a$)

Information Flow demands a planning approach that supports both, hierarchical task decomposition and the explicit notion of states. Such an approach is developed in this work.

2.2. Domain objects of the planning domain

This section introduces the domain objects of the planning domain, i. e., the domain objects the planner plans over. We analyze these objects in order to capture similarities between them. The different domain objects part of the SAP case study are explained in Subsection 2.2.1. Based on this introduction we derive a general behavioral model for the domain objects in Subsection 2.2.2. We explain our best practices how to describe the behavior of domain objects in the context of IT change planning and emphasize the differences to previous approaches.

2.2.1. View on SAP systems in the Deployed state

This subsection gives an overview of the model components of an SAP system as they are part of the Deployed state of a SLiM model. An SAP system consists of a database (DB) and a set of application servers. Application servers might be either running a central-instance (CI) or a dialog-instance (DI). A fully deployed SAP system always contains one database, one central-instance, and at least one dialog-instance.
Figure 2.2 shows a possible configuration of an SAP system consisting of a database, a central-instance, and a dialog-instance. An SAP system within the SLiM project comprises the following components:

Services:

The database, the central-instance, and the dialog-instances of an SAP system are modeled as services within the SLiM project. The expression service is used interchangeably with *Grounded Execution Service* (*GES*).
The following three services occur within the SAP case study:

- **database service:** The *database* service models the database software component of an SAP system.
- **central-instance service:** The *central-instance* service models an SAP central *Instance*[2] profile running on a VM.
- **dialog-instance service:** The *dialog-instance* service models an SAP dialog Instance profile running on a VM.

Besides these different types of services, there are no other Grounded Execution Services present in the SAP case study. All services go through the same states. For example, a service can be in states “not installed”, “installed”, or “running”, compare Figure 2.3.

[2]an *SAP Instance* is a collection of application server components [26] defined by a profile.

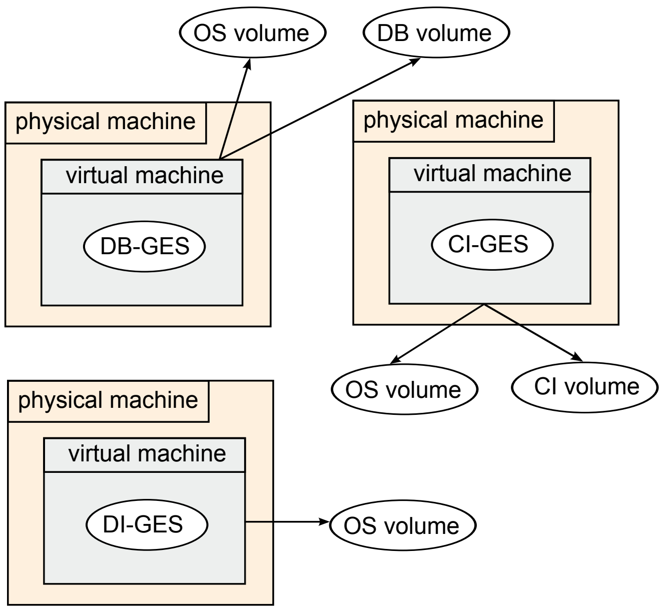


Figure 2.2.: A *decentralized SAP* system

If we extend the scope beyond the SAP case study, there might be other services with a different lifecycle. For example, services that can be paused and resumed. This would add new transitions from and to a new “paused” state. A planning approach needs to provide enough flexibility to describe this behavior in a simple and adaptable way.

Virtual machines:

The Grounded Execution Services described by the SAP case study are executed in virtual machines. The virtual machines can be based on different virtualization techniques like *XEN* [48] or *VMware* [47]. A planning approach needs to take this into account providing a mechanism to easily define different behavior behind the same domain concepts. Depending on which virtual machines the services are executed, we can speak of two different SAP deployment configurations:

- **centralized system:** In a centralized SAP system the database and the central-instance services are running on the same virtual machine. The dialog-instances are running on different dedicated virtual machines, all distinct from the database / central-instance machine.
- **decentralized system:** In a decentralized SAP system the database and the central-instance are running on dedicated virtual machines. Each dialog-instance has its own virtual machine, distinct from the database and the central-instance machine. Figure 2.2 shows a decentralized system with one dialog-instance.

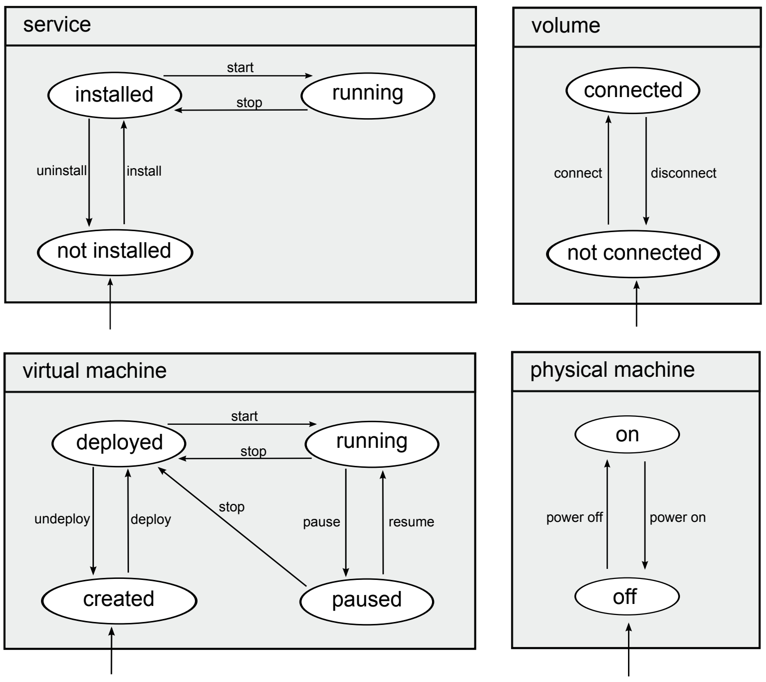


Figure 2.3.: State-transition systems for virtual machines, physical machines, services, and volumes

SAP systems have a variable amount of dialog-instances. If the user demand increases, new virtual machines hosting dialog-instances can be added.
Figure 2.3 shows the state-transition system of virtual machines. It consists of four states. The initial state is the “created” state meaning that a virtual machine exists but is not yet deployed. The “deploy” transition deploys a virtual machine thus leading to the “deployed” state. The virtual machine is not yet running in this state. From the “deployed” state the machine can be either started (“start” transition) or undeployed by executing the “undeploy” transition. If the virtual machine is started, it is in state “running”. From there the “pause” transition can pause the virtual machine. It can be resumed from this state. The machine can be stopped if it is in the “running” or “paused” state. This is only one possible description of a virtual machine. There might be other kinds of virtual machines with different states and transitions.

Physical machines:

Virtual machines are running on physical machines. There can be more than one virtual machine running on a physical machine. Figure 2.2 shows a decentralized SAP system where each virtual machine is running on a dedicated physical machine. Physical machines do not necessarily need to have the same characteristics, for example, they can have different memory capacities.

The transition system of a physical machine consists of two states, “on” and “off”. As it can be seen in Figure 2.3, the transitions “power on” and “power off” switch between these two states.

Volumes:

Volumes describe disc images that can be mounted within a virtual machine. The *SLiM* project distinguishes between the following types of volumes:

- **DB volume:** The *DB volume* contains the database.
- **OS volume:** The *OS volume* contains the operating system. Such a volume is connected to each virtual machine.
- **CI volume:** The *CI volume* contains central-instance specific data. A *CI volume* needs to be connected to a VM hosting a central-instance service.

Figure 2.3 shows the transition system for volumes. A volume can be either “not connected” or “connected”. The transitions “connect” and “disconnect” switch between the two states.

2.2.2. A general behavioral model for domain objects

In this subsection we summarize important characteristics of the domain objects introduced in Subsection 2.2.1. We extract the similarities behind the behavior of the domain objects in order to get a general model underlying the behavior of domain objects.
Our general model of domain objects is based on the observation from Subsection 2.2.1 that the domain objects can be considered to have a state. Previous work done in the field of IT change planning does not consider this.
Figure 2.3 in Subsection 2.2.1 shows the state-transition systems describing some of the domain objects. For example a service, i. e., a Grounded Execution Service, can be in states “not installed”, “installed”, and “running”. In order to change the state of a service, a transition needs to be executed. For example the “stop” transition leads from the “running” state to the “installed” state. Figure 2.3 in Subsection 2.2.1 shows one state transition-system accounting for all Grounded Execution Services. Thus, all services of the planning domain do have the same lifecycle. The advantage of the hybrid approach proposed in this work is that we only need to specify this state-transition system once and then it accounts for all Grounded Execution Services. Also note that the transitions might be differently implemented for different kinds of services. For example, in a database the “stop” transition might consist of an action to stop the database and one to backup all database tables. In contrast to this, the “stop” transition of a central-instance might only consist of an action to stop the CI. The hybrid approach takes this into account, enabling us to specify different refinement behaviors depending on the domain object a transition system is associated to. Thus, the more general lifecycle behavior is clearly separated from hierarchical refinement to implement transitions in state-transition

systems. In addition to that, the hybrid approach facilitates the reuseability of domain specifications. For example, if a new service is added to the planning domain, the state-transition system can be reused and one only needs to define new hierarchical problem solving behavior to implement the transitions.
Related approaches alloy the behavior of domain objects with hierarchical task refinement strategies. The general behavioral model underlying the hybrid approach is the state-transition system. It separated from the description of hierarchical task refinement.
Be aware that even if domain objects cannot be described as state-transition systems, the hybrid approach still supports the specification of task refinement strategies not related to state-transition systems.

2.3. Dependencies in the SAP case study

The following section gives an overview of the dependencies a planner needs to take into account when planning in the domain of the SAP case study. They are only accounting when the SAP system has been completely deployed on a data center, i. e., in the Deployed state of the model (compare Subsection 2.1.2).
This section assumes that the state-transition systems described in Figure 2.3 in Subsection 2.2.1 are associated with the domain objects in order to describe the dependencies properly. Figure 2.3 shows the state-transition systems associated with services, physical machines, virtual machines, and volumes. A dependency always accounts for a transition in a state-transition system. This means dependencies need to be fulfilled in order to do a transition in the transition system.
First of all, Subsections 2.3.1 to 2.3.3 introduce the dependencies according to the type of service (DB, CI, or DI service). After that, Subsection 2.3.4 explains additional dependencies regarding virtual machines and volumes. Finally, Subsection 2.3.5 extracts similarities from the previously mentioned dependencies to pave the way for a general model of dependencies occurring in the SAP case study. This general model builds the foundation for the notion of dependencies in the hybrid approach.

2.3.1. Dependencies of a database Grounded Execution Service

This subsection explains the dependencies for a database Grounded Execution Service. It introduces the dependencies according to the transitions a database service has. See the state-transition system of a service in Figure 2.3 in Subsection 2.2.1 for these transitions. Table 2.1 shows the transitions of a database service and the dependencies linked to each transition.

Each row of the table describes a transition and at most one dependency holding for it. The first column holds the name of the transition we are looking at. The second column holds the description of a dependency. If it and the following columns are empty, the transition does not have a dependency. The “domain object” column describes the domain object that is affected by the dependency. Dependencies within the SAP case

Table 2.1.: Dependencies of database transitions

transition name	**dependency description**	**domain object**	**state to achieve**
start	—	—	—
stop	stop CI before DB	CI	installed
install	—	—	—
uninstall	uninstall CI before DB	CI	not installed

study demand to bring another domain object into a certain state in order to perform a transition. The "domain" object column describes which domain object we need to bring into a well defined state in order to execute the transition mentioned in the first column. The last column names the state we need to achieve in the domain object affected by the dependency.
As it can be seen in Table 2.1, there are no dependencies for the "start" and "install" transitions of a database service. The database is the lowest tier in the multi-tier architecture of an SAP system. The database depends on nothing to perform its service. Thus, we can start or install it without considering any dependencies. However, the central-instance depends on the database. It can only run if the database is running. This results in dependencies for the "stop" and "uninstall" transitions. The second row shows the dependency for the "stop" transition. In order to stop the database the central-instance needs to be stopped first. The dependency is called "stop CI before DB" and demands that the central-instance (see "domain object" column) is in state "installed". Similarly, if we want to uninstall the database we need to uninstall the CI first because we assume that there are dependencies checked during the uninstallation process. Dependency "uninstall CI before DB" in Table 2.1 describes this dependency.

2.3.2. Dependencies of a central-instance Grounded Execution Service

This subsection describes the dependencies of a central-instance service. Table 2.2 gives an overview of the dependencies based on the transitions a central-instance service has. Remember that Figure 2.3 in Subsection 2.2.1 shows the state-transition system for a service.

Table 2.2.: Dependencies of central-instance transitions

transition name	**dependency name**	**target**	**state of target**
start	start DB before CI	DB	running
stop	stop all DIs before CI	all DIs	installed
install	install DB before CI	DB	installed
uninstall	uninstall all DIs before CI	all DIs	not installed

The central-instance depends on the database to deliver its service. This leads to two dependencies, one for the "start" and one for the "install" transition of the central-instance. We cannot start the central-instance if the database is not running because

the central-instance relies on code stored in the database. Thus, the database needs to be in state “running” in order to do transition “start” of the central-instance. The same principle accounts for the “install” transition. During installation of a central-instance, the existence of a database is checked. Thus, the database needs to be at least in state “installed” (see row three of Table 2.2).
All dialog-instances depend on the central-instance in order to deliver their service. This leads to dependencies for the “stop” and “uninstall” transitions. The “stop all DIs before CI” dependency is shown in the “stop” row of Table 2.2. It says that we need to stop all dialog-instances before we can stop the central-instance because the dialog-instances depend on the service provided by the central-instance. The dependency for the “uninstall” transition demands that all dialog-instances need to be uninstalled before the central-instance can be uninstalled. Thus, we need to bring all dialog-instances into state “not installed”.

2.3.3. Dependencies of a dialog-instance Grounded Execution Service

This subsection examines the dependencies that account for a dialog-instance service. They are shown in Table 2.3.

Table 2.3.: Dependencies of dialog-instance transitions

transition name	**dependency name**	**target**	**state of target**
start	start CI before DI	CI	running
stop	—	—	—
install	install CI before DI	CI	installed
uninstall	—	—	—

The dialog-instance depends on the services delivered by a central-instance. Thus, there are dependencies for the “start” and “install” transition of a DI service. Row one shows the “start CI before DI” dependency. In order to start the dialog-instance, the central-instance needs to be started first, i. e., it needs to be in state “running”. Row three of Table 2.3 shows the similar requirement for the “install” transition. A dialog-instance can only be installed if the central-instance is already installed because the install routine of the DI checks the presence of the CI.
As nothing depends on the dialog-instance, we can stop or uninstall it without considering any dependencies.

2.3.4. Additional dependencies regarding other model components

Besides the dependencies services have to other services, there are other model components whose behavior is massively driven by dependencies. Independent from the kind of a service, a transition within a service (i. e., “install”, “start”, “stop”, or “uninstall”; compare Figure 2.3 in Subsection 2.2.1) can only be done if the VM the service is running on is in state “on”. This models the fact that any changes to software can only be done when the machine is running.

Any transition within a virtual machine (compare Figure 2.3) can only be performed if there is at least an OS volume connected to the virtual machine. This models the invariant that virtual machines cannot work without an OS volume hosting the operating system. Similarly, we have to demand that a database or central-instance volume is connected to the virtual machine in which the state of a database or central-instance service is to be changed.

2.3.5. A general model for dependencies between domain objects

In this subsection we explain how the previously mentioned dependencies can be unified in a general model for dependencies. A dependency is always linked to the execution of a transition. For example, the “start CI before DI” dependency is linked to the “start” transition of a central-instance as shown in Table 2.3 in Subsection 2.3.3. This means, that the dependency “start CI before DI” has to be solved before the “start” transition can be taken. Thus, a dependency always accounts for a transition and references a state to be achieved in another domain object. Assuming that the general behavior model as described in Subsection 2.2.2 accounts, a dependency can be described as a Classical Planning problem. Classical Planning problems [22] can be considered as searching through the state space of a transition system. This is exactly the solution in order to achieve the dependency. In the given example we need to find a sequence of transitions that bring the central-instance to state “running” when executed. All in all, the hybrid approach holds the view that dependencies can be described as Classical Planning problems. This idea separates dependencies, described by Classical Planning problems, from hierarchical problem solving behavior and the description of domain object behavior.
Be aware that even if some domain objects do have the same state-transition systems the dependencies linked to the transition might be different. For example, compare the “start” transition of a service. A database does not have a dependency associated with the “start” transition as it can be seen in Table 2.1 in Subsection 2.3.1. This is different to a dialog-instance which has a dependency associated (see Table 2.3 in Subsection 2.3.3). The hybrid approach takes this into account. Compared to previous work done in the area of IT change planning the hybrid approach offers a more intuitive way to specify dependencies because they are made explicit. In a pure HTN approach like Trastour’s [46] dependencies can only be described as part of a hierarchical decomposition. In the hybrid approach we provide our own concept for dependencies and can thus separate dependencies from the description of hierarchical problem solving behavior.

2.4. Workflows within the SAP case study

This section gives an example of sequential and parallel workflows a planner has to plan for. Planning for sequential and parallel execution is essential for a planner in the context of the SAP case study.
Workflows that are dominated by dependencies as introduced in Section 2.3 are a good

example for totally[3] ordered task decomposition. Given an SAP system with a DB, a CI, and one DI Grounded Execution Service, stopping the database service implies stopping the central-instance and before that the dialog-instance. Due to the imposed temporal constraints there is no parallel decomposition possible.

An example that uses sequential, i. e., totally ordered, as well as parallel decomposition is the high-level change request "setup new SAP system". It can be accomplished through a totally ordered sequence of CRs like "create new virtual machines", "start VMs", "install software on VMs", and "start software on VMs". It would be very inefficient to further decompose the "create new virtual machines" subtask into "create one VM" tasks in sequential order because virtual machines can be created in parallel. In addition to that, a plan which sets up VMs totally ordered would result in a very long execution time because setting up a VM takes some time and each step waits for the previous one to complete. If a software installation routine checks dependencies to other software components during installation, a sequential decomposition is needed in the "install software on VMs" change request. As described in Section 2.3, starting a Grounded Execution Service involves starting all the services one is dependent on first. Thus we have to decompose the last node "start software on VMs" in sequential order. The proposed approach also needs to be able to plan by task refinement in order to capture the best practices of the IT Change Management domain. Although the example provided here suggests either a total order or parallel decomposition of tasks, the hybrid approach supports planning using partial order decomposition. Note that a parallel decomposition can be expressed by a partial order but not every partial order can be expressed by a pure parallel decomposition.

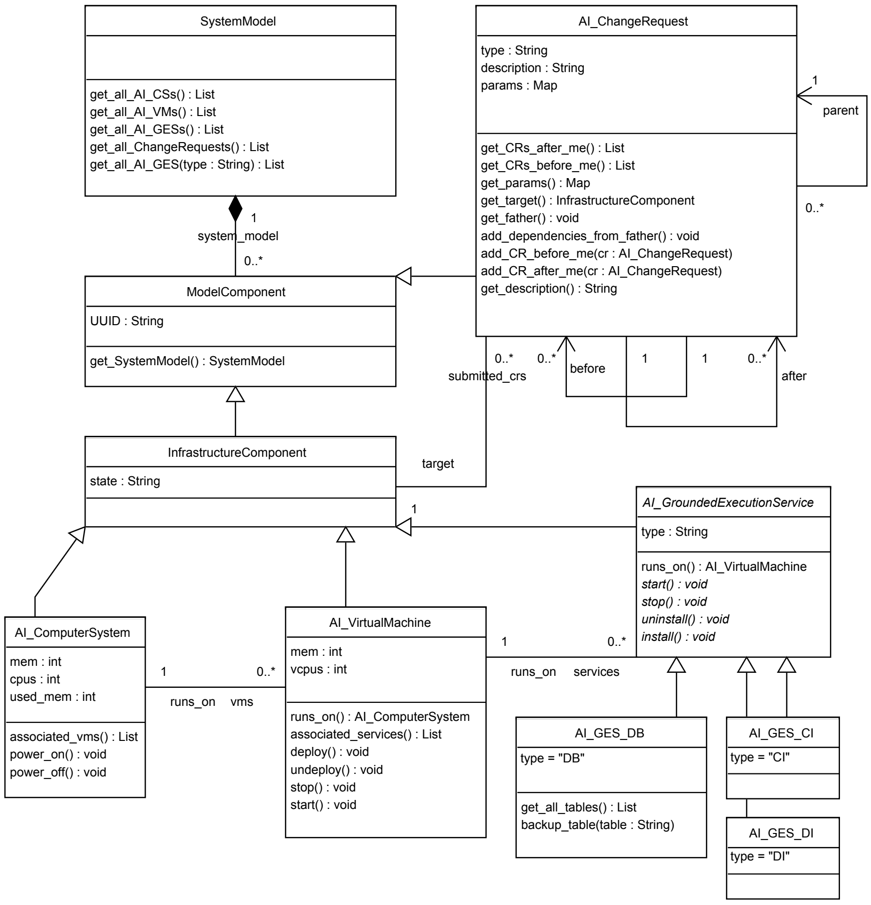
2.5. Model of the planning domain

This subsection explains the object oriented model which describes the components of the infrastructure the planner plans over. In addition to that, it explains the characteristics of a change request modeled by the class "AI_ChangeRequest" and how it relates to the classes describing the infrastructure.

Figure 2.4 shows an UML class diagram of the object oriented model, which is used for planning by the hybrid approach. Be aware that the hybrid approach is not limited to planning over this model. We can easily add new classes to the model as subclasses of the class "InfrastructureComponent". Existing classes can be specialized or new classes can be added to model new domain objects. The strength of the hybrid approach is that it is fully customizable to the model. This leads to great flexibility regarding the specification of state-transition systems linked to classes.
The class "SystemModel" is the anchor of the model. Every class in the model can be reached from "SystemModel". We can receive the "SystemModel" from every subclass of "ModelComponent" by calling the "get_SystemModel()" method. A "SystemModel" is an aggregation of "ModelComponents". Every other class in the model is a subclass or indi-

[3] a total or linear order $<$ is a binary relation which is antisymmetric (i. e., $\forall a \forall b : a < b \wedge b < a \rightarrow a = b$), transitive (i. e., $\forall a \forall b \forall c : a < b \wedge b < c \rightarrow a < c$) and total (i. e., $\forall a \forall b : a < b \vee b < a$))

Figure 2.4.: An UML class diagram describing the planning domain and the characteristics of the class “AI_ChangeRequest”



rect subclass of “ModelComponent”. The class “InfrastructureComponent” adds a “state” attribute and an unique identifier to a “ModelComponent”. It models elements of the planning domain that have a state. Every object part of the infrastructure is a subclass of “InfrastructureComponent”. We call these classes / objects “domain objects” because they describe the planning domain. Currently, there is “AI_ComputerSystem” modeling a physical machine. On it “AI_VirtualMachines” are running that are itself running “AI_GroundedExecutionServices”. The abstract class “AI_GroundedExecutionService” is specialized by a database (class “AI_GES_DB”), a central-instance (“AI_GES_CI”), and a dialog-instance (“AI_GES_DI”). We can navigate the domain objects using the named associations shown in Figure 2.4.

The class “AI_ChangeRequest” is of particular interest because it models a change request and its relationships to the domain objects. It has the following attributes and references:

- **description:** A string describing the change request.
- **type:** A string describing the type of the change request. The hybrid approach comprises different types of change requests that are described by the value of the "type" attribute.
- **params:** A Groovy map holding the parameters of the change request. The key describes the name of a parameter and the value the value of the parameter.
- **target:** A reference describing an instance of class "InfrastructureComponent", a domain object. It points at the domain object the change request relates to.
- **after:** A set of references to "AI_ChangeRequests" describing the change requests that need to be executed after the completion of the change request.
- **before:** A list of references to class "AI_ChangeRequest" describing the change requests that need to be completed before the change request can be executed.

By describing the change requests that need to happen before or after a change request we describe a partial order over all change requests. The formalization of dependencies using "after" and "before" references is redundant. If two change requests $cr1$ and $cr2$ need to happen in sequence, then $cr2$ is contained in the after list of $cr1$ and $cr1$ is contained in the before list of $cr2$. For convenience we assume this redundancy. From a graph theoretic point of view it is sufficient to either look at the "before" or "after" relation. We name the relation describing the after references the *happens_before* relation, i. e., for two change requests cr_1 and cr_2 $(cr1, cr2) \in \mathit{happens\_before}$ accounts if $cr2$ is contained in $cr1$'s after list. Change requests that are not part of the transitive closure of the *happens_before* relation can be executed in parallel.
Be aware that for a change request neither the "after" nor the "before" change requests need to be specified. In this case the change request can be executed in parallel to all other change requests.

2.6. Requirements for a planning approach

The following section collects requirements for the planner that are implied by the introduction of the SAP case study and best practices common in AI Planning. Especially the dependencies explained in Section 2.3, the workflows introduced in Section 2.4, and the Model Information Flow impose a set of requirements the planner needs to fulfill. The developed hybrid approach is to be measured according to these requirements.

- **R1 : Parallel and sequential task decomposition**
  Complex tasks as shown in Section 2.4 need to be decomposed into easier change requests. Due to efficiency reasons there needs to be the possibility to decompose tasks either in sequence or in parallel. A planning approach needs to take this into account.

- **R2 : Sound planning algorithm**
  A deterministic[4] planning system is called sound, i. e., it fulfills the soundness property, if, whenever it is invoked on a planning problem P and returns a plan $result\ != failure$ then $result$ is guaranteed to be a solution for the planning problem according to the predefined semantics of a solution [22].

- **R3 : Complete planning algorithm**
  A deterministic planning system is called complete (i. e., it fulfills the completeness property) if, whenever it is invoked on a planning problem with an existing solution the algorithm guarantees to return a plan $result\ != failure$ where $result$ is a semantical solution to the planning problem [22].

- **R4 : Possibility to express dependencies**
  There need to be concepts within the planning algorithm and the domain description to express dependencies as they were explained in Section 2.3. There needs to be a domain concept for dependencies which serves as input to the planner. This concept needs to take into account that dependencies can be different on the level of instances. For example, there can be different dependencies for different kinds of Grounded Execution Services. All dependencies introduced in Section 2.3 are referencing states of other objects' state-transition systems which leads to *R6* demanding a powerful concept for states.

- **R5 : Preserving of dependencies during planning**
  The planning approach needs to be aware of the dependencies and must fulfill these dependencies during planning. Dependencies are not allowed to be violated. This requirement is closely linked to the soundness property (*R2*) because the violation of *R5* leads to a violation of *R2*.

- **R6 : Notion of state**
  A planner needs to be aware of the states the domain objects are in. Furthermore, it needs to know which change requests need to be done in order to get to another state. The behavior of a domain object strongly depends on the state it is in. As dependencies are referencing the state of other domain objects, it is wise to have a good and powerful notion of state incorporated into the planning algorithm.

- **R7 : Planning changes according to the Model Information Flow**
  The Model Information Flow is an important part of the SLiM project. Thus, the planning algorithm needs to be able to plan for changes of the MIF. A typical change request in this context would be the “change state of model to 'Deployed' ” change request. The fact that the MIF heavily makes use of the notion of states enforces requirement *R6*.

- **R8 : Possibility to describe workflows**
  Workflows in the context of Change Management are describing best practice steps

[4]A planning system is called deterministic if for any given pair of state and action there is at most one state successor defined by the state transition underlying the planning problem.

in order to achieve a certain goal. For example, updating a software component includes stopping, updating, and starting the software again. A planning approach within the SLiM project should provide means to define these best practice steps to generate plans adhering to them.

- **R9 : Task refinement capabilities**
  The planner needs to be able to plan the refinement of tasks because this is central to the domain of IT change request planning. Because solving a dependency can be part of a task refinement process, the algorithm needs to be able to solve a planning problem over state-transition systems as part of a task refinement process.

- **R10 : Reuse of behavior and dependency specification**
  If there are components with a similar behavior in the planning domain, then there should only be one specification of this behavior. Components with similar behavior need to be added easily to the domain description. If a dependency is the same for a set of components, then these dependency should only be specified once and reused.

Besides soundness (requirement *R2*) and completeness (requirement *R3*), there is a third criteria, the *admissibility* criteria, introduced in [22]. An *admissible* planning algorithm returns an optimal solution whenever a planning problem P is solvable. Admissible planning algorithms need a measure to evaluate the optimality of a solution. This work focuses on plan generation that is heavily driven by dependencies and by best practices reducing the room for optimization heavily. Finding optimal plans within the small set of plans satisfying the constraints, makes it necessary to introduce a function to assess the changes to infrastructures. We plan to address this topic in a future work.

3. Evaluation of existing planning approaches

Several planning approaches, solving different kinds of planning problems, do already exist. Two of them are promising to solve the problems imposed by the SAP case study. First of all, the two approaches are introduced in Sections 3.1 and 3.2. Having introduced the approaches, Section 3.3 has a closer look at them under different viewpoints. For each viewpoint we describe the capabilities of the two approaches. In addition to that, we explain the advantages of the hybrid approach regarding every viewpoint. Finally, Section 3.4 summarizes the findings from Section 3.3 to justify the research done in the hybrid approach.

3.1. Classical Planning

This section gives an introduction to Classical Planning, one of the two planning techniques used by the hybrid approach. Subsection 3.1.1 introduces a conceptual model for planning which is then restricted by a set of constraints in Subsection 3.1.2. This restricted conceptual model is the foundation of the Classical Planning approach which is finally introduced in Subsection 3.1.3. This section follows the introduction to Classical Planning given in [22].

3.1.1. A conceptual model for planning

This subsection provides a simple theoretical model in order to describe the main elements and characteristics of a planning problem. Be aware that such a conceptual model does not define computational or semantical aspects of a planning problem. The Classical Planning approach introduced in Subsection 3.1.3 is based on this conceptual model.
According to [22], a planning problem can be described by a *state-transition system* (sometimes also called *discrete-event system*) which formally is a 4-tupel $\sum = (S, A, E, \gamma)$ where:

- $S = \{s_1, s_2, ...\}$ is a finite or recursively enumerable set of states.
- $A = \{a_1, a_2, ...\}$ is a finite or recursively enumerable set of actions.
- $E = \{e_1, e_2, ...\}$ is a finite or recursively enumerable set of events.
- γ is a state-transition function with signature $\gamma : S \times A \times E \to 2^S$, where 2^S is the powerset of S, i. e., $2^S = \{S' \mid S' \subseteq S\}$.

The transition function γ depends on the current state, an action, and an event. While actions are transitions controlled by the planner, events cannot be controlled by the planner. They may change the state of the transition system without the occurrence of an action. It is convenient to introduce an neutral action *noop* in order to model transitions that are only influenced by events. Furthermore, a neutral event ϵ to model transitions only influenced by actions is necessary. The following abbreviations are holding: $\gamma(s, noop, e) \equiv \gamma(s, e)$ and $\gamma(s, a, \epsilon) \equiv \gamma(s, a)$. Be aware that $\equiv$ is used to define an abbreviation while $=$ is used to define an equivalence. An action a is called applicable to a state s if $\gamma(s, a) \equiv \gamma(s, a, \epsilon) \neq \emptyset$. If e is an event and $\gamma(s, e) \neq \emptyset$, then e can occur in state s. That means the state-transition system can change its state to another state $s' \in \gamma(s, e)$ on occurrence of event e. There can be different semantics regarding the state-transition function γ. For example, γ could be defined to lead to a state in $\gamma(s, a)$ or $\gamma(s, e)$ if $\gamma(s, a, e)$ is invoked, i. e., actions and events are competing for execution. Given a state-transition system $\sum$ and a start state s_{init}, the purpose of planning is to find a set of actions to apply to $\sum$ in order to achieve a goal. A goal can be specified

- explicitly through a goal state $s_g \in S$ or a set of goals $S_g \subseteq S$. In this case the planner searches for a sequences of actions that lead to a goal state. Classical Planning, which is introduced in Subsection 3.1.3, uses this kind of goal specification.
- through conditions that have to account for a sequence of states of a plan. The goal of the planner is not to reach a specified state but to find a plan that satisfies constraints formulated over states. For example, we could demand the planner to construct a plan that excludes some previously defined states.
- through a task. The objective of the planner is to achieve a plan for a given task. HTN Planning represents this kind of objective specification. It is used in the hybrid approach and is introduced in Subsection 3.2.

Figure 3.1 shows a high-level architecture of a very simple planning system. The planner needs a description of a real world planning domain which is formalized as a state-transition system $\sum$. Further inputs to the planner are an initial state $s_{init} \in S$ and objectives describing the goals of the planner. The planner generates a plan which is given to a controller. The controller executes the plan on the system in order to change it. Changes to the system lead to new observations which are recognized by the controller. These observations are not passed back to the planner. This means that the planner does the planning once and does not change the plan according to observations made by the controller (*offline planning*).

3.1.2. A restricted model for planning

This subsection introduces eight assumptions that define a restricted model for planning based on the conceptual model explained in the previous subsection. Classical Planning, which is introduced in Subsection 3.1.3, bases on these assumptions. We explain why some of these assumptions hold in the planning domain. For others, we explain the

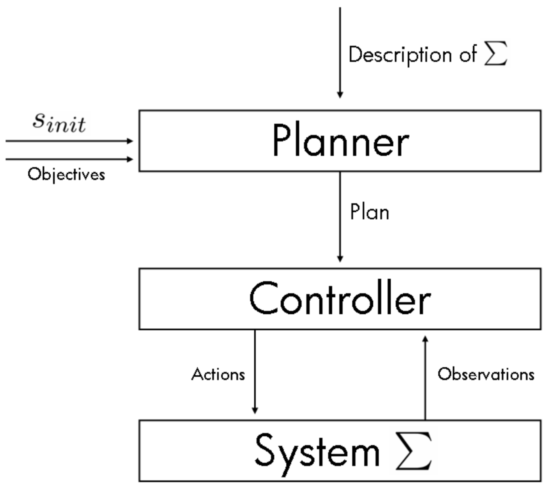


Figure 3.1.: A conceptual model for *offline planning* [22]

additional restrictions that need to be imposed. The constraints are important because Classical Planning is based on the restrictions and it can only be used in the context of the SAP case study if the restrictions account.

- **A1 - finite $\sum$:** A state-transition system $\sum = (S, A, E, \gamma)$ is called *finite* if S is finite. This accounts for the SAP case study because every state-transition system shown in Figure 2.3 in Subsection 2.2.1 has a finite amount of states. Thus, looking at a single state-transition system fulfills assumption *A1*.

- **A2 - fully observable $\sum$:** A state-transition system $\sum$ is called *fully observable* if the planner has complete knowledge about the state of the system. This accounts for the SAP case study because all used state-transition systems are fully observable.

- **A3 - deterministic $\sum$:** A state-transition system $\sum$ is called *deterministic* iff (if and only if) $\forall u \in A \cup E \quad \forall s \in S : |\gamma(s, u)| \leq 1$, i. e., for a given state and a given action or event there is maximal one successor. This also accounts for the SAP case study domain if transitions are treated as actions in the state-transition systems in Figure 2.3 in Subsection 2.2.1.

- **A4 - static $\sum$:** A planning system $\sum = (S, A, E, \gamma)$ is called *static* if there are no external events ($E = \emptyset$). This means the environment only changes its state due to actions performed by the planner. We assume that this accounts for the SAP case study, too. Transitions in Figure 2.3 in Subsection 2.2.1 model actions. There are no events and no other actions than the modeled ones.

- **A5 - restricted:** A planning system has *restricted goals* if a goal is defined explicitly as a goal state $s_g \in S$ or a goal set $S_g \subseteq S$. Allowing only change requests of type "set state of state-transition system to s", where $s \in S$, issued towards a single state-transition system as shown in Figure 2.3 in Subsection 2.2.1, results in a planning system with restricted goals. The Classical Planning part of the hybrid approach complies to this.

- **A6 - sequential:** A solution to the planning problem is a totally ordered sequence of actions. If we only focus on the lifecycle of objects, i. e., on the state-transition systems described in Figure 2.3 in Subsection 2.2.1, and on change requests of type "set state of state-transition system to s" ($s \in S$), then this restriction also accounts for the SAP case study.
- **A7 - implicit time:** The actions of a planning system do not have a duration. In the SAP case study we do not reason about the duration or length of actions. Instead, we are aiming at a binary partial order that describes after which actions an action can be executed. Thus, assumption *A7* accounts in the SAP case study.
- **A8 - offline:** In an *offline* planning problem the planner is not concerned with changes to the environment that might occur during the planning phase. We assume that only the actions executed by the controller change the state of the infrastructure. Thus, the SAP case study is conducted under assumption *A8*.

3.1.3. Classical Planning

Classical Planning is also called *STRIPS* Planning due to the name of a planner for *restricted state-transition systems* [20]. A *restricted state-transition system* $\sum$ is a state-transition system which adheres to restrictions $A1$ to $A8$ introduced in Subsection 3.1.2. Such a system $\sum$ does not have any events[1], thus $\sum = (S, A, \gamma)$ holds. If γ is applicable to s, then there is only one successor[2] state $s' \in S$, thus $\{s'\} = \gamma(s, a)$. We also write this as $s' = \gamma(s, a)$.
A *planning problem* P for a restricted state-transition system $\sum = (S, A, \gamma)$ is a triple $P = (\sum, s_{init}, G)$ where $s_{init} \in S$ is the initial state and $G \subseteq S$ a set of goal states. A solution in P is a sequence of actions $(a_1, ..., a_n)$, $n \in \mathbb{N}_0$, such that $\gamma(...\gamma(\gamma(s_{init}, a_1), a_2)..., a_n) \in G$. The expression Classical Planning generically refers to planning for restricted state-transition systems, i. e., searching through the state space of a restricted state-transition system. The restrictions as described in Subsection 3.1.2 are important in the context of this work. A hybrid approach needs to adhere to these restrictions when it comes to solve a Classical Planning problem as part of a hybrid planning problem. Furthermore, it is a characteristic of Classical Planning that goals are explicitly defined by specifying a goal state or a goal set. This is different to an HTN approach where the decomposition of a task is the goal.

[1] $E = \emptyset$ due to *A4*
[2] assumption *A3*

3.2. Hierarchical Task Network Planning

Hierarchical Task Network (*HTN*) planners differ from Classical planners (see Subsection 3.1.3) in "what they plan for and how they plan for it" ([22], p. 229). They do plan for a decomposition structure of a high-level task into subtasks, called hierarchical task network, in order to accomplish the high-level task. They do not plan for it by searching for a sequence of actions, but by searching for a task decomposition that achieves a provided high-level task. In HTN a plan is a decomposition of a high-level task, e. g., setting up an SAP system, into smaller and smaller subtasks until primitive tasks are reached. Primitive tasks cannot be decomposed further.
The basic idea behind HTN was developed in the 70s ([43], [44]). The approach was further refined into a set of available domain independent HTN planners. One of the most sophisticated ones is the *SHOP2* planner ([36], [39], [37], and [38]). HTN has been widely applied to practical planning problems with great success, for example, Mars exploration [17], workflow planning in Grid Computing [23], crisis intervention [19], control of deep space antennas [8], and recently IT change request planning [46].
This section gives a non-formal high-level introduction to HTN Planning. It bases on a conceptual model for HTN Planning introduced in [22] and some experiences made with SHOP2. First of all, the important terms in the context of HTN are introduced and explained. After that, an Hierarchical Task Network planning example from the SAP case study is given in order to provide a better understanding of the introduced terms.

A conceptual model for HTN Planning

Besides HTN Planning there is a simpler form, called *Simple Task Network* (*STN*) Planning [22] which only allows a restricted set of constraints being used during task decomposition. This frees the planner from additional complexity. For the scope of this work a partially ordered STN decomposition is sufficient. As Hierarchical Task Networks are more expressive than Simple Task Networks, everything mentioned in this section also applies for HTN. However, the differences between HTN and STN in the context of this work are marginal so that Simple Task Network Planning can be considered as Hierarchical Task Network Planning. Other publications than [22] might not even deal with this fine grained differentiation. For an explanation of the minor differences of the two approaches see [22].
An Hierarchical Task Network planner is basically defined by the following terms:

- **task:** A *task* can either be a *primitive* or a *non-primitive task*. A *non-primitive task* can be decomposed into subtasks by applying an HTN method to it. The capabilities of the HTN algorithm define whether the subtasks need to be solved in sequence, i. e., in total order or in partial order. The latter one enables the specification of subtasks that can be executed in parallel. Primitive tasks are the leaf nodes of a decomposition tree, the *hierarchical task network*. Primitive tasks can change the state of the world, e. g., by deleting or adding predicates to the knowledge base of the planner. Tasks are identified by a unique name and they have variables that enable to customize the behavior.

- **method:** *Methods* decompose non-primitive tasks into a combination of primitive and non-primitive tasks. A conceptual model for HTN methods consist of the following four parts [22] :
  - **name:** A unique name for the method, including a set of parameters that customize the method.
  - **task:** The non-primitive task which is decomposed by the method.
  - **precondition:** A precondition defined over the knowledge base and the parameters of the method. It needs to be true in order to apply the *method* to the specified task.
  - **an hierarchical task network:** An hierarchical task network describes a set of tasks and an order imposed on these tasks. The tasks can be either totally or partially ordered, depending on the capabilities of the HTN algorithm.
- **operator:** Primitive tasks cannot be decomposed further and are mapped to *operators. Operators* can change the state of the world by modifying the knowledge base. If no operator can be found for a primitive task, then this is regarded as a fault to achieve a successful decomposition.
- **planning domain:** An HTN *planning domain* is a tuple $D = (O, M)$ where O is a set of operators and M a set of methods. D is called a *partially-ordered planning domain* if every $m \in M$ is partially ordered, i. e., the hierarchical task network belonging to m. The hybrid approach uses partially ordered planning domains.
- **planning problem:** A *planning problem* describes the inputs to an HTN planner. An HTN *planning problem* is a 4-tuple $P = (s_{init}, w, O, M)$, where $s_{init} \in S$ is the initial state, w is a task network called the initial task network describing the tasks to solve, and $D = (O, M)$ is a planning domain.
- **plan:** In Simple Task Network or Hierarchical Task Network Planning approaches a *plan* is a task network that solves the initial task network of a planning problem. Readers interested in a more precise semantics of a *plan* are encouraged to have a look at [22] or [15].

An HTN example

To get a better understanding of Hierarchical Task Network Planning, Figure 3.2 shows a task decomposition of the high-level task “install_software”. This high-level task has one parameter, the ID of the SAP system on which to install the software. As mentioned in Section 2.3, there are dependencies that need to be taken into account when installing the software on an SAP system. The decomposition of a task into subtasks needs to take care of not violating these constraints. A method decomposes the non-primitive high-level task “install_software” into the three non-primitive subtasks “install_db-software”, “install_ci-software”, and “install_di-software” in total order (compare Figure 3.2). Decompositions in total order are visualized through a curved arrow. Each of the first two

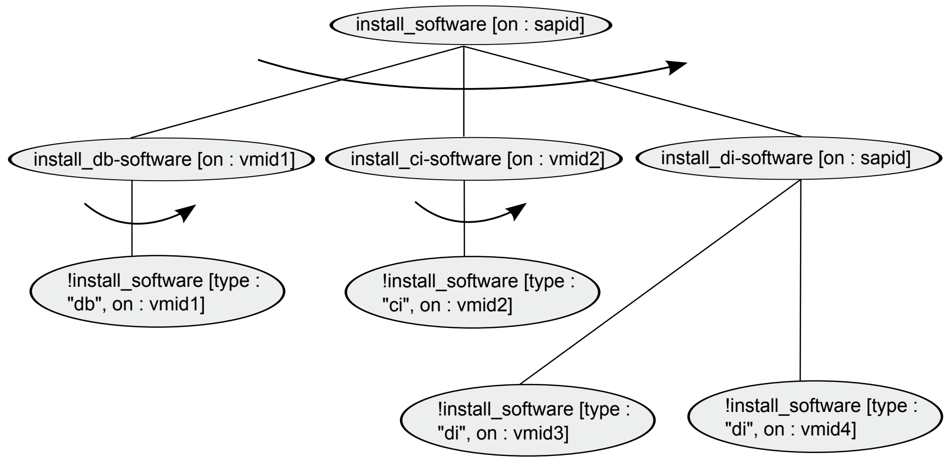

install_software [on : sapid]
install_db-software [on : vmid1]
install_ci-software [on : vmid2]
install_di-software [on : sapid]
!install_software [type : "db", on : vmid1]
!install_software [type : "ci", on : vmid2]
!install_software [type : "di", on : vmid3]
!install_software [type : "di", on : vmid4]

Figure 3.2.: An hierarchical task network for decomposing the task “install_software” in the context of an SAP system.

children of the “install_software” task has one parameter, the id of the virtual machine to install the Grounded Execution Service on. The third task has the id of the SAP system as an attribute because there might be more than one dialog-instance in the system. The language used to describe a method needs to provide computational functionalities interpreted by the planner in order to find the ids of the virtual machines.
Another method decomposes the “install_db-software” task into a sequence of one subtask, the “!install_software” task. This is a primitive-task which is mapped to an operator as shown by the exclamation mark in Figure 3.2 (SHOP2 notation). It cannot be decomposed any further. It changes the state of the knowledge base such that a database Grounded Execution Service is installed on the specified virtual machine. The “install_ci-software” decomposition is similar to the decomposition of the “install_db-software” task. Be aware that in both the more general operator “!install_software”, which takes a software type and a virtual machine id as input parameters, is used. The reuse of generic operators to build high-level change requests is characteristic for the HTN approach. The decomposition of the third task “install_di-software” is done in parallel because the software installed on dialog-instances is independent from each other. This can be seen by examining the dependency tables in Section 2.3.
If there is more than one method with a matching precondition, then the decomposition becomes indeterministic. If a decision does not lead to a successful decomposition, the planner backtracks [12] and uses another method whose precondition is satisfied to do the decomposition. The plan for the “install_software” task produced by the HTN planner is the hierarchical task network as shown in Figure 3.2. More precisely the plan consists of the operators located in the leaf nodes of the decomposition tree and a partial order imposing an execution order on the tasks. This order is defined by the sequential or parallel decompositions the operators are part of in the hierarchical task network. Be aware that the hierarchical task network shown in Figure 3.2 is the result of a planning task

issued in a knowledge base in which database, central-instance, and dialog-instance were not installed when planning was started. Figure 3.2 would show a different task network if the domain objects were in different states. We also would have to define different methods then. During plan execution only the leaf nodes are executed according to their global temporal order.

3.3. Limitations of the planning approaches

This section explains why neither a pure Classical nor a pure HTN Planning approach cope well with the characteristics inherent to the SAP case study. Depending on the viewpoint, there is always one of the two approaches which has difficulties satisfying the demands. Every viewpoint describes an aspect that is important for planning in the SAP case study. In addition to that, we explain for every viewpoint how the hybrid approach overcomes these difficulties and emphasize its advantages compared to previous work done in the area of IT change request planning.
We start by examining the capabilities of the three algorithms when planning in domains where domain objects do have a state. After that, we explain the problems that occur when changing the lifecycle of domain objects in Subsection 3.3.2. Furthermore, additional limitations when describing workflows and dependencies are explained in Subsections 3.3.3 and 3.3.4. Finally, we examine the different capabilities of Classical and HTN Planning when it comes to describe hierarchical planning domains in Subsection 3.3.5.

3.3.1. Planning over states of objects

This subsection examines how HTN and Classical Planning cope with planning in planning domains where domain objects do have a state. We also explain why the hybrid approach is well suited for planning over the states of domain objects and why previous work done on IT change request planning is not.

Characteristics of HTN and Classical Planning

Classical Planning is superior to Hierarchical Task Network Planning when it comes to plan over the states of domain objects. The Model Information Flow has states, Grounded Execution Services have states, and lots of other components, too (compare Chapter 2). An HTN approach does not make these states explicit, instead, the states are implicitly coded into the HTN methods. A set of methods are introduced with different preconditions in order to decompose the same high-level task. The preconditions evaluate to true depending on the state the object is in.
For example, consider the task "start_db-software" shown in Figure 3.3. Leaving dependencies aside for a while the decomposition only depends on the state of the database service (see Figure 2.3 in Subsection 2.2.1 for the states of a database service).

Figure 3.3 shows two decompositions of the "start_db-software" change request being the result of two different methods. The preconditions of the methods only depend on

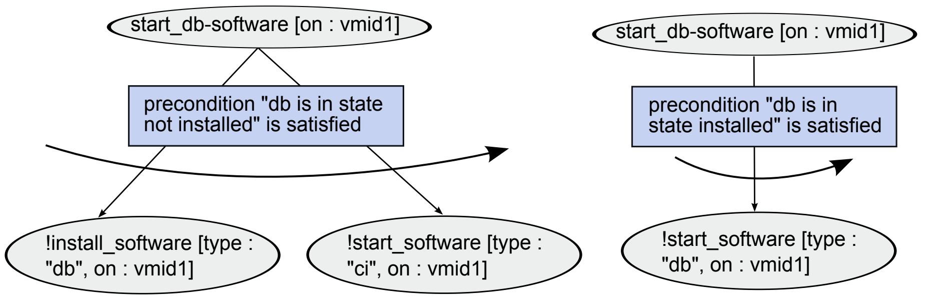

start_db-software [on : vmid1]
precondition "db is in state not installed" is satisfied
!install_software [type : "db", on : vmid1]
!start_software [type : "ci", on : vmid1]
start_db-software [on : vmid1]
precondition "db is in state installed" is satisfied
!start_software [type : "db", on : vmid1]

Figure 3.3.: HTN Planning considering states

the state of the database service. Compare the preconditions shown above the lines linking the children to their parent node in Figure 3.3. The method decomposing the left "start_db-software" task in Figure 3.3 can only be applied if the database is in state "not installed". The method decomposes the the task into a sequence of two primitive tasks. These atomic tasks install and start the service according to the two hop path in the state-transition system of a service in Figure 2.3 in Subsection 2.2.1. The second method can only be applied to the "start_db-software" task if the database is in state "installed", i. e., it is currently stopped. In this case we only need to start the database. All in all, we write an HTN method for every state a database can be in because the path to the goal state is different when we demand to start a database.
Be aware that the HTN approach does not make use of the state-transition system defined in a data structure. But the designer of the HTN methods inevitably has the states of a database in mind and designs the HTN decompositions according to the states through which a state-transition system can go. For each state $s_1, s_2, ..., s_n \in S$ that a domain object can be in and for each goal state $g_1, g_2, ..., g_n \in S$ of the state-transition system, we need to define a method with a suitable precondition in order to define the actions to execute in order to get from start to goal. The introduced example shows how HTN methods with different preconditions are used to model paths within a state-transition system. Such a solution makes the description of the behavior, the state-transition system, implicit. This results in drawbacks regarding maintainability and readability.
In comparison to HTN, Classical Planning offers an explicit notion of state. This becomes advantageous when planning over the states of objects. States do not explicitly need to be coded into the HTN methods.

Advantages of the hybrid approach

Compared to pure HTN approaches like [46] the hybrid approach makes the behavior of domain objects explicit by using state-transition systems. This behavior is not alloyed with hierarchical refinement of tasks as it is done in an HTN approach. This results in the fact, that someone who is confronted with a planning domain of the hybrid approach can easily see what the behavior of domain objects is and which methods do actually describe the refinement of tasks. By making the behavior of domain objects explicit it is easier to adapt this behavior. For example, given an HTN domain description as used by [46] it is difficult to adapt this description when the lifecycle of a domain object changes. This is caused by the fact that we cannot clearly identify the behavior of the domain objects in the HTN methods because these methods contain behavior, hierarchical problem refinement, and dependencies. The hybrid approach encourages the domain writer to reuse or to adapt the behavioral descriptions of domain objects because the behavior is clearly stated and identifiable. All in all, the hybrid approach has an explicit notion of states of domain objects but still supports hierarchical problem solving strategies.

3.3.2. Changes to the lifecycle of domain objects

This subsection examines the capabilities of HTN, Classical, and Hybrid Planning under the aspect of changing the behavior of domain objects. First HTN and Classical Planning are examined under this aspect. After that, we explain how the hybrid approach simplifies changes to the lifecycle of domain objects.

Characteristics of HTN and Classical Planning

An advantage of Classical Planning is that it can cope very well with changes to the lifecycle of a domain object. The changes can be directly propagated to the state-transition system. A domain description of a restricted state-transition system can be easily adapted to add new states like, for example, a "paused" state for virtual machines. Accomplishing these changes in an HTN domain is more difficult because we need to identify the relevant methods that implicitly code the changes to the state of a domain object (compare the example given in Subsection 3.3.1). Furthermore, it is difficult to foresee the effects of these changes as HTN methods are written with a problem solving intention that might be destroyed by the modifications. Methods are coding problem search behavior plus behavior of domain objects. Instead, state-transition systems are only modeling the behavior of domain elements. This makes changes to the lifecycle of domain object easier in Classical Planning.

Advantages of the hybrid approach

In the hybrid approach the lifecycle of a domain object is stored in a state-transition system. Dependencies are linked to a transition in the state-transition system. Hierarchical refinement rules are specified as HTN methods. Thus, the lifecycle of a domain object can be clearly identified. By making it explicit, it is easy to adapt, change, or copy the lifecycle. Compared to other approaches [46] the knowledge base designer does not need to worry that parts of the behavior description are overseen. For example, in an HTN approach the behavior of a domain object can be distributed over many HTN methods like the ones shown in Figure 3.3 in Subsection 3.3.1. Other approaches like CHAMPS [31] and Cordeiro's approach [10] do not offer the possibility to describe the behavior of domain objects by state-transition systems. They do not make the behavior of domain objects explicit.

3.3.3. Description of workflows

In this subsection we examine the three planning approaches based on their capabilities to describe workflows, i. e., best practices in the area of IT Change Management.

Characteristics of HTN and Classical Planning

The HTN approach is very well suited to describe workflows in an intuitive manner. For example, the workflow “update software” might consist of the steps “stop software”, “execute update”, and “restart software”. Writing an HTN method which has the high-level task “update software” and the mentioned three children is close to the workflow idea. Workflows cannot be specified as easily in Classical Planning. We would have to introduce a new state (or even a new state-transition system) called “software updated” and a three hop path from the current state to this state with the actions “stop software”, “execute update”, and “restart software” needs to exist. We can then describe the “update software” workflow as the change request “set state to ’software updated’ ” which computes the path consisting of the three previously mentioned actions. This solution mixes lifecycle specification from the original state-transition system with workflow behavior. Describing the steps of a workflow is something totally different from describing the steps a domain object can go through. Various problems arise when it comes to integrate this workflow behavior into a state-transition system describing the lifecycle. Thus, Classical Planning is not suitable to describe workflow behavior.

Advantages of the hybrid approach

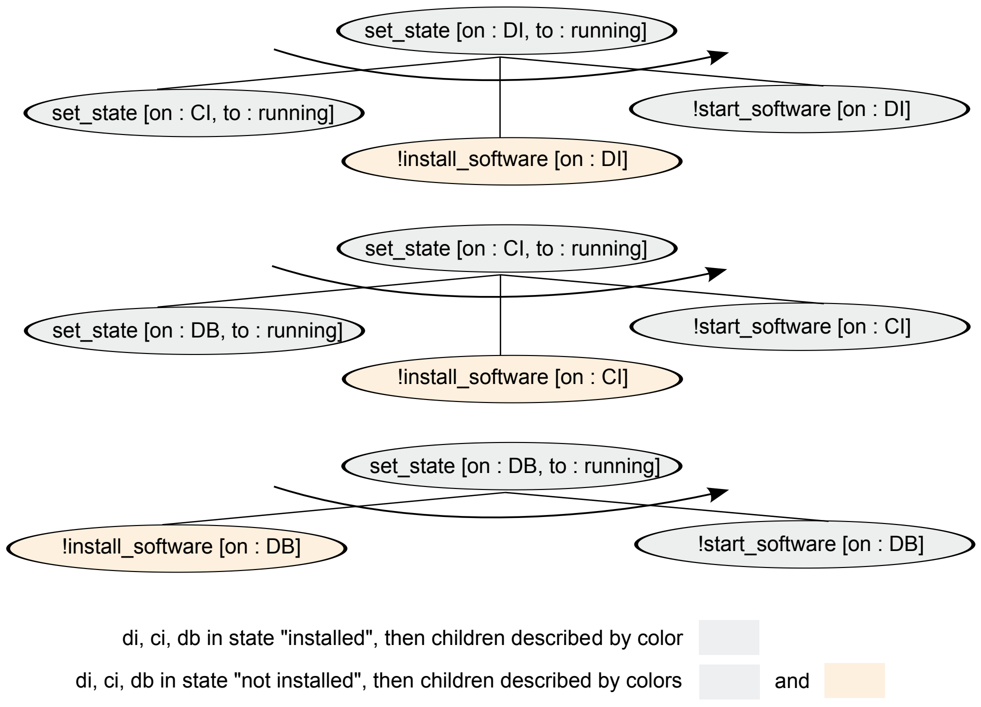
The hybrid approach offers all capabilities known from HTN planners. Thus, the hybrid approach can specify best practices and workflows by HTN methods. This is separated from the specification of the behavior of domain objects. The hybrid approach enables us to specify the behavior of domain objects by state-transition systems which is more natural than coding the elementary behavior of domain objects into HTN methods as done by other approaches.

3.3.4. Description of dependencies

This subsection examines HTN and Classical Planning from the viewpoint of dependencies. Planning according to dependencies in the hybrid approach is crucial to produce sound plans. First, we explain how HTN and Classical Planning cope with describing dependencies. After that, we highlight the advantages when it comes to specify dependencies in the hybrid approach.

Characteristics of HTN and Classical Planning

Specifying HTN methods in a domain including dependencies inevitably mixes dependency change requests, i. e., change requests that need to be introduced in the decomposition in order to take care of dependencies, and object lifecycle specific change requests.



set_state [on : DI, to : running]
set_state [on : CI, to : running]
!start_software [on : DI]
!install_software [on : DI]
set_state [on : CI, to : running]
set_state [on : DB, to : running]
!start_software [on : CI]
!install_software [on : CI]
set_state [on : DB, to : running]
!install_software [on : DB]
!start_software [on : DB]
di, ci, db in state "installed", then children described by color
di, ci, db in state "not installed", then children described by colors
and

Figure 3.4.: HTN Planning taking dependencies and states of domain objects into account

An example for this is the task to start a dialog-instance. We assume that the dependencies described in Tables 2.2 and 2.3 in Section 2.3 are holding. Thus, it is necessary to start the database first. After that, we can start the central-instance and finally the dialog-instance. Besides the occurrence of dependencies the given example is very suitable to see how the notion of states influences the definition of methods. Thus, there is some overlapping with Section 3.3.1. Figure 3.4 implicitly shows nine methods that are necessary to produce a plan which starts a dialog-instance taking the dependencies and the states of other domain objects into account.

There are three decompositions shown. The one at the top is a decomposition for the "set_state [on : DI, to : running]" task. It is a change request to set the state of the dialog-instance to "running". This task can be decomposed in three different ways depending on the state of the dialog-instance. The decomposition only containing the gray children, i. e., the first and third child, describes a method decomposing the high-level task into the two subtasks "set_state [on : CI, to : running]" and "!start_software [on : DI]" if the dialog-instance is in state "installed". Note that the first child needs to be solved because the central-instance needs to be started in order to start the dialog-instance. This is a dependency task, a task that needs to be solved to guarantee a constraint. If the dialog-instance is in state "not installed", a method is used which has an additional "!install_software [on : DI]" task between the two subtasks (task in orange color in Figure 3.4). If the dialog-instance is already in state running, there is no need to

do a further decomposition. This case is not explicitly shown in Figure 3.4, nevertheless a method needs to be defined for this case, too. As it can be seen, the dependency to start the CI before the DI, is modeled by the first subtask which demands the central-instance to be started before anything else is done to the dialog-instance. The same idea accounts for the "set_state [on : CI, to : running]" task which is shown in the second decomposition in Figure 3.4. First of all, the state of the database needs to be set to "running". After that, a set of change requests depending on the state of the CI is issued against the central-instance itself. Again three methods are needed. The last decomposition for the "set_state [on : DB, to : running]" task does not need to check any dependencies. It only issues change request against the database depending on the state of the database. Considering all possible states a DB, DI, and CI service can be in, we need nine different methods in order to achieve a complete HTN decomposition for the "set_state [on : DI, to : running]" task. These nine decompositions can deal with all states a central-instance and database can be in. The amount of methods will increase if we also consider stopping or installing a database.
The interesting question regarding dependencies is whether this decomposition reflects what is expected by the general dependency model introduced in Section 2.3. It fulfills the requirements under special circumstances. It is not enough to only introduce a dependency subtask like "set_state [on : CI, to : running]" because nothing guarantees that the effects of this subtask are still accounting when for example, the "!start_software" operator is executed (compare Figure 3.4, first decomposition). The second subtask "!install_software [on : DI]" could annul the changes of the first dependency subtask. If the precondition of the third task does not check whether the effects of the first dependency change request are still accounting, we are running into problems. Strongly formulated preconditions are needed to achieve the dependency model explained in Subsection 2.3.5.

There are some drawbacks regarding this kind of dependency specification:

- **Mixture of dependencies and states of objects:** In Figure 3.4 there is no separation between child change requests driven by dependencies and the ones that influence the state of the state-transition system. It is difficult to later adapt the methods according to new dependencies or to new states introduced within the lifecycle of domain objects. When using an already existing domain description, it might not be obvious which change requests belong to a dependency.

- **Implicit dependencies:** Dependencies play an important role in the SAP case study. They should be made explicit in a domain description. Compare requirement *R4* in Section 2.6.

- **Strong preconditions:** A lot of effort needs to be put into taking care that preconditions are strong enough. Otherwise changes to the effects of dependency driven change requests can be missed.

Compared to HTN, it is very difficult to describe dependencies in a pure Classical Planning domain. One possibility to describe dependencies in a pure Classical Planning approach is to have a global point of view on the state space. Lets assume the state space is a tuple space in which the first element of the tuple models the state of the database and the second the state of the central-instance. For example, a transition from $(installed, installed)$ to $(installed, running)$ is not possible because the database is not running in the goal state but the central-instance is. A possible solution according to this idea is not considered in this thesis because the state space of a cross product restricted state-transition system would grow very big. Furthermore, a description of dependencies in a tuple space is even less intuitive and more difficult to read than the HTN dependencies.

Advantages of the hybrid approach

Due to the importance of dependencies in the SAP case study, the hybrid approach has its own domain concept for dependencies. Dependencies are always described as Classical Planning problems to be solved. These problems are linked to a transition in a state-transition system. Before the transition can be executed, we need to solve the dependency by solving the Classical Planning problem. Dependencies are thus made explicit in the hybrid approach. They can be directly identified by looking at the state-transition system. Compared to that dependency tasks described in a pure HTN domain like in [46] can be difficult to identify. In addition to that, it becomes more difficult to change the behavior of domain objects because it is difficult to assess the scope of the dependencies. Within the hybrid approach the scope of a dependency is made explicit by associating the dependency to a transition. Another advantage of the hybrid approach is that dependencies are not alloyed with hierarchical task refinement strategies.

3.3.5. Notion of hierarchical decomposition

This subsection examines the three approaches regarding their capabilities to specify refinement strategies for abstract high-level tasks. First of all, HTN and Classical Planning are examined. After that, we explain the advantages of the hybrid approach when it comes to specify task decomposition rules.

Characteristics of HTN and Classical Planning

Classical Planning does not offer the possibility to define the hierarchical decomposition of tasks. Compared to HTN, Classical Planning always plans at the level of the operators, the leaf nodes of the tree. It does not make use of the domain specific, best practice planning strategies that are encoded within HTN methods. It only searches for a sequence of actions to accomplish a goal without having a notion of intermediate goals.

Compared to Classical Planning, HTN methods guide the planner on a path that is likely to lead to a successful decomposition. Classical Planning cannot be sure whether it follows a path that leads to the goal. This is the main aspect that helped HTN to gain wide success in lots of practical planning problems. HTN is very important in the context of the SAP case study because the domain of IT change planning is a hierarchical domain where task refinement is essential. Furthermore, the concept of a plan in the SAP case study is not a sequence of actions but an hierarchical task network which solves the high-level change request.

Advantages of the hybrid approach

The hybrid approach offers all the refinement capabilities provided by an HTN planner. Compared to previous approaches like [46] we link the transitions in a transition system to task refinement by associating a task with the transition. This task might be subject to HTN refinement when taking the transition in the transition system. Thus, we link HTN planning to Classical Planning. The hybrid approach can also trigger Classical Planning problems by specifying them within methods. This gives us the freedom to switch between both representations whenever the other one might be more convenient. This increases the readability, maintainability, and extendability of knowledge bases specified in the hybrid approach.

3.4. Need for a Hybrid Planning approach

The previous section has shown drawbacks and advantages of both HTN and Classical Planning in the context of the SAP case study. Hierarchical Task Network approaches can plan over the states of objects but then the description is rather unintuitive and complex as explained in Subsections 3.3.1 and 3.3.4. If dependencies defined over states are added to this domain, the description becomes more confusing (compare Subsection 3.3.4). In this context an HTN method based description has difficulties to cope with changes as seen in Subsection 3.3.2. Nevertheless, the HTN approach is a powerful concept which matches very well to the description of workflows (Section 3.3.3) and the hierarchical concept of change requests (Section 3.3.5). Some of the HTN drawbacks can be neutralized by Classical Planning. Subsection 3.3.1 shows that it has advantages in defining planning domains heavily determined by the states of objects. In such domains Classical Planning improves readability and maintainability of the knowledge base (Subsection 3.3.2). Furthermore, dependencies can be made explicit and do not need to be hidden within HTN method decompositions as explained in Subsection 3.3.4. The proposed hybrid approach in this thesis picks the best of both worlds. Chapter 5 introduces a Domain Specific Language (DSL) which describes the Hybrid Planning domain. This DSL describes the knowledge base and is used to generate a plan. The hybrid approach is introduced in Chapter 6.

4. Overview of the hybrid approach

This chapter gives an introduction to the basic principles of the hybrid approach. It builds a foundation for the introduction of the DSL in Chapter 5 and the algorithm in Chapter 6. We start by describing how the hybrid approach links together Hierarchical Task Network Planning and Classical Planning in Section 4.1. After that, we explain an example run of a simplified version of the algorithm in Section 4.2. Finally, Section 4.2 introduces the conceptual model of the hybrid approach.

4.1. Basic ideas behind the hybrid approach

This section introduces some elementary principles behind the hybrid approach. First of all, we explain its ability to reason about the refinement of change requests in Subsection 4.1.1. Section 4.1.2 continues by explaining some elementary characteristics of the state-transition systems used in the hybrid approach. Finally, we explain how dependencies can be described as Classical Planning problems linked to a transition in a state-transition system in Subsection 4.1.3.

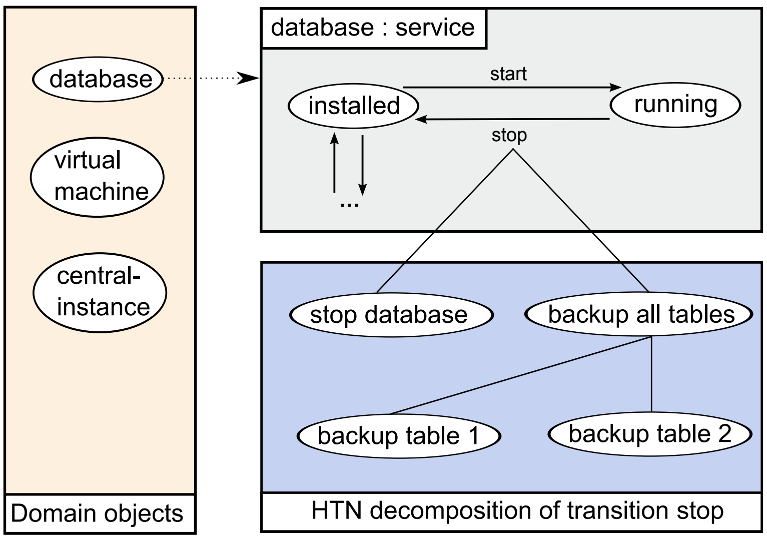
4.1.1. Hierarchical task refinement

This subsection explains the capabilities of the hybrid approach regarding the refinement of tasks. The hybrid approach is fully compatible to an HTN planner. The hybrid approach extends HTN in the way that it can reason about three kinds of tasks, non-atomic, atomic, and reserved tasks. Atomic and non-atomic tasks match to the notion of tasks in HTN. A reserved tasks, i. e., a reserved change request, describes a Classical Planning problem. An HTN method in the hybrid approach can refine a non-atomic tasks into a set of non-atomic, atomic, and reserved change requests. Be aware that only non-atomic change requests are subject to refinement by HTN methods and that HTN only deals with non-atomic and atomic tasks. This leads to full backwards compatibility with HTN approaches. As in pure HTN approaches, atomic tasks are implemented by an operator, which describes the changes to the knowledge base. A reserved change request part of the decomposition tree is dealt with by the Classical Planner of the hybrid approach. It is not subject to further refinement by a method but by the Classical Planner.

4.1.2. State-transition systems

The domain of IT change design consists of domain objects that can be considered to have a state. Thus, the hybrid approach incorporates the states a domain object can be in. This helps us to better describe dependencies and the lifecycle of domain objects. Combined with the HTN capabilities as explained in Subsection 4.1.1, a flexible way to describe the domain of IT change planning arises. Figure 4.1 shows how the hybrid approach links Classical Planning and HTN Planning.

Figure 4.1.: Example of a non-atomic change request linked to a transition of a state-transition system

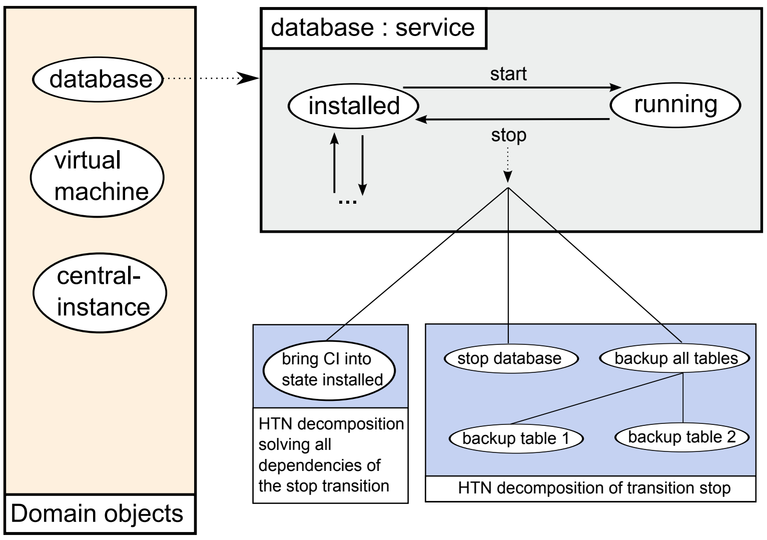


The left side of Figure 4.1 shows the knowledge base which consists of the domain objects the hybrid approach is planning over. These are for example services, virtual machines, and physical machines. Note that the domain objects match to an instance of a subclass of class “InfrastructureComponent” shown in Figure 2.4 in Section 2.5. Each domain object can be described by a state-transition system. A transition system consists of states, transitions, and a task linked to every transition. The task linked to a transition can be a non-atomic, atomic, or reserved change request. This task needs to be solved in order to successfully accomplish the transition. For example, Figure 4.1 shows a piece of a state-transition system for a database. Be aware that the shown part of the state-transition system matches to the transition system introduced for a service in Figure 2.3 in Section 2.2. The transition leading from “running” to “installed” is labeled with “stop”. Lets assume “stop” is the non-atomic task associated with the transition and the transition system is currently in state “running”, i. e., the database is running. If we define the goal as bringing the database into state “installed”, then we need to execute the “stop” transition meaning that the non-atomic task “stop” needs to be solved. Figure 4.1 also shows how the “stop” task is decomposed by methods. A method decomposes the “stop” task into tasks “stop database” and “backup all tables”. Another method decomposes the latter task into the atomic tasks “backup table 1” and “backup table 2”. Be aware that according to Subsection 4.1.1 the hybrid approach also allows to write methods that specify reserved change requests, i. e., Classical Planning problems, as subtasks. If the path to take in the state-transition system comprises more than one transition then the task network is adapted accordingly. The high-level description given here does not consider temporal dependencies between the tasks, dependencies between multiple reserved change requests, and dependencies described for a transition. The latter one is explained in the next subsection.

4.1.3. Describing dependencies in the hybrid approach

This subsection explains how dependencies are described in the hybrid approach. Previous observations made in Subsection 2.3.5 show that dependencies can be described as Classical Planning problems. For example, consider the "stop" transition of a database Grounded Execution Service as shown in Figure 4.2. In order to stop the database, the central-instance needs to be brought into state "installed" as described by the dependency in Table 2.1 in Subsection 2.3.1. After that, we can solve the task network linked to the "stop" transition. Figure 4.2 illustrates how the decomposition looks like when taking dependencies into account.

Figure 4.2.: Example of a non-atomic change request and a dependency linked to a transition

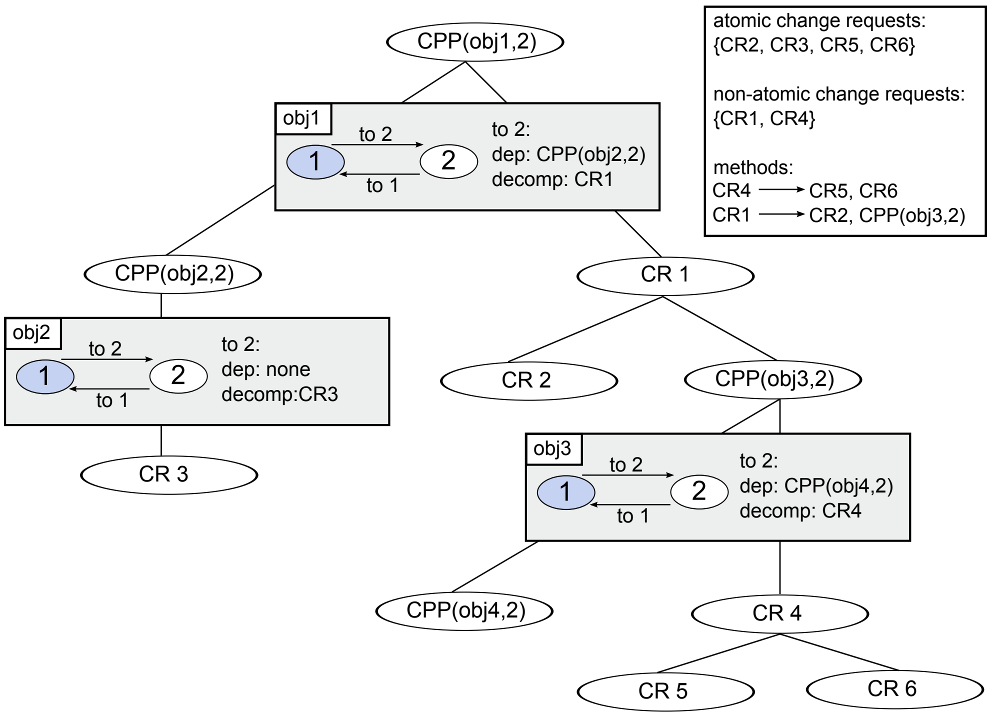


The "stop" transition of a database has only one dependency linked to it. It is described by a reserved change request, i. e., a Classical Planning problem, which demands to to bring the state-transition system associated with the dialog-instance into state "installed". Thus, the first task to solve is the reserved change request describing this Classical Planning problem. The task is shown in the first blue box in Figure 4.2. Be aware, that the dependency task in this box is not further refined in the example. There also might be more than one dependency change request linked to a transition. As soon as the reserved change requests describing the dependencies have been solved, the hybrid approach algorithm continues by solving the "stop database" and "backup all tables" tasks. These tasks are described by a method decomposing the non-atomic task “stop”. This has been previously explained in Subsection 4.1.2. Thus, in order to do a transition in a transition system, there are various tasks to solve. First of all, the reserved change requests describing the dependencies linked to the transition need to be solved. After that, the algorithm continues with planning for the task that needs to be done in order to do the transition. This task can be a non-atomic task like in Figure 4.2. In this case the task is further decomposed by a method. If the task linked to the transition is an atomic change request, then the algorithm tries to apply an operator to it. In the case a reserved change request is linked to the transition, the Classical planner plans for it. The hybrid approach takes care of setting the temporal constraints between the subtasks such that sound plans are produced.

4.2. High-level example of the hybrid approach

This section aims to provide a first high-level example of the hybrid approach. We give a simple example that is solved by a basic version of the algorithm. This section leaves aside lots of problems and more advanced questions that occur during planning, focusing on the core idea of the hybrid approach. It picks up the basic concepts and ideas introduced in Section 4.1 and puts them into an example.

Figure 4.3.: A high-level view of the hybrid approach algorithm



The top-level change request to solve is a Classical Planning problem. Classical Planning problems, i. e., reserved change requests, are formalized by the abbreviation CPP. As every change request (compare Figure 2.4 in Section 2.5) has a target domain object, a reserved change requests has one, too. In case of the top-level change request $CPP(obj1, 2)$ the target is $obj1$. The second argument of the term is a parameter of the reserved change request, the state to set the state-transition system linked to the domain object to. All in all, the $CPP(obj1, 2)$ change request demands to set the state of domain object $obj1$ to 2. Note that our top-level change request could also be an atomic or non-atomic task. Because this matches to normal HTN behavior, it is of less interest.
The box on the right side of Figure 4.3 shows the change request and methods described by the knowledge base. $CR2$, $CR3$, $CR5$, and $CR6$ are atomic change requests. $CR1$ and $CR4$ are non-atomic change requests. The knowledge base also knows two methods that can decompose the non-atomic change requests. Note that the method decomposing $CR4$ only has atomic subtasks and that the method decomposing $CR1$ describes an atomic and a reserved change request as subtasks.
Figure 4.3 also shows the states of the state-transition systems that are associated with the domain objects. A state marked blue is representing the current state of the transition system. The upper left part of the transition system shows the domain object it accounts for. The state-transition systems for $obj1$, $obj2$, and $obj3$ are all in state 1.

As mentioned in Section 4.1, there is a non-atomic, atomic, or reserved change request linked to a transition. In addition to that, a set of dependencies, i. e., reserved change requests, are specified for a transition. This is also shown in the state-transition systems given in Figure 4.3. There is one dependency task, the reserved change request $CPP(obj2, 2)$, linked to transition $to2$ in the state-transition system of domain object $obj1$. Furthermore, the non-atomic change request $CR1$ is linked to the transition, modeling the task to be solved in order to do the transition. Note that nothing is specified for transition $to1$ in every transition system because it is not of relevance to the example.

The planner starts with planning for the $CPP(obj1, 2)$ change request. Because this is a reserved change request, the Classical Planner part of the hybrid approach defines how to treat this change request. First of all, we need to get the transition system which is associated to $obj1$. The reserved change request demands to bring $obj1$ into state 2. As the state-transition system is currently in state 1 (compare the blue marked state in $obj1$ in Figure 4.3), the shortest path to the goal only consists of transition $to2$. But in order to execute a transition we need to resolve the dependency change requests, i. e., the reserved change requests linked to the transition, first. The only dependency change request described by transition $to2$ in $obj1$ is $CPP(obj2, 2)$. The dependency describes a Classical Planning problem that demands to bring domain object $obj2$ into state 2. As soon as this change request is solved, the algorithm needs to solve the non-atomic, atomic, or reserved change request linked to transition $to2$. In case of transition $to2$ in $obj1$ it is the non-atomic change request $CR1$. Thus, $CPP(obj1, 2)$ has two children, the dependency task $CPP(obj2, 2)$ and the non-atomic task $CR1$. Be aware that we apply some simplifications here. First of all, we do not consider temporal constraints between

the children. Furthermore, we chose an example in which the shortest path only consists of one transition. If there were more than one transition in the shortest path, then the same principle applies and the tasks of the n^{th} transition are added after the subtasks solving the $n-1^{th}$ transition.
The algorithm continues by planning for the $CPP(obj2, 2)$ change request. This is done by the Classical Planner because it is a reserved change request. The current state of the state-transition system linked to $obj2$ is 1. The shortest path to the goal contains only transition $to2$. There are no dependency change requests specified for this transition. Thus, there is only one child to the $CPP(obj2, 2)$ change request, the $CR3$ change request. This is the change request that is linked to the execution of the transition. Because $CR3$ is an atomic change request, the HTN planner tries to find an operator for it and applies the changes to the knowledge base. After that, the algorithm continues with planning for change request $CR1$.
$CR1$ is a non-atomic change request and it is planned for by the HTN Planner of the hybrid approach. As it can be seen in Figure 4.3 there is a method defined in the knowledge base that decomposes $CR1$ into $CR2$ and $CPP(obj3, 2)$. We assume that this method is applicable to $CR1$. Thus, $CR1$ gets two new child change requests, $CR2$ and $CPP(obj3, 2)$. The hybrid planner uses a depth-first search strategy leading to an invocation of the planner on change request $CR2$.
$CR2$ is an atomic change request thus it is dealt with by the HTN Planner. The HTN Planner searches for an operator that implements the atomic task. Having applied the effects of the operator to the knowledge base, the planner continues with planning for the $CPP(obj3, 2)$ change request. It is a reserved change request. It demands to set the state of domain object $obj3$ to state 2. The planner determines the transition system linked to $obj3$ and its current state. The shortest path to reach the goal state 2 only consists of transition $to2$. There is a dependency task linked to the transition. It demands to set the state in domain object $obj4$ to 2. The task linked to the transition is $CR4$. Both tasks become the new subtasks of $CPP(obj3, 2)$. The planner continues by planning for the dependency task.
The dependency task $CPP(obj4, 2)$ is planned for by the Classical Planner because it is a reserved change request, i. e., a Classical Planning problem. We assume that the state-transition system associated with $obj4$ is already in state 2. In this case nothing needs be done because the dependency is trivially fulfilled. The planner can thus continue with planning for $CR4$.
$CR4$ is a non-atomic change request which is planned for by the HTN Planner. It uses a method described in the knowledge base to decompose the non-atomic task into subtasks $CR5$ and $CR6$. Both are atomic change requests. The planner tries to find an operator for $CR5$ first. If it succeeds it applies the operator to the knowledge base and continues with planning for $CR6$ in the same way. After that, planning is finished and $CPP(obj1, 2)$ has been successfully decomposed.

Note that lots of simplifications were assumed in the given example. We did not consider temporal constraints between subtasks, possible conflicts between subtasks that might lead to unsound plans, shortest paths that comprise more than one transition, and parameters of change requests. These issues are dealt with when we introduce the algorithm in Chapter 6

4.3. The conceptual model of the hybrid approach

This section introduces the conceptual model of the hybrid approach. It formalizes the ideas introduced in Section 4.1 and adds some additional concepts necessary when talking about planning algorithms. A conceptual model is a simple theoretic device to describe the concepts that are important for the hybrid approach. It gives the reader an overview of concepts like state-transition systems, transitions, states, dependencies, and their relationships to each other as explained in Section 4.1. The concepts are described using sets, functions, and tuples. This is similar to the conceptual model of Classical Planning as introduced in Subsection 3.1.1.
Let $O = \{o_1, ..., o_n\}$ be the set of all domain objects belonging to a planning domain. Then the following definitions hold:

Definition 1 (*task / change request*)

A task / change request is a triple

$$c = (c\_name, c\_o, c\_params)$$

where

- c_name *is the name of the change request which is unique within the set of all names of change requests.*
- $c\_o \in O$ *is a domain object of the planning domain (see Definition 9) against which the change request is issued.*
- $c\_params = \{p_1, ..., p_n\}$, $n \in \mathbb{N}_0$ *is a set of parameters of the change request.*

A change request can be either an atomic, non-atomic, or reserved change request.

Let $CR = \{c_1, ..., c_n\}$, $n \in \mathbb{N}_0$, $c_n = (c\_name_n, c\_o_n, c\_params_n)$ be the set of all change requests. A change request can be either a non-atomic, atomic, or reserved change request. Atomic and non-atomic change requests are used within HTN Planning problems while reserved change requests describe a Classical Planning problem. Due to this categorization CR can be partitioned into three sets consisting of atomic, non-atomic, and reserved change requests. $CR = atomic\_CRs \cup non\_atomic\_CRs \cup res\_CRs$ where all three sets are pairwise disjunct. We define CR_names, the set of all names of change requests, as $CR\_names = \{c\_name \mid (c\_name, c\_o, c\_params) \in CR\} = \bigcup_{i\in\{1,...,n\}} c\_name_i$. CR_names does not contain duplicates, in the case change requests have the same name.
The function $names$ is defined as follows:

$$names : 2^{CR} \to 2^{CR\_names},$$
$$\text{where} \quad 2^{CR} \ni X \to \{name \mid (name, target, params) \in X\} \in 2^{CR\_names}$$

Given a set of change requests, *names* delivers a set containing the names of the change requests.

Definition 2 (*state-transition system*)

A state-transition system $\sum$ in the hybrid approach is a triple

$$\sum = (o, S, T)$$

where

- *$o \in O$ is the domain object for which $\sum$ accounts.*
- *$S = \{s_1, ..., s_n\}$, $n \in \mathbb{N}_0$ is a set of states.*
- *$T = \{t_1, ..., t_n\}$, $n \in \mathbb{N}_0$ is a set of transitions (see Definition 4).*

In the conceptual model of the hybrid approach a state-transition system accounts for exactly one domain object $o \in O$. For example, every instance of a subclass of class “InfrastructureComponent” (compare Figure 2.4 in Section 2.5) is a domain object and has its own state-transition system. The proposed DSL relaxes the constraints by describing state-transition systems that can account for a set of domain objects. The conceptual model does not take this into account to keep matters simple.

Definition 3 (*reserved change request*)

A reserved change request is a change request

$$c = (c\_name, c\_o, c\_params)$$

as defined in Definition 1, such that

- *$c\_name \in names(res\_CRs) = \{CPP\}$, i. e., $c\_name = CPP$.*
- *$c\_o \in O$ is a domain object of the planning domain (see Definition 9) against which the reserved change request, i. e., the Classical Planning problem, is issued.*
- *$c\_params = \{p\}$, is a set of parameters containing one parameter. p describes the goal state of the Classical Planning problem, i. e., the state the transition system associated to c_o is to be brought to. More formal let $\sum = (c\_o, S, T)$ be the state transition system for c_o, then p describes the goal state to be reached in $\sum$, i. e., $p \in S$.*

In the conceptual model a reserved change request adheres to all the characteristics of a change request as demanded by Definition 1. The name of a reserved change request is fixed to "CPP" (for Classical Planning problem) and it has only one parameter. This parameter describes the goal state of the Classical Planning problem that needs to be solved regarding the domain object c_o. For example, the reserved change request $cr = (CPP, o, state_1)$ demands that domain object o is to be brought into state $state_1$.

Definition 4 (*transition*)

A transition t of a state-transition system $\sum = (o, S, \{..., t, ...\})$ is a 5-tuple

$$t = (t\_name, start, goal, task, D)$$

where

- *t_name is the name of the transition. It needs to be unique within a state-transition system.*
- *$start \in S$ is the source state of the transition.*
- *$goal \in S$ is the target of the transition.*
- *$task$ is an atomic, non-atomic, or reserved change request to be solved when executing the transition.*
- *$D = \{d_1, ..., d_n\}$, $n \in \mathbb{N}_0$ is a set of dependencies (see Definition 5) accounting for transition t in $\sum$.*

Each transition of a state-transition system has a description (t_name). The transition also specifies the source state and the target state it leads to. Linked to a transition is a change request $task$. It is optional and describes the change request which needs to be solved when taking the transition in the transition system. Before this task can be solved, a set of dependencies D has to be fulfilled first. Dependencies are defined in Definition 5.

Definition 5 (*dependency*)

A dependency d in the context of a transition $t = (t\_name, start, goal, task, \{..., d, ...\})$ of a state-transition system $\sum = (o, S, \{..., t, ...\})$ is a pair

$$d = (d\_name, Tasks)$$

where

- *d_name is the name / description of the dependency. It needs to be unique within all dependencies belonging to a transition.*
- *$Tasks = \{cr_1, ..., cr_n\}$, $n \in \mathbb{N}_0$, $\forall_{i=1...n} \ : \ cr_i \in res\_CRs$.*

A dependency d belonging to a transition t is a pair holding the name of the dependency (d_name) and $Tasks$, a set of reserved change requests. Note that all tasks in $Tasks$ are reserved change requests, i. e., they describe Classical Planning problems. All reserved change requests contained in $Tasks$ need to be solved in order to fulfill a dependency d.

Definition 6 (*decomposition*)

A decomposition decomp is a pair

$$decomp = (ord, CRs)$$

where

- *ord is an order, $ord \in \{parallel, sequential\}$*
- *$CRs = (cr_1, ..., cr_n)$, $n \in N_0$, $\forall_{i=1...n} \ : \ cr_i \in CR$, is an ordered set of change requests.*

A decomposition can either be parallel or sequential, i. e., totally ordered. It also defines an ordered tuple of tasks. These are the tasks that need to be solved in order to achieve the decomposition.

Definition 7 (*method*)

A method m is a triple

$$m = (m\_task\_name, m\_pre, m\_decomp)$$

where

- $m\_task\_name \in names(non\_atomic\_CRs)$ *is the name of a non-atomic change request that can be decomposed by the method.*
- *m_pre is a precondition that needs to be true in order to decompose a change request.*
- *m_decomp is a decomposition defining the subtasks and an order how they need to be solved (see Definition 6).*

Let $c = (c\_task\_name, c\_o, c\_params)$ *be a non-atomic change request, i. e.,* $c \in non\_atomic\_CRs$. *m is called applicable to change request c if* $c\_task\_name = m\_task\_name$ *and m_pre is fulfilled by c_o.*

A method can only decompose non-atomic change requests. It specifies the name (*m_task_name*) of a change request that can be decomposed. A precondition (*m_pre*) checks whether the target of the change request, i. e., the second element in a change request triple, fulfills the circumstances to do the task decomposition. The decomposition *m_decomp* describes the subtasks and an order among them. The subtasks need to be solved in order to achieve the high-level change request.

Definition 8 (*operator*)

An operator op is a triple

$$op = (op\_task\_name, op\_pre, effects)$$

where

- $op\_task\_name \in names(atomic\_CRs)$ *is the name of an atomic change request the operator describes the changes for.*
- *op_pre is a precondition that needs to be fulfilled in order to apply the operator.*
- *effects describes how the knowledge base is altered through the execution of the operator.*

Let $c = (c\_task\_name, c\_o, c\_params)$ *be an atomic change request, i. e.,* $c \in atomic\_CRs$. *op is called applicable to the change request c if* $c\_task\_name = op\_task\_name$ *and op_pre is fulfilled by c_o.*

Operators describe the effects that atomic change requests have on the knowledge base. This can be more sophisticated changes like adding or deleting domain objects.

An operator describes the name of an atomic change request for which it can perform the changes to the knowledge base. A precondition is evaluated over the target of the atomic change request in order to decide whether it can be applied. Furthermore, an operator contains a concept to describe the effects on the knowledge base when executing the operator.

Definition 9 (*Hybrid Planning domain*)

A planning domain D of the hybrid approach is a 4-tuple

$$D = (\sigma, M, OP, O)$$

where

- $\sigma = \{\sum_1, ..., \sum_n\}$, *$n \in N_0$ is a set of state-transition systems as defined in Definition 2.*
- $M = \{m_1, ..., m_n\}$, *$n \in N_0$ is a set of methods (see Definition 7).*
- $OP = \{op_1, ..., op_n\}$, *$n \in N_0$ is a set of operators (see Definition 8).*
- $O = \{o_1, ..., o_n\}$, *$n \in N_0$ is a set of domain objects.*

A *Hybrid Planning domain* aggregates all state-transition systems, methods, operators, and domain objects to a planning domain. A Hybrid Planning domain is used to describe a Hybrid Planning problem.

Definition 10 (*A Hybrid Planning problem*)

A planning problem P of the hybrid approach is a pair

$$P = (D, CR')$$

where

- *D is a Hybrid Planning domain as defined in Definition 9.*
- *$CR' \subseteq CR$ is a set of change requests to plan for.*

The *Hybrid Planning problem* consists of a Hybrid Planning domain and a set of change requests to plan for.
Furthermore, we need to define what a plan looks like in the domain of the hybrid approach:

Definition 11 (*plan*)

A plan p is a pair

$$p = (CR', happens\_before)$$

where

- $CR' \subseteq atomic\_CRs$.
- $happens\_before \subseteq CR' \times CR'$ *is a partial order such that* $(cr_1, cr_2) \in happens\_before$ *if* cr_1 *needs to be finished before* cr_2 *can be started.*

In the hybrid approach a plan is a pair consisting of a set of atomic change requests and a relation called $happens\_before$. This relation is a partial order $<$ such that $cr_1 < cr_2$ means that cr_1 needs to be completed before cr_2 can be started. Because $<$ is a partial order, there can be tasks $cr_1, cr_2 \in CR'$ such that $(cr_1, cr_2) \notin happens\_before$ and $(cr_2, cr_1) \notin happens\_before$. In this case cr_1 and cr_2 can be executed in parallel.

The conceptual model describes the ideas and concepts that are important to the hybrid approach. It shows the relationships between transitions, dependencies, change requests, task networks, and other concepts. For example, a dependency always belongs to a transition. The proposed DSL needs to adhere to these constraints. However, the conceptual model does not specify how the concepts are expressed in the DSL. It only says which concepts need to be expressed by the DSL and how they relate to each other.

5. A Domain Specific Language describing the Hybrid Planning domain

This chapter gives an introduction to the Domain Specific Language used to describe the hybrid approach planning domain. The DSL only describes one possible way how to express the planning domain. It needs to conform to the definitions introduced in Section 4.3 defining the conceptual model of the hybrid approach. Section 5.1 introduces the Groovy language, which is used to express the proposed Domain Specific Language. Groovy is explained to the extent it is needed to understand the constructs to define the Domain Specific Language. After that, we present the DSL for state-transition systems in Section 5.2. The DSL is described syntactically by EBNF notation. Furthermore, semantical constraints are explained. Sections 5.3 and 5.4 explain, how the Domain Specific Language can describe HTN methods and operators.

5.1. The Groovy language for DSL specification

This section introduces the programming language Groovy, which is used to describe the Domain Specific Language of the planning domain. The DSL makes intense use of some important concepts of the Groovy language. Thus, it is important to understand these concepts in order to understand the DSL and how it can be used by the planner. Section 5.1.1 gives a high-level overview of the characteristics of the Groovy language. Closures, which are an important concept of Groovy, are introduced in Subsection 5.1.2. In addition to closures, Groovy supports dynamic object orientation features, like catching method calls to undefined methods. This is explained in Section 5.1.3. Finally, Section 5.1.4 sums up the advantages of Groovy in the context of the hybrid approach. Readers interested in a more comprehensive introduction to the Groovy language are encouraged to have a look at [32].

5.1.1. Short overview of Groovy

"Groovy is an agile dynamic language for the Java Platform with many features that are inspired by languages like Python, Ruby, and Smalltalk, making them available to Java developers in a Java-like syntax" ([32], p.4). Groovy is closely linked to Java because all Groovy code is directly translated into Java code. The fact that Groovy code is actually Java code enables full integration with Java. Classes defined in Java can be directly instantiated in Groovy code giving all the power of the Java class libraries to the Groovy programmer. In addition to that, Groovy has its own powerful libraries, called *GDK*. Groovy code can be statically (like in Java) or dynamically typed. The language offers very good scripting capabilities. But Groovy goes far beyond the capabilities of a scripting language. It offers a concept, called closure, to encapsulate executable code into objects. Furthermore, it is great for XML processing, database querying, and file processing. In addition to that, Groovy offers dynamic object orientation features. This enables us to change the behavior of objects at runtime. All these features make Groovy an excellent candidate to implement a planner.

5.1.2. Groovy closures

This subsection gives an introduction to Groovy closures. *Closures* are Groovy objects that hold executable code. Furthermore, parameters can be passed to a closure when executing it. As closures are Groovy objects, their reference can be passed as a parameter to a method call. The method can then use this closure to customize its behavior according to the code defined in the closure. Closures are very useful for the planner because we can write Groovy code accessing our model elements into a closure. This closure can then be used during planning to evaluate a precondition over the model or to change the model when it comes to implement an operator. A first simple usage of a closure is shown in Listing 5.1.

Listing 5.1: Defining and calling a closure

```
1 def my_closure = {
2         int x = 5 + 5;
3         println "value of x : " + x; return x
4 }
5
6 int ret_value = my_closure()
7 println("return value of closure : " + ret_value)
```

```
value of x :  10
return value of closure :  10
```

Below each listing there is a box which shows the results when executing the code from the listing above in a Groovy interpreter. Lines 1-4 define a closure over an untyped variable, called “my_closure”, holding a reference to a closure object. In a closure definition everything written within curved brackets is the code that is held by the closure object. The closure’s code consists of three statements. The first assigns the result of the addition $5+5$ to x (line 2). After that, a Groovy function named “println” to print the value of x is called. It is the equivalent of Java’s “System.out.println”. In line 3 the closure returns the value of x.

Line 6 calls the previously defined closure and the return value of the closure is stored in the integer variable “ret_value”. Be aware that the statement in line 6 does not need to be ended by a semicolon because this is optional in Groovy. The last line prints the value of the integer returned by the closure. Looking at the output in the box below Listing 5.2, we can see how the statements within the closure are executed when calling the closure in line 6. As a closure is an object, its reference can be passed to methods. Listing 5.2 shows how a closure is defined and then passed to a method in order to “customize” it.

Listing 5.2: Defining and passing a closure to a method

```
1  def filter = { int x -> return x > 10 }
2
3  def test_method(int x, Closure closure) {
4          if (closure(x)) {
5                  println "Filter approved " + x
6          }else{
7                  println "Filter did not approve " + x
8          }
9  }
10
11 test_method(9, filter)
12 test_method(11, filter)
```

```
Filter did not approve 9
Filter approved 11
```

Line 1 defines a closure called “filter”. The closure has one integer parameter, which is written on the left side of the arrow. The right side of the arrow defines the actual code that makes up the closure. The closure returns a boolean value describing whether the parameter passed to the closure is bigger than 10. A method named “test_method” is defined between lines 3 and 9. It expects two parameters, an integer and a closure. In line 4 the passed closure is called with the passed integer in order to receive a boolean value that describes whether the parameter is filtered or not. Depending on the return value of the closure a message is printed. The last two lines are calling the method with different integers and the same “filter” closure. The example given in Listing 5.2 shows how code defined in a closure can be used to customize a method. The “filter” closure could define a precondition of an HTN method and “test_method” from Listing 5.2 could be a general plan method. In order to use closures in the DSL, we need a more intuitive way of how to define them than in line 1 in Listing 5.2. Listing 5.3 shows three different ways how to do calls to a closure which takes a closure as its last parameter.

A closure “test_closure” is defined in lines 1-6. It takes two parameters, a string called “test” and a closure called “closure”. The parameter “test” is printed out in line 3 and then the closure “closure” is executed. We also define a closure “second_param” in line 8, which prints out the string “hello world” when called. The last three lines of Listing

Listing 5.3: Different possibilities how to pass a closure to a closure

```
1  def Closure test_closure = {
2          String test, Closure closure ->
3          println test;
4          closure();
5          println ""
6  }
7
8  def Closure second_param = {println "hello world"}
9
10 test_closure("first param" , second_param)
11 test_closure("first param" , {println "hello world"})
12 test_closure("first param") {println "hello world"}
```

```
first param
hello world

first param
hello world

first param
hello world
```

5.3 show three different possibilities how to call the closure “test_closure”. All calls are semantically equivalent to each other. The first call in line 10 passes the two parameters to the closure as known from ordinary Java method calls. In line 11 we pass a locally defined closure as the second parameter to “test_closure”. It should not be surprising that it delivers the same result. The last line shows an abbreviation of the call in line 11 in Listing 5.3. If the last parameter passed to a closure is a locally defined closure, then Groovy allows to put the parameters before the last one into round brackets (in case there are any). The curved brackets defining the last parameter, a locally defined closure, can be written outside the round brackets. This is shown in line 12 in Listing 5.3. Thus, we put the first parameter in round brackets and write the locally defined closure outside the round brackets. This closure is treated as the second parameter. This only works for locally defined closures. The closure “{println “hello world”}” in line 12 cannot be replaced by the previously defined “second_param” closure. As we can see on the output shown below Listing 5.3, all three calls deliver the same result.
Listing 5.4 shows a simple DSL for a state-transition system making use of this abbreviation. The listing is syntactically valid Groovy code. The idea behind the DSL is explained according to the control flow when executing the “transition_system” closure from Listing 5.4.

Like in the previous examples, the code shown in Listing 5.4 only works if the inter-

Listing 5.4: A simple DSL for state-transition systems

```
1  transition_system("system1") {
2          states {
3                  state {"1"}
4                  state {"2"}
5          }
6
7          transitions {
8                  transition {"1 to 2"}
9                  transition {"2 to 1"}
10         }
11 }
```

preter knows of the definitions of the closures that are called. To limit the length of code snippets the definitions are shown in Listing 5.5. Remember from the previously given examples that the definition of closures has to be processed before the closures are invoked. After all closure definitions from Listing 5.5 have been read, the Groovy interpreter can start to call the closure "transition_system" in Listing 5.4, line 1. The "transition_system" closure is defined in lines 1-5 of Listing 5.5. The closure takes two parameters, a string called "name" and a closure called "states_and_transitions". The parameter "name" is mapped to "system1" and the closure "states_and_transitions" is mapped to the closure defined through the content within the curved brackets in lines 1 and 11 in Listing 5.4. The control flow proceeds by executing the code stored in the "transition_system" closure. Thus, the first output of the program will be the "println" statement in line 3 of Listing 5.5. After that, the closure "states_and_transitions" is executed. Thus, the code between the curved brackets in lines 1 and 11 in Listing 5.4 is executed. This is again valid Groovy code. It is a sequence of calls to the closure "states" and the closure "transitions". The "states" closure is called and one parameter, the closure between the curved brackets in lines 2 until 5 in Listing 5.4, is passed to it. To see what happens when the "states" closure is executed we need to look at the definition of this closure in lines 7-9 in Listing 5.5. The "states" closure executes its only parameter, a closure. This means that lines 3 and 4 from Listing 5.4 are executed. This is valid Groovy code again because it is a sequence of two closure calls each to the closure "state". First the "state" closure is called with one parameter, the closure {"1"}. To see what happens with the parameter, we need to look at the definition of the closure "state" in Listing 5.5, lines 15-18. It shows that the closure {"1"}, which is semantically equivalent to {return "1"}, is executed and its return value is stored in an untyped variable called "name_of_state". After that, the value of this variable is printed out and the control flow returns to the execution of the second "state" closure. The "transitions" closure is then treated analogously to the "states" closure.
The output to the command line when executing the "transition_system" closure is shown in the box below Listing 5.5.
As shown, Groovy offers excellent possibilities to specify planning domains in a very elegant way. Listing 5.5 can be hidden from the user who specifies a planning domain. The

Listing 5.5: Code to define the closures from Listing 5.4

```
def Closure transition_system = {
        String name, Closure states_and_transitions ->
        println "name of trans. system" + name
        states_and_transitions()
}

def Closure states = { Closure body ->
        body()
}

def Closure transitions = { Closure body ->
        body()
}

def Closure state = { Closure body ->
        def name_of_state = body()
        println "came accross state " + name_of_state
}

def Closure transition = { Closure body ->
        def name_of_transition = body()
        println "came accross transition " + name_of_transition
}
```

```
came accross state 1
came accross state 2
came accross transition 1 to 2
came accross transition 2 to 1
```

writer of the planning domain only needs to be aware of the syntax and the semantics of a DSL like the one shown in Listing 5.4. Note that with small modifications done to Listing 5.5 a state-transition system data structure can be built when executing the DSL.

5.1.3. Dynamic object orientation

Groovy offers the possibility to catch calls to unknown methods in a well defined method. This allows the specification of subtasks of an HTN decomposition by calls to undefined methods in a closure. The calls are intercepted by a well defined Groovy method, which then can generate the new change request objects according to class “AI_ChangeRequest” in Figure 2.4 in Section 2.5. The parameters of the method call are also provided to this well defined Groovy method. We can then set the parameters by assigning a value to the “params” map of the “AI_ChangeRequest” instance. An example of a Groovy DSL describing a set of subtasks is given in Listing 5.6.

Listing 5.6: A DSL using calls to unknown methods

```
subtasks{
        change_request1([f : "t1", s : "t2"])
        change_request2([f : "t3"])
}
```

Listing 5.7 shows the necessary closure and method definitions in order to process the DSL from Listing 5.6. Each Groovy class has an "invokeMethod" method. If a non-existing method is called, the control flow is redirected to this method. A default implementation is automatically inherited behind the scenes. In order to customize the behavior of a class, we need to override this method. This also works within Groovy scripts because Groovy scripts are translated to Java classes behind the scenes. Listing 5.7 shows the definition of the "invokeMethod" method. It takes a string, the name of the method, and an object modeling the parameters. It is assumed that we only pass a Groovy map when calling non-existing methods. This map matches to the "params" map stored by an "AI_ChangeRequest". The given example prints the name of the change request, i. e., the name of the unknown method called, and extracts the map from the object. Lines 6-8 print out the key value pairs of the map by making use of a closure. The method "keySet" in line 6 of Listing 5.7 delivers a list of keys. On this list the method "each" is called which takes a closure as a parameter. This closure defines what is done to each element from the list of keys. The Groovy interpreter will call the closure with each element from the list. By convention the passed element is referenced by the identifier "it" within the closure. In addition to that, the "subtasks" closure needs to be defined (line 12 in Listing 5.7). The output of the program is shown in the box below Listing 5.7. The proposed DSL for the hybrid approach makes use of this idea in order to specify task decompositions.

5.1.4. Advantages of Groovy as a language for planning

There are some advantages when using Groovy as the underlying programming language to implement the hybrid approach:

- **Java compatibility:** As Groovy is based 100% on Java, it is fully backwards compatible with Java. This enables us to define our model as explained in Figure 2.4 in Section 2.5 as an EMF model to make use of the deepcopy mechanism to backup and restore the model during planning. Furthermore, the planner can be written in Groovy making use of features like closures and method interception.
- **Closures:** Closures can be used to describe preconditions that are directly formulated over objects belonging to the model. They can specify the Groovy code which is to be executed on an object in order to check whether a precondition is fulfilled. The object can be passed to the closure by a parameter. A planner can

Listing 5.7: Method to catch calls to unknown methods

```
public Object invokeMethod(String name, Object args) {
        println "new change request : " + name
        List list = (List) args
        Map arguments = (Map) list[0]
        println "parameters :"
        arguments.keySet().each {
                println "key " + it + ", value " + arguments[it]
        }
        println "\n"
}

def Closure subtasks = {Closure body -> body()}
```

```
new change request :  change_request1
parameters:
key f, value t1
key s, value t2

new change request :  change_request2
parameters:
key f, value t3
```

then store this closure in a data structure and use it at planning time to evaluate preconditions directly over the model. There is no need to construct a transformation of the model to a predicate based knowledge base as it is used by lots of planners like SHOP2.

- **Possibility to catch method calls:** Groovy's dynamic object orientation capabilities enable us to catch calls to unknown methods by overriding a well defined method. This can be used to define a decomposition of a task by writing Groovy code into a closure. This nicely integrates with a DSL describing state-transition systems similar to the one shown in Listing 5.4 in Subsection 5.1.2.

5.2. The DSL to describe state-transition systems

This section introduces the DSL for state-transition systems used by the planner. The conceptual model presented in Section 4.3 shows that a state-transition system comprises states and transitions. The latter one is a more complex concept which is linked to the concepts of decomposition and dependencies. Especially transitions cannot be looked at detached from transition systems. The same accounts for dependencies and transitions. This results in the issue that the DSL for transitions and dependencies needs to be explained together with the DSL for state-transition systems. This section explains the

DSL for state-transition systems and the concepts linked to it in a step by step manner. First, the DSL specifying the frame of a transition system is explained in Subsection 5.2.1. After that, we show how transitions are formalized in the DSL in Subsection 5.2.2. Finally, Subsection 5.2.3 explains how dependencies can be described for the transitions. For each subsection an EBNF describing the syntax and an example adhering to it are given. Semantical constraints that need to account but are difficult to express in EBNF are textually explained by comments.

5.2.1. The frame to describe state-transition systems

The DSL for state-transition systems follows the idea introduced in Listing 5.4 in Subsection 5.1.2. It uses the recursive principle that closures can have closures as parameters. To introduce the DSL, we start at the highest closure and refine the closures in a step-wise manner. As a running example, an instance of the DSL describing the dependencies among services is given. This example is then refined together with its EBNF. Listing 5.8 shows the first part of the DSL adhering to the EBNF from Figure 5.1. The EBNF presented adheres to the EBNF specification in [28]. Non-terminal symbols are written in blue typewriter font, terminal symbols in the usual black font.

Figure 5.1.: EBNF for the frame of state-transition systems

(1) `Transition_system` =	’transition_system(target : ’ `Classname_tran` ’) {’ `Tran_system_body` ’}’ ;
(2) `Classname_tran` =	(* Name of a class for which the transition system accounts. *) ;
(3) `Tran_system_body` =	`Valid_closure` `States_closure` `Transitions_closure` ;
(4) `Valid_closure` =	’valid{’ `Valid_closure_body` ’}’ ;
(5) `Valid_closure_body` =	(* valid Groovy code returning a boolean value. The identifier “it” references any instance of the class named in rule 2. *);
(6) `States_closure` =	’states{’ {`State_closure`} ’}’ ;
(7) `State_closure` =	’state{’ `State_name` ’}’ ;
(8) `State_name` =	(* a valid Java string representing the name of a state. Needs to be unique within the bodies of all “state” closures. *);
(9) `Transitions_closure` =	(* see EBNF rule 1 in Figure 5.2 for definition. *) ;

The closure which is called when executing the DSL in Listing 5.8 is the “transition_system” closure. It takes two parameters, the first is a map and the second is a closure (compare Listing 5.8, line 1). The map needs to have a “target” key. The value of the key is the name of a class occurring in the model. In the given example it is the class “AI_GroundedExecutionService”, a class representing database, dialog-instance, and central-instance services. Semantically this means that each domain object of instance “AI_GroundedExecutionService” is a potential candidate to have the described state-transition system associated. Whether it is really associated to an instance of “AI_GroundedExecutionService” is determined by the “valid” closure. It is shown in line 3 in Listing 5.8 and defined by the non-terminal “Valid_closure” in rule 4 in Figure 5.1. To make the following textual explanations more readable we call a closure x that is given to a closure y as the last parameter the body of y. For example, the closure defined

Listing 5.8: A DSL describing the frame of state-transition systems

```
1  transition_system(target : AI_GroundedExecutionService) {
2
3          valid {true}
4
5          states {
6                  state {"not_installed"}
7                  state {"installed"}
8                  state {"started"}
9          }
10
11         transitions{ ... }
12
13 }
```

through the curved brackets between lines 1 and 13 in Listing 5.8 is called the body of the “transition_system” closure. Alike the closure “{true}” in line 3 is called the body of the “valid” closure.
The body of the “transition_system” closure is always a sequence of three closure invocations. EBNF rule 3 in Figure 5.1 shows that this sequence is fixed and none of the three closures, i. e., non-terminals on the right side, are optional. The “valid” closure is very simple. It is described by EBNF rules 4 and 5. The body of the “valid” closure needs to return a boolean value. The “valid” closure decides whether an arbitrary domain object that is an instance of the target class has the described transition system associated. For example, this becomes advantageous for the different services that are all a subclass of “AI_GroundedExecutionService” (compare Figure 2.4 in Section 2.5). This enables us to associate the same state-transition system to different kind of services if the body of the “valid” closure returns “true” as shown in Listing 5.8.

Listing 5.9: Possible “valid” closures

```
valid {it.type == "DI"}
valid {it.type == "CI"}
valid {it.type == "DB"}
valid {System.currentTimeMillis() > 5}
```

Listing 5.9 shows different possible bodies of the “valid” closure, i. e., closures passed to the “valid ”closure. The planner always calls the “valid” closure with one parameter, which is an instance of the class provided in the “target” key value pair of the “transition_system” closure (compare line 1 in Listing 5.8). In the given example the code in the body of the “valid” closure needs to be executable on each instance of an “AI_GroundedExecutionService” class. If only one parameter is passed to a closure, then it is always referenced by the identifier “it” within the closure. Grounded Execution Services have an attribute called “type” (compare Figure 2.4 in Section 2.5) describing the kind of service. The “valid” closure could thus select more precisely for which instance

of class “AI_GroundedExecutionService” the transition system holds. Using the first closure from Listing 5.9 would only associate the transition system to dialog-instances. The last closure shows that arbitrary code returning a boolean value can be written into the closure. In this case the transition system is only linked to a Grounded Execution Service if the system time is greater than 5 milliseconds at the time the planner evaluates the closure. The example given in Listing 5.8 just returns true because the transition system accounts for databases, central-instances, and dialog-instances. Remember that in contrast to the states and transitions the dependencies are different for database, central-instance, and dialog-instance services.
The “states” closure in lines 5-9 is defined by EBNF rules 6-8 in Figure 5.1. The body of a “states” closure consists of an optional amount of “state” closure calls. It is basically allowed to define a transition system without states but the “states” closure always has to exist (compare rule 3). A “state” closure models one possible state of the transition system. The closure given to it needs to return a string which describes the name of the state as described by rule 7 and 8 in Figure 5.1. There must not exist two “state” closures whose bodies return the same string, i. e., state names need to be unique within a transition system.

Comparison with the conceptual model

There are some slight differences to the conceptual model. According to Definition 2 in Section 4.3, a state-transition system $\sum$ is a triple $\sum = (o, S, T)$. The set of states S matches very well to the “states” closure whose body consists of a sequence of “state” closure calls. The same accounts for the set of Transitions T which is described in the same manner in Subsection 5.2.2. The only tuple element of $\sum$ that cannot be found directly in the DSL is the domain object o to which $\sum$ is associated. In the conceptual model every domain object o has its own transition system $\sum$. In the worst case we have to specify the same state transition system for different domain objects if they have the same behavior. This redundancy is convenient for a theoretical model because it simplifies the conceptual model. But it is not practicable for the DSL. It should avoid redundant specifications and facilitate the reuse of transition systems where possible as demanded by Requirement *R10* in Section 2.6. This is achieved by using the “valid” closure and the “target” map in order to select a set of objects for which a transition system accounts.

Advantages of the DSL

The DSL shown in Listing 5.8 enables the reuse of state-transition systems without specifying them twice. For example, the transition system given in Listing 5.8 accounts for all subclasses of “AI_GroundedExecutionService”. Looking at the UML model of the planning domain in Figure 2.4 in Section 2.5, we can see that the transition system accounts for all three subclasses of “AI_GroundedExecutionService”, i. e., the classes describing a database, central-instance, and dialog-instance service. Another advantage of the hybrid approach is that we can directly write Groovy code into the body of the “valid” closure

in order to check whether a domain object is associated with the state-transition system. This enables us to write more complicated associations not only based on classes. For example, we can assign different state-transition systems depending on the value of an attribute or the associations of a domain object.
All in all, the DSL shown in Listing 5.8 introduces the notion of states (compare Requirement *R6* in Section 2.6) and enables the domain writer to make reuse of the behavior specification of domain objects (Requirement *R10*).

5.2.2. Describing transitions

This subsection describes the body of the "transitions" closure in more detail. Listing 5.10 shows an example of the body of a "transitions" closure which describes the "transitions" closure left empty in Figure 5.8, line 11. The given example adheres to the EBNF defined in Figure 5.2.

Figure 5.2.: EBNF for transitions and subtasks

(1) `Transitions_closure` =	'transitions{' { `Transition_closure` } '}' ;
(2) `Transition_closure` =	'transition(' `Transition_name` ' , [from : ' `State_name` ', to : ' `State_name` ']) {' `Transition_body` '}' ;
(3) `Transition_name` =	(* a Java string containing the name of the transition. The name needs to be unique within the transition system. *);
(4) `State_name` =	(* a Java string representing the name of a state. It needs to match the name of a state produced by rule 8 in Figure 5.1. *);
(5) `Transition_body` =	`Subtask_closure` `Dependencies_closure` ;
(6) `Subtask_closure` =	'subtask{' `Subtask_closure_body` '}' ;
(7) `Subtask_closure_body` =	(* Groovy Code with a call to at most one undefined method describing the task to solve when executing the transition. The identifier "it" points to the domain object the state-transition system is associated to. *);
(8) `Dependencies_closure` =	(* See EBNF rule 1 in Figure 5.3 for definition. *) ;

Listing 5.10: The part of the DSL describing transitions

```
transitions{
  transition("install" , [from : "not_installed", to : "installed"]) {

    subtask {
      install_AI_GES on : it
    }

    dependencies {...}
  }
}
```

The body of a "transitions" closure always consists of any number of "transition" closure invocations (see rules 1-2, Figure 5.2). If there are no transitions, the body of the

"transitions" closure is left empty. The "transitions" closure itself is non-optional. A "transition" closure takes three parameters. The first one is a string giving the transition a name. The name of the transition needs to be unique within the state-transition system. See EBNF rule 3 for a description. The second parameter is a map as described by EBNF rule 2. It has the two keys "from" and "to". The values of these keys need to be strings describing the name of the states the transition originates and leads to. The strings need to match to states that have been previously defined within the body of a "state" closure. This is done by rules 7 and 8 of the EBNF in Figure 5.1. The third parameter of the "transition" closure is a closure which we call the body of the "transition" closure. The content of the body is described by EBNF rule 5 in Figure 5.2. The body of a "transition" closure always consist of the invocation of a "subtask" and a "dependencies" closure. The "subtask" closure describes at most one atomic, non-atomic, or reserved change request which is to be solved when doing the transition. This task is described by a call to an undefined Groovy method. The name of this method describes the name of the change request. Within the "subtask" closure "it" references the domain object the transition system is associated to. This is described by rule 7 in Figure 5.2. The example given in Listing 5.10 shows a method call to the Groovy method "install_AI_GES" in line 6. By convention calls to an unknown method always have one parameter, a map. This map needs to have at least a key called "on". The "on" key holds a reference to the domain object which will be set as the target of the newly created change request (compare class diagram in Figure 2.4 in Section 2.5). Besides this reserved key value pair there can be other key value pairs describing additional parameters of a change request. The key value pairs modeling parameters of the HTN method can then be stored in the "params" map of the created "AI_ChangeRequest" object.
The second closure invocation in the body of the "transition" closure is a call to the "dependencies" closure. The body of this closure is defined in Subsection 5.2.3.

Comparison with the conceptual model

Definition 4 of the conceptual model defines a transition t as a tuple $t = (t\_name, start, goal, task, D)$. t_name, the name of the transition, matches the first parameter of the "transition" closure. The start and goal state are covered by the second parameter of the "transition" closure, the map with the reserved keys "from" and "to". The whole "subtask" closure invocation models $task$, the task to be solved when performing transition t. There is only a minor difference to the conceptual model. In the conceptual model each domain object has its own state-transition system, thus its own transitions and thus its own dependencies. The DSL describes this in a more general way. State-transition systems are matched to a set of domain objects. As dependencies are dependent on the type of a domain object, there needs to be a construct to find out whether a dependency accounts for a concrete instance of a state-transition system. This is done by an additional "valid" closure belonging to a "dependency" closure as explained in Subsection 5.2.3. All in all, the set of dependencies D from the conceptual model can be matched to the information described by the "dependencies" closure.

Advantages of the DSL

The part of the DSL introduced in this section has the advantage that it clearly separates domain object behavior from hierarchical problem solving behavior. The behavior of domain objects is described by the sates and the transitions between them. By specifying an atomic, non-atomic, or reserved change request in the “subtask” closure we can separate hierarchical knowledge from behavior knowledge. If we want to know exactly how the “install” transition is achieved, we need to have a look at the methods or operators that treat the “install_AI_GES” task. Note that if we chose to associate a non-atomic task with the transition, there can be more than one method decomposing this task. This enables us to specify a domain where different domain objects have the same state-transition systems but the transitions are achieved in different ways. For example, all Grounded Execution Services have the same state-transition system but the “stop” transition in a database is differently refined from the “stop” transition in a central-instance. We do not mix the refinement strategies with the description of the behavior as it is done in other approaches like [46]. This gives us great flexibility to describe state-transition systems accounting for more than one domain object (compare Subsection 5.2.1) and different hierarchical refinement behavior for the same transition. Both is clearly separated from each other in the DSL and the conceptual model.

5.2.3. Describing dependencies

Dependencies are part of a transition and they need to be fulfilled in order to execute a transition. Listing 5.11 shows the DSL describing the dependencies for the “install” transition of a Grounded Execution Service. It fills the body of the “dependencies” closure left empty in Listing 5.10, line 8. The EBNF defining the syntax is shown in Figure 5.3.

Figure 5.3.: EBNF for dependencies

(1) `Dependencies_closure` =	'dependencies {' { `Dependency_closure` } '}' ;
(2) `Dependency_closure` =	'dependency{' `Name_closure` `Valid_closure_dep` `Subtasks_closure` '}' ;
(3) `Name_closure` =	'name {' `Name_closure_body` '}' ;
(4) `Name_closure_body` =	(* A string describing the name of the dependency. *) ;
(5) `Valid_closure_dep` =	'valid{' `Valid_closure_dep_bo` '}' ;
(6) `Valid_closure_dep_bo` =	(* Groovy code returning a boolean value describing whether the dependency is valid for the domain object belonging to the transition system. The domain object is referenced by the “it” identifier. *);
(7) `Subtasks_closure` =	'subtasks {' `Subtasks_closure_body` '}' ;
(8) `Subtasks_closure_body` =	(* Groovy code doing calls to reserved change request, i. e., the “Classical_Planning_problem” method. “it” within the closure references the domain object the transition system is associated to. *);

The body of the “dependencies” closure consists of $n \in \mathbb{N}_0$ “dependency” closure invocations (compare EBNF rule 1 in Figure 5.3). According to EBNF rule 2, the body of

Listing 5.11: DSL describing dependencies for the “install” transition

```
1  dependencies {
2
3   dependency {
4    name {"install_CI_before_DI"}
5    valid {it.type == "DI"}
6    subtasks {
7      for (AI_GroundedExecutionService ci : it.get_SystemModel().
8      .get_all_AI_GES("CI")) { Classical_Planning_problem on: ci,
9      goal_state : "installed" }
10   }
11  }
12
13  dependency {
14   name {"install_DB_before_CI"}
15   valid {it.type == "CI"}
16   subtasks {
17     for (AI_GroundedExecutionService ci : it.get_SystemModel().
18     .get_all_AI_GES("CI")) { Classical_Planning_problem on: ci,
19     goal_state : "installed" }
20   }
21  }
22
23 }
```

a “dependency” closure is a sequence of three non-optional closure invocations. First of all, the “name” closure is invoked. It takes only one parameter, a closure that returns a string. Compare rules 3 and 4 of the EBNF in Figure 5.3. This string models the name resp. the textual description of the dependency. It needs to be unique within the state-transition system. The second closure is the “valid” closure. It is similar to the “valid” closure introduced in Subsection 5.2.1, Listing 5.8, line 3. The closure there decides whether a state-transition system is valid for a domain object or not. The “valid” closure invoked within the body of the “dependency” closure returns a boolean value. This value determines whether the dependency holds in a state-transition system that is associated to a concrete instance of a domain object. Within the closure “it” references this domain object. For example, the first “dependency” closure in Listing 5.11 in lines 3-11 is only valid for state-transition systems associated to an “AI_GroundedExecutionService” of type dialog-instance. This constraint is checked by examining the value of the attribute “type” of the Grounded Execution Service. It needs to be set to “DI”. Thus, the “install_CI_before_DI” dependency only accounts for “install” transitions in dialog-instances. When the body of the “valid” closure in line 5 in Listing 5.11 is executed by the planning algorithm, it passes the domain object the transition system is associated to, to the closure as a parameter. The last closure in the body of the “dependency” closure is the “subtasks” closure. It is described by EBNF rule 7 in Figure 5.3. Within the closure only calls to a Groovy method describing a Classical Planning problem are allowed to be done. Thus, only calls to the Groovy method “Classical_Planning_problem” are allowed to be

done. The method calls describe the Classical Planning problems that need to be solved in order to execute the transition. The Groovy code written in the bodies of the "subtasks" closures in Listing 5.11 is a little bit more complicated than the previous examples. The "install_CI_before_DI" dependency uses a for loop in order to generate the new dependency change requests (see lines 7-8 in Listing 5.11). As usual the identifier "it" references the domain object the transition system is associated to. Due to the value of the "target" key (compare Subsection 5.2.1, Listing 5.8, line 1), we can be sure that the code is only executed with "it" being an instance of class "AI_GroundedExecutionService". Lets assume calling the methods "get_SystemModel()" and "get_all_AI_GES("CI")" on any "AI_GroundedExecutionService" in a row delivers a list containing all central-instance services of an SAP system. Note that according to the UML class diagram in Figure 2.4 in Subsection 2.5 these method calls are well defined. We can iterate over this list and create new change requests by doing calls to unknown methods.
Each call to the "Classical_planning_problem" method shown in lines 8-9 in Listing 5.3 describes a reserved change request. In the map passed to this method, the key "on" describes the target of the change request. According to Definition 3 a reserved change request also holds a parameter that describes the goal state of the Classical Planning problem. We specify this goal state by the value of the key "goal_state". For example, the goal state of the reserved change requests created in line 9 in Listing 5.11 is the "installed" state. All in all, lines 7-9 in Listing 5.11 describe a Classical Planning problem for every dialog-instance present in the SAP system. Each of the problems demands to set the state of a dialog-instance to "installed".
All "dependency" closures and method calls to the undefined "Classical_Planning_problem" method in Listing 5.11 can be directly matched to entries in the tables shown in Section 2.3. The first dependency given in the DSL is due to the entry in the "install" row of Table 2.3. The second "dependency" closure originates from the "install" entry of Table 2.2.

Comparison with the conceptual model

The conceptual model defined a dependency d as pair $d = (d\_name, Tasks)$. The description d_name of a dependency can be directly mapped to the "name" closure. $Tasks$, the set of reserved change requests to be solved in order to achieve dependency d, is described by the calls to the 'Classical_Planning_problem" method in the "subtasks" closure. The "valid" closure from Listing 5.11 cannot be found in the conceptual model. In the conceptual model each domain object has its own transition system, which has its own transitions and each transition has its own dependencies. This means that each dependency from the conceptual model belongs to exactly one transition system. In the DSL a transition system can account for more than one domain object. Nevertheless, not all dependencies should account for all of these domain objects as shown by the example given in Listing 5.11. The "valid" closure specifies for which domain objects the dependency holds when a transition is executed. It is not needed in the conceptual model due to the decisive mapping between dependencies and domain objects used there.

Advantages of the DSL

The DSL for dependencies introduced in this subsection comes with several advantages. First, we can specify different dependencies for the same transition. The dependencies only account if their "valid" closure is satisfied over the domain object. This enables us to specify one state-transition system for a set of domain objects that have different dependency behavior. In our example all services in the SAP domain have the same transition system, i. e., the same basic behavior. But their behavior differs in the dependencies the different services need to take into account. The DSL of the hybrid approach enables us to reuse the state-transition system and to describe the dependencies on a finer grained level. The example given in Listing 5.11 shows how different dependencies for domain objects with the same states and transitions can be specified. We can reuse the dependencies, states, and transitions as demanded by Requirement *R10*. Second, the DSL and the hybrid approach offer a semantical concept to specify dependencies. They are specified as Classical Planning problems. Dependencies are made explicit by referring to the state of domain objects as demanded by Requirement *R4* and *R6*.

5.3. A DSL to describe HTN methods

This section describes the DSL for HTN methods. Listing 5.12 gives an example of a method that decomposes a change request called "stop_software" into a sequence of two change requests "stop_AI_GES" and "backup_database". Both have the same target as the "stop_software" change request. Figure 5.4 shows the EBNF rules that describe the syntax of a method.

Figure 5.4.: EBNF description for HTN methods

(1) `Method` =	'method(' `CR_name_m` ', target : ' `Classname_method` ') {' `Precondition_m` `Subtasks_clo_m` '}' ;
(2) `CR_name_m` =	(* a valid Java string representing the name of a non-atomic change request that can be decomposed by the method. *);
(3) `Classname_method` =	(* Name of the class the change request needs to have as a target to do the decomposition. *);
(4) `Precondition_m` =	'precondition {' `Precondition_m_b` '}' ;
(5) `Precondition_m_b` =	(* Groovy code returning a boolean value describing whether this method can be applied to decompose the task the method is applied to. The identifier "it" within the closure references the change request to decompose by the method. *);
(6) `Subtasks_clo_m`	'subtasks {' `Subtasks_clo_m_bo` '}' ;
(7) `Subtasks_clo_m_bo`	'sequential {' `Method_tasks` '}' | 'parallel {' `Method_tasks` '}' ;
(8) `Method_tasks`	(* Valid Groovy code doing calls to atomic, non-atomic, or reserved change requests. The identifier "it" references the change request the method is applied to. *);

An HTN method is defined by a "method" closure. This closure takes three parameters as described by EBNF rule 1 in Figure 5.4. The first one is a string describing

Listing 5.12: DSL describing an HTN method

```
1  method("stop_software", target : AI_GroundedExecutionService) {
2
3   precondition {it.get_target().type == "DB"}
4
5   subtasks {
6    sequential {
7     stop_AI_GES on : it.get_target()
8     backup_database on : it.get_target()
9    }
10  }
11
12 }
```

the name of the non-atomic change request the method can decompose. Listing 5.12 gives an example of a method that can decompose a non-atomic change request called “stop_software”.
The second parameter is similar to the first parameter of the “transition_system” closure as introduced in Listing 5.8 in Subsection 5.2.1. It is a map with key “target” and the name of a class as its value. The planning algorithm uses this map for a sanity check. Each “AI_ChangeRequest” object references a target object. An algorithm takes the change request object and tries to find a method to decompose the task. The target map specifies the class which the object referenced by the target attribute needs to be an instance of. In the given example the target of the change request needs to be an instance of class “AI_GroundedExecutionService”. From an algorithmic point of view it is not necessary to specify this class name. Nevertheless, it makes the writer of a domain description aware of where “it” points to in the “sequential” or “parallel” closure (lines 6-9), namely an instance of class “AI_GroundedExecutionService”, the target of the change request.
The third parameter is a closure. According to rule 1 in Figure 5.4 the closure always consists of calls to two non-optional closures, the “precondition” and the “subtasks” closure. Similar to the previous occurrences of a “valid” closure, the “precondition” closure returns a boolean value, specifying whether the method can be applied to decompose a change request depending on its target object. “it” within the closure references the change request the method is applied to. The body of a “subtasks” closure consists either of the invocation of a “sequential” or “parallel” closure. Compare rules 6-7 in the EBNF defined in Figure 5.4. There are no restrictions as to which method calls are allowed to be made in the body of one of the closures. Recall that there were restrictions imposed for the method calls within a “dependency” closure in Subsection 5.2.3. The subtasks described by method “stop_software” are “stop_AI_GES” and “backup_database” (compare lines 7-8 in Listing 5.12). Both change requests do not have parameters. The target of the new child change requests is set to the same target as the target of the “stop_software” change request, which is decomposed by the method. Note that it is possible to access the parameters of the “stop_software” change request within the body

of the “precondition” and the “sequential” / “parallel” closure. Because “it” references the “stop_software” change requests, its parameters can be accesses by examining its “params” attribute. Thus, we can describe preconditions depending on the values of the parameters of the change request.
As we do not offer unification capabilities, the DSL needs to offer additional computational constructs to describe different possible decompositions implied by one method. For example, consider a method that deploys a virtual machine on a physical machine. As there are many possibilities how to deploy the VM on the different machines, we need to describe different decompositions by which the high-level task can be achieved. In logic based HTN planners this is done by unification which binds parameters of a task to different values. The planner then tries to satisfy the subtasks described by any of these bindings. To model this in an object oriented knowledge base, we introduce the “try_all” closure. The “try_all” closure can be invoked within the body of the “sequential” / “parallel” closure. Within the body of the “try_all” closure the identifier “it” references the change request to decompose. The body of the “try_all” closure can generate many change requests. One of these change requests needs to be satisfied in order to finish the task described by the “try_all” closure.

Listing 5.13: “try_all” closure describing alternatives in the decomposition

```
subtasks {
 sequential {
  try_all {
   CR1 on : it.get_target()
   CR2 on : it.get_target()
  }
  CR3 on : it.get_target()
 }
}
```

Listing 5.13 shows a “subtasks” closure of a method that makes use of the “try_all” closure. The decomposition described is sequential. The “subtasks” closure describes a decomposition into two change requests. First of all either “CR1” or “CR2” needs to be successfully resolved. After that, change request “CR3” needs to be solved. All in all, the “subtasks” closure says that either “CR1”, “CR3” or “CR2”, “CR3” needs to be achieved in order to successfully decompose the high-level task.

Comparison with the conceptual model

Definition 7 defines a method m as a triple $m = (m\_task\_name, m\_pre, m\_decomp)$. m_task_name describes the name of the task which can be decomposed by the method. It directly matches to the first parameter of the “method” closure. m_pre is defined as a precondition evaluated over the target of the change request. This matches the “precondition” closure in the body of the method. The DSL goes a step further than the conceptual model by introducing the “target” map (see line 1 in Figure 5.12, second parameter of “method” closure). The algorithm only evaluates the precondition if the target

of the change request is a subclass of the provided class name. This is not more powerful than what has been described in the conceptual model. The DSL uses a compound precondition that first checks whether the domain object is an instance of a certain class and then evaluates the "precondition" closure over the object. As *m_pre* from the conceptual model is an arbitrary precondition evaluated over the target, every compound precondition (class selector + "precondition" closure) from the DSL adheres to the precondition of a conceptual model. Thus, every DSL combination of a target class and a "precondition" closure can be expressed as a precondition consistent with the conceptual model.
The other direction holds, too. Every precondition *m_pre* evaluated over the target of the change request can be written as a DSL precondition consisting of a class selector statement and a "precondition" closure. No matter whether *m_pre* consists of class selection statements or not, it is always possible to set the target attribute to "Object" and directly use *m_pre* as the body of the "precondition" closure. Thus, the DSL has the same expressiveness as the conceptual model when defining preconditions.
The last element from the triple, *m_decomp*, is a decomposition to solve the top-level task. It can be directly matched to the "subtasks" closure.

5.4. A DSL to describe HTN operators

This section describes how HTN operators are formalized in the DSL. Listing 5.14 shows the DSL describing an operator. The example follows the EBNF given in Figure 5.5.

Figure 5.5.: EBNF describing HTN operators

(1) `Operator` =	'operator(' `CR_name_atomic` ', target : ' `Classname_action` ') {' `Precond_o` `Effects_o` '}' ;
(2) `CR_name_atomic` =	(* A string describing the name of the atomic change request the operator can be applied to. *);
(3) `Classname_action` =	(* Name of a class the atomic change request needs to have as a target to apply the operator. *);
(4) `Precond_o` =	'precondition{' `Precond_o_body` '}' ;
(5) `Precond_o_body` =	(* Groovy code returning a boolean value describing whether this operator can be applied to the atomic task. The identifier "it" within the closure references the atomic change request the operator is applied to. *);
(6) `Effects_o` =	'effects {' `Effects_o_body` '}' ;
(7) `Effects_o_body` =	(* Groovy code changing the model. The identifier "it" references the domain object, which is the target of the atomic change request. *);

An operator in the hybrid approach gives a semantics to an atomic task. It describes how the knowledge base is altered by the atomic change request. The HTN operator is described by a closure called "operator". Its parameters are very similar to the ones of a method (compare Subsection 5.3). They are described by EBNF rule 1 in Figure 5.5. The first parameter is a string representing the name of the atomic change request the operator can be applied to (compare rule 2 in Figure 5.5). As with methods, the

Listing 5.14: DSL describing an HTN operator

```
1 operator("stop_AI_GES", target : AI_GroundedExecutionService) {
2
3  precondition {it.get_target().state == "running"}
4
5  effects {it.get_taget().stop()}
6
7 }
```

second parameter describes the class the target of the atomic change request needs to be an instance of in order to apply the operator. The body of the “operator” closure consists of calls to a “precondition” and an “effects” closure. The “precondition” closure from Listing 5.14, line 3 has the same syntax as the “precondition” closure already known from methods.
The second closure that is called within the body of the “operator” closure is the “effects” closure. Its syntax is described by EBNF rules 6 and 7 in Figure 5.5. The “effects” closure takes a closure as a parameter describing the code to implement the effects of the operator on the model. Whenever the algorithm comes across an atomic change request, it searches for an operator to implement the change request. It will then execute its “effects” closure in order to make the changes persistent to the knowledge base. Within the “precondition” and “effects” closures, “it” references the change request the operator is applied to. In the given example the operator calls the “stop()” method on the “AI_GroundedExecutionService” which is the target of the “stop_AI_GES” change request.

Comparison with the conceptual model

The conceptual model defines an operator *op* as a triple $op = (op\_task\_name, op\_pre, effects)$ where op_task_name is the name of the operator. It is implemented by the first parameter of the “operator” closure. op_pre is modeled by the target map, the second parameter given to the “operator” closure, and the “precondition” closure. This follows exactly the same idea which is used to describe the combined precondition of methods. In addition to that, the same observations regarding the expressiveness of the preconditions of the conceptual model and the combined preconditions of the DSL are holding. The last element *effects* from the triple describes the effects of the operator when applied to an atomic change request. It can be directly mapped to the “effects” closure.

6. The hybrid HTN/Classical Planning approach

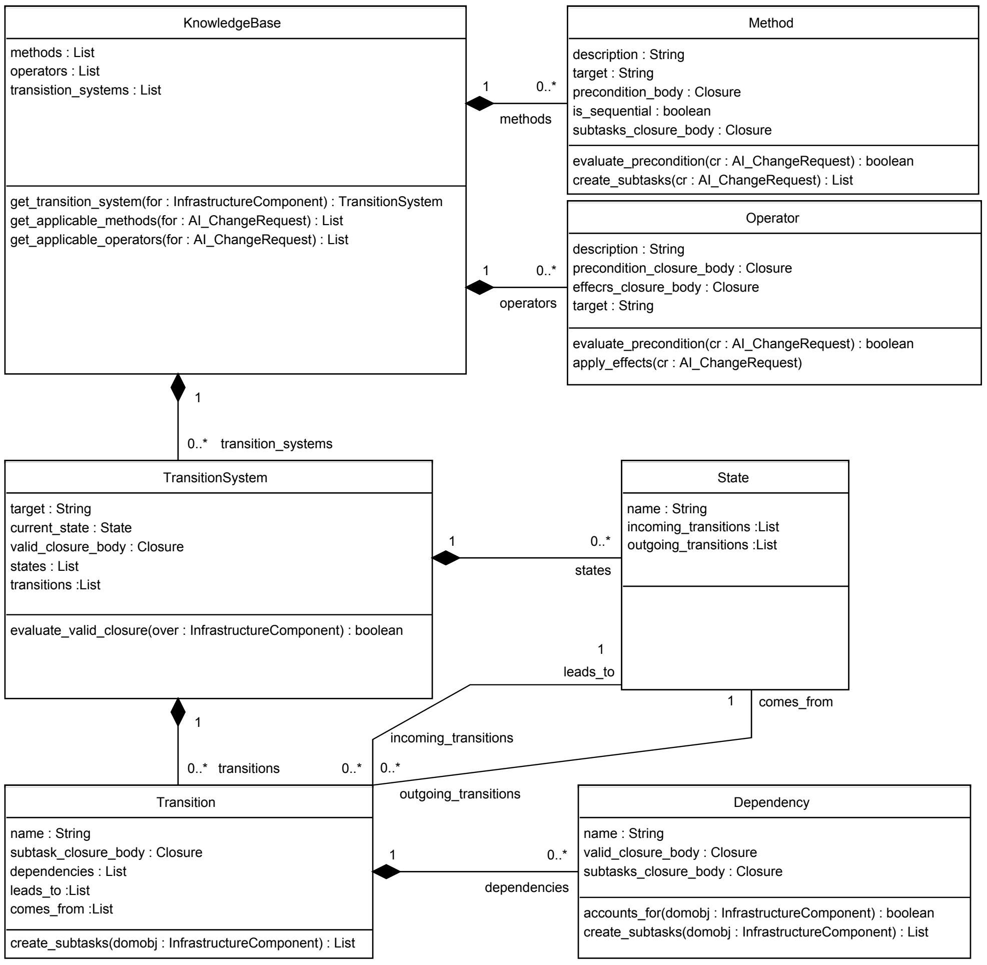
This chapter introduces the algorithm of the hybrid approach and explains advanced concepts like dependencies among tasks and consistency of plans. We start by explaining the model describing the knowledge base of the planner in Section 6.1. It contains the contents defined by an instance of the DSL. The planner uses the model for planning together with the previously introduced model of the infrastructure in Section 2.5. Having introduced the model, we explain the algorithm of the hybrid approach by means of language independent pseudo code in Section 6.2. After that, we give an example in Section 6.3 showing how the algorithm solves a planning problem. Finally, we cover more advanced topics like dependencies between change requests, consistency of plans, and the detection of plan inconsistencies in Section 6.4.

6.1. Model of the knowledge base defined by the DSL

This section explains the structure of the knowledge base used by the planner. It puts the concepts the DSL can express in an object oriented model. The planner uses this model together with the model describing the infrastructure and change requests as introduced in Subsection 2.5 to do the planning. Figure 6.1 shows the UML class diagram describing the knowledge base of the planner.

The "KnowledgeBase" class aggregates the operators, the state-transition systems, and the methods that are defined by the DSL describing the planning domain. The "Method" class describes an HTN method in the Hybrid Planning domain. It holds an attribute "description" which stores the description / name of the method. This name equals the string specified in the first parameter of the "method" closure in Listing 5.12 in Section 5.3 (in case of the given example method its value matches "stop_software"). The "target" attribute of the "Method" class holds the name of a class. The change request that is to be decomposed by this method needs to have a target which is an instance of this class. In the example for a method given in Section 5.3 the value is "AI_GroundedExecutionService". A method described by the DSL also consists of a "precondition" closure. The attribute "precondition_body" holds the body of this closure. In case of the provided example method in Listing 5.12 in Section 5.3 it is the closure "{it.get_target().type == "DB"}". Because the decomposition described by the method is sequential, the value of attribute "is_sequential" is set to true. The closure "subtasks_closure_body" stores the body of either the "sequential" or "parallel" closure. In case of the example given in Section 5.3 it is the closure "{stop_AI_GES on : it.get_target() backup_database on : it.get_target()}". In addition to the attributes, the class provides methods to

Figure 6.1.: UML class diagram describing the knowledge base of the planner


KnowledgeBase
methods : List
operators : List
transistion_systems : List
get_transition_system(for : InfrastructureComponent) : TransitionSystem
get_applicable_methods(for : AI_ChangeRequest) : List
get_applicable_operators(for : AI_ChangeRequest) : List
Method
description : String
target : String
precondition_body : Closure
is_sequential : boolean
subtasks_closure_body : Closure
evaluate_precondition(cr : AI_ChangeRequest) : boolean
create_subtasks(cr : AI_ChangeRequest) : List
Operator
description : String
precondition_closure_body : Closure
effecrs_closure_body : Closure
target : String
evaluate_precondition(cr : AI_ChangeRequest) : boolean
apply_effects(cr : AI_ChangeRequest)
1 0..* methods
1 0..* operators
1 0..* transition_systems
TransitionSystem
target : String
current_state : State
valid_closure_body : Closure
states : List
transitions :List
evaluate_valid_closure(over : InfrastructureComponent) : boolean
State
name : String
incoming_transitions :List
outgoing_transitions :List
1 0..* states
1 leads_to
1 comes_from
incoming_transitions
0..* 0..*
outgoing_transitions
1 0..* transitions
Transition
name : String
subtask_closure_body : Closure
dependencies : List
leads_to :List
comes_from :List
create_subtasks(domobj : InfrastructureComponent) : List
1 0..* dependencies
Dependency
name : String
valid_closure_body : Closure
subtasks_closure_body : Closure
accounts_for(domobj : InfrastructureComponent) : boolean
create_subtasks(domobj : InfrastructureComponent) : List

evaluate the closures at planning time. The method "evaluate_precondition()" takes an "AI_ChangeRequest" and evaluates the closure stored in "subtasks_closure_body" over this change request. Thus, the algorithm can decide whether a method can be applied to decompose a given change request. Furthermore, the "create_subtasks()" method can be used to create the subtasks of a change request when applying the HTN method to it. This is done by executing the closure "subtasks_closure_body" with the change request passed to it. It returns a list of lists of change request, where each list holds one possible decomposition of the method applied to the change request. The "Operator" class is similar to the "Method" class thus its description is omitted.
A "KnowledgeBase" also aggregates many "TransitionSystem" objects, each representing the data described by a "transition_system" closure. For example, the attribute "valid_closure_body" holds the body of the "valid" closure. In case of the example given in Listing 5.8 in Subsection 5.2.1 its value is the closure "{true}". The "evalu-

ate_closure_body()" method enables us to evaluate this closure over an "Infrastructure-Component" in order to find out whether the transition system can be associated with the domain object. A transition system consists of many "State" and many "Transition" objects. The attribute "subtask_closure_body" in the "Transition" class stores the body of the "subtask" closure, i. e., the one shown in lines 5-6 in Listing 5.10 in Subsection 5.2.2. Be aware that the class also offers a method to create the subtasks from this closure. A "Transition" aggregates many "Dependency" objects, each modeling a dependency nested within a "transition" closure as shown in Listing 5.11 in Subsection 5.2.3. The attributes of the "Dependency" class are pretty straight forward. The method "accounts_for()" uses the body of the "valid" closure to decide whether a dependency accounts for a particular domain object. The "create_subtasks()" method is similar to the one used in a "Transition" object. It creates the subtasks described by the dependencies by executing the "subtasks_closure_body" closure.
All in all, the class diagram in Figure 6.1 puts the concepts described by the DSL into an UML class diagram which describes the characteristics of the knowledge base. It does not only store the contents described by the DSL but adds methods that can be used by a Groovy implementation of the planner to conveniently do the planning.

6.2. The algorithm of the hybrid approach

This section introduces the algorithm of the hybrid approach by means of language independent pseudo code. First, we explain a hybrid approach planning method in Subsection 6.2.1 which delegates the planning task either to the HTN or Classical planner. After that, the HTN Planning algorithm is explained in Subsection 6.2.2 followed by the algorithm of the Classical planner in Subsection 6.2.3.

6.2.1. Hybrid approach top-level algorithm

This subsection gives an overview of the top-level planning method of the hybrid approach. It delegates a planning task either to an HTN planner or a Classical planner depending on the type of the first task passed to the function.

Algorithm 1 : $Hybrid\_Planner(< t_1, ..., t_n >, O, M)$

```
1: if $n = 0$ then
2:    return $<>$
3: else
4:    if $t_1$ is atomic || $t_1$ is non-atomic then
5:       return $HTN\_Planner(< t_1, ..., t_n >, O, M)$
6:    else
7:       return $Classical\_Planner(< t_1, ..., t_n >, O, M)$
8:    end if
9: end if
```

Algorithm 1 shows the pseudo code of the "Hybrid_Planner()" method. It takes three

parameters, an ordered sequence of tasks $< t_1, ..., t_n >$, $n \in \mathbb{N}_0$, a set of HTN operators O, and a set of HTN methods M. If there are no tasks to plan for, then the algorithm returns the empty plan in line 2. Otherwise, there is at least one task to plan for. We then need to check whether the first task is an atomic or non-atomic change request. In the latter case, the HTN planner is called on the same planning problem and its generated plan is returned in line 5. Otherwise, t_1 is a reserved change request describing a Classical Planning problem which needs to be solved by the Classical Planner. Thus, we return the plan given back by the invocation of the Classical planner in line 7.
The "Hybrid_Planner()" method can be used if we do not exactly know whether the first element in the task queue is an HTN or Classical Planning task. It will then delegate the call to the appropriate planner. Nevertheless, the HTN and Classical planner also delegate a call to the other planner if they cannot plan for the first task in the queue.

6.2.2. HTN Planning algorithm

This subsection introduces the HTN algorithm used by the hybrid approach on a high-level. We introduce the algorithm using pseudo code. The HTN planner works together with the Classical planner in order to solve a planning problem in the hybrid approach. Be aware that the following description of the algorithm is only declarative, i. e., it is only stated what to do but not how to do it.

Algorithm 2 shows the pseudo code of the HTN algorithm. It follows the principle ideas of the Classical partial-order *Simple Task Network* (*STN*) depth-first search algorithm introduced in [22]. Like the "Hybrid_Planner()" method, the "HTN_Planner()" method is called with the same parameters. These comprise a set of tasks to plan for, i. e., atomic, non-atomic, or reserved tasks, a set of operators O, and a set of methods M. Be aware that normally a knowledge base is passed to the planning method, too. We do not do this because according to our domain model (see Figure 2.4 in Subsection 2.5) a task holds a reference to a target object belonging to the knowledge base. In addition to that, we assume that the knowledge base as described in Subsection 6.1 is globally available within both planning methods.
If the list of change requests to plan for is empty, we return the empty plan in line 2 in Algorithm 2. In case there is at least one element in the list of passed tasks we need to check whether the first one, i. e., t_1, is a reserved change request. If so, we need to call the Classical planner and return the plan computed by it because this planner deals with reserved change requests. In case t_1 is a non-atomic / atomic task we need to distinguish whether to plan for a method decomposition or the application of an operator. We first explain the behavior of the algorithm when planning for a method decomposition.

Behavior regarding non-atomic change requests

The pseudo code describing the planning for non-atomic tasks is shown in lines 8-22 in Algorithm 2. To decompose the non-atomic change request t_1, we need to find a method that can decompose t_1. M, the set of methods, is passed to the HTN Planning method

Algorithm 2 : $HTN\_Planner(< t_1, ..., t_n >, O, M)$

```
1: if $n = 0$ then
2:   return $<>$
3: end if
4: if $t_1$ is reserved change request then
5:   return $Classical\_Planner(< t_1, ..., t_n >, O, M)$
6: end if
7: if $t_1$ is non-atomic then
8:   Let $M' \subseteq M$ be the set of methods applicable to $t_1$.
9:   if $M' =<>$ then
10:     return $failure$
11:   end if
12:   Pick a method $m \in M'$ non-deterministically to do the decomposition.
13:   Let $Subtasks$ be a set of sets, where each set $S =< s_1, ..., s_k >$, $k \geq 0$, $S \in$
      $Substasks$ holds one possible sequence of subtasks that need to be achieved in
      order to achieve $t_1$.
14:   Let $snap$ be a snapshot of the current knowledge base.
15:   for $S =< s_1, ..., s_k > \in Subtasks$ do
16:     $plan \leftarrow Hybrid\_Planner(< s_1, ..., s_k > \circ < t_2, ..., t_n >, O, M)$
17:     if $plan! = failure$ then
18:       return $plan$
19:     end if
20:     revert to snapshot $snap$.
21:   end for
22:   return $failure$
23: else
24:   Let $O' \subseteq O$ be the set of operators applicable to $t_1$.
25:   if $O' =<>$ then
26:     return $failure$
27:   end if
28:   Pick an operator $o \in O'$ non-deterministically.
29:   Apply the effects described by operator $o$ to the model.
30:   $plan \leftarrow Hybrid\_Planner(< t_2, ..., t_n >, O, M)$
31:   if $plan = failure \mid\mid (t_1 \circ plan)$ $is\ inconsistent$ then
32:     return $failure$
33:   else
34:     return $t_1 \circ plan$
35:   end if
36: end if
```

as a parameter. $M' \subseteq M$ is the set of methods applicable to t_1 (line 8). A method $m \in M$ is called applicable to a task t_1 if the following three conditions are met:

- The description of t_1, i. e., the name of the non-atomic change request, equals the description of method m.
- The target of the non-atomic change request t_1 is an instance of the class described by the target attribute of method m.
- The precondition of m is satisfied by the parameters and the target of t_1.

If M', the set of methods that can be applied to decompose t_1, is empty, failure is returned in line 10. This is usual HTN behavior as known by other HTN planners like SHOP2 [37] where non-atomic tasks that cannot be decomposed are regarded as a failure. If M' is not empty we nondeterministically choose a method $m \in M$ to decompose t_1 in line 12. Be aware that we do not try another method if the decomposition using m fails. Thus, the presented version does not consider backtracking on failure. Note that by leaving backtracking aside the shown version of the HTN planner is not complete thus violating Requirement *R3* from Section 2.6. However, the backtracking technique [12] can be easily added by introducing an additional loop over the methods in M' resulting in a complete HTN planner.
Traditional HTN planners over a logic-based knowledge base, e.g. SHOP2 [37], use unification to bind multiple values to the parameters of a change request. This is particular useful when a non-atomic task is decomposed by a method in always the same children but with different parameters given to the child change request. Unification can be built over a logic based knowledge base but is particular difficult to implement over an object oriented knowledge base. Our answer to this problem is the previously introduced "try_all" closure in Section 5.3 which creates many tasks out of which one has to be successfully solved. The set *Subtasks* in line 13 is a set of sets, where each set holds a sequence of tasks which needs to be achieved in order to achieve t_1. Planning succeeds if there is at least one sequence of tasks in *Subtasks* which achieves a successful decomposition of t_1.
Before trying any of the possible decompositions described by m, a backup, also called a *snapshot*, of the knowledge base needs to be done. It enables us to return to the saved state of the planning domain if one possible decomposition did not result in a successful decomposition. The concept of snapshots was introduced by [10]. We assume that we can do snapshots at any time. The snapshot taken in line 14 is named *snap*.
The *for* loop between lines 15 and 21 tries to solve t_1 by applying different decompositions to t_1 described by the same method. It calls the 'Hybrid_Planner()" algorithm in line 16 on a new list consisting of the subtasks of the non-atomic change request and the rest of the tasks that were originally passed on the invocation of the "HTN_Planner()" method. The subtasks are added at the beginning of the list, because we first have to completely solve the subtasks of t_1. If the recursive invocation in line 16 returns a valid plan, we can return it in line 18. Otherwise, we need to revert the model to snapshot *snap* (line 20) and need to try another decomposition in the next loop. If none of the decompositions returns a valid plan, $failure$ is returned in line 22.

Behavior regarding atomic change requests

If t_1 is an atomic change request, the *if* statement in line 7 in Algorithm 2 evaluates to false and the *else* block in lines 24-35 is executed. In order to successfully plan for an atomic change request, we need to find a set of operators $O' \subseteq O$ that can be applied to the atomic change request t_1 (line 24). Similar to a method, an operator $o \in O$ can be applied to an atomic task t_1 if:

- The description of t_1, i. e., the name of the atomic task, equals the description of operator o.
- The target of t_1 is an instance of the class described by the target attribute of operator o.
- The precondition of o is satisfied by the parameters and the target of t_1.

The algorithm returns failure if there are no applicable operators in O' (line 25). Otherwise, an operator $o \in O'$ is picked non-deterministically. We can then apply the effects of the operator o to the model. The effects are described by the body of the "effects" closure of an operator. As soon as the effects are made persistent to the domain objects we can call the "Hybrid_Planner()" method on the rest of the remaining tasks in line 30. If the plan returned does not match $failure$, we can add t_1 to this plan but only if t_1 is not in conflict to the atomic tasks that are already part of the plan. To keep matters simple, we do not examine plan consistency here but defer the examinations to Section 6.4.

6.2.3. Classical Planning algorithm

This subsection introduces the planning algorithm of the Classical planner part of the hybrid approach. The Classical planner is called whenever a Classical Planning problem is to be solved, i. e., a reserved change request is at the beginning of the task queue. Algorithm 3 shows the pseudo code of the Classical planner algorithm.

The "Classical_Planner()" method takes the same input parameters as the HTN planner method introduced in Subsection 6.2.2. The parameters are a sequence of tasks $< t_1, ..., t_n >$, $n \in \mathbb{N}_0$, a set of HTN methods M, and a set of HTN operators O. The basic idea behind the Classical planner used within the hybrid approach is that it solves a slightly extended Classical Planning problem [22] within the decomposition tree of an HTN problem. As HTN problems can trigger Classical Planning problems and Classical Planning problems describe a mixture of HTN and Classical Planning problems again, a full integration between both approaches exists.

The Classical Planning method first needs to examine whether the passed sequence of tasks is empty in line 1 in Algorithm 3. If there is nothing to plan for, line 2 returns the empty plan. Otherwise, t_1, the first task in the sequence, needs to be examined. Because the Classical planner only plans for reserved change requests, the "HTN_Planner()" method needs to be called if t_1 is an atomic or non-atomic change request (compare lines 4-6 in Algorithm 3). In case the control flow continues to line 7, t_1 is a reserved change

Algorithm 3 : $Classical\_Planner(< t_1, ..., t_n >, O, M)$

```
1: if $n = 0$ then
2:   return $<>$
3: end if
4: if $t_1$ is atomic || $t_1$ is non-atomic then
5:   return $HTN\_Planner(< t_1, ..., t_n >, O, M)$
6: end if
7: Let $target$ be the target of $t_1$, $goal$ the goal state specified by $t_1$, $sts$ the transition
   system of $t_1$, and $current$ the current state of $sts$.
8: if $t_1$ violates any other reserved change request then
9:   return $failure$
10: end if
11: if $goal = current$ then
12:   return $Hybrid\_Planner(< t_2, ..., t_n >, O, M)$
13: end if
14: Be $T$ a set of transitions of $sts$ leading from $current$ to $goal$, preferably the shortest
    path.
15: $Class\_Planning\_Problem =<>$
16: $task\_to\_solve\_old = null$
17: for transition $t \in T$ do
18:   Let $D$ be the set of all dependencies described for transition t.
19:   Let $D' \subseteq D$ be the set of all valid dependencies for transition t.
20:   $All\_dependency\_tasks =<>$
21:   for dependency $d \in D'$ do
22:     Let $CRS =< cr_1, ..., cr_n >$, $n \geq 0$ be the set of all reserved change requests
        described by dependency $d$.
23:     $All\_dependency\_tasks = All\_dependency\_tasks \circ CRS$
24:   end for
25:   Be $task\_to\_solve$ the task to be solved when executing transition $t$
26:   $Add\_dependencies\_to\_children(All\_dependency\_tasks, task\_to\_solve,$
      $task\_to\_solve\_old)$
27:   $Class\_Planning\_Problem = Class\_Planning\_Problem \circ$
      $All\_dependency\_tasks \circ task\_to\_solve$
28:   $task\_to\_solve\_old = task\_to\_solve$
29: end for
30: return $Hybrid\_Planner(Class\_Planning\_Problem \circ < t_2, ..., t_n >, O, M)$
```

request. As explained in Definition 3 of the conceptual model given in Section 4.3, a reserved change request is a well defined change request. It has one fixed parameter describing the goal state the transition system linked to the target of the reserved change request needs to be brought to. This parameter is extracted in line 7. Furthermore, *current* is the current state of this transition system.
After that, we need to verify whether the reserved change request t_1 violates any other reserved change request out of the set of reserved change requests that have been planned for already. If so, failure needs to be returned. In the case no reserved change request is violated, the control flow continues to line 11. We do not describe here how possible violations look like and how they can be handled. To keep matters simple this is done in Section 6.4.
If no other reserved change requests are violated, we need to check whether the transition system is already in the goal state, i. e., $goal = current$. In this case t_1 is trivially satisfied and a recursive call to the "Hybrid_Planner()" on $< t_2, ..., t_n >$, the remaining sequence of tasks, is done. If the control flow continues to line 14 in Algorithm 3, there is at least one transition in *sts* to be taken in order to reach the goal state defined in *goal*. This means, we need to construct a plan that changes the state of *sts*. To compute the set of transitions from *current* to *goal* state, a *single-pair shortest path* problem needs to be solved. For example, the Floyd-Warshall algorithm [21] can be used prior to planning to solve the *all-pairs shortest path problem* in $O(|S|^3)$, where S is the set of states of the state-transition system. Alternatively, Dijkstra's algorithm [14] can be used to compute the shortest path between the *current* state and every other state in up to $O(log(|S|) + |T|)$ where S is the set of states and T the set of transitions. The sequence T in line 14 of Algorithm 3 holds the transitions belonging to the shortest path from *current* to *goal*.
In line 15 the variable *Class_Planning_Problem* is initialized. After the *for* loop shown in lines 17-29 ends, *Class_Planning_Problem* holds all the tasks that need to be solved in order to solve the Classical Planning problem described by t_1. This includes reserved change requests described by the dependencies linked to a transition as well as the atomic, non-atomic, or reserved change request described by the task linked to the transition. *Class_Planning_Problem* collects these change requests for each transition. In lines 17-29 the algorithm iterates over all transitions in T, the transitions of the shortest path. Within one iteration, D in line 18 is the set of dependencies linked to transition t in state-transition system *sts*. Remember that a dependency belonging to a transition accounts depending on the domain object the transition system belongs to. Thus, we have to compute $D' \subseteq D$, the set of dependencies accounting for transition t. A dependency $d \in D$ accounts for a state-transition system *sts* if its "valid" closure is satisfied over domain object *target* (see line 7 for the initialization of *target*). For each dependency d accounting for transition t we then calculate the reserved change requests that are described by the dependency. This is done in the *for* loop shown in lines 21-24. The ordered set $CRS =< cr_1, ..., cr_n >$, $n \in \mathbb{N}_0$, describes the reserved change requests that need to be fulfilled in order to solve dependency d. In line 23 we add CRS to the list *All_dependency_tasks*. *All_dependency_tasks* aggregates all dependency tasks for transition t. When the *for* loop ends in line 24, *All_dependency_tasks*

holds all reserved change requests described by all valid dependencies of transition t. Thus, it holds all the Classical Planning problems that need to be solved in order to do transition t. The Classical Planning problems are described by reserved change requests in *All_dependency_tasks*. The execution of a transition is linked to solving a task. This task, i. e., an atomic, non-atomic, or reserved change request is assigned to *task_to_solve*. Finally, we can assemble the set of tasks that need to be accomplished in order to execute transition t. These tasks include the dependency tasks stored in *All_dependency_tasks* and the task linked to the transition stored in *task_to_solve*. This is done in line 27. When planning for the next transition, the algorithm also has to remember the previous value of *task_to_solve* in order to establish proper dependencies between the tasks defined by different transitions. These dependencies are established in line 26 where the auxiliary method "Add_dependencies_to_children()" is called. Adding the right temporal dependencies among the children is a non-trivial issue. To keep matters simple here, we explain in Subsection 6.4.3 which temporal constraints need to be set among the subtasks of a reserved change request.
After the iteration over all transitions of the shortest path ends in line 29, *Class_Planning_Problem* holds all tasks that are needed in order to bring *sts* from *current* state to *goal* state. Thus, the Hybrid planner is called on the tasks contained in *Class_Planning_Problem* and the original task list not including t_1. This will trigger Classical Planning problems first if there were dependencies for the first transition and then an atomic, non-atomic, or reserved change request to implement the transition.

6.3. Example of the hybrid approach algorithm

This section gives an example of the algorithm of the hybrid approach described in Section 6.2. We assume that the planning domain consists of three instances of "AI_Grounded-ExecutionService", a database described by an instance of "AI_GES_DB", a central-instance ("AI_GES_CI"), and a dialog-instance which is an instance of "AI_GES_DI". Their current state is "running" which is described by their "state" attribute inherited from the superclass "InfrastructureComponent" as shown in Figure 2.4 in Subsection 2.5. The state-transition system for an "AI_GroundedExecutionService" is described in Listing A.1 in Appendix A. The listing also describes the methods and operators used in the example. We refer to it when necessary.
Due to readability reasons we abbreviate calls to the method "Hybrid_Planner()" by "HYP()", to method "Classical_Planner()" by "CLP()", and to "HTN_Planner()" by "HTP()". The top-level change request to be solved in the given example is a reserved change request, i. e., a change request describing a Classical Planning problem. It demands to set the state of the database to "installed", the state in which it is stopped (compare the DSL in Appendix A for the states of an AI_GroundedExecutionService). Due to the dependencies accounting in the domain as explained in Section 2.3, the dialog-instance and the central-instance need to be stopped first. In addition to that, we want to do a backup of the tables of a database, after it has been stopped. We abbreviate change requests in the form $CRname(target, param_1, ..., param_n)$ where $CRname$ is the

description of the change request, *target* the target of the change request, and $param_i$, $i \in \mathbb{N}_0$, are the parameters of the change request. If the change request is a reserved change request, then *CRname* does not hold the description of the atomic or non-atomic task but the value *CPP* for Classical Planning problem. Thus, the high-level task to solve can be written as "CPP(db,"installed")", describing a Classical Planning problem which demands to set the state of the database to "installed" (compare Appendix A for the description of the states of an AI_GroundedExecutionService). Thus we call the "Hybrid_Planner()" method with a list of tasks only containing the "CPP(db,"installed")" task:

```
HYP(<CPP(db,“installed”)>, 0, M)
```

This call is redirected to

```
CLP(<CPP(db,“installed”)>, 0, M)
```

because the first task t_1 is a reserved change request that is planned for by the Classical planner (compare Algorithm 1 in Subsection 6.2.1). t_1 is the task CPP(db,"installed"). Within Algorithm 3 in Subsection 6.2.3 *target* is set to "db", *goal* is set to "installed", and *current* holds the value "running" because the database is currently in state "running". t_1 does not violate other reserved change requests. Thus, the body of the *if* statement in line 9 is not executed. In the remainder of this section we always assume that reserved change requests are never in conflict to each other. In addition to that, the example is constructed such that the reserved change requests are not in conflict to each other. We explain in Section 6.4.3 in more detail how conflicting reserved change requests can be identified. The *goal* state is not already reached, thus line 12 is not executed. The shortest path is stored in T, i. e., T holds the value "<"stop">" because the shortest path from state "running" to "installed" consists only of the stop transition. To verify the shortest path see the definition of the state-transition system in Appendix A. Thus, the *for* loop in lines 17-29 is only executed once. In its only run t holds transition "stop" (line 17). D is the set of all dependencies described for transition t. Lines 81-102 in Listing A.1 in Appendix A show that there are two dependencies defined for transition "stop", thus D holds the value "<"stop_CI_before_DB", "stop_DI_before_CI">". $D' \subseteq D$ contains only "stop_CI_before_DB" because only this dependency is valid over a database. To see this compare the "valid" closures of the two dependencies in lines 85 and 95 in the DSL in Appendix A. Only the body of the "valid" closure of dependency "stop_CI_before_DB" evaluates to true over a database. Thus d in line 21 in Algorithm 3 in Subsection 6.2.3 can only hold the "stop_CI_before_DB" dependency. The set *CRS* contains the reserved change requests described by dependency d. As we can see in lines 87-89 in the DSL in Appendix A, these change requests are described by Classical Planning problems over all central-instances. These demand to set the state in each central-instance to "installed". Because there is only one central-instance, which we call "ci" here, *CRS* holds the value "<CPP(ci,"installed")>". By solving this Classical Planning problem we can resolve the preconditions that are necessary to execute transition t, the "stop" transition of the database. Line 23 in Algorithm 3 initializes *All_dependency_tasks* with the value "<CPP(ci,"installed")>". This variable aggregates all change requests described by all valid dependencies of transition t. In line 25 of the pseudo code *task_to_solve* is initialized with the task linked to transition t, i. e., the "stop" transition. As it can be seen in line 78 of the DSL description in Appendix A, this task is the "stop_software" task. Thus, *task_to_solve* holds the value "stop_software(db)". Finally, line 27 in the Classical planner algorithm defines the *Class_Planning_Problem* as

```
<CPP(ci,"installed"), stop_software(db)>
```

This leads to the new plan defined by the call to the Hybrid planner in line 30 of the pseudo code:

```
HYP(<CPP(ci,"installed"), stop_software(db)>, 0, M)
```

Because t_1, the first task in the list of tasks, is a reserved change request, the Hybrid planner delegates the planning to the "Classical_Planner()" method. This leads to:

```
CLP(<CPP(ci,"installed"), stop_software(db)>, 0, M)
```

The behavior of the Classical planner is the same as before when CPP(db,"installed") was processed. t_1 is now the CPP(ci,"installed") task. The shortest path T is computed as <"stop"> again. The *for* loop in lines 17-29 in Algorithm 3 in Subsection 6.2.3 is only executed once. The set of dependencies D holds the same dependencies as before because we are looking again at the "stop" transition of an "AI_GroundedExecutionService". But now D', the set of valid dependencies, is different from before. As it can be seen in lines 81-102 in the DSL in Appendix A, the dependency "stop_DI_before_CI" is now valid because the "valid" closure evaluates to true over a central-instance. Dependency "stop_CI_before_DB" only accounts for databases and is thus not contained in D'. All in all, D' holds the value "<"stop_DI_before_CI">". Line 22 computes the subtasks described by all dependencies. In our case only the subtasks for dependency "stop_DI_before_CI" are computed. Lines 97-99 in the DSL in Appendix A show the code to create the new reserved change requests. The code in the body of the "subtasks" closure creates a reserved change request for every dialog-instance. This reserved change request demands the state of the dialog-instance to be set to "not_installed". Because there is only one DI present in the model, *CRS* holds "<CPP(di,"installed")>" in line 22. The *task_to_solve* is again the "stop_software" task but now with the central instance as target. Thus, *Class_Planning_Prob* holds the value

`<CPP(di,"installed"), stop_software(ci)>` after the *for* loop in line 29

This leads to the following recursive call in line 30:

```
HYP(<CPP(di,"installed"), stop_software(ci), stop_software(db)>, 0, M)
```

Because t_1 is a reserved change request, the Hybrid planner delegates the planning to the Classical planner again. This leads to the new invocation:

```
CLP(<CPP(di,“installed”), stop_software(ci), stop_software(db)>, O, M)
```

Again, the shortest path is computed as T=<“stop”> because the dialog-instance is in state running, too. But now D', the set of all dependencies valid for transition “stop” is empty. The DSL does not describe a valid dependency for the transition “stop” of a dialog-instance. All the bodies of the “valid” closures of the dependencies evaluate to false over a dialog-instance because the type attribute of a dialog-instance holds the value “DI”. This leads to an empty *All_dependency_tasks* set when the *for* loop in lines 21-24 ends. *task_to_solve* holds the “stop_software(di)” task again. In line 27, *Class_Planning_problem* only holds the task “stop_software(di)” because there were no dependency tasks computed. This leads to the following recursive invocation in line 30 in Algorithm 3 in Subsection 6.2.3:

```
HYP(<stop_software(di), stop_software(ci), stop_software(db)>, O, M)
```

The first task is not a reserved change request any more, but a non-atomic change request. Due to this reason the Hybrid planner directs the planning to the HTN planner for the first time:

```
HTP(<stop_software(di), stop_software(ci), stop_software(db)>, O, M)
```

Algorithm 2 in Subsection 6.2.2 describes the HTN algorithm. t_1 is a non-atomic change request because there is no operator with the description “stop_software” defined in the DSL in Appendix A. Consequently, the *if* statement in line 7 is true and M, the set of methods, holds “<stop_software(1), stop_software(2), backup_database>”. Where “stop_software(1)” is the first “stop_software” method shown in lines 143-154 in the DSL and “stop_software(2)” is the one shown in lines 156-165. M' is the set of methods applicable to t_1. Only “stop_software(2)” is applicable due to two reasons. First, the target of the “stop_software” change request, i. e., the dialog-instance, is an instance of a subclass of the abstract class “AI_GroundedExecutionService” (compare target key in line 156 of the DSL in Appendix A which demands this). Second, the body of the “precondition” closure (see lines 158-159 of the DSL in Appendix A) evaluates to true over domain object “di”. Thus, the algorithm can only pick m=“stop_software(2)” in line 12. *Subtasks* in line 13 in Algorithm 2 in Subsection 6.2.2 describes all possible decompositions of t_1 by method m. As it can be seen in the DSL (line 163), t_1 is decomposed into one task, the “stop_AI_GES(di)” task. Substituting t_1 by this task leads to the new recursive call:

```
HYP(<stop_AI_GES(di), stop_software(ci), stop_software(db)>, O, M)
```

The first task t_1 is now an atomic task, delegating the processing to the HTN planner:

```
  HTP(<stop_AI_GES(di), stop_software(ci), stop_software(db)>, O, M)
```

There is only one applicable operator for “stop_AI_GES” described in lines 186-192 in the DSL in Appendix A. It is applicable because the current state of the dialog-instance is still running. This operator is picked by the HTN algorithm in line 28 in Algorithm 2. Its effects are applied to the knowledge base. The effects are described in line 190 of the DSL. The body of the “effects” closure executes the method “stop()” implemented by every subclass of “AI_GroundedExecutionService”. We assume that the implemented “stop()” method automatically adjusts the “state” attribute of the domain object. Otherwise, we would have to set the attribute manually in the body of the “effects” closure. Finally, we get the following plan:

```
  <stop_AI_GES(di)> ∘ HYP(<stop_software(ci), stop_software(db)>, O, M)
```

Note that the first task is not part of the planner invocation any more because it is already an operator part of the plan. The call to the Hybrid planner is again redirected to the HTN planner which decomposes the task into the atomic task “stop_AI_GES(ci)”. The same operator as with the previous task is then used to implement the changes to the knowledge base. This leads to the plan:

```
  <stop_AI_GES(di), stop_AI_GES(ci)> ∘ HYP(<stop_software(db)>, O, M)
```

The dialog-instance and the central-instance are now both in state “installed”. The Hybrid planner redirects the call again to the HTN planner which now deals with the “stop_software(db)” task. It is a non-atomic change request. M holds three methods, “stop_software(1)”, “stop_software(2)”, and “backup_database”. Only “stop_software(1)”, matching the method shown in lines 143-154 in the DSL in Appendix A, can be applied. It describes two subtasks “stop_AI_GES(db)” and “backup_database(db)” that are substituted for t_1 leading to the new plan:

```
  <stop_AI_GES(di), stop_AI_GES(ci)> ∘
HYP(<stop_AI_GES(db), backup_database(db)>, O, M)
```

The invocation is again redirected to the HTN planner. t_1 holds the atomic task “stop_AI_ GES(db)”. The set of applicable operators only consists of the “stop_AI_GES” operator, shown in lines 186-192 in the DSL in Appendix A because there is no other operator matching this description. The changes described by this operator are made persistent to the knowledge base, i. e., the method “stop()” is executed on the database (compare “effects” closure in line 190 in the DSL). This leads to the following plan:

```
  <stop_AI_GES(di), stop_AI_GES(ci), stop_AI_GES(db)> ∘
HYP(<backup_database(db)>, O, M)
```

The call is redirected to the HTN planner because "backup_database(db)" is a non-atomic change request. This leads to the new invocation:

```
  <stop_AI_GES(di), stop_AI_GES(ci), stop_AI_GES(db)> ∘
HTP(<backup_database(db)>, 0, M)
```

The HTN algorithm shown in Algorithm 2 in Subsection 6.2.2 initializes t_1 with the "backup_database(db)" task. M', the set of applicable methods to decompose t_1, only consists of the "backup_database" method shown in lines 169-180 in the DSL in Appendix A. The database, the target of the change request, is an instance of class "AI_GES_DB". It is in state "installed", i. e., the precondition of the method is satisfied. Furthermore, the name of the change request and the method are matching. We assume that the method "get_tables()" returns the List "<"table1","table2">" when called on database "db". Under this assumption the variable *Subtasks* in line 13 in Algorithm 2 in Subsection 6.2.2 holds the value "< <backup_table(db,"table1"), backup_table(db,"table2")> >". The *for* loop in lines 15-21 is only executed once because *Subtasks* only contains one set. Finally, the following recursive call is done in line 16:

```
  <stop_AI_GES(di), stop_AI_GES(ci), stop_AI_GES(db)> ∘
HYP(<backup_table(db,"table1"), backup_table(db,"table2")>, 0, M)
```

The call to the Hybrid planner is redirected to the HTN planner. The "backup_table" task is an atomic task. There is only one operator with description "backup_table" defined, the one in lines 218-224 in the DSL in Appendix A. The precondition is satisfied because the database is in state "installed". The HTN algorithm makes the changes to the knowledge base persistent by executing the "backup_table()" method on the database. The string describing the name of the table is given to the method as a parameter (see line 222 of the DSL). The name of the table to backup can be accessed by the "params" map of the change request because it is a parameter of this change request. The behavior of the algorithm regarding the second "backup_table" task is the same. In the end the following plan is delivered:

```
  <stop_AI_GES(di), stop_AI_GES(ci), stop_AI_GES(db)> ∘
<backup_table(db,"table1"), backup_table(db,"table2")>
```

6.4. Temporal constraints and plan consistency

This section explains the temporal constraints included in a plan, how they are maintained during task decomposition, and how a plan can be checked for consistency regarding these constraints. With temporal constraints we mean a binary relation over change requests expressing which change request needs to be finished before another one can be started. Definition 11 in Subsection 4.3, which defines a plan, already gave a definition of such a relation. The introduced *happens_before* relation part of plan describes which atomic change requests need to be finished before another one can be started. The *happens_before* relation is only defined over atomic change requests. For the remainder of this section we relax the *happens_before* relation to an $after$ relation which does not only formalize temporal constraints between atomic change requests, but between all three kind of change requests. $after$ is defined as follows:

Definition 12 (*after relation*)

Let $CRS = \{cr_1, ..., cr_n\}$, $n \in \mathbb{N}_0$, *be the set of all change requests. Then* $after$ *is defined as:*

$$after \subseteq CRS \times CRS,$$

such that $(cr_i, cr_j) \in after$, $i \leq n$, $j \leq n$, $i \neq j$ *if* cr_i *needs to be finished before* cr_j *can be started.*

The $after$ relation needs to be kept updated by the HTN and the Classical planner to keep track of the temporal dependencies. Lets assume that both planning methods keep this relation updated and a successful decomposition of a task into atomic change requests is achieved. We can then receive the *happens_before* relation of the plan (compare Definition 11) by restricting the $after$ relation to all tuples (a, b), such that a and b are atomic change requests.

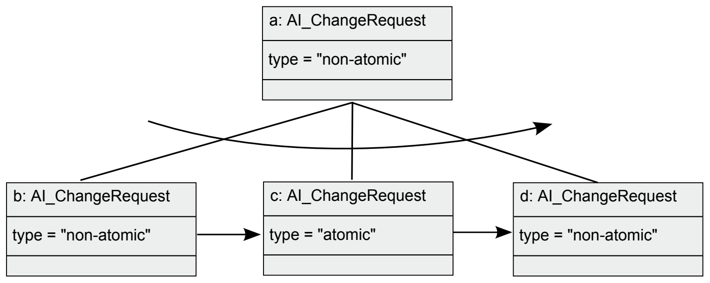
The remainder of this section is organized as follows: First of all, we explain the temporal constraints involved in HTN task decomposition in Subsection 6.4.1. In Subsection 6.4.2 we explain the temporal constraints that need to be checked or added when a new action is added to a plan. Finally, Subsection 6.4.3 explains which temporal constraints need to be added when the Classical planner decomposes reserved change requests.

6.4.1. Maintaining temporal constraints during task decomposition

This subsection explains how temporal constraints are maintained during HTN task decomposition. The HTN pseudo code introduced in Algorithm 2 in Subsection 6.2.2 does not explain how the algorithm keeps track of temporal dependencies. Especially in line 13, where the subtasks are created, it is not mentioned which temporal constraints account among these subtasks and other change requests part of the decomposition tree. Figure 6.2 shows the first part of an example explaining how the $after$ relation introduced in Definition 12 is adapted during task decomposition.

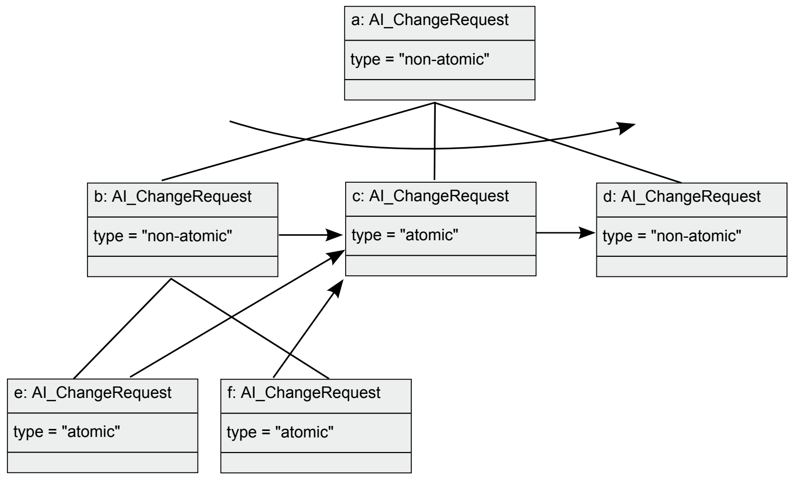
The example in Figure 6.2 shows the decomposition of the high-level task a into three subtasks. Be aware that the shown change request objects are a simplification of the “AI_ChangeRequest” class introduced in Figure 2.4 in Subsection 2.5. Because a is a non-atomic change request, its “type” attribute holds the value “non-atomic”. It is decomposed into the three tasks b, c, and d. The first and the last one are non-atomic change requests and c is an atomic change request. The curved arrow over the lines

Figure 6.2.: Decomposition tree after expansion of change request a



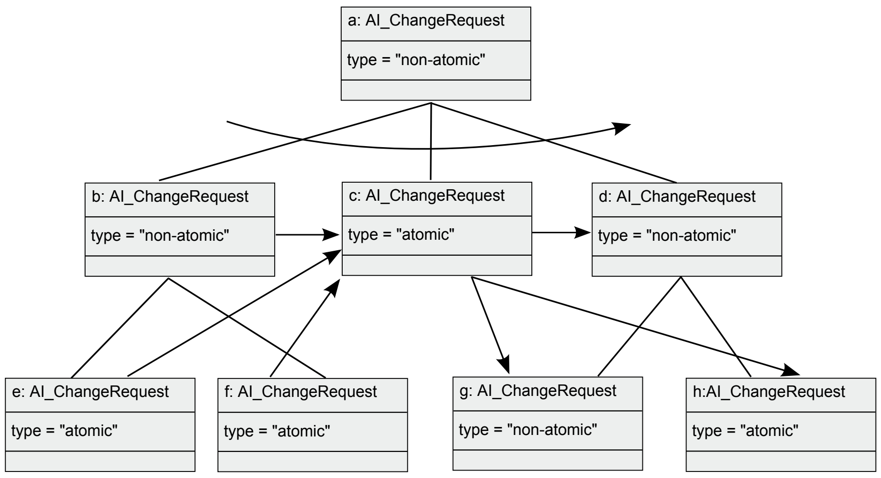
leading to the children symbolizes a sequential decomposition, i. e., a totally ordered method is used to decompose a. The arrows between b and c as well as c and d describe the $after$ relation. Be aware that the after relation comprises atomic and non-atomic change requests. Because the parent node a is not part of the $after$ relation we do not need to consider additional constraints when creating the subtasks. We assume that the arrows above are added to the $after$ relation when the *Subtasks* are assembled in line 13 in the HTN pseudo code in Algorithm 2 in Subsection 6.2.2. Figure 6.3 shows the situation after b has been decomposed into e and f, two parallel atomic subtasks.

Figure 6.3.: Decomposition tree after expansion of change request b



In the case of a parallel decomposition it is not necessary to add a temporal constraint between e and f because they can be executed in parallel. But we need to take the dependencies into account the parent b is participating in. Before the decomposition, $(b, c) \in after$ accounts. As e and f are children of b, $(e, c) \in after$ and $(f, c) \in after$ accounts, too. The $after$ relation is symbolized by the arrows in Figure 6.3. Be aware that if parent b were part of the $after$ relation in the way that $\exists\ x \in CRS$ such that $(x, b) \in after$, then we had to adapt the $after$ relation in the second argument. This is not the case in Figure 6.3 because there is no incoming arrow in b. The next task expanded by the HTN algorithm is node d because c is an atomic change request. Figure 6.4 shows the decomposition tree after the expansion of change request d.

Figure 6.4.: Decomposition tree after expansion of change request d



Change request d in Figure 6.4 is decomposed into two parallel tasks, g and h. Thus, (g, h) is not added to the $after$ relation. The parent node d is involved in the relation (c, d) part of the $after$ relation. This can be seen by the arrow in Figure 6.4 leading from c to d. As g and h are children of d, (c, h) and (c, g) are added to the $after$ relation. Only g is a non-atomic change request which needs to be decomposed further. The result of its decomposition is shown in Figure 6.5.

g is decomposed into i and j by a method describing a sequential decomposition. Thus, (i, j) needs to be added to $after$. In addition to that, we need to look at the parent's involvement in the $after$ relation. The only pair in $after$ containing g is (c, g). Because i and j are children of g, they also need to obey the constraints imposed on their parent. Consequently (c, i) and (c, j) are added to $after$ which is shown by the arrows between the participating change requests in Figure 6.5.

The final plan can be derived from the decomposition tree and the after relation by restricting the $after$ relation to pairs of atomic change requests. The set of atomic change requests in Figure 6.5 is $\{e, f, c, i, j, h\}$ and the $after$ relation, restricted to atomic change requests, is $\{(e, c), (f, c), (c, i), (c, j), (i, j), (c, h)\}$. The set of atomic change requests and the restricted $after$ transition on them is called a plan for change request a. Be aware that such a plan is consistent with Definition 11 in Section 11. The plan is visualized in Figure 6.6. It can be received from Figure 6.5 by removing every arrow leading from or to a non-atomic change request (this equals the restriction of the $after$ relation to atomic tasks). In addition to that, non-atomic tasks and the lines connecting a task with its parent task need to be removed .

Figure 6.5.: Decomposition tree after expansion of change request g

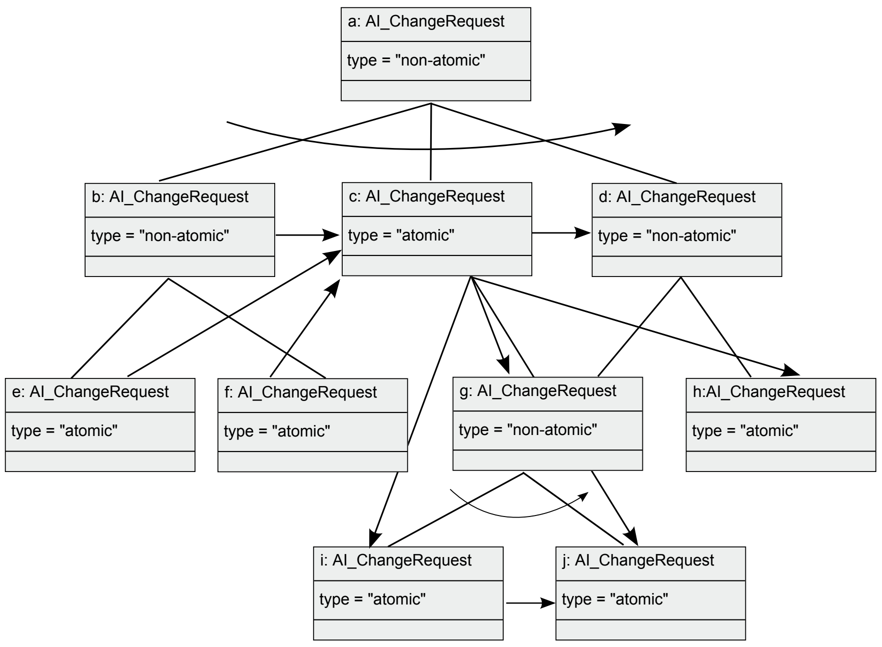
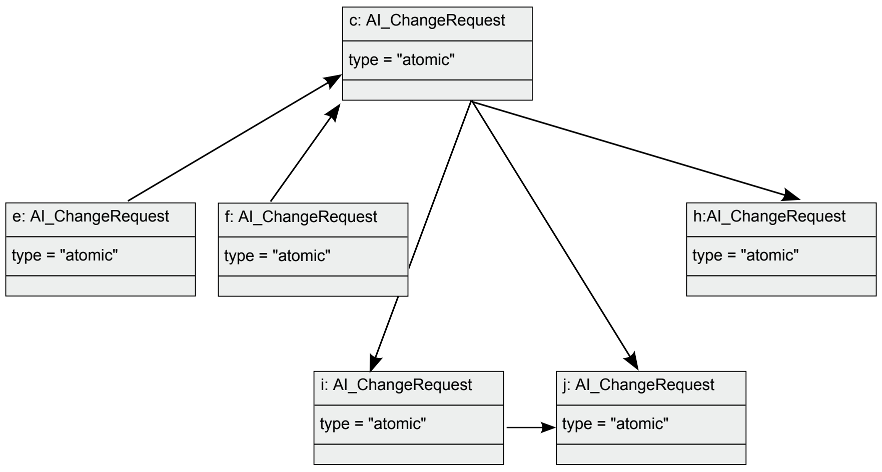


Figure 6.6.: Final plan consisting of atomic tasks and *after* relation



From the plan shown in Figure 6.6 we can also easily derive a possible execution order of the change requests. A change request cr can be executed if $\neg\exists cr' \in CRs$, such that $(cr', cr) \in after$ and cr' has not been finished yet. To find these change requests we need to look for atomic change requests in Figure 6.6 that have no incoming arrows or only arrows from change requests whose execution is already finished. Change requests e and f fulfill this requirement. They can be executed in parallel. After both have been finished, c can be executed because it only has incoming arrows from finished change requests. After c is finished, i and h can be executed in parallel. As soon as i is finished, j can be executed (still in parallel to h).

Note that there is a slight difference between how the $after$ relation is stored in the conceptual model and how it is represented in the UML diagram shown in Figure 2.4 in Subsection 2.5. In the conceptual model we describe temporal constraints using a binary relation called $after$ (see Definition 12 in Section 6.4). In the UML model the $after$ relation is described by the "after" references of class "AI_ChangeRequest". Because the pseudo code presented in Section 6.2 is based on this UML model, there is no explicit formalization of the $after$ relation in the pseudo code. In the UML model every change request only knows about the temporal constraints it participates in. This is in contrast to the $after$ relation which provides a global view on temporal constraints.

6.4.2. Plan extension and plan consistency

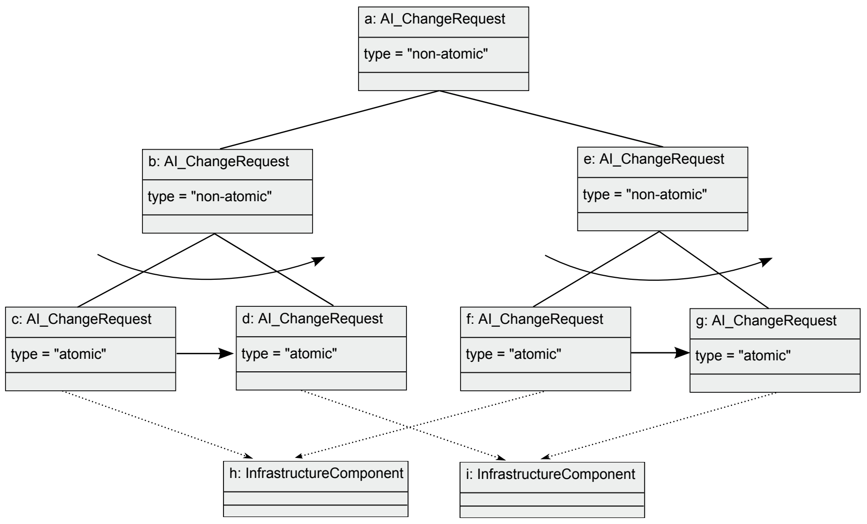
This subsection explains what needs to be considered when adding an atomic change request to a plan and how plan consistency can be checked. Both is done by the HTN planner when t_1 is added to the old plan *plan* in line 31 in the HTN algorithm in Subsection 6.2.2 and the resulting plan is checked for consistency. In this subsection we look at both issues. First of all we explain which temporal constraints need to be considered when a new action is added to the plan. After that, we explain how the new plan can be checked to contain only consistent temporal constraints.

Adding an atomic task to a plan

Adding a new atomic task to a plan, i. e., a set of atomic tasks and a $happens\_before$ relation over them as defined by Definition 11 in Section 4.3, might not be as trivial as it seems to be at first sight. An atomic task which is to be added to the plan is part of parallel and / or sequential decompositions as described in Subsection 6.4.1. The action which is added to the plan is part of such a combination of sequential and parallel decompositions. Because we assume that these dependencies are automatically captured as explained in Subsection 6.4.1 we do not need to consider them when the action is added to the plan. But there are also other temporal constraints imposed on an action than the ones defined by the decomposition through parallel or sequential tasks . Figure 6.7 provides an example of these constraints.

It shows the decomposition of the top level change request a into the parallel change requests b and e. Both are non-atomic tasks. They are decomposed by sequential methods

Figure 6.7.: Unspecified temporal relationships between parallel subtasks



(compare the curved arrows below nodes b and e) into the atomic change requests c, d and f, g. The arrows between c, d as well as f, g describe the $after$ relation as defined in Definition 12 in Section 6.4. In addition to that, the decomposition tree is augmented by the targets of the change requests. i, an instance of class "InfrastructureComponent" is the target of change requests d and g. Furthermore, h is the target of the atomic change requests c and f. As both change requests can be executed in parallel (there is no arrow between c and f), there are two parallel change requests targeting the same domain object. The same accounts for the atomic tasks d and g both targeting i. They can be executed in parallel after c and f have been finished. Normally we would demand from the domain expert not to write methods that are inconsistent. They are inconsistent because the designer of the knowledge base specified that b and e can be solved in parallel but some of their children are targeting the same domain objects.

This leads to possibly unsound plans. To see this examine the atomic change request f. It is planned for in a state of the knowledge base in which c has already changed target h. But the plan which is returned by the algorithm allows us to execute f in parallel with c which is in contrast to how the planner planned for action f. Normally this is not a problem if parallel actions target different domain objects because then the preconditions of one action do not rely on the effects of another action. In this case we do not need to add additional temporal constraints. But if two actions can be executed in parallel and they target the same domain object, a temporal relationship reflecting the order how the planner planned for it needs to be added. Otherwise, we are facing unsound plans.

Generally the following needs to be done when adding an action t_1 to the plan in line 31 in Algorithm 2 in Subsection 6.2.2: For every $cr \in plan$ we need to verify whether cr and t_1 are targeting the same domain object. If they do so and (t_1, cr) is not part of the transitive closure of the $after$ relation, i. e., t_1 and cr can be executed in parallel, then we need to add this constraint to the $after$ relation. Regarding our proposed model of

an “AI_ChangeRequest” this means that we need to add cr to the “after” references of change request t_1. The other way round holds for t_1 whose “before” reference needs to be updated.
Definition 13 describes what it means to add an atomic task to an already existing plan in terms of the conceptual model:

Definition 13 (*adding an atomic task to a plan*)

Let p, $p = (\{cr_1, ..., cr_n\}, after)$, $cr_i = (cr\_name_i, cr\_o_i, cr\_params_i)$, $i \in \{1, ..., n\}$ be a plan as described in Definition 11 in Section 4.3. Let $cr_0 = (cr\_name_0, cr\_o_0, cr\_params_0)$ be the atomic change request to add to plan p. Then $p' = cr_0 \circ p$ is defined as:

$$p' = (\{cr_0, cr_1, ..., cr_n\}, after'), \text{ such that}$$
$$after' = \{(cr_0, cr_i)|\ i \in \{1, ..., n\}\ \wedge\ (cr\_o_0, cr\_o_i) \notin after\ \wedge\ cr\_o_0 = cr\_o_i\} \cup after$$

Plan consistency

After the new atomic change request has been added to the old plan in line 31 in Algorithm 2 in Subsection 6.2.2, the new plan needs to be checked for consistency. In the conceptual model a consistent plan is defined as follows:

Definition 14 (*consistent plan*)

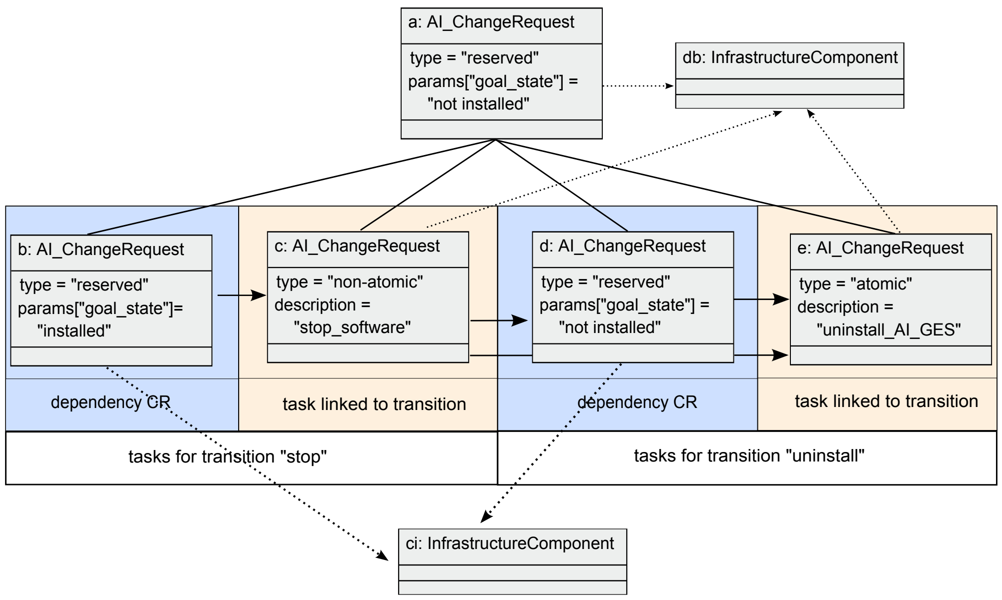
A plan $p = (CR, happens\_before)$ as defined in Definition 11 in Section 4.3 is called consistent if the graph $G = (V, E)$ where $V = CR$ and $E = happens\_before$ does not have a cycle.

A plan is consistent if its *happens_before* relation does not have a cycle. The *happens_before* relation can be derived from the $after$ relation by restricting it to pairs of atomic change requests. If the *happens_before* relation has a cycle, then there is a cyclic dependency which cannot be resolved. Thus, there is no execution sequence of tasks that fulfills the dependencies. For example, consider the decomposition given in Figure 6.7. The algorithm conducts a first-depth search. The last recursive call done by the planner is to the atomic change request g when line 30 in Algorithm 2 in Subsection 6.2.2 is executed. This results in the concatenation of the empty plan $(\emptyset, \emptyset)$ and g leading to the plan $(\{g\}, \emptyset)$. Later f is added to this plan. When adding f to the plan (f, g) is added to the $after$ relation because all dependencies of f are added. This leads to the new plan $(\{f, g\}, \{(f, g)\})$. We do not need to add additional constraints because g is targeting a different domain object than f. The new plan is consistent because it is cycle free. The next action to add from Figure 6.7 is d. But d targets the same domain object as g, namely i. If d and g where part of the $after$ relation this would not matter. Unfortunately they are not and can thus still be executed in parallel. When adding d to the plan we thus also need to add (d, g), an additional constrained as demanded by Definition 13, to the $after$ relation. This leads to the new plan $(\{d, f, g\}, \{(f, g), (d, g)\})$. The plan is consistent because it does not contain a cycle in its $after$ relation. On adding c to the plan, we need to add (c, d) to the $happens\_before$ relation of the plan. Furthermore f, which is already in the plan, targets the same domain object as c. As $(c, f) \notin after$, this pair needs to be added, too. This leads to the final plans $(\{c, f, g\}, \{(f, g), (d, g), (c, d), (c, f)\})$. The final plan is circle free. Be aware that this is done behind the scenes when lines 31-35 are executed in Algorithm 2 in Subsection 6.2.2.

6.4.3. Dependencies for reserved change requests

In the previous sections we only examined how dependencies are maintained during task decomposition (Subsection 6.4.1) and how additional dependencies need to be added when plans are concatenated in Subsection 6.4.2. In addition to that, we have looked at plan consistency in Subsection 6.4.2. We have not yet examined how temporal constraints are set when reserved change requests are solved. This is particularly important because this gives a semantics to the Classical Planning problems we solve within the HTN tree. Figure 6.8 shows an exemplary decomposition of a reserved change request and the dependencies accounting among its children.

Figure 6.8.: Dependencies between the subtasks of a reserved change request



The example given in Figure 6.8 is similar to the example provided in Section 6.3. The top-level change request given there is a reserved change request demanding to stop the database. The "AI_ChangeRequest" a given in Figure 6.8 goes a step further. It demands to set the state of the database to "not installed", i. e., it demands to uninstall

the database. Note that the "params" map of change request a holds a key "goal_state" describing the goal of the Classical Planning task. We assume that the database is currently in state "running". Thus, two transitions need to be executed to reach the "not installed" state, the "stop" and then the "uninstall" transition. The Classical Planning algorithm shown in Algorithm 3 in Subsection 6.2.3 determines the subtasks as follows: First, it looks up the valid dependencies for transition "stop". As it can be seen in the DSL in lines 75-102 in Appendix A, there is one valid dependency for this transition in a database. This dependency describes a Classical Planning problem (lines 87-89) demanding to set the state of the dialog-instance to "installed", i. e., the state where it is stopped. This is expressed by change request b. After that, the task linked to the transition is planned for. Line 78 in the DSL shows that this is the "stop_software" task. It is represented by change request c. The algorithm repeats the same thing for the "uninstall" transition. It looks up the valid dependencies and creates the children described by them. Only one dependency, the one shown in lines 114-122, is valid for the transition "install" of a database. It demands to set the state of the central-instance to "not installed". Task d represents this change request. After that, the task linked to the "uninstall" transition is added. In this case it is an atomic task described in line 109 of the DSL and shown in the "AI_ChangeRequest" e in Figure 6.8. The dashed arrows between the tasks mark the targets of the change requests.
The Classical Planning algorithm given in Subsection 6.2.2 does not state how the $after$ relation is defined among these subtasks. For each transition the algorithm determines a set of "dependency" subtasks, i. e., subtasks defined by the dependencies linked to a transition, and the task linked to the transition. Compare Figure 6.8 to see which change requests are dependency change requests and which ones are the tasks linked to either transition "stop" or "unistall".
On processing a reserved change request, the following dependencies need to be set:

- For each change request defined by a dependency, i. e., a dependecy CR, there needs to be a sequential constraint to the "task linked to the transition" that follows them. For example, change request b in Figure 6.8 is a dependency change request because it is described by a dependency linked to transition "stop". It has an arrow leading to c which is the task linked to the transition. The same accounts for the arrow leading from d to e because d is a dependency change request of transition "uninstall" and e is the task linked to the "uninstall" transition.
  If there were more than one dependency change request belonging to a transition, then each of these would have an arrow to the task linked to the transition. Be aware that there are no sequential constraints between the reserved change request describing the dependencies of one transition. Thus, change requests resolving dependencies for the same transition are considered to be executed in parallel.

- If the reserved change request demands to execute more than one transition then there is an arrow from the task linked to the previous transition to every change request belonging to the following transition. For example, in Figure 6.8 two transitions, "stop" and "uninstall", need to be executed. c is the task linked to transition "stop". "unistall" is the subsequent transition to "stop". d and e implement the "uninstall" transition. Thus, there is an arrow from c to d and from c to e. This is necessary in order to guarantee the sequential execution of the change requests that belong to different transitions.

The "Add_dependencies_to_children()" method called in line 26 in Algorithm 3 in Subsection 6.2.3 sets the dependencies as described above. Algorithm 4 summarizes the previous findings describing the "Add_dependencies_to_children()" method.

Algorithm 4 *Add_dependencies_to_children*(T, t_to_solve, t_old)

```
1: for $t \in T$ do
2:     add $(t, t\_to\_solve)$ to $after$ relation
3: end for
4: if $t\_old != null$ then
5:     for $t \in (T \cup \{t\_to\_solve\})$ do
6:         add $(t\_old, t)$ to $after$ relation
7:     end for
8: end if
```

Algorithm 4 is called for every transition by Algorithm 3. It takes three parameters, all dependency tasks linked to the transition (set T), the task which is directly linked to this transition (t_to_solve), and t_old, the task linked to a possibly existing previous transition. For example, consider the change request a given in Figure 6.8. For transition "stop" Algorithm 4 is called with the parameters $\{b\}$, c, and $null$: $\{b\}$ contains all dependency tasks of transition "stop", c is the task linked to the transition, and $null$ denotes there is no transition advancing the "stop" transition. The *for* loop in lines 1-3 in Algorithm 4 adds (b, c) to the $after$ relation, i. e., the arrow between b and c in Figure 6.8. The following *if* statement evaluates to false. When the *for* loop in lines 17-29 in Algorithm 3 in Subsection 6.2.3 is executed with $t = uninstall$, the "Add_dependencies_to_children()" function is called with $\{d\}$, e, and c as parameters. The set $\{d\}$ holds all dependency tasks valid for the "uninstall" transition in a database, e is the task linked to the transition, and c is the task of the previous "stop" transition. Algorithm 4 adds (d, e) to the $after$ relation when lines 1-3 are executed. Now the *if* statement in line 4 evaluates to true because there has been a previous transition before "uninstall". In lines 5-7 the algorithm iterates over every $t \in T \cup \{t\_to\_solve\} = \{d\} \cup \{e\} = \{d, e\}$ and adds them in pairs with c to the relation. All in all, (c, d) and (c, e) are added to the after relation. To sum it all up, the algorithm creates constraints such that all dependency change requests for a transition are executed before the task linked to the transition is executed. If there is more than one transition that needs to be taken in order to solve the Classical Planning problem, additional temporal constraints are added between the task implementing the previous transition and all tasks belonging to the subsequent transition. Thus, we can ensure that tasks are executed in the order transitions occur in the state-transition system. Be aware that the algorithm uses a first-depth search and respects this order.

Detecting violation of reserved change requests

Reserved change requests are different from atomic and non-atomic change requests because they describe Classical Planning problems. Two reserved change requests might occur in parallel, leading to plans that allow to change the state of the same transition system in parallel. These inferences lead to unsound plans. It is thus desirable to ban reserved change requests that occur in parallel and change the same state-transition system. More formally we define in the terms of the conceptual model:

Definition 15 (*conflicting reserved change request*)

Two reserved change requests $cr_1 = (cr\_name_1, cr\_o_1, cr\_params_1)$ *and* $cr_2 = (cr\_name_2, cr\_o_2, cr\_params_2)$ *are called conflicting if*

- cr_1 *and* cr_2 *have the same target domain object, i. e.,* $cr\_o_1 = cr\_o_2$.
- $(cr_1, cr_2) \notin after \wedge (cr_2, cr_1) \notin after$*, i. e.,* cr_1 *and* cr_2 *can occur in parallel.*

The Classical Planning algorithm shown in Algorithm 3 in Subsection 6.2.3 checks if the reserved change request, which is planned for, violates another reserved change request already part of the decomposition tree. Because the $after$ relation does not only contain constraints between atomic change requests, but also between change requests of other types, it also contains an order over the reserved change requests. We can thus directly verify whether Definition 15 holds. If the change request conflicts with another reserved change request then we return failure, otherwise planning is continued. The pseudo code shown in lines 8-10 in the Classical Planning algorithm in Subsection 6.2.3 performs the checks according to Definition 15.
Be aware that due to our definition of conflicting change requests every plan containing reserved change requests that target the same domain object and can be executed in parallel are considered as not valid. Looking at the decomposition, which is generally applied to reserved change requests as shown in Figure 6.8, this means the following:
When the "Classical_Planner()" method is called on change request b it checks whether it is in conflict with another reserved change request. The only reserved change request that has been touched by the planner before and can be executed in parallel to b is a because there is no arrow between a and b. Because a and b are targeting different domain objects there is no conflict between the two change requests. When the planner plans for d it needs to check relations to a and b. Only a can be executed in parallel to d. But they are not in conflict to each other because a is targeting the database and d the central instance (compare Figure 6.8).
Be aware that reserved change requests part of the transitive parent / child relation (e.g. b and a) do not have a temporal constraint between them. Applying Definition 15 to these pairs results in the fact that two reserved change requests cr_1, cr_2, where cr_2 is the descendant of cr_1, are not allowed to be in conflict with each other. This means the descendant cr_2 of a reserved change request cr_1 is not allowed to change the state of the

same state-transition system as its ancestor. This is perfectly fine because otherwise, we would produce unsound plans by contradicting the intention of a higher-level change request. This one might already try to achieve a specific state in a domain object.

7. Related work

This chapter provides an overview of important related work in the context of the hybrid approach. Related work can be roughly categorized into two areas. The first comprises related work done on hybrid HTN approaches that combine HTN with other planning approaches, mainly Operator-based Planning. These approaches are introduced in Section 7.1. The second area comprises related work done in the field of IT change request planning. Relevant works from this field are introduced in Section 7.2.

7.1. Related Hybrid HTN Planning approaches

This section gives an introduction to related work regarding hybrid HTN Planning approaches. We can distinguish between different kinds of hybrid HTN planning techniques. The first area comprises hybrid approaches that combine Hierarchical Task Network Planning [37], [16] with Operator-based Planning [40]. These approaches offer both, Classical and HTN Planning capabilities. The second kind of hybrid approaches are Hybrid Planning approaches that are not comprised of two different planning approaches, but of data structures from different approaches. Such approaches can be considered as hybrid, too.

Chien and his coworkers [9], [8] introduced a hybrid approach which combines HTN Planning with Operator-based Planning. It has been applied to two different planning domains, the automatic generation of scripts for image processing [9] and the automatic generation of tracking plans of communication antennas [8]. We first examine the approach used for image processing.

The *MVP/VICAR* planner [9], [7] is meant to generate scripts in a language for image processing, called *VICAR*. These scripts are based on high-level image processing requests. According to [9] there are two main reasons for the introduction of the hybrid approach. First, a pure Operator-based planner was not able to plan for the big search space without adding additional control knowledge. Second, analysts of the image processing domain think hierarchically about how to decompose a complete image processing request into a sequence of more simpler steps. These observations match partly with our motivation for the introduction of a hybrid approach. The domain of change request planning is undoubtedly an hierarchical domain in which high-level change requests are decomposed into finer-grained change requests. This is similar to the problem solving strategies applied in the VICAR domain. In contrast to VICAR, we do not introduce a hybrid approach because we suffer from an exploding search space. We introduce Hybrid Planning because a pure HTN approach comes at certain drawbacks like decreased knowledge base readability and maintenance. Instead, a hybrid approach consisting of

HTN and Classical Planning improves the readability, maintainability, and reuseability of our knowledge base.
There are also differences how the two planning approaches are linked together. The MVP/VICAR planner starts with a high-level planning goal which is further decomposed by HTN methods. The decomposition either ends with atomic goals, i. e., primitive tasks, or with goals describing an Operator-based Planning problem. These goals are then solved by an Operator-based planner. Finally, the different plans are integrated to one global plan. Thus, Chien's approach can trigger Operator-based Planning problems from HTN Planning problems. Our approach is different in the way that we also allow to trigger HTN Planning problems from Classical Planning problems. This is important to our planning domain because the change of the state of a domain object can involve more complex tasks that need further refinement. This also means that we have to enrich the original Classical Planning approach by additional concepts, leaving the paths of well explored Classical Planning approaches. This is a drawback compared to Chien's approach which still can make use of the findings regarding Operator-based Planning. By triggering HTN Planning problems from Classical Planning problems and vice versa we gain the freedom to switch between both representations at every hierarchical level of the decomposition tree. This is especially useful for expressing constraints in our planning domain that involve solving an hierarchical task network that is most easily specified by a Classical Planning problem.
Chien also argues in favor of a separation of the Hierarchical Task Network planner and the Operator-based planner. We do not separate these two approaches. Both access the same knowledge base. We even go a step further by reducing the Classical Planning problem to an HTN problem. Classical Planning problems are solved within the HTN decomposition tree. This leads to a closely coupled integration of HTN and Classical Planning resulting in great flexibility when switching the planning approach everywhere within the decomposition tree. This is especially important in our domain, where Classical Planning problems need to be solved as part of an hierarchical planning task, e. g., when it comes to resolve constraints.

Chien et al. [8] also constructed a planner similar to MVP/VICAR, the *DPLAN* planner. It is a planning system that can generate antenna tracking plans for radio science telecommunications antennas. It was developed after VICAR and uses the same algorithmic principle. Thus, lots of the evaluation given on the MVP/VICAR algorithm also accounts for DPLAN. The planner creates an antenna tracking plan based on high-level service requirements, e. g., an uplink to a device in space, and on available hardware equipment. Similar to MVP/VICAR it applies a hybrid HTN/Operator-based Planning approach where Operator-based Planning is done in the leaf nodes of the decomposition tree. HTN is chosen due to its abilities to express more complex order constraints than Operator-based Planning can do. The domain of change request planning is similar to this one where more complex order constraints are given that cannot be based on simple pre- and post conditions like in Operator-based Planning. In addition to that, we tend to think about change requests hierarchically. The motivation to introduce Classical Planning in our approach comes from the fact that lots of our domain objects can be considered to have a state. Thus, we introduce it for convenience reasons. Our motivation for Classical Planning is different to Chien's who uses Operator-based Planning because some problems of the DPLAN planning domain do not have an hierarchical nature. The DPLAN planner uses HTN for the decomposition of abstract tasks, i. e., tasks in the DPLAN domain which are independent of the underlying hardware. The planner applies Operator-based Planning for more specific directives, e. g., when the hardware has been chosen. Similar to this we apply HTN to decompose abstract change requests. Different to DPLAN, we nearly exclusively use Classical Planning to resolve constraints in our domain. But the use of Classical Planning within our approach is not limited to this. It can also be used as part of an HTN decomposition when it just might be easier to specify a subtask of a higher-level task that demands to bring a domain object into a well defined state. All in all, MVP/VICAR and DPLAN are both operating in a domain that demands a hybrid approach.

Having done the work on MVP/VICAR and DPLAN, Estlin et al. [18] summarized the findings from Chien's two projects. They argue in favor of a hybrid HTN/Operator-based approach as described in MVP/DPLAN for hierarchical domains with implicit constraints, i. e., constraints that cannot be easily expressed by pre- and post conditions, and demand for generality to express planning problems. As our domain of change request planning has an hierarchical structure, too, we can confirm their justifications for HTN. Because we use Classical Planning, a simpler form of Operator-based Planning, in our hybrid approach, we cannot directly match the mentioned advantages of Operator-based Planning to our domain. Compared to [18] we use Classical Planning as part of our hybrid approach because it enables us to give our domain objects a state. Furthermore, we can easily describe constraints by referencing to the states of domain objects, i. e., by specifying a Classical Planning problem.

Further research regarding a hybrid HTN/Operator-based approach bas been done by Kambhampati et al. [30]. Their work focuses on providing a principled framework for a hybrid HTN/Operator-based approach. It focuses mainly on the theoretical aspects of hybrid HTN/Operator-based Planning. The work on a hybrid approach is motivated by the existence of partially hierarchical planning domains, i. e., planning domains where not everything can be pressed into an hierarchical order. According to [30] there needs to be the possibility to plan in an Operator-based fashion in such domains. Early studies of our work have shown that we can describe our problem domain completely in HTN but this comes at the price of clarity and maintainability of the knowledge base as mentioned before. Using Classical Planning makes our domain description easier to understand.
In [30] the hybrid HTN approach is developed on top of a previously defined formalization for Action-based Planning. Be aware that our approach develops a solution for Classical Planning problems which is directly integrated into the decomposition tree of an HTN approach. Thus, we provide an example how to solve Classical Planning problems, i. e., search through a restricted state-transition system, within an HTN decomposition tree. Kambhampati [30] agrees with Chien's work [8], [9] that on a high-level HTN Planning is used until tasks cannot be decomposed any further by methods. Afterwards, Operator-

based reasoning is used to solve tasks not yet satisfied. In our work we extend this in such a way that the solution to Classical Planning problems describes a task network to be solved. In the simplest case the solution to a Classical Planning problem can be a task network only containing atomic tasks modeling Kambhampati's and Chien's approaches. Be aware that we can mimic their approaches if the domain described by Operator-based Planning can be formalized as a restricted state-transition system.

Another hybrid approach different to ours and the previous ones is *GraphHTN* [34], [33] by Lotem, Nau, and Handler. It is not a hybrid HTN/Operator-based Planning approach but a pure HTN Planning algorithm doing task refinement with the help of an additional data structure. This data structure is developed in the *Graphlan* [6] approach and is extended for the purpose of HTN Planning in [34] resulting in the GraphHTN approach. While our work and the ones of Chien and Kambhampati consider hybrid approaches in the sense that they support multiple different planning approaches, GraphHTN can be considered as a hybrid approach that uses ideas of data structures brought forward in other planning algorithms. However, GraphHTN can only do HTN Planning and thus does not fall within our perception of hybrid approaches.

7.2. Related Change Management approaches

This section gives an overview of related work done in the area of IT change request planning. Lots of related work deals with Change Management in general. Change request planning is an important part of Change Management. We mostly focus on the planning approaches proposed in the related works because this is the main theme of our work.

One of the first systems that automated the construction of change plans is *IBM CHAMPS* by Keller et al. [31]. Similarities between CHAMPS and our approach is the strong focus on dependencies. Our approach supports a well defined semantics for dependencies according to which the planner can reason. Compared to CHAMPS, we mainly focus on dependencies on a higher software level, e. g., on application server instances, instead of single servlets deployed into an application server. Nevertheless, our model of dependencies is capable of expressing low level dependencies, too.
Besides these similarities there are also some differences. First of all, CHAMPS does planning and scheduling of tasks while our approach only focuses on planning. By introducing Temporal Constraint Networks [13], [46], we can extend our approach to reason about the deadline and duration of tasks, too. An external scheduler can then schedule the atomic change requests according to these constraints. CHAMPS automatically generates dependencies from deployment descriptors and software artifacts. Planning is then done according to these dependencies. Our approach is based on the view that dependencies need to be specified by a domain expert aware of the system's software and hardware that is planned over. Our approach enables the specification of dependencies on all granularity levels while CHAMPS focuses on low level dependencies. In addition to that, CHAMPS exclusively plans according to dependencies. There is no notion of

hierarchical task refinement or the possibility to specify best practice knowledge from IT Change Management by workflows. Our approach supports workflow description by HTN method specification and task refinement. The "planning" approach introduced in [31] cannot be considered as a planning approach from an Artificial Intelligence point of view because it is not aware of the current state of the world and how actions do change it. It does not take the preconditions and effects of actions into account. Thus, CHAMPS does not guarantee the soundness of plans.

Cordeiro et al. [11] proposed a *template-based* approach to formalize, preserve, and reuse experiences gained regarding IT changes. Their work on *ChangeLedge* introduces the idea of *request-* and *plan templates*. A *request template* specifies the characteristics of a high-level *Request for Change* (*RFC*). In the hybrid approach request templates can be expressed by the notion of a high-level HTN method with well defined parameters. *Plan templates* describe a preliminary plan implementing an RFC / request template. This plan describes large-grained steps required to accomplish the RFC. Plan templates can be modeled in our approach by HTN methods that further decompose high-level tasks, i. e., request templates.
Although the focus of [11] is the knowledge reuse, an algorithm for task refinement is proposed. Compared to CHAMPS [31] Cordeiro et al. propose an algorithm based on task refinement. However, the refinement is limited compared to our approach. The algorithm proposed by Cordeiro et al. takes an RFC and searches for a preliminary change plan, i. e., a plan template, to implement the RFC. After that, planning is done exclusive by solving dependencies. Although Cordeiro et al. suggest the nesting of templates, the proposed algorithm does not take this into account. Thus, it is not possible to switch between planning according to dependencies and task refinement at every step in the planning process. This means that task refinement based on workflow descriptions can only be performed on the highest level by finding a suitable plan template. After that, the activities of the plan template are refined according to dependencies and not due to hierarchical problem solving behavior. Our approach is capable to plan for decomposition of tasks and resolution of dependencies at every hierarchy level of the plan interchangeably.
In addition to that, Cordeiro's approach makes it necessary that an operator specifies the mapping of parameters between an RFC and the plan template. Thus, similar to CHAMPS, ChangeLedge does not present an automated planning approach that can be executed without human intervention. The focus of our work is on eliminating human intervention to automate infrastructure and service management. Similar to CHAMPS, ChangeLedge has no AI Planning background. The planner is not capable of reasoning about the effects and preconditions of actions. The planner can not detect inconsistencies specified by an IT practitioner in plan templates, e. g., when two parallel actions change the same domain object. Thus, from an AI Planning point of view, the algorithm proposed by Cordeiro et al. is not sound. Our algorithm detects this and outputs a sound sequential plan.

Aware of the drawbacks of their previous approach [11], which does not take preconditions and effects of actions into account, Cordeiro et al. [10] proposed a new algorithm that takes the effects of actions into account. After CHAMPS [31] and their first work on ChangeLedge [11], it can be seen as a first approach into the direction of an AI Planning algorithm. The proposed solution is based on the templates and architectural models introduced in their previous work. Again the first refinement is done by the plan template and then refinement is done based on dependencies. Thus, workflow and hierarchical problem solving behavior can only be specified in the first refinement step. After that, refinement planning not induced by dependencies becomes impossible. According to [10] a plan is totally refined if all dependencies are satisfied. Cordeiro's work does not take into account that task refinement might not be always defined by dependencies. Especially workflow behavior does not necessarily need to be based on dependencies. Instead, our approach enables the specification of hierarchical problem solving behavior at every hierarchy level. In addition to that, solving dependencies can be part of this hierarchy or new workflow behavior can be triggered by solving dependencies. In the hybrid approach dependencies and hierarchical problem solving behavior are clearly separated by different concepts in the knowledge base. Cordeiro's algorithm delivers unsound plans under certain circumstances. If two actions are planned for in parallel and change the same domain object, then there are no mechanisms to prevent the parallel execution of these actions. As mentioned before, the planning approach proposed in this work is still sound because an additional temporal constraint is added. Unfortunately [10] does not explain how the changes to the underlying model are described. All in all, Cordeiro's second approach extends the algorithm described in [11] such that it changes the knowledge base during planning, although it is not stated how this is done exactly. Our work proposes Groovy and the concept of closures to describe changes to an object oriented knowledge base.

Trastour et al. [46] recently applied HTN Planning to the domain of IT change request planning. They propose *ChangeRefinery*, an automated planning approach based on HTN Planning. It is the first work that applies methods from AI Planning to change request planning. Compared to Cordeiro's [10] previous work on change request planning, Trastour's approach now incorporates planning based on hierarchical refinement on all levels and based on best practices described by workflows.
Trastour's work [46] very much focuses on providing a generic architecture that captures models, policies, and best practices of the IT change planning domain. Compared to our work, they propose an approach in which an *IT practitioner* guides the decomposition by choosing different applicable HTN methods. Our approach focuses on planning without human intervention. While Trastour et al. mainly focus on providing a general architecture, we show how the planning domain can be described by clearly separating behavior of domain objects from hierarchical problem solving behavior. Our research is based on the drawbacks emerging from a pure HTN domain description as it is applied in their work.
From an algorithmic point of view the HTN Planning approaches used in our work and Trastour's are very similar. Minor differences exist regarding the management of temporal constraints. While Trastour et al. [46] use *Simple Temporal Networks* [13] to reason

about deadlines and duration of actions, we only reason about actions in the form that some actions need to be completed before another one can be started. We apply a simpler model for temporal constraints because by using Classical Planning within the HTN approach we need a simple theoretic model to express the temporal relationships implied by Classical Planning. Nevertheless, our approach can be adapted to incorporate Simple Temporal Networks [13] to reason about the duration and deadlines of actions. From an algorithmic point of view ChangeRefinery and our approach differ in the fact that our algorithm incorporates HTN and Classical Planning. This gives us the possibility to plan in an hierarchical fashion over a domain that is very much driven by the states of domain objects. This makes our domain description easier to maintain because we clearly separate behavior of domain objects from hierarchical problem solving strategies. In addition to that, we describe dependencies not as HTN tasks but as Classical Planning problems. We consider this to be a more natural perception of constraints in IT change management.
Additional differences regard the implementation of the planner. While Trastour et al. use the *Hibernate Query language* to query their underlying model for domain objects, we directly write Groovy code in our DSL to query our model. Thus, we are not limited to the capabilities of a query language because we can write our own query language by providing appropriate methods in our model. In addition to that, we do not try to mimic an unification approach as known from logic knowledge bases. Instead, we provide our own computational constructs to describe multiple bindings over an object oriented model.
All in all, Trastour's approach differs from ours in the way that we provide a Hybrid Planning approach that takes states of domain objects into account. This enables us to make dependencies between domain objects explicit. Furthermore, the hybrid approach uses knowledge bases that clearly separate between domain object behavior, hierarchical task solving behavior, and dependencies. This leads to a more readable, maintainable, and adaptable domain specification than in a pure HTN approach.

Although not directly related to IT change request planning we want to mention that Machado [35] et al. proposed a solution to revert parts of a change plan if some of its actions fail during execution. Our work and the related work done in IT change planning assumes that the execution of change plans does not fail. If one assumes that the execution of change plans can fail, then there needs to be support to rollback all the actions belonging to one transaction.

There has also been lots of research regarding the scheduling of actions of a change plan. As we only focus on change plan generation, a deeper overview of related work in this area is not within the scope of this work. Readers interested in an overview of scheduling work in the context of IT change request planning may be referred to the related work section in [46].

8. Conclusion and future work

In this work we have described the drawbacks of a pure HTN approach when planning over a hierarchical domain whose domain objects have a state. The planning domain of IT change requests is such a domain. Although a pure HTN approach is capable of describing this domain, its usage comes with various drawbacks. First, hierarchical problem solving strategies are mixed with the behavior description of domain objects. This leads to an implicit coding of state-transition systems into HTN methods. Second, dependencies relating to the state of domain objects are not made explicit in a pure HTN approach. This leads to decreased maintainability, extendability, and reuseability of an HTN knowledge base.
To overcome these problems we have proposed a hybrid HTN / Classical Planning approach which copes well with the characteristics of hierarchical domains where the behavior of domain objects is best described by state-transition systems. Using our approach, we can describe the knowledge base in a modular way because domain object behavior, hierarchical refinement of tasks, and description of dependencies are clearly separated from each other. This increases the quality of the knowledge base. The approach offers great opportunity to describe hierarchical refinement of tasks and keep the notion of states of domain objects at the same time. In addition to that, we have shown how an dynamic object oriented language like Groovy can be used to do planning over an object oriented model.

Because we have introduced a new planning approach, topics linked to the general context of Change Management were not in the focus of this work. Further research in a holistic architecture that uses the hybrid approach planning algorithm is necessary. We also simplified the temporal constraints accounting between tasks to a partial order in order to achieve simple rules how to describe temporal constraints between tasks defining a Classical Planning problem. Thus, reasoning about deadlines and durations of change requests was out of the scope of this work. It still needs to be learned how reasoning about deadlines and task duration can be brought together with temporal dependencies imposed when solving a Classical Planning problem.

Our immediate next steps are to evaluate the algorithm using an implementation based on Groovy. In addition to that, we plan to extend the case study beyond SAP systems, e. g., to *Alfresco* [1] or *TikiWiki* [45], both Open Source Content Management systems, to prove the broader applicability of the hybrid approach. We are confident that our perception of software and infrastructure elements as state-transitions systems and of dependencies referencing states in transition systems copes with the challenges posed by new applications. We also plan to extend our approach using Temporal Constraint

Networks [13] and to use scheduling techniques to schedule the generated plans according to the temporal constraints. Finally, we want to examine how the hybrid approach can profit from an refinement of state-transition systems. We are aiming at an refinement of states such that states of a high-level transition system are refined by a transition system itself. This enables us to describe large, complex systems as state-transition systems whose behavior is based on finer grained transition-systems. Combining this with an hierarchical refinement of transitions could turn out to be a powerful approach to plan for IT change requests.

A. Appendix

Listing A.1 describes the planning domain underlying the example given in Section 6.3. The DSL describes a state-transition system for "AI_GroundedExecutionServices" (lines 1-137), three HTN methods (see lines 143-180), and five HTN operators in lines 184-224. The syntax of the DSL is explained in Section 5.2.

Listing A.1: Pseudo code to plan for the "resolve_dependency" change request

```
1  transition_system(target : AI_GroundedExecutionService) {
2
3   valid {true}
4
5   states {
6    state {"not_installed"}
7    state {"installed"}
8    state {"running"}
9   }
10
11  transitions {
12
13   transition("install" , [from : "not_installed", to : "installed"]) {
14
15     subtask {
16       install_AI_GES on : it
17     }
18
19     dependencies {
20
21      dependency {
22       name {"install_CI_before_DI"}
23       valid {it.type == "DI"}
24       subtasks {
25        for (AI_GroundedExecutionService ci : it.get_SystemModel().
26        .get_all_AI_GES("CI")) { Classical_Planning_problem on: ci,
27        goal_state : "installed" }
28       }
29      }
30
31      dependency {
32       name {"install_DB_before_CI"}
33       valid {it.type == "CI"}
34       subtasks {
35        for (AI_GroundedExecutionService db : it.get_SystemModel().
36        .get_all_AI_GES("DB")) { Classical_Planning_problem on: db,
37        goal_state : "installed" }
38       } // end subtasks
39      } // end dependency
```

```
   } // end dependencies
  } // end transition


  transition("start" , [from : "installed", to : "running"]) {

   subtask {
    start_AI_GES on : it
   }

   dependencies {

    dependency {
     name {"start_DB_before_CI"}
     valid {it.type == "CI"}
     subtasks {
      for (AI_GroundedExecutionService db : it.get_SystemModel().
      .get_all_AI_GES("DB")) { Classical_Planning_problem on: db,
      goal_state : "running" }
     }
    }

    dependency {
     name {"start_CI_before_DI"}
     valid {it.type == "DI"}
     subtasks {
      for (AI_GroundedExecutionService ci : it.get_SystemModel().
      .get_all_AI_GES("CI")) { Classical_Planning_problem on: ci,
      goal_state : "running" }
     } // end subtasks
    } // end dependency
   } // end dependencies
  } // end transition


  transition("stop" , [from : "running", to : "installed"]) {

   subtask {
    stop_software on : it
   }

   dependencies {

    dependency {
     name {"stop_CI_before_DB"}
     valid {it.type == "DB"}
     subtasks {
      for (AI_GroundedExecutionService ci : it.get_SystemModel().
      .get_all_AI_GES("CI")) { Classical_Planning_problem on: ci,
      goal_state : "installed" }
     }
    }

    dependency {
      name {"stop_DI_before_CI"}
      valid {it.type == "CI"}
      subtasks {
       for (AI_GroundedExecutionService di : it.get_SystemModel().
       .get_all_AI_GES("DI")) { Classical_Planning_problem on: di,
       goal_state : "not_installed" }
      } // end subtasks
     } // end dependency
    } // end dependencies
   } // end transition


   transition("uninstall" , [from : "installed", to : "not_installed"]) {

    subtask {
     uninstall_AI_GES on : it
    }

    dependencies {

     dependency {
      name {"uninstall_CI_before_DB"}
      valid {it.type == "DB"}
      subtasks {
       for (AI_GroundedExecutionService ci : it.get_SystemModel().
       .get_all_AI_GES("CI")) { Classical_Planning_problem on: ci,
       goal_state : "not_installed" }
      }
     }

     dependency {
      name {"uninstall_DI_before_CI"}
      valid {it.type == "CI"}
      subtasks {
       for (AI_GroundedExecutionService di : it.get_SystemModel().
       .get_all_AI_GES("DI")) { Classical_Planning_problem on: di,
       goal_state : "not_installed" }
      } // end subtasks
     } // end dependency
    } // end dependencies
   } // end transition

 } // end transitions
} // end transition_system



// methods:

method("stop_software", target : AI_GroundedExecutionService) {

 precondition {it.get_target().type == "DB"}

 subtasks {
  sequential {
   stop_AI_GES on : it.get_target()
   backup_database on : it.get_target()
  }
 }

}

method("stop_software", target : AI_GroundedExecutionService) {

 precondition {it.get_target().type == "CI" ||
               it.get_target().type == "DI" }

 subtasks {
  sequential {
   stop_AI_GES on : it.get_target()
  }
 }

}

method("backup_database", target : AI_GES_DB) {

 precondition {it.get_target().state == "installed"}

 subtasks {
  sequential {
   for (String table_name : it.get_target().get_all_tables())
   { backup_table on: it.get_target(), name : table_name }
  }
 }

}



// operators:

operator("stop_AI_GES", target : AI_GroundedExecutionService) {

 precondition {it.get_target().state == "running"}

 effects {it.get_taget().stop()}

}

operator("start_AI_GES", target : AI_GroundedExecutionService) {

 precondition {it.get_target().state == "installed"}

 effects {it.get_taget().start()}

}

```

```
operator("install_AI_GES", target : AI_GroundedExecutionService) {

 precondition {it.get_target().state == "not_installed"}

 effects {it.get_taget().install()}

}

operator("uninstall_AI_GES", target : AI_GroundedExecutionService) {

 precondition {it.get_target().state == "installed"}

 effects {it.get_taget().uninstall()}

}

operator("backup_table", target : AI_GES_DB) {

 precondition {it.get_target().state == "installed"}

 effects {it.get_target().backup_table(it.get_prams()["name"])}

}
```

Bibliography

[1] Alfresco - Open Source Enterprise Content Management. URL: http://www.alfresco.com/.

[2] Amazon Elastic Computing Cloud (EC2). URL: http://aws.amazon.com/ec2/.

[3] Amazon Simple Storage Service (S3). URL: http://aws.amazon.com/s3/.

[4] Animoto Blog. URL: http://blog.animoto.com/2008/04/21/amazon-ceo-jeff-bezos-on-animoto/.

[5] Arnold, S. How Google's Internet Search is Transforming Application Software, Infonortics. URL: http://www.infonortics.com/publications/google/google-legacy.html.

[6] Blum, A., and Furst, M. Fast Planning Through Planning Graph Analysis. *Artificial Intelligence 90* (1997), 281–300.

[7] Chien, S. A. Using AI Planning Techniques to Automatically Generate Image Processing Procedures: A Preliminary Report. URL: http://trs-new.jpl.nasa.gov/dspace/handle/2014/33804.

[8] Chien, S. A., Govindjee, A., Estlin, T. A., Wang, X., and Jr., R. W. H. Automated Generation of Tracking Plans for a Network of Communications Antennas. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI) and Ninth Innovative Applications of Artificial Intelligence Conference (IAAI)* (July 1997), pp. 963–970.

[9] Chien, S. A., and Mortensen, H. Automating Image Processing for Scientific Data Analysis of a Large Image Database. *IEEE Transactions on Pattern Analysis and Machine Intelligence 18*, 8 (August 1996), 854–859.

[10] Cordeiro, W., Machado, G., Andreis, F., Santos, A., Both, C., Gaspary, L., Granville, L., Bartolini, C., and Trastour, D. A Runtime Constraint-aware Solution for Automated Refinement of IT Change Plans. In *Proceedings of the Nineteenth IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM)* (2008).

[11] Cordeiro, W., Machado, G., Daitx, F., Both, C., Gaspary, L., Granville, L., Sahai, A., Bartolini, C., Trastour, D., and Saikoski, K. A Template-based Solution to Support Knowledge Reuse in IT Change Design. In

Proceedins of the Eleventh IFIP/IEEE Network Operations and Management Symposium (NOMS) (2008), pp. 355–362.

[12] Cormen, T., Leiserson, C., Rivest, R., and Stein, C. *Introduction to Algorithms*, second ed. MIT Press, McGraw-Hill, 2001.

[13] Dechter, R., Meiri, I., and Pearl, J. Temporal Constraint Networks. *Artificial Intelligence 49* (1991), 61–95.

[14] Dijkstra, E. A note on two problems in connexion with graphs. *Numerische Mathematik 1*, 1 (December 1959), 269–271.

[15] Erol, K., Hendler, J., and Nau, D. Semantics for Hierarchical Task-Network Planning. *Technical Report CS-TR-3239, UMIACS-TR-94-31, ISR-TR-95-9, Computer Science Department, University of Maryland* (1994).

[16] Erol, K., Hendler, J., and Nau, D. UMCP: A Sound and Complete Procedure for Hierarchical Task Network Planning. In *Proceedings of the Second Conference on Artificial Intelligence Planning Systems (AIPS)* (June 1994), pp. 249–254.

[17] Estlin, T., Castano, R., Anderson, R., Gaines, D., Fisher, F., and Judd, M. Learning and Planning for Mars Rover Science. In *Eightteenth International Joint Conference on Artificial Intelligence (IJCAI), workshop notes on Issues in Designing Physical Agents for Dynamic Real-Time Environments* (2003).

[18] Estlin, T. A., Chien, S. A., and Wang, X. An Argument for a Hybrid HTN/Operator-Based Approach to Planning. *Springer Lecture Nodes in Computer Science 1348* (1997), 182–194.

[19] Fernández-Olivares, J., Castillo, L., García-Pérez, O., and Palao, F. Bringing users and planning technology together: Experiences in SIADEX. In *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling (ICAPS)* (2006), AAAI Press, pp. 11–20.

[20] Fikes, R., and Nilsson, N. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence 2*, 3–4 (1971), 189–208.

[21] Floyd, R. W. Algorithm 97: Shortest path. *Communications of the ACM 5*, 6 (1962), 345.

[22] Ghallab, M., Nau, D., and Traverso, P. *Automated Planning: Theory and Practice.* Morgen Kaufmann, 2004.

[23] Gil, Y., Deelman, E., Blythe, J., Kesselman, C., and Tangmunarunkit, H. Artificial Intelligence and Grids : Workflow Planning and Beyond. *IEEE Intelligent Systems 19*, 1 (2004), 26–33.

[24] Gottfrid, D. New York Times Blog - Self-service, Prorated Super Computing Fun! URL: `http://open.blogs.nytimes.com/2007/11/01/self-service-prorated-super-computing-fun/`, November 2007.

[25] Hamilton, J. R. An Architecture for Modular Data Centers. In *Proceedings of the Third Biennial Conference on Innovative Data Systems Research (CIDR)* (January 2007), pp. 306–313.

[26] Hernandez, J. *The SAP/R3 Handbook.* Computing Mcgraw-Hill, 1997.

[27] Hoelzle, U. The Google Linux Cluster. URL: `http://www.uwtv.org/programs/displayevent.asp?rid=1680`, September 2002.

[28] ISO/IEC 14977 : 1996(E): Information Technology - Syntactic Metalanguage - Extended BNF. URL: `http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf`.

[29] IT Infrastructure Library: ITIL Service Transition, version 3. London: The Stantionery Office, p. 270 , 2007.

[30] Kambhampati, S., Mali, A., and Srivastava, B. Hybrid Planning for partially hierarchical domains. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI)* (1998), pp. 882–888.

[31] Keller, A., Hellerstein, J., Wolf, J., Wu, K., and Krishnan, V. The CHAMPS System: Change Management with Planning and Scheduling. In *Proceedings of the Ninth IEEE/IFIP Network Operations and Management Symposium (NOMS)* (April 2004), pp. 395–408.

[32] Koenig, D., Glover, A., King, P., Laforge, G., and Skeet, J. *Groovy in Action.* Manning Publications, January 2007.

[33] Lotem, A., and Nau, D. New Advances in GraphHTN: Identifying Independent Subproblems in Large HTN Domains. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS)* (2000), pp. 206–215.

[34] Lotem, A., Nau, D., and Hendler, J. Using Planning Graphs for Solving HTN Planning Problems. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI) and Eleventh Conference on Innovative Applications of Artificial Intelligence (IAAI)* (July 1999), pp. 534–540.

[35] Machado, G., Daitx, F., Cordeiro, W., Both, C., Gaspary, L., Granville, L., Saikoski, K., Sahat, A., Bartolini, C., and Trastour, D. Enabling Rollback Support in IT Change Management Systems. In *Proceedings of the Eleventh IEEE/IFIP Network Operations and Management Symposium (NOMS)* (2008), pp. 347–354.

[36] NAU, D. S., AU, T.-C., ILGHAMI, O., KUTER, U., MUÑOZ-AVILA, H., MURDOCK, J. W., WU, D., AND YAMAN, F. Applications of SHOP and SHOP2. *IEEE Intelligent Systems 20*, 2 (2005), 34–41.

[37] NAU, D. S., AU, T.-C., ILGHAMI, O., KUTER, U., MURDOCK, J. W., WU, D., AND YAMAN, F. SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research (JAIR) 20* (2003), 379–404.

[38] NAU, D. S., MUÑOZ-AVILA, H., CAO, Y., LOTEM, A., AND MITCHELL, S. Total-order planning with partially ordered subtasks. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI)* (2001), pp. 425–430.

[39] NAU, D. S., SMITH, S., AND EROL, K. Control Strategies in HTN Planning: Theory Versus Practice. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI) and Tenth Innovative Applications of Artificial Intelligence Conference (IAAI)* (July 1998), pp. 1127–1133.

[40] PENBERTHY, J. S., AND WELD, D. S. UCPOP: A Sound, Complete, Partial Order Planner for ADL. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning (KR'92)* (October 1992), pp. 103–114.

[41] RightScale Blog: Animoto's Facebook scale-up. URL: `http://blog.rightscale.com/2008/04/23/animoto-facebook-scale-up/`.

[42] ROLIA, J., BELROSE, G., BRAND, K., EDWARDS, N., GMACH, D., GRAUPNER, S., KIRSCHNICK, J., STEPHENSON, B., VICKERS, P., AND WILCOCK, L. Adaptive Information Technology for Service Lifecycle Management. *Technical Report HPL-2008-80, HP Labs* (2008).

[43] SACERDOTI, E. D. The Nonlinear Nature of Plans. In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence (IJCAI)* (1975), pp. 206–214.

[44] TATE, A. Generating Project Networks. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence (IJCAI)* (1977), pp. 888–893.

[45] TikiWiki - Groupware / Content Management Solution, Homepage. URL: `http://info.tikiwiki.org/tiki-index.php`.

[46] TRASTOUR, D., FINK, R., AND LIU, F. ChangeRefinery: Assisted Refinement of High-Level IT Change Requests. Submitted to the Eleventh IFIP/IEEE International Symposium on Integrated Network Management (IM), June 2009.

[47] VmWare Homepage. URL: `http://www.vmware.com/`.

[48] Xen Homepage. URL: `http://www.cl.cam.ac.uk/research/srg/netos/xen/`.