Domain-Independent Planning for Services in Uncertain and Dynamic Environments

Eirini Kaldeli

Printed by CPI Koninklijke Wöhrmann

RIJKSUNIVERSITEIT GRONINGEN

Domain-Independent Planning for Services in Uncertain and Dynamic Environments

Proefschrift

ter verkrijging van het doctoraat in de
Wiskunde en Natuurwetenschappen
aan de Rijksuniversiteit Groningen
op gezag van de
Rector Magnificus, dr. E. Sterken,
in het openbaar te verdedigen op
vrijdag 10 mei 2013
om 16:15 uur

door

Eirini Kaldeli

geboren op 14 januari 1983
te Athene, Griekenland

ISBN: 978-90-367-6155-0
ISBN-Electronic: 978-90-367-6214-4

To Rozina and Christophoros

Contents

Acknowledgments

Although this dissertation is to a great extent about planning, the long process of getting here has been full of developments, dilemmas and emotions that were "out of plan". Now that the thesis is to be sent out for printing, it may all seem clear and obvious. Looking back, however, I cannot but realize how fortunate I am to have encountered a number of people who have helped and supported me through all the uncertainties. It is a great pleasure to thank them here.

My deepest feelings of gratitude are owed to my advisor, Marco Aiello, who has been there with his unstinting guidance and encouragement at all stages of this project. His belief in me, and his readiness to help by all means is what kept me going at times of disillusionment. His advice has always been to the point and prompt, no matter how busy he was, and his patience limitless. I believe he rightfully deserves the title of the most supportive advisor, and I will always be thankful to him.

Special thanks go to Alexander Lazovik, who has acted (although not officially) as my second advisor. He is the one who gave me the initial push at a point that I was feeling lost, and part of the material presented in this dissertation is due to his contribution. Our long conversations have been stimulating, his strong programming skills have proven invaluable, and through his guidance I have become a better programmer.

I am also indebted to the members of the administrative staff at the University of Groningen, and particularly to Ineke Schelhaas, Esmee Elshof, Desiree Hansen, and Yvonne van der Weerd. Thanks to their effective and timely assistance, I never had to worry about complex paperwork and practical matters. I also want to thank Antonis for designing the cover of the thesis.

They say that a PhD is a lonely venture, and this is absolutely true. However, without my friends and colleagues, I would never have managed to be at piece with

this loneliness. Jelle and Electra are among the ones I connect my best times in Groningen with. I will always heartwarmingly remember the young activist who studies to change the world, and the blond Greek girl who passes for a native Dutch speaker. During my stay in Groningen, I have changed many places and housemates, some of who remain good friends till today. Lama and Kristiana have been around during the toughest period of my PhD, pushing me to be more social, and sharing their food with me when I had no time or mood for cooking. Narges and Tiago may have left Groningen before me, but I will never forget the nice time we have spent together. Other Greek expats in Groningen have contributed to creating an atmosphere that feels like home, balanced with the diverse cultures I have got to know through my international friends. I thank them all!

My colleagues at the University of Groningen have done their best to create a warm and cooperative environment. When I first came to the Distributed Systems group, we were only two PhD students, me and Elie El-Khoury, with whom I shared the anxieties and joys of a new beginning. Since then, the group has expanded with new members, who I would like to thank not only for our scientific collaboration and opinion exchange, but also for the nice experiences we have shared in our travels and going outs. In particular: Ilche Georgievski, Ehsan Ullah Warriach, Viktoriya Degeler, Pavel Bulanov, Tuan Anh Nguyen, Heerko Groefsema, Andrea Pagani, Ando Emerencia, and Mahir Can Doğanay, Faris Nizamic, Saleem Anwar, and Fatima al-Saif. Nick van Beest may come from a different department, however I consider him as "one of us", and owe him special thanks for our close and fruitful collaboration. For the enjoyable time we have had at the office and beyond, I am thankful to Artemis Kontogogos, George Azopardi, Giannis Giotis, Giuseppe Papari, Kerstin Bunte, Ahmad Waqas, and Aree Witoelar from the neighboring research groups, as well as to Elena Lazovik.

It is not only my new friendships and acquaintances from Groningen that have kept me upbeat throughout my PhD, but also my old dear friends from Athens. I have not been seeing them as frequently as I would like, however they were always there when I needed them: Vasilis with his discrete concern, Charalampos with his lighthearted spirit, Stephanos who counts more as a friend than a cousin, and Eva, despite her leaving from Greece. Lamprini, Andreas, Christina, Thodoris, Vasiliki, Grigoris, Dimitris, and others, too many to mention individually, have all contributed to making an expatriate's time at her home country fun and cosy.

My broader family has also expressed their support in many forms. I would like to especially mention the handmade delicatessen of my aunt Despoina, and the satiric collages of my aunt Amalia, which have been arriving to Groningen by post all these years.

Giorgos has been the reason behind my discovery of Groningen, and my decision

to start a PhD there at the first place. Things have not developed as planned since, and at the end, I am still undecided whether he has been a constructive or distracting influence on my PhD. So, I am not sure whether he should have my acknowledgements here. But he certainly has my love.

Following the thread that has lead me to pursuing a PhD, I ultimately find my parents, Rozina and Christophoros, who have always encouraged me to learn more. It is to them that I dedicate this dissertation.

Eirini Kaldeli
Groningen
March 24, 2013

Chapter 1

Introduction

The concept of *service* is used in Information Technology to abstract away from the specifics of software implementation through the engagement of standardized interfaces and well-defined functional descriptions. Services are made available over some network, and can interact with each other by exchanging messages using a communication protocol. Web-based standards, such as HTTP for transportation and XML-based languages for the description of message and operations, are usually employed to prescribe these interoperable machine-to-machine interactions, in which case services are implemented as *Web Services* (WSs).

Services, and especially Web Services, have emerged as the fundamental elements for building distributed applications. Traditionally, WSs have been used in enterprise IT systems or made publicly available online via the Internet. However, thanks to recent advancements in the hardware and software industry, service-oriented applications are also becoming applicable to a variety of physical objects with computational and networking capacities, such as embedded devices, sensor and actuator networks, electronic appliances, or RFID-tagged objects. The result is an environment with access to a vast amount of information and diverse operations, which, if exploited collectively, can pave the way for added-value applications that deliver something more than just the sum of their parts. What we are looking for is a *composition* of different services that can in synergy realize more complex objectives. So, how can we compose the data provided by different sources, from sensors to search engines, along with the operations offered by the available wired, wireless, or mobile devices and computers? How does such a composition connect with human or business needs, and how can these needs be expressed? Besides its formation, how should a composition react to failures or changes in a constantly evolving environment?

All these aspects are of concern to a wide range of scenarios, from smart homes and virtual tourism, to e-government and robotics. To take the case of domestic environments, compositions can represent daily routines of home inhabitants. For example, every time a user wakes up, he may want the radio to start playing, the coffee machine to prepare his coffee, the heating to adjust depending on weather conditions etc. The standard way to describe a composition of this kind is in the form of some complex process, using a workflow-style language like WS-BPEL [OASIS, 2007]. However, such a specification involves considerable manual effort and is tailored towards a specific objective, assuming a closed world, where services are static. If the requirements of the objective or the dynamics of the world change, then the predefined composite process is not applicable anymore, and has to be re-written.

Research in the discipline of AI (Artificial Intelligence) can contribute towards the realization of service infrastructures that go beyond basic interoperation technologies and ad hoc process specifications, and offer highly automated functionalities that are adaptable to changing user needs and environmental conditions. Within the last years, several approaches inspired by work in the AI planning community have been proposed in order to automate the process of service composition, e.g., [Sirin et al., 2004; Peer, 2005*a*; Sohrabi et al., 2006; Berardi et al., 2008]. *Planning* is the process of "choosing and organizing *actions* by anticipating their expected outcomes", with the aim of achieving some pre-stated goal [Ghallab et al., 2004]. The analogies with the problem of service composition are evident: actions correspond to functionalities offered by different services, and the goal is derived from some user request or inferred by some situation that calls for a combination of services to take care.

The underlying premise that enables AI methods such as planning to pervade the territory of service infrastructures is that services come along with appropriate semantic markups. Adding semantics to services, so as to specify their properties, capabilities, interfaces, and effects, enables minimizing human intervention for a number of tasks, such as service discovery, composition, and execution monitoring [Cardoso and Sheth, 2006]. In the context of planning approaches, services are usually represented either in terms of preconditions and effects or as state-transition diagrams, as e.g., in [Berardi et al., 2003; Traverso and Pistore, 2004]. We adopt the former view, and treat service operations as atomic actions, each of which is modeled by a set of *preconditions*, which should hold before the operation can be invoked, and *effects*, which model the results of an invocation.

The composition approach advocated by the current thesis is driven by the general aim of combining services automatically and on-demand, relying solely on individual descriptions of loosely-coupled software components, and a declarative goal that specifies *what* has to be achieved, but not *how*. The idea is to maintain a generic and modular repository that comprises a number of diverse service operations, from booking flights to arranging appointments with a doctor, and can serve a variety of different objectives with minimal request-specific configuration. In order to move towards such versatile domains, we use *domain-independent planning*, and propose an extended language for expressing complex goals in a declarative fashion, detached from the particularities and interdependencies of the available services. This is unlike many previous approaches that restrict the applicability of the domain to a set of anticipated user needs, predefined in the form of some procedural template, e.g., [McIlraith, 2004; Sirin et al., 2004]. A quite powerful language allows the specification of extended goals, which can capture requirements over state traversals, beyond mere reachability properties.

1.1 Planning domain representation

Domain-independent planning relies on an abstract, general model of the actions that are available in a domain. Conceptually, a planning domain can be seen as a state transition system, where the application of actions changes the state of the world. Given a model of the domain, the task of a planner is to find an appropriate structure of actions (e.g., a totally or a partially ordered set of actions), which when applied to some initial situation achieves some given objective. In the simplest case, a plan is a sequence of actions that lead from some initial state to a goal state.

There are different ways to represent the set of states and the transition function. The most established representation approach is to consider a state as a set of propositions. These propositions are derived from *ground* predicates, which define the relation between the different objects that exist in the world (for example, $\mathit{adjacent}(\mathit{robot1}, \mathit{loc1})$). The truth value of some of these predicates, called fluents, can be changed by applying some *planning operator*. Planning operators are described by a set of (input) parameters, preconditions and effects. Preconditions and effects are logical formulas on predicates, and parameters refer to the free variables that are used in the preconditions or effects of the operator. For example, a simple operator which moves a robot from some location l to another one d is described like this:

$\mathit{moveRobot}(l, d)$
  precs: $\mathit{robotAt}(l) \ \wedge \ \mathit{adjacent}(l, d)$
  effects: $\mathit{robotAt}(d) \ \wedge \ \neg \mathit{robotAt}(l)$

The variables l, d are the input parameters, and can take values from some respective sets of objects-constants (different robots and locations in the domain). To apply a schematic planning operator, its parameters have to be first substituted by concrete objects. In that respect, an action corresponds to a grounded instance of an operation. E.g., $\mathit{moveRobot}(\mathit{loc1}, \mathit{loc2})$, where $\mathit{loc1}, \mathit{loc2}$ are constants, is an action.

Most planning approaches work on grounded instances of the planning domain, and it has been shown that finding a plan given a grounded representation is an exponentially easier task than planning at the schematic level [Erol et al., 1995]. However, all these approaches deal with planning domains where the cardinality of the free variables that are arguments to predicates and operators is kept small. Such an assumption is not realistic in the case of service domains, which are *data intensive*, i.e., they deal with variables that range over very large domains, such as prices, dates etc. This means that if the behavior of a service operation is to be modeled by some planning operator, then this has quite frequently to involve fluents with numeric-valued arguments. For example, an operation for reserving some airline ticket is parametrized by data associated with the flight, such as dates, locations, airlines etc. Whether the application of such an operation will have the intended effect or not, depends on the choice of the right value to the input parameters of the operation. The right values for the input parameters may depend on the outcomes of previous operations in the composition-plan, and thus reasoning about their instantiation is part of the responsibility of the planning process. In the worst case (depending on the domain), considering an operator which involves k arguments with cardinality $|V|$ each, there can be permissible $|V|^k$ grounded instances. For large $|V|$s and many operators that involve variables with high cardinalities, the number of ground operators, and, as a consequence the size of the search space, can grow enormously. If one also considers grounding sensing operations that return numeric-valued outputs, something quite common in many service domains, the size of the grounded domain becomes intractable, and memory explodes.

For these reasons, we choose to work directly at the schematic level of the planning domain, and in the remainder of the thesis we use the terms *planning operator* and *action* interchangeably. Following the so-called Multi-Valued Task (MPT) paradigm [Helmert, 2009], we use state variables rather than predicates as the basic elements for describing the world. Under that view, a state is considered as a tuple of values to variables, and these values change through the application of some action. This approach leads to a more compact encoding of the set of feasible states, since it does not unnecessarily consider the combinations of predicates which are mutually exclusive. Most importantly, state variables can conveniently model *outputs* of sensing operations which take numeric values, rather than just deriving the truth value of some predicate. This is an important requirement for domains that involve many data source lookup services. Under the state-variable perspective, planning operators are modeled in a conceptually different way. For example, considering the planning operation for moving a robot, the state-variable representation would introduce a variable *locRobot* modeling the location of each robot in the domain:

```
$moveRobot(d)$
  precs: $adjacent(locRobot, d)$
  effects: $locRobot := d$
```

1.1.1 Planning as a Constraint Satisfaction Problem

In order to satisfy the requirements for domain-independent planning and dealing with variables ranging over large domains, we propose to use a planner which is based on translating the domain and the goal into a Constraint Satisfaction Problem (CSP), and applying a standard constraint solver for computing a solution-plan. Planning as CSP fits well with many aspects that are of particular concern for service domains. The encoding expected from the constraint solver suits the MPT-like schematic planning representation, and constraint solving supports the efficient handling of numeric expressions and variables ranging over large domains. Moreover, such an encoding suits well with most standard WS description languages, which are based on state variables rather than predicates. Complex goals can also be expressed in the form of constraints. Moreover, the CSP-based encoding of the planning problem can be such so as to lead to plans that include parallel actions, something particularly useful at execution time, since response times of certain service operations are frequently quite long.

Dynamic constraint solving, which allows the efficient addition and removal of constraints, can be applied to serve the need for continuously modifying the CSP instance, so that it reflects the evolution of a changing environment. This enables the planner to constantly incorporate new facts about the environment or remove obsolete ones, check for possible inconsistencies, and react accordingly.

The main shortcoming of resorting to planning as CSP at the schematic level is that it has less inferential power compared with modern domain-independent planners, and may suffer from performance issues for planning domains that require complex combinatorial reasoning. The powerful heuristics used by best-performing planners such as [Richter and Westphal, 2010; Hoffmann and Nebel, 2001], depend on a propositional encoding, while the level at which the constraint solver operates is quite detached from the structure of the planning domain. However, the principal raison d'être of the proposed planning framework is not achieving high performances in the domains used in planning competitions, but rather demonstrating its capability to address the special requirements put forth by service environments.

1.2 Uncertainty about the initial state

Classical planning is founded on the assumption that the planner has complete knowledge about the world. In many domains, however, the planner is likely to start from a state where it possesses only partial knowledge about the current state of the world, and has to resort to some *sensing* operations in order to acquire the information that it misses, and which is needed for the successful completion of the planning task. Service domains consisting of services publicly available on the Web are usually dominated by services that are data sources, and offer mainly operations which provide information about the current state of the world rather than changing it [Fan and Kambhampati, 2005]. A successful composition-plan may be conditioned on that information. For instance, given a request for buying a book from `amazon.com` if this costs less that a maximum price, the purchase action should only be performed after having retrieved the price by calling the appropriate sensing action.

In such situations, we say that the planner has to deal with *uncertainty* about the initial state, i.e., it has to consider that there is a number of different possibilities about the actual values of certain variables. The actual value of an unknown variable can be returned after the invocation of some sensing action, which is alternatively referred to as *knowledge-gathering* or *observational* action. Such actions include some special effects (analogously called observational or knowledge-gathering or sensing effects), which, as opposed to world-altering effects, do not modify any variable, but observe its current value and return it to the planner in the form of some output. The information provided by observation actions frequently refers to variables that range over very large domains. In most service domains one can assume *full observability*, i.e., that there is at least one sensing action for each initially unknown variable, and that these actions, when invoked, will instantiate the variable to a specific value.

In a domain-independent setting, the planning agent is expected to *plan* for sensing as well. This means that the planner should be in the position to identify what knowledge it lacks for satisfying the goal and reason how to seek for it, instead of relying on sensing actions or queries that are explicitly specified in some imperative WS template. The planner should also proactively take care of the data flow aspect in the plan, referring to how the acquired outputs are used by subsequent actions in the plan. For example, a user may like to mail a parcel to some person, whose address he does not currently know. The plan has thus to consult a white pages service first to retrieve the initially unknown address, and then give the order for posting by passing the right address value to the respective input parameter.

To deal with the problem of incomplete knowledge, we adopt a *knowledge-level* representation that reflects whether the values of the variables participating in the domain are known to the planning agent at some given state or not. The knowledge-level representation is generated automatically, given the definition of the variables, planning operators and the goal. Given that, plans are constructed based on the knowledge of the planner at some given state, and how this knowledge evolves through the application of actions.

1.3 Offline planning and online execution

Uncertainty does not only refer to the initial state and planning-time non-determinism concerning the different possible concrete values of the outputs of sensing actions. Non-determinism about the state resulting after an action's execution also stems from other sources of contingencies. A service invocation may behave in unexpected ways, such as return a failure, not respond at all, or even act in a way different than the one prescribed by their description. The actual outcomes of a service invocation only become visible at *runtime*, when the plan is exposed to the actual environmental conditions. Thus, the problem of uncertainty is directly correlated with the interaction between planning and execution. As the actions included in a plan constructed *offline* under partial knowledge are being invoked step-by-step, the planner enriches its knowledge about the environment, based on the feedback it can collect, be it some newly sensed information, some failure indication, or other notifications about how the world changes. Under the light of the feedback acquired *online*, the planner may have to revise certain decisions and look for alternative solutions.

1.3.1 A motivation example

Suppose that a user is happy to learn that in the following days a singer he is fond of is making a tour in the country where he lives. What he wants is to book a ticket and a hotel room for the nearest upcoming concert whose date and location meet some criteria referring to the weather conditions, the distance from his hometown, his availability according to his agenda, as well as about the price he is willing to pay for his overnight stay.

These requirements are expressed in the form of some declarative extended goal. The satisfaction of this goal requires the collaboration of services coming from diverse business domains—namely related to traveling, entertainment events, maps, calendar and weather services—in a manner that can hardly be anticipated in advance. Depending on the information returned at runtime, there are clearly many different possible ways that this goal can be fulfilled. For example, it may turn out that the place of the first upcoming concert is too far, or that there is no hotel available on that date within his budget, etc. In such cases, the original plan has to be interrupted and revised, so that the conditions regarding the whereabouts and date of the next concert are looked up. To further complicate things, at any moment a service may fail. So, if, e.g., the booking service of the first selected hotel that meets the users criteria happens to be in a permanent failure state, an alternative hotel has to be searched, and depending on the result, the goal may finally be satisfied or not.

1.3.2 Orchestration through continual plan revisions

One way to tackle the unpredictable nature of the environment is to enumerate all possible states that may arise during execution, and construct a conditional plan which includes branches for all possible outcomes of the various sources of uncertainty that may affect the plan, as, e.g., performed in [Pistore, Marconi, Bertoli and Traverso, 2005; Hoffmann et al., 2010]. However, given the large number of possible outcomes of sensing actions and unforeseen contingencies, planning for all potential circumstances that may appear during execution is not a recommended strategy for most service domains. In order to deal with this high degree of uncertainty, we adopt an approach that interweaves planning, monitoring and execution. The approach resorts to *continual planning*, so that the upcoming plan steps anticipated offline can be revised as execution proceeds, in face of inconsistencies that stem either from the newly acquired information or from erroneous service behavior. The process starts with an offline plan constructed on *optimistic* assumptions about the outcome of service invocations (usually referred to as a weak plan [Cimatti et al., 2003]). The execution progress is monitored, and assumed conditions are continuously checked towards the discovered reality. If an inconsistency is detected, then the planner is asked to modify the plan.

In order to describe this alternating sequence of offline planning and online execution, we borrow the concept of *orchestration*, which is used by the service-oriented computing community to denote an executable composition. In a service infrastructure, the orchestration engine is a central coordinator which interacts with the component WSs in accordance to the composite process specifications. We reserve a similar role for an orchestration component that interacts with the environment, informs the planner about the information it has collected, and decides about when to switch from planning to execution and vice versa. Depending on the situation, re-using parts of the existing plan may speed up the process of plan revision. The orchestrator can ask the planner to compute a new plan from scratch, or to attempt a refinement of the existing one, by updating the values of some input parameters to actions, or adding some extra actions. Moreover, a non-blocking strategy is adopted with respect to waiting for the response of sensing actions, so that the framework can go on with planning and execution of actions that do not depend on the expected response. Figure 1.1 presents a high-level overview of the interactions between the planner, the orchestrator, and the environment. Continual planning is realized via gradually altering the CSP instance, so that it reflects the most current information about the environment, and, depending on the plan revision policy, preserves or disregards previous plan commitments.

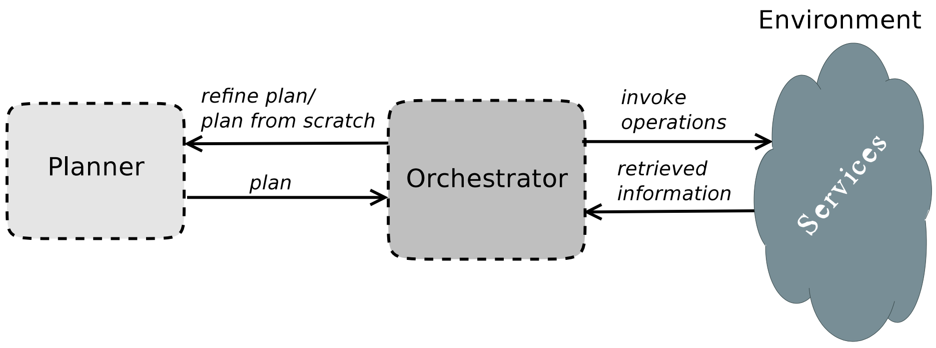


Figure 1.1: Basic schema of the interleaving planning and execution framework.

1.3.3 Dealing with dynamic environments

In most service environments, operations offered by services are accessible to many stakeholders, who may work on commonly shared data and pursue their own tasks simultaneously. This means that the dynamics of the domains change not only as a result of the deliberate actions executed by the planning agent, but also due to the activity of other exogenous agents who are active in the same environment. The activity of other actors may have repercussions on the composition-plan under execution, and render it invalid, either because some information it relies on has in the meantime become obsolete, or because certain scheduled actions are not applicable anymore under the new circumstances. For example, considering a partially executed composition which involves a robot moving around, if an external actor puts an obstacle in the robot's way in the middle of execution, the robot may fall unless it revises its decisions about how to move. Moreover, in many service environments dynamicity also applies to the availability of services, e.g., the services offered by a mobile phone appear and disappear depending on the location of the phone. With a few notable exceptions [Au et al., 2005; Bertoli et al., 2009; Klusch and Renner, 2006], the problem of domain dynamicity has been by large overlooked by existing planning approaches to WSC.

Depending on the service infrastructure within which it operates, the orchestrator monitors the execution through different ways and to varying levels of details. Traditionally, the communication between the orchestrator and the services takes place by following a request-reply protocol, e.g., Java-RMI (Remote Method Invocation) or CORBA (Common Object Request Broker Architecture). However, in many service frameworks, it is feasible to implement and take advantage of publish-subscribe mechanisms, e.g., in the OSGi framework (`www.osgi.org`) or the Amazon Simple Notification Service for clouds. Such a mechanism allows software components to raise events (publishers) by sending messages to interested parties which have expressed interest in these events (consumers). By registering for certain event types, the orchestrator can asynchronously receive notifications about changes that occur in the environment, and react accordingly. This way, it is kept informed about the results of the activities of other agents as well as about the actual effects of the plan's actions, and can detect whether the actual world state deviates from the state that was predicted by the plan.

1.4 Thesis scope and organization

The contribution of the current thesis is balanced between the design aspects of a planning system that can address the special requirements brought forth by service composition, and the more technical aspects regarding how such a planner can be integrated and put to work in different service-oriented platforms in order to facilitate complicated and laborious tasks. Such platforms include a domain consisting of diverse services publicly available on the Web, a Smart Home equipped with intelligent devices exposed as services, and a framework for Business Process (BP) recovery in case of process interference. The role reserved for the planner in each of these environments varies from satisfying user requests to ensuring the consistency of some broader system, be it a Smart Home or a set of concurrent BPs. Issues related to how the planner interacts with the other components of the service architecture, and how the expected planning representation is related to service or workflow descriptions are of particular importance in the course of realizing service-oriented platforms that can address real-world situations.

Depending on the application area, the characteristics of the planning system become more or less important, and put together enhance the extent of scenarios that can be represented and effectively dealt with. These characteristics include a knowledge-level representation to model uncertainty about the initial state; efficient handling of numeric-valued variables, which can appear as input to actions or output of observational effects; production of plans with a high level of parallelism; support for extended goals; and continual plan revision to deal with sensing outputs, failures, long response times or timeouts, and exogenous events. All these features are realized in a way that respects the requirement for domain independence.

The organization of the thesis is weighted between the description of the concepts, algorithms and techniques used by the planning system on one hand, and the design and implementation of broader service architectures with emphasis on practical case-studies on the other.

Chapter 2 gives an overview of the existing literature on the various aspects that concern this thesis, including planning approaches to WS composition, planning as CSP, planning under uncertainty, Smart Homes and the Web of Things, and Business Process recovery and adaptation.

Chapter 3 contains the basic definitions and algorithms that pertain to the offline working of the planning system, which we refer to as the *RuG planner*. In this chapter, we describe the representation of the planning domain extended with additional variables to model the knowledge-level representation, as well as its transformation into a CSP. We also present the syntax and the semantics of the language for expressing extended goals, their transformation into constraints, as well as a graphical editor for assisting the specification of goals. We show how the resulting CSP is solved by a constraint solver, and give an example of an optimistic plan produced by the offline planner.

Chapter 4 demonstrates the applicability of the RuG planner in *domotic* applications, concerned with the realization of intelligent home environments, which can enhance the convenience, comfort, and security of modern home residents. This chapter introduces a layered service-oriented architecture for Smart Homes, starting from the lower device interconnectivity level up to the higher application layers that undertake the load of complex functionalities, such as composition, and provide a number of services to end-users. The RuG planner stands at the core of this architecture, and interacts seamlessly with the other components, such as the context-awareness module and the human-computer interaction interface. Its task is to compute compositions that can satisfy the goals issued by the users or inferred by the home itself. A fully working prototype that realizes such an architecture is evaluated both in terms of performance as well as from the end-user point of view, so as to provide an assessment of the acceptability and usability of the solution. The scenarios demonstrated in this chapter assume complete information about the state of the world, collected by a set of sensors spread allover the house, and consider that services are executed sequentially in a successful manner, without the interference of external agents.

The interesting case where offline planning has to be interleaved with execution is investigated in Chapter 5. The focus of this chapter is the design and implementation of an orchestration framework, which is characterized by a high level of non-blocking concurrency and can deal with a number of inconsistencies that arise due to the uncertain and dynamic nature of service environment: with sensing outputs that violate the optimistic offline assumptions, with erroneous service behaviors that contradict the expected effects, with long response times, and with exogenous events that interfere with the plan execution. The orchestration approach is based on continuously revising partially executed plans via altering the CSP, and by either reusing fractions of the previous solution or replanning from scratch. The orchestration framework is evaluated on a number of simulated scenarios, which demonstrate the instantiation of output-to-input parameters matchings, the trade-off between plan refinement and planning from scratch, and the case of dealing with actions that take too long to respond.

Chapter 6 is concerned with the problem of dynamicity due to changes caused by external events in a setting of concurrently running Business Processes that access and modify common resources. This problem is known as *process interference*, and may lead a BP to some inconsistent state or to undesirable business outcomes, e.g., when performing an action based on outdated information. One way to deal with such issues is to annotate fragments of the BP with dependency scopes, which specify a set of desired properties-goals that should be achieved to recover the BP from certain inconsistencies. When during execution a modification event about some volatile information is detected, the RuG planner can be used to automate the generation of a recovery process, depending on the runtime conditions. To achieve this level of automation, the BP description is annotated with appropriate semantics, and then transformed to a planning domain. The approach is tested on a real case study taken from the Dutch e-government.

The thesis concludes with Chapter 7, which discusses the main achievements of the presented work and indicates some directions for possible continuations.

1.5 Publications

The work presented in the thesis has been published or submitted for publication in several contexts. Table 1.1 gives an overview of how the various papers are related with the aspects addressed in each of the main chapters of the thesis. The contributions are to be considered joint with the respective co-authors.

Chapter	**Venue**	**Citation**
3	ICAPS 2009	[Kaldeli et al., 2009*a*]
4	ACM TWEB	[Kaldeli et al., 2013]
	IFAC SYROCO 2012	[Caruso et al., 2012]
	ICSOC 2010	[Kaldeli et al., 2010]
	ICSOC 2010 Demo Session	[Warriach et al., 2010]
	ICAPS 2009 Application Showcase	[Kaldeli et al., 2009*b*]
5	AAAI 2011	[Kaldeli et al., 2011]
	journal paper to be submitted to AI journal	
6	KIBP 2012 (co-located with KR)	[van Beest et al., 2012]
	journal paper under review in Elsevier Information Systems	

Table 1.1: Overview of publications, and manuscripts under review or preparation in relation with the aspects addressed in the chapters of the thesis.

Chapter 2

Related Work

The topics investigated in the current thesis touch upon different fields, including work in the areas of Service Composition, Planning as CSP, Planning with Incomplete Knowledge and with Extended Goals, Replanning and Interaction with Execution, Domotics and the Web of Things, and Business Process Recovery and Adaptation. Given the large amount of relevant literature in all these areas, in the followings we discuss in more detail approaches that we consider as the most pertinent to the main issues of our concern, while only providing an overview of the broader picture. For more detailed discussions of related work across several dimensions, the interested reader is directed to surveys in the respective fields, some of which are cited in the followings.

2.1 Planning for service composition

A great number of approaches have been proposed in the literature about describing, constructing, executing and maintaining Web Services compositions, with research approaching the topic from different viewpoints, including issues related to service discovery and matchmaking, e.g., [Skoutas et al., 2008; Pilioura and Tsalgatidou, 2009], support for process evolution and migration, e.g., [Ryu et al., 2008; Orriëns and Yang, 2006], Quality of Service requirements, e.g., [Baligand et al., 2007; Hassine et al., 2006; Yu et al., 2007], support for dynamic reconfiguration, e.g., using the channel-based coordination language Reo [Lazovik and Arbab, 2007; Krause et al., 2011]. Several methodologies inspired from work in AI have been applied to deal with problems associated to WSC, ranging from reinforcement learning [Wang et al., 2008] to model checking and theorem proving [Rao, Küngas and Matskin, 2006; Papapanagiotou and Fleuriot, 2011].

In order to move towards compositions characterized by a higher degree of automation, customizability and context-awareness, AI planning methodologies have been employed for composing WSs. The common premise underlying these approaches is that services come along with semantic markups that describe their functionality in some convenient format, usually in terms of pre- and postconditions, which make them akin to planning operators. Several ontologies for the semantic description of WSs have been proposed in that respect, such as the influential OWL-S [W3C, 2004] (formerly DAML-S [DARPA, 2002]), WSMO [W3C, 2005*a*] and WSDL-S [W3C, 2005*b*]. Most planning approaches consider exact concept matches between variables, inputs and outputs, or assume some ontology (or multiple ontologies) and accompanying reasoning mechanism that take care of heterogeneities, e.g., [Sirin et al., 2003; Grau et al., 2004; Akkiraju et al., 2006; Lin et al., 2007; Hatzi et al., 2010]. In [Pistore et al., 2006] the procedural and the ontological information about semantic WSs are kept separately, and the link between the two is provided by appropriate semantic annotations. The problem of how to incorporate background ontologies into planning tools is investigated in [Sirbu and Hoffmann, 2008; Hoffmann et al., 2007, 2009], where concept subsumption relations are modeled through forward effects. The interesting challenge of planning in domains which are incompletely specified is acknowledged in [Kambhampati, 2007]. Shallow descriptions of WSs are very often the case in public repositories as affirmed by the survey performed in [Fan and Kambhampati, 2005].

Given that there is no commonly agreed definition of the problem of service composition from the planning perspective, different approaches depart from different starting points regarding what is given and what is to be achieved, as well from different restrictive assumptions about the expected behavior of the services. As a result, each approach considers its own test cases and scenarios that fit the particular features it seeks to demonstrate. Therefore, due to the lack of some commonly agreed benchmark, no direct comparison in terms of performance, expressivity or other well-defined requirements can be made. In the followings, we provide a brief high-level overview of the different approaches that use AI-planning techniques for the purpose of WSC.

A simple domain-independent planner for computing trivial compositions of services modeled as STRIPS-style [Fikes and Nilsson, 1971] operators is used in [Sheshagiri et al., 2003]. The approach proposed in [McDermott, 2002] constructs conditional plans depending on the outcomes of information retrieval, however its applicability is limited to very simple service domains. The PKS [Petrick and Bacchus, 2004] (Planning with Knowledge and Sensing) planning system is used for generating compositions at the knowledge level in [Martínez and Lespérance, 2004]. Services are modeled as primitive actions specified in a STRIPS-like formalism, however domain-specific design rules are required to capture additional effects triggered by sensing actions and search control constraints. A Partial Order Planning approach is adopted in [Peer, 2005*a*], which provides for sensing, service failures and replanning after exposure to the environment. In [Hoffmann et al., 2010], an adaptation of the FF [Hoffmann and Nebel, 2001] (Fast Forward) planner is used to construct Business Processes from atomic IT entities described in a planning-like manner. A conformant FF adjusted to consider on-the-fly output constants is used in [Hoffmann et al., 2009].

The approaches mentioned so far stick to domain-independent planning methods: they generate compositions relying on loosely-coupled individual descriptions of independent services and keep the ad hoc knowledge about how these can be linked to the minimum. Another line of research, assumes the availability of some generic template description, which specifies the basic steps of the composition at an abstract level. In that respect, planning is not fully automatic, however the additional control knowledge makes these approaches more powerful in terms of performance as well as expressivity, since they can usually handle advanced plan constructs such as loops and/or branches. This tradition is followed by [McIlraith and Son, 2002; McIlraith, 2004; Sohrabi et al., 2006, 2009], which build on modeling the WS domain in situation calculus and using versions of the Golog programming language. The general idea behind this proposal is to describe a set of user objectives in terms of a sufficiently generic Golog program, which includes many different non-deterministic choices to provide for variability. Then, the task of composition amounts to the customization of this generic template at runtime with respect to specific user constraints and preferences, which may also refer to non-functional requirements [Sohrabi et al., 2006, 2009]. Regarding the problem of incomplete knowledge, a middle-ground interpreter is employed, which senses online to obtain the missing information, while only simulating the effects of world-altering actions, until all necessary information has been gathered, and a correct plan has been found and can actually be executed.

Hierarchical Task Networks (HTN) have also been used as a means to represent generic procedures, e.g., [Wu et al., 2003; Sirin et al., 2004; Kuter et al., 2005; Au et al., 2005; Lin et al., 2008]. SHOP2, a highly optimized HTN planning system, is used to decompose process models, translated from DAML-S OWL-S to SHOP2 methods, into primitive operators/atomic services. In [Kuter et al., 2005], information gathering is performed during planning time, by issuing a list of appropriate queries to collect all information that is missing at the initial state. An extension of the algorithm for dealing with information that may change during the operation of the composition is presented in [Au et al., 2005], by considering that a solution is correct only within some expiration time. The trustworthiness of a composition is the focus of [Kuter and Golbeck, 2009], where users history of service ratings is taken into account by SHOP2. HTN planning is also used in [Madhusudan and Uttamsingh, 2006], where a solution is selected for execution among alternatives based on some cost, and replanning may be triggered upon the discovery of a failure. An hybrid approach which combines domain-independent planning with HTN is adopted in [Klusch and Gerber, 2005, 2006]. The XLPLAN planner is used, which can exploit information about hierarchical decomposition to speed up fast-forward heuristic search in the action space.

An architecture that can choose between a number of different planners, depending on the requirements associated with the specifics of the domain and goal each time, is proposed in [Peer, 2004]. Other approaches propose a semi-automatic composition procedure, where users can intervene and control the search over the possible plans constructed at the planning graph level. In the Synthy [Agarwal, Chafle, Dasgupta, Karnik, Kumar, Mittal and Srivastava, 2005; Agarwal, Dasgupta, Karnik, Kumar, Kundu, Mittal and Srivastava, 2005] architecture for end-to-end WSC, the stage of functional level synthesis is taken care by a GraphPlan-like contingent planner [Mediratta and Srivastava, 2006] which returns probably incomplete plan branches according to user preferences. A mixed-initiative approach is adopted in [Rao, Dimitrov, Hofmann and Sadeh, 2006], where users can specify high-level procedures and select from possible branches generated by a version of GraphPlan. In [Beauche and Poizat, 2008], adaptation features are added to an extension of GraphPlan with HTN-like decomposition constraints.

As opposed to approaches that view services as atomic planning operators, there is a track of research which considers stateful services, where behavioral descriptions impose constraints on the possible interactions that a service can be engaged with. In [Traverso and Pistore, 2004; Pistore, Traverso and Bertoli, 2005; Pistore, Marconi, Bertoli and Traverso, 2005; Pistore et al., 2006; Bertoli et al., 2006, 2010] component services are seen as state transition systems, e.g., derived from a BPEL description as explained in [Traverso and Pistore, 2004; Pistore, Traverso and Bertoli, 2005], where transitions correspond to asynchronous message exchanges resulting from some atomic action's execution. The requirements of the desired composite service are expressed in some temporal logics-like language, and symbolic techniques inspired by model checking are used for computing an executable process, which includes conditions and loops. An extension for data flow requirements is described in [Marconi et al., 2006]. In [Bertoli et al., 2009], the approach is extended to support requirements about how to handle uncontrollable events, such as a flight delay.

Another interesting approach which abstracts services as transition systems is the so-called "roman model" advocated by the work presented e.g., in [Hull et al., 2003; Bultan et al., 2003; Berardi et al., 2003; Berardi, Calvanese, Giacomo, Lenzerini and Mecella, 2005; Berardi et al., 2008]. From that perspective, the composition problem is treated as a problem of coordinating the executions of a given set of available services described as finite state automata. The objective itself is described in terms of a target service-transition diagram that conforms to some desired interactions. In [Berardi et al., 2003] available services are modeled as deterministic transition systems, i.e., given a state and an action the result of the action on the service is a unique state. The approach is extended to allow non-determinism in the target service in [Berardi et al., 2004], and non-determinism in the case of available services is investigated in [Berardi, Calvanese, Giacomo and Mecella, 2005], to provide for cases where the result of an interaction cannot be foreseen offline. A technique for precomputing the maximal simulation of all possible compositions, and choosing at execution the next step (transition) according to runtime information is proposed in [Berardi et al., 2008]. Distributed extensions of the Roman model, in cases where there is no central orchestration that can meditate between the services, are investigated in [Sardiña et al., 2007].

Planning algorithms have been applied for solving the WSC challenge, e.g., in [Oh et al., 2007; Zou et al., 2012]. From that perspective, the composition problem is defined as a data integration problem, and the aim is to find a chain of services which given some set of input concepts produces a set of output concepts. In such a context, WSs are seen as mere data sources, and all preconditions regard the availability of some input parameters. Dependencies between actions amount to matchings between sets of input and output parameters, and search can be performed much quicker.

A number of surveys have went through planning approaches to WSC across different categorization and comparison lines. In [Chan et al., 2007; Kster et al., 2005; Peer, 2005*b*], several approaches are classified according to the planning techniques they use and in association with the features that characterize them. In [Agarwal et al., 2008] the focus is on the relation between the offline composition and the execution stage. Planning approaches are discussed along with other approaches such as workflow management in [Srivastava and Koehler, 2003; Rao and Su, 2004; Dustdar and Schreiner, 2005; Alamri et al., 2006; Eid et al., 2008].

2.2 Planning domains and goal specifications

PDDL (Planning Domain Description Language) has become the standard language for defining planning problems which is used in the International Planning Competitions since 1998 [McDermott and the AIPS-98 Planning Competition Committee, 1998], and has undergone several extensions [Fox and Long, 2003; Gerevini and Long, 2006]. PDDL represents the world using objects and predicates, and numeric expressions are not allowed to appear as arguments to predicates or values of action parameters. The variable/value domain representation used by the RuG planner is similar in concept with the Multi-valued Planning Task(MPT) encoding [Helmert, 2009, 2006]. An algorithm for automatically translating a PDDL domain description into a MPT one is described in [Helmert, 2009]. Since our focus is on representing Web Services as planning operators, translating a planning domain encoded in PDDL to the form that the RuG planner expects as input is not a main concern. Our experience with modeling real services as planning actions actually showed that an encoding based on state variables rather than predicates follows more intuitively from service domain descriptions (see Section 4.4 about the transformation of devices represented in OSGi-UPnP and Section 6.4.2 about translating a BP into a planning domain).

The extended declarative goal language supported by the RuG planner as described in Section 3.4 enhances the traditional specification a goal as a set of final states by providing a number of additional features that allow the expression of constraints over state trajectories and hands-off observational requirements. A short overview of its basic operators has been presented in [Kaldeli et al., 2009*a*]. Many elements of the language are inspired by XSRL (XML Service Request Language) [Papazoglou et al., 2002; Aiello et al., 2002; Lazovik et al., 2005] for formulating complex requests against standard business processes.

PDDL3 [Gerevini and Long, 2006] extends the previous versions of PDDL, by supporting a richer goal language which provides for state trajectory constraints which should be respected by the entire sequence of plan states, as well as with soft goals which are desired but necessary to achieve. The goal language supported by the RuG planner is less expressive than PDDL3 and does not capture preference goals (although the `under_condition_or_not` goal operator could be seen as a form of soft requirement), but several parallels can be drawn with some of PDDL3 modal operators, such as always, sometime, and sometime-before.

The RuG planner goal language shares many concerns with the work presented in [Golden and Weld, 1996; Golden, 1998; Golden et al., 1996], which deals with meeting user goals in environments similar to the Unix operating system. Since incomplete information is intrinsic in such domains (see also Section 2.4), distinguishing between satisfaction and mere observational goals is essential. The operators "initially", "satisfy" and "hands-off" goals in [Golden, 1998] can be seen as analogous to the combinations of the RuG planner's achieve, find-out and maintainability constructs described in Section 3.4. A clear distinction between achievement and information gathering goals is also kept in [Peer, 2005*a*], for the purpose of composing semantically annotated WSs transformed into PDDL operators. A model which supports partial satisfaction of goals, making a distinction between core and context-specific goals, is proposed in [Vukovic and Robinson, 2005].

Systems that follow the planning as Model Checking approach have built-in support for temporally extended goals, which allow imposing constraints on the state trajectory, e.g., specification of safety or liveness properties. In [Traverso and Pistore, 2004; Pistore, Traverso and Bertoli, 2005], the EAGLE goal language, based on temporal logics extended with preferences, is used for composing WSs modeled as state transition systems. Several goal specifications for composing WSs move away from the purely domain-independent declarative spirit, and require that the set of possible solutions is pre-defined in some form of procedural template, either in the form of HTN methods, e.g., [Au et al., 2005], as a Golog program, e.g., [Sohrabi et al., 2006], or as a target state automaton, e.g., [Berardi, Calvanese, Giacomo, Hull and Mecella, 2005]. In such a context, runtime synthesis is responsible for customizing the high-level procedural specification with respect to user constraints and preferences. From that perspective, the work in [Sohrabi et al., 2006, 2009] extends the approach presented in [Sohrabi et al., 2006], so that Golog generic procedures can be customized not only based on hard but also on soft constraints, yielding compositions which are optimal with respect to the latter. The work presented in [Lin et al., 2008] investigates how qualitative user preferences expressed in PDDL3 can be incorporated in HTN planning for WSC, and [Sohrabi and McIlraith, 2010] deals with optimal service compositions by considering constraints and preferences over how HTN tasks should be parametrized and decomposed. An interesting proposal for fusing procedural and declarative goals to allow greater flexibility in expressing goals is made in [Shaparau et al., 2008].

2.3 Planning as CSP

A great amount of research has invested in exploiting constraint satisfaction techniques for solving planning and scheduling problems. However, CSP-based planners do not perform as well as state-of-the-art heuristic-based and SAT-based planners in the domains used in planning competitions. A direct transformation of the planning problem into a CSP has been presented in [Ghallab et al., 2004], where constraints describe the preconditions and effects of actions along with frame axioms. A rather different formulation based on successor state constraints similar to the ones captured by the planning graph is proposed in [Lopez and Bacchus, 2003], yielding improved performance. In [Barták and Toropila, 2008], some enhanced reformulations based on multi-valued state variables and transformations to ad-hoc tabular constraints are applied. Several techniques that aim at reducing search space and improving the efficiency of search strategies have been investigated in [Barták and Toropila, 2009]. The multi-valued representation is used in [Gregory et al., 2010] for problems with action costs, where cost-optimal sequential plans are generated by identifying compositions/macros of actions. A CSP encoding for producing parallel plans is proposed in [Barták, 2011], through the use of constraints that model the synchronization transitions that are possible between assignments to the same state variable.

A compilation of GraphPlan's planning graph into a CSP and using constraint satisfaction search techniques to improve Graphplan's backward search has been proposed in [Kambhampati, 2000; Do and Kambhampati, 2001]. By encoding the planning graph rather than the original planning problem, this approach is able to capture more characteristics of the structure of the planning problem in the CSP encoding. Constraints have also been used in the context of partial order planners [Vidal, 2004].

Mixed CSPs, which distinguish between controllable decision variables and uncontrollable parameters corresponding to environmental uncertainty and contingent events, have been used for modeling domains with incomplete knowledge and contingent events [Fargier et al., 1996; Guettier and Yorke-Smith, 2005]. In [Guettier and Yorke-Smith, 2005] mixed CSP is used for solving a control problem for the aerospace domain. Although the planning problem is rather particular and defined in terms of constraint-based automata and environmental constraints, an interesting online solution is followed. New contingent plans are built from scratch incrementally for increasing planning horizons/points in time, and these plans provide decisions for an increasing subset of possible world states. A CSP encoding for the conformant probabilistic planning problem, with no observability and probabilistic actions, is used in [Hyafil and Bacchus, 2003, 2004]. An approach that integrates constraint-based reasoning into the planning graph for temporal domains with predictable exogenous events that happen at known times is described in [Gerevini et al., 2006].

Constraint satisfaction techniques have been used extensively for scheduling problems that reason about time and resources, e.g., [Laborie, 2003; Gerevini et al., 2006]. A survey on CSP techniques used in the context of planning and scheduling is presented in [Barták et al., 2010]. To the best of our knowledge, CSP-based planners have been used so far for generating offline plans for grounded propositional domains, and are decoupled from the execution environment. The suitability of constraint solving techniques in a domain-independent planning setting for problems that involve uncertainty, sensing and unpredictable external events has not yet been investigated. In such a context, dynamic CSP and solution reuse/repair techniques, which use information collected from previous searches to speed up the search in the altered CSP, may prove helpful. Such a method, which makes use of no-good recording, is proposed in [van der Krogt and de Weerdt, 2005]. Performance improves when solving a CSP that differs by one constraint (added or removed) from a previously solved one. In [Wallace et al., 2009; Wallace and Grimes, 2010], some heuristics that exploit information about certain important features of the CSP that are not affected by the alterations are used, yielding considerably better performance for randomly changed CSPs.

2.4 Planning with incomplete information and sensing

Planning approaches that seek to deal with service environments have to take into account uncertainty about the initial state and unpredictable runtime behavior. A straightforward way to address these issues is to enumerate a priori all possible states that may arise at execution time, and then construct a conditional/contingent plan for each alternative runtime outcome. However, in service domains, the search space resulting from such an approach may become too large to explore if one considers the enormous number of ground actions and distinct outcomes as a consequence of the large cardinality of input parameters as well as outputs.

In the XLPLAN planning system [Klusch and Gerber, 2005, 2006], external procedure calls are implemented through linked call-back functions, which return a Boolean indication of whether a predicate has been added or deleted from the next world state. In [Hoffmann et al., 2010] non-determinism stemming from the set of alternative action outcomes is treated through "determinization", i.e., each non-deterministic action is compiled into a set of deterministic actions, one for each possible outcome. Clearly, although performance may be acceptable for binary variables, strategies that resort to determinization are not effective when the cardinality of possible outcomes increases.

Some approaches address the problem of incomplete information by only simulating world-altering effects during the composition process, assuming complete independence between sensing and world-altering actions, and setting limitations on the interleaving between knowledge-providing and world-altering actions. In [McIlraith, 2004; Sohrabi et al., 2006], information providing services are modeled as external function calls within the Golog programs. The approach relies on the assumption that information persists for a reasonable amount of time (until all actions that make use of it are executed), and that it is not altered by any subsequent actions inside or outside the composition. It is also taken for granted that all sensing actions can be performed even if the world-altering effects of actions that precede them in the plan have not been materialized (but only simulated). Similarly, in [Kuter et al., 2005; Au et al., 2005], information gathering and execution is treated as a task disconnected from planning, and execution is ceased until all sensing actions return. Analogous assumptions are made in [Peer, 2004, 2005*a*], where the subset of the plan consisting exclusively from sensing actions is extracted and executed first. If the outcome of the actions violates the causal relations following from the domain and goal, replanning is triggered.

Different algorithms for searching at the knowledge level have been proposed by the research line focusing on composing services as state transition diagrams, based on some temporally extended goal. Binary Decision Diagrams (BDDs) are used for the compact representation of beliefs, which amount to sets of states. The composition is actually a conditional plan, depending on the outputs/resulting states of knowledge-intensive transitions. One of the shortcomings of the initial algorithm presented in [Pistore, Traverso and Bertoli, 2005] is that it can only deal with Boolean-valued data. Non-Boolean data is considered in [Pistore, Marconi, Bertoli and Traverso, 2005], however, tests are still limited to low cardinality, and performance remains poor for reasonably complex compositions. A search algorithm on the AND-OR graph corresponding to the belief-level space is applied in [Bertoli et al., 2006], which however suffers from degrading performance as the number of branches of the solution grows. In all approaches mentioned above the domain description is proposition-based, the amount of outputs that can be generated is limited, and the state-explosion problem cannot be avoided when data cardinality increases. As shown in [Bertoli et al., 2010], belief-level construction grows exponentially with the branching factor of the conditional solution.

A different knowledge-level formulation as instructed by PKS is used in [Martínez and Lespérance, 2004]. In PKS, the planner's knowledge state is represented by a set of databases, which are updated whenever sensing actions are executed. Although the version of PKS used in [Martínez and Lespérance, 2004] cannot deal with high ranges of possible outcomes, it would be interesting to investigate the applicability and performance of an extension of PKS presented in [Petrick, 2011]. This extension allows the generation of conditional plans that cover numeric-valued outcomes by means of interval-valued functions, which are used to cut down the branching factor.

Dealing with the data flow dimension, i.e., the relation between outputs of operators with inputs of other operators in the plan, is an important issue associated with incomplete knowledge. For data intensive service domains determining the parameters for an action can be equally difficult as determining which actions belong to the plan. Since almost all state-of-the-art planners resort to some kind pre-processing for compiling the PDDL domain into a fully grounded encoding, on-the-fly handling of runtime outputs is difficult to implement. The problem of incorporating data production and flow into a plan has been investigated in [Golden, 2003; Golden and Pang, 2004]. Although [Golden and Pang, 2004] considers a planning graph approach, its basic idea of adopting a CSP encoding which amounts to a lifted (not grounded) representation is also adopted by the RuG planner. In [Hoffmann et al., 2009], data production is addressed by considering sets of additional potential constants to instantiate outputs, and by applying an adapted version of conformant FF. Input-output matchings are dealt with based on some axiomatizations [Hoffmann et al., 2007; Hoffmann, 2008] describing the ramifications entailed by sensing, i.e., implications entailed by the outputs/newly created constants of services/operators. However, the approach is limited to propositional effects, and the problem of search space explosion when considering many output constants remains.

Independently of the problem of WSC, in the planning community, there have been large advances in the performance of contingent planners which operate under uncertainty. For example, besides symbolic methods similar to the ones used in [Pistore, Marconi, Bertoli and Traverso, 2005; Bertoli et al., 2010], subtle logical formulas have also been applied for the compact representation, pruning and search in AND/OR graphs at the belief state space [To et al., 2011]. Instead of an explicit encoding of all possible states, some approaches advocate an implicit representation beliefs by keeping a history of actions and observations made, and inferring from those whether a proposition holds, e.g., [Hoffmann and Brafman, 2005; Shani and Brafman, 2011]. The conformant subcase where uncertainty comes only from the initial state, while observation actions are deterministic is discussed in [Palacios and Geffner, 2009; Albore et al., 2011]. An action language that provides for sensing actions with probabilistically and qualitatively non-deterministic effects is proposed in [Iocchi et al., 2009], and belief graphs are used to compute conditional plans. However, to the best of our knowledge, all these versions of contingent planning only consider observational effects that are propositional. If the application domain is characterized by an intractably large set of contingencies, a plan monitoring and repair approach is probably more appropriate.

2.5 Replanning and interactions with the environment

In dynamic and uncertain domains, acting, sensing and planning has to be intertwined, so that the plan is continuously adapted to the knowledge acquired during execution. In order to take into account online developments, the execution progress is monitored to ensure that certain conditions assumed offline actually hold. If some deviation is detected, then the original plan should be revised. Deviations between the premised contextual conditions and the actual ones may result not only from the uncertainty entailed by sensing actions, but also from changes incurred by reasons beyond the control of the planning agent, such as unexpected failures or the actions of other agents that are present in the same environment.

Some planning approaches to WSC provide for simple reaction mechanisms to some kinds of contingencies, which are however usually hand-coded and domain dependent. In [Bertoli et al., 2009], exogenous events are treated via reaction goals, which state what should be done when certain actions take place, while preferences over goals are also dealt with. The computed composition is a conditional, tree-structured plan, including branches regarding recoverable goal states. Therefore, the approach suffers from performance problems when the branching factor grows. In [Peer, 2005*a*], a partial order planner is used, and success conditions are included in actions' effects specifications. Replanning is triggered whenever some causal link indicating an interdependency between actions is violated due to some inconvenient outcome at runtime, and those violated links are avoided by the replanning search process. In [Klusch and Renner, 2006], the XLPLAN planning system is extended with an event listener about new facts, and changes in operators availability or the goal, and offers replanning capabilities, relying however on a closed-world assumption.

If the dynamics of the domain are known or can be learned, then these can be incorporated into a probabilistic planning domain representation, where different action effects occur with some probability. Markov Decision Processes (MDP) constitute an established mathematical model for probabilistic planning problems, and there are many planners which deal with probabilistic state models, e.g., [Yoon et al., 2007; Göbelbecker et al., 2011]. Replanning has been extensively employed by approaches which work on the determinized version of probabilistic domains, like FF-Replan first presented in [Hoffmann and Nebel, 2001], and further extended, e.g., in [Yoon et al., 2008]. The basic idea is to remove the non-determinism and probabilistic information from the domain by determinizing the possible outcomes/effects (either by creating one action per outcome or a single action for one of the possible outcomes), and then perform a search on that deterministic classical domain. During execution, if confronted with an unexpected state, search is repeated with the unexpected state as the initial state. A common principle that is shared between FF-Replan and the RuG planner is that both rely on optimistic assumptions about the future, i.e., they compute a solution by selecting the most convenient outcome. A strategy for identifying actions with unrecoverable outcomes, and adding precautionary actions to the optimistic plan so as to avoid dead-ends is described in [Foss et al., 2007].

There are several approaches which work on probabilistic domains with partial observability and sensing actions. In [Shani and Brafman, 2011], the replanning approach for the determinized representation is extended for such domains. Non-determinism stemming from incomplete knowledge at the initial state, and from sensing actions is removed by considering a single distinguished initial state from the set of possible (grounded) initial states. As the plan is being executed, belief states are updated accordingly, and replanning is triggered if the initial belief state sampling is not consistent with the world. This approach is not tested in domains with numeric observation effects. A framework that switches from classical planning to planning in small abstractions of the problem when encountering a sensing action whose outcome is uncertain is proposed in [Göbelbecker et al., 2011]. This approach can deal with noisy sensors, but the set of possible outcomes is kept small.

Many approaches to replanning try to reuse parts of the existing plan to guide the search for the new one. The idea is that under certain circumstances the work of adapting the current plan requires less time than planning from scratch, without sacrificing quality. A refinement heuristic for partial order plans is proposed in [van der Krogt and de Weerdt, 2005], which involves removing potentially problematic actions from the current plan, and incrementally adding extra actions to it until reaching a valid plan. An approach that focuses on preserving plan stability, i.e., replanning with minimum changes to previous plans, is presented in [Fox et al., 2006]. However, depending on circumstances and the kind of changes in the state of the world, the work required for repairing an old solution may be greater than planning by completely disregarding the previous solution [Nebel and Koehler, 1995]. A balanced approach between replanning from scratch and plan repair is proposed in [Borrajo and Veloso, 2012], where the plan is used as a bias to the heuristic search for the new problem. In this case, search expands by probabilistically choosing between heuristic search for the new goal and reuse of actions and goals of the past plan. The approach is used to speed up the planning time for classical deterministic domains. All above approaches are propositional.

Attention has been paid to examining the role of interactions between automated planning and execution, and the possible conflicts between plans and the environment. Approaches that consider dynamic environments have to deal with the tradeoff between investing too much effort in planning to ensure valid and optimal plans and quick commitment to probably bad choices, which may affect the rest of the plan and lead to dead-ends. In [Martínez et al., 2012], later parts of the plan are computed based on an abstracted domain resulting from the lifting of some manually selected predicates, with the aim of reducing computational effort for planning while avoiding dead-ends. The approach is tested on propositional domains with simple action failures, i.e., no exogenous events, and with no observations from sensing.

Previous frameworks that tightly integrate planning, monitoring, execution and information gathering include [Golden and Weld, 1996; Golden, 1998; Golden et al., 1996; Knoblock, 1995], which are concerned with building planning agents for dynamic and uncertain environments such as the Unix operating system. The RuG planner shares many concerns with this work, regarding tractable closed world reasoning with updates, knowledge preconditions, and observation effects that assign values to runtime variables. In [Knoblock, 1995], sensing is realized through the instantiation of these runtime variables (an idea analogous to the response variables used in the RuG planner) which can be used by other actions, while failed actions are treated via domain-specific failure handlers. In the context of WSC, a generic algorithm which performs continual replanning from scratch after every invocation of a knowledge-providing action is described in [Lazovik et al., 2003, 2006], however no evaluation is provided.

More recently, a continual planning framework for multi-agent planning under incomplete knowledge has been presented in [Brenner and Nebel, 2009]. Decisions depending on yet unknown facts are postponed through the use of assertions, special virtual actions that trigger replanning whenever their knowledge preconditions are achieved at execution time. Replanning annotations in that context lead into postponing sensing till just before the actions that need the information to be observed, which can be inefficient in terms of total execution time if a lot of time-consuming sensing is required. Interestingly, assertions can also be used to learn new operators that become available to the planning agent during execution. Similarly to the representation adopted by the RUG planner, multi-valued state variables are used to model the domain rather than a propositional encoding.

2.6 Comparative summary

Table 2.1 illustrates an aggregate outline of some of the main approaches to WSC which make use of planning techniques. Each column clusters together the collection of work by a group of authors which share a common viewpoint to WSC, and estimate it as a whole by taking into account the most recent improvements and capabilities added to their line of work. The assessment is performed across five dimensions, on each of which every approach is rated. The rates range from '★', indicating limited or inefficient support for the respective capability, to '★★★', signifying extended and efficient support.

Domain independence is assessed by considering the amount of effort required to model the domain, and the diversity of user needs it can cover. Approaches that represent the domain in terms of decoupled atomic actions are thus regarded as domain-independent, while the use of procedural templates which predefine the possible combinations of activities is regarded as domain-specific knowledge. *Support for data* refers to the extent to which numeric-valued variables and expressions

Approach	Domain independence	Support for data	Goal expressivity	Sensing	Contingencies
McIlraith, Shorabi et al.	★ (Golog procedure)	★★★ (not grounded)	★★★ (preferences)	★★ (restricting assumptions)	★ (predefined choices upon failures)
Nau, Kuter, Sirin, Wu et al.	★ (HTN methods)	★ (supported, but not explicitly discussed)	★ (imperative)	★★ (restricting assumptions)	★★ (timeouts, outdated info)
Bertoli, Traverso, Pistore et al.	★ (STS)	★★ (predefined data exchange)	★★★ (temporal logics, preferences)	★★ (conditional plan)	★★ (predefined reactions to changes)
de Giacomo, Berardi, et al.	★ (STS)	★★ (predefined data exchange)	★ (target STS)	★ (predefined in target STS)	★ (non-deterministic STS)
Peer et al.	★★★ (PDDL-like)	★★ (sets of possible substitutions)	★ (final state, find-out goals)	★★ (proactive but restricting assumptions)	★ (replanning for failures)
Klusch et al.	★★ (FF+HTN)	★★ (call-back functions, grounding)	★ (final state)	★ (predefined, not explicitly discussed)	★★ (replan for external events)
Aiello, Lazovik et al.	★ (process-like)	★★ (booleanization)	★★★ (XSRL)	★★ (interleaving)	★ (support, but not discuss)

Table 2.1: Comparative summary of planning approaches to WS composition.

on them are efficiently handled with. The basic requirement for the data criterion is the support for numeric-valued outputs of sensing operations. *Goal expressivity* assesses whether the respective approach can satisfy goals over state traversals and preferences, and whether the goal is of declarative or imperative nature.

Sensing refers to how observational actions are included to the composition (whether they are predefined or the approach reasons about them), and how observational effects are allowed to be interleaved with world-altering ones during execution. The latter consideration concerns the extent of restricting assumptions about the relation of sensing actions to the overall composition, e.g., that sensing actions do not depend on any world-altering effects.

The term *contingencies* encompasses failure responses, timeouts or any other service behavior that deviates from the expected one, as well external events and information changes due to factors other than the composition agent. Each approach is judged with respect to the kind of contingencies it can effectively deal with (e.g., whether it only considers a success-failure distinction, or also takes care of external events), as well as the amount of required extra manual specifications, and to what degree these are domain-dependent.

2.7 Service coordination in domotic environments

The main contribution of the work presented in Chapter 4 concerns the design and implementation of a framework based on the concept of dynamic coordination of intelligent devices exposed as services in the environment of a Smart Home. In the followings, we give a short overview of research on the Web of Things, discuss previous work related to composition and the employment of planning and other similar AI-inspired approaches for pervasive systems, and finally refer to some selected domotics project close to ours in spirit.

2.7.1 The Web of Things

The Web of Things is a term to describe Web-like infrastructures where the interconnected objects can be physical ones, for which there is a virtual representation in the software architecture. These objects can be physically accessed and manipulated by human beings. Wireless sensors, embedded devices or RFID-tagged items are integrated into a pervasive network and can communicate with other objects and services using Web-based principles, from SOAP and WSDL to Ajax and REST. The Cool Town project [Kindberg et al., 2000] is one of the first examples proposing the application of the Web paradigm for interlinking physical objects. These interact by exchanging messages via HTTP connections and by the use of a standard interface, rather than having heavy middleware applications running on each device. The standard Web technologies are extended to support discovery, mobility and location-awareness, and devices are indexed via Web pages, which make their services available to users.

The principle of RESTful services is broadly used for providing a uniform HTTP interface to interacting with smart things, independent of their platform protocol. In [Duquennoy et al., 2009], it is demonstrated that putting Web Servers directly on resource-constrained devices is a feasible solution. The authors of [Trifa et al., 2010], on the other hand, argue for the use of smarts gateways, which hide the underlying specific network protocols of the connected devices, and can thus be used for providing aggregate functions, based e.g., on composing single lower-level services. An extended discussion of different approaches building upon Web principles is provided in [Guinard et al., 2011], where it is shown how the notion of Web mashups can be applied to physical objects, in order to offer more customization possibilities to end-users. Since the focus of the present treatment is on realizing home smartness via dynamic service composition, independently of the underlying invocation mechanism, we do not enter into the debate of RESTful vs. Web Service based architectures.

In recent years, several SOAs such as UPnP and Jini[Apache, n.d.] have emerged to provide interoperability with minimum human intervention. The OSGi platform has been widely used as a platform- and application-independent residential gateway that enables interconnection, discovery, and coordination of different devices, thus offering more flexibility to domain designers, e.g., [Zeadally and Kubher, 2008; Lee et al., 2009]. Moving towards a semantic annotation of the OSGi description is proposed in [Gouvas et al., 2007] to improve the discovery process. The work presented in [Aiello, 2006] investigates the use of Web Services in the domestic network, and in [Aiello and Dustdar, 2008] the application of the Web Service stack is proposed as a means to solve the interoperability problem at home. The architecture builds on using WS-Notification as an event-based mechanism for addressing emergency situations in the home, most notably, the fall of an elder, however the aspects of context and coordination of service are not addressed beyond the basic action/reaction interactions. In [Cabezas et al., 2008], an architecture for extending the OSGi registry with semantic terms is proposed, which allows the automatic parsing of services by software agents, however no tasks more complex than service registration and invocation are considered.

2.7.2 Service composition and AI Planning in pervasive systems

WS-BPEL [OASIS, 2007], the standard for expressing WS compositions, has also been proposed to guide the coordination of pervasive systems. In [Lazovik et al., 2009], the RuG visualization platform presented in Section 4.5.3 is coupled with a BPEL engine to demonstrate some composition scenarios. The approach proposed in [Redondo et al., 2008] describes how composite services deployed as BPEL processes can be made available in a semantically enriched OSGi platform. However, BPEL processes are pre-compiled and thus support limited dynamicity. In [Etzioni et al., 2010], BPEL processes in a smart home are enhanced with a runtime fault management mechanism, where the receipt of a fault-indicating event triggers an appropriate predefined fault template according to the semantically inferred type of the fault. However, these approaches do not overcome the limited flexibility and adaptability deriving from the rigid nature of predefined processes.

In the context of domotic systems, AI-inspired techniques have been used for coordinating intelligent components in a ubiquitous environment, without however emphasizing on services in any specific way. In [Pecora and Cesta, 2007], each device is represented as a software agent and the problem of service integration is cast to Distributed Constraint Optimization. The coordination takes place in a purely distributed manner, relying on the communication between independent agents. On the negative side, modeling the home behavior involves specifying all possible inter-relations between the variables comprising the domain in terms of complex constraints, which makes it a fairly cumbersome process, even for a limited number of services. The requests the user can make to the system are limited to a set of rather simple commands, which only involve the interaction of a limited set of predefined agents.

An Hierarchical Task Network planning approach is adopted in [Gravot et al., 2006] for controlling a humanoid robot so that it performs certain cooking tasks. The planner bases on the description of predefined methods expressed in terms of the actions the robot can perform. A multi-agent approach is adopted by [Davidsson and Boman, 2005] to control a smart building's conditions with the objective of saving energy and increasing users' satisfaction. This approach however focuses on triggering some predefined rules as a reaction to certain events rather than on computing complex compositions of different services. In [Aker et al., 2011] the action description language C+ is used for modeling a housekeeping domain: multiple robots have to collaborate to tidy up by moving objects around the house, and the causal reasoner CCALC is applied to plan the robots' activities in a safe way. In [Giacomo et al., 2012], an extension of the Roman Model [Berardi, Calvanese, Giacomo, Hull and Mecella, 2005] with goal-based processes is used for composing domotic stateful services. In the proposed framework, the user can state some declarative goals that refer to transitions of some target process, i.e., a transition system which models some user's complex routine. In [Caruso et al., 2012] a framework for Home automation which combines offline composition based on the Roman Model approach with dynamic on-demand composition by using the RuG planner is presented.

A review of different uses of planning in Smart Homes is presented in [Simpson et al., 2006], where several research issues surrounding planning, such as plan recognition and knowledge representation, are considered. In [Bronsted et al., 2010], the main requirements and open challenges for service composition in pervasive environments are analyzed, including issues related to context awareness, contingency management, and device heterogeneity.

2.7.3 Smart Homes projects

A number of research and industrial projects focusing on supporting a wide range of household devices over heterogeneous network environments have been performed and are underway. The work presented in Chapter 4 has been conducted in the context of the European project Smart Homes for All (SM4ALL) [SM4All, 2008-2011], which builds upon the concept of service, and uses composability and semantic techniques, in order to guarantee dynamicity, dependability and scalability, while preserving the privacy and security of the home and its users. SOCRADES [Spiess et al., 2009] focuses on an SOA-based integration architecture which enables the collaboration of ubiquitous devices in the manufacturing domain with services offered at the enterprise application level. Like in our case, Web Services are embedded to devices and a publish-subscribe mechanism is used to handle events. However, only execution of pre-defined descriptions of service compositions, such as BPEL or mash-ups, is supported, and runtime flexibility is limited to selecting the right instances for a fixed sequence of service types.

HYDRA [Eisenhauer et al., 2010; Zhang and Hansen, 2008] proposes a service-oriented middleware platform for networked embedded systems, which supports a model-driven development of ambient intelligence applications, based on ontologies of semantic devices. Semantic rules are used for diagnosing possible malfunctioning in the system, however there is no support for intelligent composition generation to deal with such situations. The RUNES middleware [Costa et al., 2007] for embedded systems requires explicit connectors between components which may have inter-related functionalities, and which can be further organized into groups that can form stacks of overlay services. This design leads to a layered architecture, with limited however dynamicity and no automatic reasoning capabilities. The SM4ALL platform tackles the problem of home automation at a higher application level, and focuses on dynamic and runtime compositions of embedded services connected to the network, thus realizing a system that is more user-centric, customizable and context-adaptable with respect to the aforementioned infrastructures.

Several approaches stimulated from the field of Artificial Intelligence have been adopted by projects that seek to leverage the smartness exposed by homes equipped with smart sensors and actuators. In the context of the MavHome project, learning algorithms are employed in [Rao and Cook, 2004] to predict the occurrence of common activities that take place in a home, and decide whether it should take them over automatically. The Intelligent Buildings project, e.g., [Davidsson and Boman, 2005], builds upon an agent-based approach which has already been discussed in Section 2.7.2. In the course of the ThinkHome project, the use of neural networks is proposed in [Kastner et al., 2010] to learn the optimal values for the parameters of automation activities with respect to context and user preferences, such as specifying the best time to start heating a room based on weather conditions.

2.8 Business Process recovery

The approach for runtime BP reconfiguration by employing AI planning presented in Chapter 6 shares many concerns with previous work in the areas of BP recovery, adaptation and process interference, as discussed in Section 2.8.1. A comparative overview of previous approaches which employ automated planning for the purpose of runtime BP reconfiguration is presented in Section 2.8.2.

2.8.1 BP adaptation and repair

Design-time verification cannot cover for unexpected and unspecified runtime data interactions. In order to handle such unforeseen runtime events, runtime process changes can in many cases not be avoided. Changeability of business processes is a large research area focusing on the capabilities to change business processes at design-time or runtime, and a number of different frameworks have been proposed in that context.

The ADEPT project is designed to support the synchronization between several running instances of the same process. Any changes made by the user are incorporated into all of the running instances without interrupting their execution [Dadam and Reichert, 2009]. An improved version of the framework, ADEPT2, enables ad-hoc flexibility for process instances and controlled evolution of process schemas [Göser et al., 2007; Reichert and Dadam, 2009]. [Weske, 2001] provides an approach for enhancing flexibility by dynamic adaptation of running workflow instances. However, existing changeability frameworks are primarily requirements-driven and their adaptation capabilities are specially tailored to facilitate and support new business requirements (and, therefore, improve flexibility). As such, they do not incorporate the mechanisms to adapt the process in order to prevent erroneous business outcomes.

Adaptation is the ability of the implemented processes to cope with exceptional circumstances by modifying the process during runtime. A number of techniques have been proposed to support recovery from process execution inconsistencies. AGENTWORK [Müller et al., 2004] is a workflow management system which supports automated business process adaptations in a comprehensive way. Exceptions and necessary workflow adaptations are specified through a rule-based approach. Using this approach, the system is able to react to process-failures like unavailable resources or data. In [Xiao and Urban, 2008], an approach is proposed that deals with recovery of failing processes using dependency tracking based on incremental data changes. A global schedule of these data changes is used to detect data dependencies, in order to determine the impact of process failure and recovery procedures. In [Friedrich et al., 2010], repair processes are generated as a response to failing activities, based on a set of rules that specify how individual activities can be repaired through compensation or substitution. However, existing runtime solutions for process adaptation with the purpose of recovery are based on failing processes. That is, only those processes that fail during execution and terminate in an improper way are recovered. In practice, however, process interference does not necessarily cause processes to fail. More often, the processes finish regularly without any system errors from an internal perspective, leading however to erroneous results.

A solution more specific to process interference in Service-Oriented Computing is provided by [Urban et al., 2011]. Predefined (design-time) rules are used to specify the required compensation actions in case of interference. In addition to failing processes, this approach incorporates events like exceptional conditions or unavailable activities. Nevertheless, this approach does not consider problems occurring at a regularly executing process due to the use of inaccurate data. In [van Beest, Bulanov, Wortmann and Lazovik, 2010], a runtime intervention approach is proposed to repair BPs upon interference. However, the design-time specification of the required IPs still requires an extensive manual effort. In order to automate the IP generation, some extra semantic annotations are required for describing the BP. The benefits of adding semantics to BPs have long been acknowledged by the work in the field of Semantic Business Process Modeling, and exploited for a number of different purposes, such as automating process verification [Henneberger et al., 2008], which rely on a description in terms of preconditions and effects, or process model generation [Weber et al., 2010].

2.8.2 Automated planning for BP reconfiguration

The advantages of integrating AI planning techniques for several applications in the field of Business Process Management have long been acknowledged. For instance, different planning approaches can assist at the business process definition phase [Rodríguez-Moreno and Kearney, 2002; Rodríguez-Moreno et al., 2007; Madhusudan et al., 2004], while in [Jarvis et al., 1999], the use of planning in case of domain state changes. In order to facilitate (semi-)automatic adaptation at runtime, AI planning techniques have been used from different viewpoints in the literature. Goal-driven methods have been proposed for enabling variability, e.g., a planner based on situation calculus is used in [Liaskos et al., 2012] for computing sets of admissible configurations according to the constraints specified by users in terms of extended goals. In [Beckstein and Klausner, 1999], the use of an intelligent assistant based on AI planning techniques is discussed, which can suggest compensation workflows or the re-execution of activities as a response to execution failures, with the help of meta-level knowledge incorporated in the workflow semantics.

In [Ferreira and Ferreira, 2006], the use of machine learning is proposed in order to infer the preconditions and effects of activities, and generate a partially ordered execution plan that complies to these rules. The framework aims at providing a candidate process that is able of achieving some business goals. At execution time, if an activity fails, an alternative candidate plan is provided. Although the objective is different than strictly resolving process interference, a common concern with this framework's approach is the decoupling of the BP-specific constraints from the generic service repository, thus allowing the planner to generate partially ordered plans with a high degree of flexibility.

A line of work close to ours is the approach to BP adaptation through planning presented successively in [de Leoni et al., 2007, 2009; Marrella and Mecella, 2011]. This work uses several versions of Golog, which is based on planning by means of the situation calculus, to adapt a running process in case mismatches between the environment and the internal system representation are detected. In Golog, the goal to be achieved has to be specified in a procedural way, as a non-deterministic program, as opposed to the use of high-level declarative goals, as the ones used by domain-independent planners, like the one presented herein. This implies that the adaptation process has to be pre-specified in an action-centric way, which requires domain-specific knowledge of the available services and arduous hand-coding by a human expert. One advantage of the approach proposed in [Marrella and Mecella, 2011] is that it can manage any unforeseen event, by continuously comparing the environment with the expected outcomes according to the BP specification at each step of execution. The approach, however, only provides recovery policies that lead to the expected state as specified in the original process, and can thus not cover situations like the ones presented herein, which necessitate the fulfillment of extra requirements or the use of compensation activities.

A goal-driven approach, which uses an extended version of the model-based planner MBP [Shaparau et al., 2006], is adopted in [Bucchiarone et al., 2011, 2012] for BP adaptation. In that approach, service operations as well as context properties (e.g. "address") are modeled as state transition systems, which requires considerable manual effort. Every time a context change is observed, this has to be verified against all policies-goals, and if an inconsistency is detected then an adaptation is triggered. This process would be unable to detect inconsistencies in case of some data modification that does not violate the goal-policies, but still leads to erroneous results such as the delivery to an obsolete address. Although the approach supports uncertain effects, it only considers non-numeric context variables, and would suffer from state explosion and poor performance if monitored variables range over large domains.

Chapter 3

The RuG CSP-based Planner

In most service domains, the planning agent does not only have to act, but also to learn about the state of its surroundings. To achieve the latter, the planner can resort to sensing operations to acquire all the knowledge it misses and which is necessary to fulfill a goal. Sensing operations return exactly one outcome amongst a (possibly very large) set of deterministic choices, i.e., we assume full observability. The plan is constructed based on its limited offline knowledge and its information about how this knowledge state evolves through the execution of certain actions. The agent has to be able to reason about the possible outcomes of sensed information at plan time, and construct plans under uncertainty. However, things become more complicated by the fact that WS application domains involve to a large extent numeric-valued fluents, many of which have to be sensed. Because it is very common for observation actions to return numeric information, e.g., the price of an item, the date of an event, the temperature in a room etc. the planner must be able to consider a potentially very large number of possible resulting states, i.e., configurations of the physical world. A successful plan may be conditioned on the outcomes of such actions, e.g., the user may want to go ahead with buying a concert ticket only if it is guaranteed that some conditions about the weather, cost etc. hold. Similarly, some actions may have to be called taking as input arguments the yet unknown output of some observation action, e.g., the user may wish to deliver an item to some address that is retrieved by looking into an online catalog.

Given these characteristics of service domains, computing a contingent plan [Albore et al., 2009; Pryor and Collins, 1996], i.e., a plan which includes all possible branches resulting from the observed values, can be very expensive. Thus, models that work with propositional representations, and consider all potential runtime circumstances at the offline level to compute conditional plans, such as [Hoffmann and Brafman, 2005; To et al., 2011; Bertoli and Cimatti, 2002; Bryce et al., 2006], are not a recommended strategy for WS domains. Applications of contingency planning that seek to address the problem of WS composition with incomplete knowledge and sensing, like [Pistore, Marconi, Bertoli and Traverso, 2005; Bertoli et al., 2010; Hoffmann et al., 2010], perform poorly in scenarios where a large number of possible outcomes are involved and disregard numeric expressions.

Rather than that, we adopt a knowledge-level approach where plans are built based on the agent's knowledge and the way that this is changed by executing actions [Petrick and Bacchus, 2004], and under *optimistic* assumptions: plans are constructed with the anticipation that all knowledge-gathering actions which are necessary to provide the information necessary for achieving the requested goal return outcomes that are in accordance with the goal and the preconditions of the subsequent actions that lead to the desired situation. The intuition behind this behavior is that human agents often act in a similar sense under conditions of incomplete knowledge [Golden, 1998]: they plan ahead certain activities (e.g., open a door) assuming some favorable state of the world (e.g., that the door is unlocked). If at runtime they find out that the assumption they made is not true (e.g., the door is locked), they would have to come up with an alternative plan (e.g., try to first unlock the door). In most scenarios in WS application domains, planning in anticipation of convenient outcomes is a good strategy, given the large amount of available resources during search, e.g., it is likely to find some hotel room that satisfies some reasonable criteria. How the offline plan can be later revised under the light of runtime information and contingencies is described in Chapter 5. In this chapter, we focus on the offline plan which is computed considering the agent's knowledge state and the way that this is changed by actions.

3.1 Planning as constraint solving

Under conditions of uncertainty due to the incomplete knowledge about the initial state, the search space is no longer the set of states of the domain, but its power set. A plan in this sense represents a traversal between sets of states rather than complete description of states, and has only the *potential* to achieve the goal, if there is *some* auspicious state sequence that could arise from the plan's execution. Such a plan is usually referred to as *weak* in the literature [Cimatti et al., 2003]. We should emphasize that finding such a context-dependent plan is a much simpler task than computing a strong contingent plan with conditional branches which would satisfy the goal in *all* possible state sequences that could arise from observational effects. However, as already explained, postponing the computation of alternative contingent branches till more information is acquired from the environment seems to be a more feasible approach for WS scenarios that involve many numeric variables and output-to-input mappings, and being optimistic is likely to be paid off. Thus, all sources of non-determinism, where the actual behavior of actions at execution time contradicts the expected effects as modeled in the planning domain or external agents alter the world in unanticipated ways, are left to be treated by interleaving planning with execution and performing continual planning as described in Chapter 5.

Besides the requirement for dealing with incomplete knowledge and numeric sensed information, in order to model real-world service domains, one has also to consider numeric properties (e.g., the temperature must be greater than 0) both in preconditions and the goal, and be able to apply arithmetic operations (e.g., reduce a distance by 1 meter). Moreover, input arguments of actions, such as dates, places and so on, also range over large domains, and thus grounding actions may lead to an intractably large state space. Modeling the ultimate encoding as a *Constraint Satisfaction Problem* (CSP) is a particularly well-suited approach, if efficient handling of numeric expressions and variables of high cardinalities is at stake. CSP formalisms are expressive, since constraints in the context of a multi-valued encoding allow us to naturally go beyond logical formulas, and use arithmetic formulas in preconditions and effects without sacrificing efficiency. Complex declarative goals under uncertainty, which accommodate for ordering constraints, maintainability properties, numeric expressions and hands-off requirements can be also supported in the context of the CSP representation as is explained in Section 3.4.1. Equipping users with a rich language that allows them to specify a variety of different complex requests in a declarative way is particularly important for realizing user-centric and adaptable WS-enabled marketplaces [Papazoglou et al., 2002]. Moreover, describing the world in terms of variables which may range over a domain of possible values is particularly convenient for representing "under-defined" states: the range of the allowed values depends on the current knowledge of the agent and is narrowed down by the application of action effects. With such an encoding, states at which variables are not restricted to a specific value, represent sets of states, thus naturally encompassing uncertainty.

In general, CSP-based planners [Barták and Toropila, 2009] are not yet as competitive as the best-performing planners in International Planning Competitions, such as Lama [Richter and Westphal, 2010], or SAT-based planners. It should be mentioned though, that realistic service domains are usually not as structurally complex as the domains used in the International Planning Competitions (for example, the broadly used travel domain [Papazoglou et al., 2002] contrasted with the PDDL domains in IPC[1]). In service environments, complexity does not usually stem from highly transitive interdependencies between actions/service operations, but challenges come from other sources, such as incomplete knowledge and sensing, output-to-input parameter passings regarding variables that range over large

[1] `ipc.icaps-conference.org`

domains, the expressive power of the domain representation, the dynamic nature of context, and the support for complex goals. In the followings, we present how a CSP-based planner, called the *RuG planner*, can provide for a highly expressive action schema, endowed with a number of features that fit the characteristics of service domains, and are frequently overlooked by previous approaches. These features include parallel actions, which are very important given the large response times of many operations (especially sensing ones), handling of numeric variables and input parameters, numeric preconditions, and a large variety of effects. A powerful language for expressing complex goals, beyond the mere statement of properties that should hold in the final state, is supported. The planning problem modeling accommodates for incomplete knowledge and information-gathering, which is realized by an intuitive knowledge-level representation which is generated automatically, given the high-level description of the domain and the goal.

3.2 Planning Domain

Unlike most traditional planning formalisms that are predicate-based, in CSP-based planners the planning problem is usually described in terms of multi-valued state variables. Following this trend, the domain modeling we use herein is based on the Multi-valued Planning Task (MPT) encoding [Helmert, 2009], which leads to a smaller number of variables ranging over larger domains, something that is known to improve the performance of constraint solvers. We believe that the multi-valued state variable representation is particularly well-suited for modeling service domains. As we see in Section 4.4, this formalism is closer to the variable-based device-level descriptions which follow specification standards such as OSGi-UPnP [OSGi Alliance, 2009]. A similar observation holds for standard service description languages such as WSDL, or for workflow definition languages such as WS-BPEL, which also build on variables rather than propositions. Most importantly, the multi-valued encoding is a formalism that naturally supports variables ranging over large domains, which are often neglected in the classical planning literature. Constraints in the context of such an encoding allow to conveniently go beyond logical formulas, and use arithmetic formulas in preconditions and effects without special effort or major sacrifices in efficiency.

The planning domain that is supported by the RuG CSP-based planner is described in Definition 1. In order to deal with incomplete knowledge and sensing, the planning schema is enriched with a knowledge-level representation to model observational actions (sensing effects). Some extra types of effects, such as *invalidate*, are supported to address some special situations related to planning for Business Process recovery (see Chapter 6). Conditional effects are also provided for. Moreover, the planning formalism accommodates for numeric functions and effects beyond mere assignments, such as increase/decrease, which are necessary to model service operations such as increasing the overall price of an online shopping basket, or paying in some amount to a bank account. These are features that standard planning task descriptions used by CSP-based planners usually do not accommodate, e.g., compare with [Barták and Toropila, 2009]. This extra expressivity, along with the support for complex goals and the incorporation of runtime information, come at the cost of computational efficiency. Extensive use of implications, i.e., disjunctive constraints which are known to compromise the efficiency of constraint solvers, becomes unavoidable, while the alternative CSP formulations discussed in [Barták and Toropila, 2008] have restricted applicability (see Section 3.7).

Another characteristic of the supported planning schema is that input arguments to actions may range over numeric-valued domains just as all other variables. This is usually not allowed in traditional planning task descriptions [Fox and Long, 2003], which would rule out some actions which are very common in the field of services. For example, flying to a certain altitude would expect as input a number-valued argument. It should be noted that this input might be the output of some previous operation in a composition, and thus the planning domain formalism should be able to capture such input-output passings. The RuG planner resorts to an ungrounded (not instantiated) action schema. Therefore, in the followings we use the term *action* to denote what is usually called a *planning operator*. This should be kept in mind when comparing with other planning approaches to service composition, which by "number of actions" refer to the number of grounded operators. However, the grounded task can be exponentially larger than the original one, since an operator with n input arguments, either one having a domain of size D, results into $|D|^n$ many ground instances. This is a particularly important remark given that the input parameters of operators which model services commonly range over large domains, and thus, a single "action" in our representation may correspond to hundreds or thousands of actions reported by planning approaches that work on the grounded problem.

Definition 1 (Planning Domain (PD))**.** A Planning Domain is a tuple $\mathcal{PD} = \langle Var, Par, Act \rangle$, where:

- Var is a set of variables. Each variable $v \in Var$ ranges over a finite domain D^v.
- Par is a set of variables that play the role of input parameters to members of Act. Each variable $p \in Par$ ranges over a finite domain D^p. Par and Var are disjoint sets.

- A is the set of actions. An action $a \in \mathit{Act}$ is a quadruple $a = (\mathit{id}(a),\ \mathit{in}(a),\ \mathit{precond}(a), \mathit{effects}(a))$, where:

- $\mathit{id}(a)$ is a unique identifier
- $\mathit{in}(a) \subset \mathit{Par}$ are the input parameters of a
- $\mathit{precond}(a)$ is a propositional formula over $\mathit{Var} \cup \mathit{Par}$, which conforms to the following syntax:
  $\mathit{precond}(a) ::= \mathit{prop} \mid \mathit{precond}(a) \wedge \mathit{precond}(a) \mid$
  $\qquad \mathit{precond}(a) \vee \mathit{precond}(a) \mid \neg \mathit{precond}(a)$
  $\mathit{prop} \quad ::= \mathit{var} \circ \mathit{val} \ \mid\ \mathit{var_1} \circ \mathit{var_2} \ \mid (\mathit{var_1} \diamond \mathit{var_2}) \circ \mathit{val} \ \mid$
  $\qquad \mathit{known}(\mathit{var}) \ \mid\ \mathit{brel}(\mathit{var_1}, \ldots, \mathit{var_n})$
  where $\mathit{var}, \mathit{var_1}, \ldots, \mathit{var_n} \in (\mathit{Var} \cup \mathit{in}(a))$, val is some constant, $\circ$ is a relational operator ($\circ \in \{=, <, >, \neq, \leq, \geq\}$), $\diamond$ a binary operator ($\diamond \in \{+, -\}$), $\mathit{known}(\mathit{var})$ a boolean relation indicating that var is known, and brel an n-ary Boolean relation. We write $\bigwedge_i \mathit{precond_i}(a)$ to denote a sequence of conjunctions on preconditions, and likewise $\bigvee_i \mathit{precond_i}(a)$ for disjunctions.
- $\mathit{effects}(a)$ is a conjunction of any of the following elements:
  - $\mathit{assign}(\mathit{var}, v)$, where v is some constant or $v \in \mathit{Var}$
  - $\mathit{assign}(\mathit{var}, f(v_1, v_2))$, where $v_1, v_2 \in (\mathit{Var} \cup \mathit{in}(a))$ or v_1, v_2 are constants, and f the sum or the subtract function
  - $\mathit{increase}(\mathit{var}, v)$ or $\mathit{decrease}(\mathit{var}, v)$, where $v \in \mathit{Var} \cup \mathit{in}(a)$ or v is some constant
  - $\mathit{sense}(\mathit{var})$, where $\mathit{var} \in \mathit{Var}$
  - $\mathit{cond\_effect}(\mathit{prop}, \mathit{effect}(a))$, which models a conditional effect, that is applied at the next state only if prop holds at the current state.
  - $\mathit{sense\text{-}cond\_effect}(\mathit{prop}, \mathit{effect}(a))$, which models a conditional effect that is applied at the next state only if prop holds at the next state. This is used for effects that should materialize only if the outcome of a sensing effect satisfies prop.
  - $\mathit{invalidate}(\mathit{var})$, where $\mathit{var} \in \mathit{Var}$. This effect states that var becomes unknown.

Variables in Var can only change values due to some action application, while input parameters are left free to be assigned by the planner any value that is convenient for the purpose of achieving the goal. A planning state s is defined as a relation $s = \{(x, D_s^x) \mid \forall x \in \mathit{Var} \cup \mathit{Par}\}$, where $D_s^x \subseteq D^x$, and D^x is the domain of x. Thus, the notion of a state adopted herein encompasses a *set* of traditional states representing assignments of values to the variables, and allows us to accommodate for incomplete knowledge. The domain of x at state s is given by the *state-variable* function $[\![x]\!](s)$, so that $[\![x]\!](s) = D_s^x$ if $(x, D_s^x) \in s$. If $|D_s^x| = 1$, this means that x at s has a specific value. An action a is applicable on state s if its preconditions hold at s, and its execution leads to a successor state s'. The propositions in $precond(a)$ refer to the values of variables Var and parameters Par at state s, whereas the updates instructed by $effects(a)$ refer to the variables Var at state s'. We say that $precond(a)$ holds at s (or alternatively that s *satisfies* $precond(a)$) if $precond(a)$ evaluates to true for *all* possible assignments to values that are consistent with the domains of the variables at s. This implies for example that given a state $s = \{(v_1, \{1\}), (v_2, \{1, 2\})\}$ and an action a which has a precondition $v_1 = v_2$, a is not applicable at s. As we see in Section 3.3 $effects(a)$ also amount to a conjunction of propositions that should hold at s'.

The effects of type $sense(var)$ are called *observational* or knowledge-providing, i.e., they observe the current value of a variable, while the rest types of effects are *world-altering*, i.e., actively change the value of a variable. Variables that are part of sensing effects correspond to WS outputs, e.g., indicated with annotations in the XML Schema used in WSDL documents. An action may have both observational and world-altering effects. An interesting remark at this point is that the investigation performed in [Fan and Kambhampati, 2005] acknowledges that the majority of WS operations driven from public WS repositories only update the agent's knowledge state rather than the world state. To provide for incomplete information and sensing, the domain is extended by additional variables to model the knowledge-level representation and distinguish between sensing and world-altering actions. These variables are generated automatically given a domain description $\mathcal{SD}$. First, for each $var \in Var \cup Par$, a new boolean variable var_known is introduced, which indicates whether var is known at state s ($[\![var\_known]\!](s) = true$) or not ($[\![var\_known]\!](s) = false$). If proposition $known(var)$ holds at state s, this is equivalent to $[\![var\_known]\!](s) = true$. The role of these knowledge-base variables becomes obvious when we explain how effects and goals are translated into constraints. For every variable $kvar \in Var$ that participates in an observational effect, we introduce a new variable $kvar\_response$, which is a placeholder for the value returned by the respective sensing operation. Since this value is unknown until execution time, $kvar\_response$ ranges over $kvar$'s domain ($kvar\_response \in D^{kvar}$). We also maintain for every variable $cvar \in Var$ that is part of at least one world-altering effect a boolean flag $var\_changed$, which becomes true whenever this effect takes place. Thus, we end up with an extended set of variables $V = Var \cup Kb \cup Cv \cup Rv$, where Kb is the set of knowledge-base variables, Cv the set of the change-indicative variables, and Rv the response variables. States are also extended to include all variables in $V \cup Par$.

On top of the action descriptions in $\mathcal{PD}$, there may be a set of general constraints, which capture a simple aspect of the ramification problem [Finger, 1987], i.e., indirect effects of actions on variables. General constraints resemble the rules for the derived predicates discussed in [Edelkamp and Hoffmann, 2004]. For the RuG planner, a general constraint is an implication constraint which states that if the value of a variable $\mathit{var_1} \in \mathit{Var}$ has some specific value(s), then a unique value of $\mathit{var_2} \in \mathit{Var}$ can be concluded as well. A general constraint has the form $\bigvee_i \mathit{var_1} = v_i \Rightarrow \mathit{var_2} = v$, where v_i are some constants $v_i \in D^{\mathit{var_1}}$ and $v \in D^{\mathit{var_2}}$. For example, let us consider an action which moves an actor between certain locations which model topological places that in turn belong to rooms (see Figure 3.1). The effect of the action refers to an assignment to the variable *robotLoc* which indicates the location of the moving robot, which however may also imply a change to variable *robotRoom* that models the room in which the robot currently is. The latter can be modeled as a function of the former, since knowing the location we can infer the room. As we see in Section 3.3, the relation between the variables involved in general constraints has to be considered in the axioms that are generated to address the frame problem.

3.3 Encoding the Planning Domain into a CSP

Following a common practice in many planning approaches, we consider a *bounded* planning problem, i.e., we restrict our target to finding a plan of length at most k for an a priori given integer k. In the followings, we explain how the service domain is encoded into a CSP, for some given integer k. The process is similar to the one described in [Ghallab et al., 2004] (alternative encodings based on the planning graph are proposed in [Kambhampati, 2000; Do and Kambhampati, 2001]).

A constraint satisfaction problem and its solution are defined as follows:

Definition 2 (CSP)**.** A *Constraint Satisfaction Problem* is a triple $CSP = \langle X, \mathcal{D}, \mathcal{C} \rangle$ where:

- $X = \{x_1, \ldots, x_n\}$ is a finite set of n variables.
- $\mathcal{D} = \{D^1, \ldots, D^n\}$ is the set of finite domains of the variables in X, so that $x_i \in D^i$.
- $\mathcal{C} = \{c_1, \ldots, c_m\}$ is a finite set of constraints over the variables in X. A constraint c_i involving some subset of variables in X is a proposition that restricts the allowable values of its variables.

Definition 3 (Solution to a CSP)**.** A solution to a CSP $\langle X, \mathcal{D}, \mathcal{C}\rangle$ is an assignment of values to the variables in X, $\{x_1 = v_1, \ldots, x_n = v_n\}$, with $v_i \in D_i$, that satisfies all constraints in $\mathcal{C}$.

Considering a planning domain extended with the knowledge-level representation $\mathcal{PD}' = \langle V, Par, Act\rangle$, the aim is to encode $\mathcal{PD}'$ into a $CSP = \langle X, \mathcal{D}, \mathcal{C}\rangle$. First, the variables X are derived as follows: for each variable $x \in V \cup Par$ ranging over D^x, and for each $0 \leq i \leq k$, we define a CSP variable $x[i]$ in CSP with domain D^x. Actions are also represented as variables: for each action $a \in Act$ and for each $0 \leq i \leq k-1$ a boolean variable $a[i]$ is defined. This way the computed plan can include parallel actions, a fact that may save time at execution.

After deriving the CSP state variables X, the actions' preconditions and effects are encoded into constraints. Given an action $a = (id(a), in(a), precond(a), effects(a))$, we use the notation $precond(a)[i]$, $prop[i]$ and $effect(a)[i]$ to indicate the preconditions, propositions and effects on the state variables corresponding to state i. Thus, $precond(a)[i]$ ($effect(a)[i]$ respectively) results from substituting every variable $x \in X$ which appears in $precond(a)$ ($effect(a)$) by its corresponding state variable $x[i]$. For each action a, and for each $0 \leq i < k$:

- We add the constraint
  $a[i] = true \Rightarrow precond(a)[i] \ \wedge \ \bigwedge_{v \in precond(a)} v\_known[i] = true$
- We add a constraint which enforces that all input parameters should be known:
  $a[i] = true \Rightarrow \bigwedge_{p \in in(a)} p\_known[i] = true$
- For every $effect_j$ in $effects(a)$, we add a constraint of type
  $a[i] = true \Rightarrow \bigwedge_j constr(effect_j)[i+1]$,
  where $constr(effect_j)[i+1]$ is a constraint derived depending on the type of $effect_j$:

  - Case $assign(var, v)$:
    - $var[i+1] = v[i] \wedge var\_know[i+1] = true \wedge$
      $var\_changed[i+1] = true \ \wedge v\_known[i] = true$, if $v \in Var \cup in(a)$
    - $var[i+1] = v \ \wedge \ var\_know[i+1] = true \wedge$
      $var\_changed[i+1] = true$, if v some constant

    Similarly, for the effect of type $assign(var, f(v_1, v_2))$
  - Case $increase(var, v)$:
    - $var[i+1] = var[i] + v[i] \ \wedge \ var\_changed[i+1] = true \wedge$
      $v\_known[i] = true$, if $v \in Var \cup in(a)$
    - $var[i+1] = var[i] + v \wedge var\_changed[i+1] = true$, if v some constant

Respectively, for the effect of type $decrease(var, v)$.

- Case $sense(var)$:
  $var[i + 1] = var\_response \ \wedge \ var\_known = true$
- Case $cond\_effect(prop, effect)$:
  $prop[i] \ \Rightarrow \ constr(effect)[i + 1]$
- Case $sense\text{-}cond\_effect(prop, effect)$:
  $prop[i + 1] \ \Rightarrow constr(effect)[i + 1]$

Note that the translation of effects into constraints may entail the addition of constraints that should hold at the previous state (precondition). For example, to assign some variable v to some other variable var, v should already be known. In cases where $precond[i]$ includes some Boolean relation $brel(var_1, \ldots, var_n)[i]$, this is substituted by a proposition on these variables, according to translation rules specific to the relation *brel*. Depending on the relation, the resulting constraints may be less or more complex. For example, *adjacent_same_room(loc1, loc2)* (see Figure 3.1) is translated into a disjunction which includes all possible allowed value pairs of variables $loc1, loc2$ (i.e., grounding is not avoided): $\bigvee_{c1_i, c2_j}(loc1 = c1_i \wedge loc2 = c2_j)$, for all location values $c1_i, c2_j$ which are adjacent and belong to the same room. On top of the domain description, restrictions referring to the initial state are expressed in the form of a conjunction of propositions $\bigwedge_i prop\_init_i$, where $prop\_init_i$ are propositions on the variables $x \in X$. The encoding of the (partial) description of the initial state corresponds to the addition of the constraints $prop\_init[0]$ for each proposition *prop_init* that refers to the initial state.

A strong requirement that all variables involved in the preconditions should be known is also added as part of the precondition constraints. This ensures that the preconditions hold for all possible assignments to variables consistent with their allowed domain at a given state, as already mentioned. On the other hand, this excludes the applicability of an action in some cases that would be admissible. For example, given a state $s = \{(v, \{1, 2\}), (v\_known, \{false\})\}$ and an action a with precondition $v < 3$, a cannot be applied at s. This restriction is necessary however to prohibit undesirable situations, such as allowing the application of a at a state $s = \{(v, \{1, 2, 3\}), (v\_known, \{false\})\}$. In such a case, the constraint solver would be able to find *some* assignment that satisfies the constraints, which however is undesirable, since we cannot be sure if the application of a is safe, given the uncertainty about v's actual value. Only if there is a way to sense v's value, should the application of a be permitted. This restriction implies that the RuG planner is not able to handle problems with partial observability such as the ones addressed in [Petrick, 2011], where actions can be applied even if some variable in the preconditions is not known.

Considering the general constraints, these are also translated at the level of CSP variables as constraints that should hold at all states. Thus, for each general constraint $\bigvee_i \mathit{var_1} = v_i \Rightarrow \mathit{var_2} = v$ and for each $0 \leq i < k$ the constraint $\bigvee_i \mathit{var_1}[i] = v_i \Rightarrow \mathit{var_2}[i] = v \ \wedge\ \mathit{var_2\_known}[i] = \mathit{true}$ is added. Frame axiom constraints are also generated, which guarantee that variables cannot change between subsequent states, unless some action that affects them takes place. For every $v \in (V - Rv)$ and for each $0 \leq i \leq k-1$, we add the constraint

$$\bigwedge_j (\mathit{actionAff}(v)_j = 0) \Rightarrow v[i] = v[i+1],$$

where $\mathit{actionAff}(v)_j$ are the actions affecting v, i.e.,the actions for which v appears in the left side of some equality involved in the constraints derived from their effects. If v appears in the right side of the implication of a general constraint, then actions whose effects involve the variable on the left side of the respective general constraint are also included in $\mathit{actionAff}(v)_j$.

The set of constraints that comprise *CSP* are further extended by additional constraints that constitute the goal (see Section 3.6), yielding the planning problem in the form of a CSP that are passed to the constraint solver. The handling of the sensing effects allows the offline solver to assign arbitrary values to unknown variables, however if the corresponding knowledge variable *var_known* is false, this value is of no validity: the formulation of actions' preconditions and effects do not allow it to be assigned to any other variable, or considered as satisfying the goal. By adopting such an encoding, the required sensing actions are determined pro-actively, depending on the goal and the knowledge the user already possesses. Another effect of the encoding is that the planner always generates an optimistic plan, i.e., anticipating that all knowledge-gathering actions return information that is in accordance with the user's requirements, and all actions are executed successfully. This happens because the solver is free to make any assignments to the unknown variables, that fit its purposes, as long as they do not violate the constraints entailed by the domain and goal constraints. This initial plan is revised during execution as we see in Chapter 3.

3.3.1 Some action examples

Figure 3.1 illustrates two simple examples of actions expressed in terms of preconditions and effects, and shows the constraints associated with them, as a result of the process described in Section 3.3. The usability of effects like *invalidate* and *sense-cond_effect* is exhibited in Section 6.4, where it is shown how these effects model deferred choices in a Business Process. Examples of BP activities represented in terms of preconditions and effects can be found in Appendix A.2.1.

payIn(amountPar, accIdPar)
  `prec:` $\emptyset$
  `effects:` $\mathit{increase}(\mathit{accBalance}, \mathit{amountPar})$

CSP constraints, $\forall_{0 \leq i < k}$:
`prec constraints:` /*parameters known*/
$\mathit{payIn}[i] = \mathit{true} \Rightarrow$
$(\mathit{amountPar\_known}[i] = \mathit{true} \wedge \mathit{accIdPar\_known}[i] = \mathit{true})$
`effect constraints:` /*world-altering*/
$\mathit{payIn}[i] = \mathit{true} \Rightarrow (\mathit{accBalance\_changed}[i+1] = \mathit{true} \wedge$
$\mathit{accBalance}[i+1] = \mathit{accBalance}[i] + \mathit{amountPar}[i])$

findAccBalance(accIdPar)
  `prec:` $\emptyset$
  `effects:` $\mathit{sense}(\mathit{accBalance})$

CSP constraints $\forall_{0 \leq i < k}$:
`prec constraints:` /*parameters known*/
$\mathit{findAccBalance}[i] = \mathit{true} \Rightarrow (\mathit{accIdPar\_known}[i] = \mathit{true} \wedge$
`effect constraints:` /*sensing*/
$\mathit{findAccBalance}[i] = \mathit{true} \Rightarrow \mathit{accBalance\_known}[i+1] = \mathit{true} \wedge$
$\mathit{accBalance}[i+1] = \mathit{accBalance\_response}[i+1]$

moveRobot(robotLocPar, robotRoomPar)
`prec:` $\mathit{robotLoc} \neq \mathit{robotLocPar} \wedge$
$(\mathit{adjacent\_same\_room}(\mathit{robotLoc}, \mathit{robotLocPar}) \vee$
$(\mathit{adjacent\_diff\_rooms}(\mathit{robotLoc}, \mathit{robotLocPar}) \wedge$
$\mathit{door\_open}(\mathit{robotRoom}, \mathit{robotRoomPar})))$
`effects:` $\mathit{assign}(\mathit{robotLoc}, \mathit{robotLocPar})$

Location - room general constraints
$\mathit{robotLoc} = \mathit{LOC11} \vee \mathit{robotLoc} = \mathit{LOC12} \Rightarrow \mathit{robotRoom} = \mathit{ROOM1}$
$\mathit{robotLocPar} = \mathit{LOC11} \vee \mathit{robotLocPar} = \mathit{LOC12} \Rightarrow$
$\mathit{robotRoomPar} = \mathit{ROOM1}$ etc.

Figure 3.1: Three action examples expressed in terms of constraints.

CSP constraints $\forall_{0 \leq i < k}$:
prec constraints:
$moveRobot[i] = true \Rightarrow (robotLocPar\_known[i] = true \wedge$
$robotRoomPar\_known[i] = true \wedge$
$robotLoc = robotLocPar \wedge$
$((robotLoc[i] = LOC11 \wedge robotLocPar[i] = LOC12) \vee$
$(robotLoc[i] = LOC12 \wedge robotLocPar[i] = LOC11) \vee \ldots) \vee$
$((robotLoc[i] = LOC11 \wedge robotLocPar[i] = LOC21 \wedge$
$doorROOM1\_ROOM2[i] = OPEN) \vee$
$(robotLoc[i] = LOC21 \wedge robotLocPar[i] = LOC11 \wedge$
$doorROOM1\_ROOM2[i] = OPEN) \vee \ldots)$
effect constraints:
$moveRobot[i] = true \Rightarrow robotLoc[i+1] = robotLocPar[i] \wedge$
$robotLoc\_known[i+1] = true \wedge \quad robotLoc\_changed[i+1] = true$

CSP-level general constraints, $\forall_{0 \leq i \leq k}$:
$robotLoc[i] = LOC11 \wedge robotLoc[i] = LOC12 \Rightarrow$
$robotRoom[i] = ROOM1 \wedge robotRoom[i]\_known = true$

Figure 3.1: (continued)

3.3.2 Implicit predicates in the knowledge base

Although knowledge-level variables reflect whether a state variable is known or not, they cannot capture the presence or absence of information about functions (propositional or not). The question is how to model the fact that the planner knows that, e.g., a hotel room has been booked for some specific place and room parameters, i.e., that *bookedHotel* is true for some certain values of *hPlacePar* and *hDatePar*. As already explained, grounding the domain is not a feasible option because of the many input parameters which range over large domains. Therefore, in order to keep track of the knowledge about variables that are predicated on certain input parameters, a separate modeling is required, to allow distinguishing between information that refers to different input parameters, and enables the planner to make the appropriate output-to-input assignments. As new observations are made at execution time, the knowledge base facts and the respective constraints change, as we see in Chapter 5. How parametrized goals are expressed is discussed in Section 3.4.1.

The planner maintains two structures to store its knowledge about the values of variables. *knowlBase* is a map that keeps the values of variables predicated on certain parameter values, i.e., the fact that $var = val$, where $var \in Var$ and

$val \in D^{var}$, given some set $\{(p_1 = c_1), \ldots, (p_n = c_n)\}$, where $p_i \in Par$ and $c_i \in D^{p_i}$. *knownVars* stores the known values of variables which do not depend on any parameters.

For each entry of *knowlBase* $\{(p_1 = c_1), \ldots, (p_n = c_n)\} \mapsto (var, val)$, a *virtual KB action* is added to the planning domain. This virtual action has as input parameters the list $\{p_1, \ldots, p_n\}$, preconditions $\bigwedge_i (p_i = c_i)$, and an effect called virtual assign $virtAssign(var, val)$. The constraints capturing this assignment are the same as the ones of the standard $assign(var, val)$, except that the change indicatory action *var_changed* is not set to true. Thus, a virtual KB action simulates a sensing action whose output is known in advance. This way, grounding is performed only with respect to what the planner knows, which is expected to be limited, especially in comparison with the number of all possible configurations that could exist. Virtual KB actions are considered in the formation of frame axioms just like other actions. The planner will always try virtual actions before actual actions.

Regarding the *knownVars* list, for each $(var = val) \in knownVars$, the CSP variable $var[0]$ is assigned to *val*, and $var\_known[0]$ to true. The planner always starts from an initial state where all variables which are not part of *knownVars* are unknown. This implies that the plan has to include virtual actions into the plan, to transition to a state that represents what it actually already knows. The information included in *knowlBase* and *knownVars* may be annotated by some timestamp and expiration time, after which it is removed from the map, i.e., considered not to be known anymore. Therefore, the initial state and the set of virtual KB is constructed anew every time the planner is triggered.

The CSP solver may choose whatever virtual actions reflecting the *knowlBase* map suit its purposes, and assign their input parameters accordingly. For example, consider a goal about delivering a parcel to the address where some given person, “Peter Pan”, lives. If *knowlBase* already contains the entry *namePar* = “*PeterPan*” $\mapsto$ *catalAddress* = “*Neverland*”, then the planner will will choose the respective virtual action *KBSense1* with input parameters *namePar* = “*PeterPan*” to retrieve the desired value for *catalAddress*, and then proceed to the reservation action. A complete example is shown in Section 3.4.3

3.4 Goal language

Till now, we have described the representation and encoding into a CSP of the planning domain and the initial state. In the followings we present the syntax and semantics of the goal, and show how this can be translated into a set of constraints which together with the constraints formulating the planning domain and initial state constitute the final CSP which is passed to the constraint solver. The goal language supported by the RuG planner equips the user with potent constructs for expressing complex goals, beyond the mere statement of properties that should hold in the final state. Conditions over state traversals, maintainability properties, and distinguishing between wish to observe the environment and wish to change it are some of the features this language supports. These aspects are expressed in a declarative way, so that the user doesn't have to know about the operational details of the available operations and how they can be combined. The basic goal operators have been first presented in [Kaldeli et al., 2009*a*]. The RuG planner goal language shares many common concerns with the aspects presented in [Golden and Weld, 1996], such as the distinction between hands-off observations and accomplishment goals, while many constructs of its formalization resemble the syntax of XSRL [Papazoglou et al., 2002; Lazovik et al., 2005].

3.4.1 Goal syntax

The goal syntax is defined as follows:

$$
\begin{array}{lcl}
\textit{goal} & ::= & \bigwedge_i(\textit{condition-goal}_i \mid \textit{condition\_or\_not-goal}_i \\
 & & \mid \textit{subgoal}_i) \\
\textit{condition-goal} & ::= & (\textit{subgoal})\ \texttt{under\_condition}\ \textit{goal} \\
\textit{condition\_or\_not-goal} & ::= & (\textit{subgoal})\ \texttt{under\_condition\_or\_not}\ \textit{goal} \\
\textit{subgoal} & ::= & \texttt{final}\ (\textit{props}) \mid \\
 & & \texttt{achieve}(\textit{props}) \mid \\
 & & \texttt{achieve-maint}\ (\textit{props}) \mid \\
 & & \texttt{all\_states}\ (\textit{props}) \mid \\
 & & \texttt{find\_out-maint}\ (\textit{props}) \mid \\
 & & \texttt{find\_out}\ (\textit{props})
\end{array}
$$

where *props* is a propositional formula as the $precond(a)$ defined in Definition 1, with $var, var_1, \ldots, var_n \in (Var \cup Par)$. All variables and parameters not specified in the goal (or the initial state constraints) are assumed to be undefined (i.e., their respective knowledge-level variables are set to false).

The `final` subgoal is satisfied if *props* holds at the last state, while `achieve` requires that *props* should be true at *some* state over the state traversal. The `maint` annotation adds the requirement that once the respective propositions become true at some state, they should remain true in all subsequent states till the final one. `all_states` imposes that *props* should be true at all states, and is usually applied on input parameters whose values are set by the user. The `find_out` type of subgoals enforces a hands-off requirement on the variables the respective propositions involve, i.e., the planner tries to satisfy the propositions at some state without allowing any world-altering effect on these variables before that state. `find_out-maint` ensures that the involved variables should remain intact at all states of the plan. For instance, the goal `find_out`(*account_balance* > *100*) will be satisfied if the sensed value of the variable *account_balance* is greater than 100, without however allowing any action to alter the variable's value before the sensing action. On the other hand, if the goal is `achieve`(*account_balance* > *100*), the planner will do everything possible in order to fulfill the proposition, e.g., it might invoke a *pay_in* action that increases the *account_balance* by some amount.

Subgoals can be further on combined through the condition goal constructs, which impose some conditions that should be assured before the fulfillment of the subsequent subgoal. $subgoal_0$ `under_condition` $goal_1$ is satisfied if $subgoal_0$ is satisfied for the first time at some state *s* (see Definition 7) and $goal_i$ is satisfied in the state sequence preceding *s*. `under_condition` thus imposes a before-then relation between goals over the state traversal, and is particularly useful in cases where the user would like to go ahead with altering some variable, only if its sensed value satisfies some property beforehand. `under_condition_or_not` allows the expression of what can be seen as some kind of soft requirements: $subgoal_0$ `under_condition_or_not` $goal_1$ will also be fulfilled if $goal_1$ is not satisfiable; if however it is, then $subgoal_0$ has to be as well. It should be mentioned that the `under_condition_or_not` structure works as intended only if the variables involved in $goal_1$ are known at planning time. The formal semantics of the goal language is provided in Section 3.5.

3.4.2 Some goal examples

Several examples of goals which combine various constructs and express the requirements of the different scenarios dealt with in this thesis can be found in Chapters 4,6 and 5. In the followings, we present two simple examples to demonstrate the use of some basic constructs supported by the goal language:

Goal 1
`achieve-maint`(*bookedConcert* = *TRUE*) `under_condition`
  (`find_out-maint`(*temperature* > *0*))

Goal 2
`achieve-maint`(*bookedHotel* = *TRUE*) ∧ (
`achieve-maint`(*bookedConcert* = *TRUE*)
 `under_condition_or_not` (`find_out-maint`(*temperature* > *0*)))

Goal 3
$\bigwedge_i \texttt{achieve}(\textit{robotLocation} = \textit{room}_i)$

Goal 1 is accomplished if s is the first state at which *bookedConcert* = *TRUE* is satisfied, and $\texttt{find\_out-maint}(\textit{temperature} > \textit{0})$ is satisfied in the state sequence preceding s (in this example, the maintainability requirement imposed by `find_out-maint` is in practice redundant because there is no way to change the weather). If *temperature* < *0*, then Goal 1 fails. On the other hand, Goal 2 ensures that *bookedConcert*= *TRUE* will be satisfied if the temperature is not below zero, while if it is, then only *bookedHotel* = *TRUE* will be looked after. Goal 3 expresses that a robot should visit *all* rooms in a house, leaving the order of visits to be computed by the planner depending on the structure of the house.

3.4.3 Goals with parameters

It still remains unclear how to represent functions in the goal, e.g., how to say that *bookedHotel*(*hPlacePar*, *hDatePar*) is desired, where *hPlacePar* and *hDatePar* can be either a specific value or refer to some other variable, that may correspond to the yet unknown outcome of some other action (e.g., *hDatePar* = *eventDate*). In approaches where actions and propositional functions are grounded, to reach a final state that satisfies *bookedHotel*(*"Groningen"*, *"12-04-2012"*), the respective propositional variable should appear in the effects of some grounded instance of some operator (e.g., *bookHotel_Groningen_12042012*). But how can such a goal be expressed and satisfied in a variable-based ungrounded context? Actually, what an expression like *bookedHotel*(*"Groningen"*, *"12-04-2012"*) implies is that the input arguments of the action which fulfills *bookedHotel*, should be set to *"Groningen"*, and *"12-04-2012"* respectively. The effects of this action satisfy the proposition by setting the variable *bookedHotel* to true.

To capture such expressions we introduce a special notation *prop* `withParams` $\bigwedge_j par_j = v_j$, where *prop* refers to some proposition that should hold at state i, and $\bigwedge_j par_j = v_j$ with $par_j \in Par$ should hold at state $i-1$. v_j can be either a constant $v_j \in D^{par_j}$ or a variable $v_j \in Var$. According to this notation we would thus write:

bookedHotel = *true* `withParams` (*hPlacePar* = *"Groningen"*∧
hDatePar = *"12-04-2012"*).

A goal that requests booking two hotel rooms on different dates would look like:

`achieve`(*bookedHotel* = *true* `withParams`
(*hPlacePar* = *"Groningen"* ∧ *hDatePar* = *"12-04-2012"*)) ∧
`achieve`(*bookedHotel* = *true* `withParams`
(*hPlacePar* = *"Rotterdam"* ∧ *hDatePar* = *"13-04-2012"*)).

Output-to-input matchings can be captured in the goal by binding parameters to other variables. Recalling the example in Section 3.3.2, the goal would look like this:

`final`($delivery = true$ `withParams` ($destinationPar = catalAddress$))
  `under_condition`
   `find_out-maint`($known(catalAddress)$ `withParams` ($namePar =$ "*PeterPan*"))

This expresses the wish to perform some delivery to a destination which represents the address of "Peter Pan", which can be retrieved, e.g., from some sensing action provided by an online catalog. That sensing action should be performed with the right assignment ($namePar =$ "*PeterPan*"), and the delivery should be performed on the respective output, otherwise the goal cannot be satisfied (see also the concrete semantics of the goal constructs in Section 3.5).

If the knowledge base already includes the entries
($namePar =$ "*PeterPan*") $\mapsto$ ($catalAddress =$ "*Neverland*"), and
($namePar =$ "*Alice*") $\mapsto$ ($catalAddress =$ "*Wonderland*")
then the following two virtual KB actions are added to the planning domain, as described in section 3.3.2:

KB1($namePar$)	*KB1*($namePar$)
prec: $namePar =$ "*PeterPan*"	prec: $namePar =$ "*Alice*"
eff: $virtAssign(catalAddress,$ "*Neverland*")	eff: $virtAssign(catalAddress,$ "*Wonderland*")

Given these facts, the planner produces the plan:

KB1($namePar =$ "*PeterPan*"), $deliver(destinationPar =$ "*Neverland*").

A goal which requests a delivery to both Alice and Peter Pan is also satisfiable, by a plan like:

KB1("*PeterPan*"), *deliver*("*Neverland*"), *KB2*("*Alice*"), *deliver*("*Wonderland*")

(for readability reasons, we put directly the input parameters values along with the actions). The applicability of *knowlBase* becomes more evident in Chapter 5, where the entries in the knowledge base change according to the observations made during execution.

3.5 Representing the planning problem

Based on the planning domain as described in Definition 1, a *State Transition System* (STS) Σ evolves by specifying a *state-transition function* γ on the state and action sets. The γ function is applied to a state and leads to a *set* of states. This is due to the fact that *effects*(a) do not only model assignments to values, but also to outcomes that are unknown offline, i.e., do not have a specific value. As described in the process of constraints derivation in Section 3.3, *effects*(a) entail a conjunction of propositions *constr_effect*$_j$ that should hold at the successor state. Recalling that a state s satisfies a propositional formula *props* only if all possible combinations of values that are members of the domains of the variables at s satisfy *props*, it follows that an effect of type *sense*(var) leads to a set of states: the proposition $var = var\_response$ should hold at the successor state, with $var\_response \in D^{var}$, which amounts to $|D^{var}|$ different states. This way, the γ function captures a sense of incomplete knowledge and offline non-determinism in the case of knowledge-gathering actions.To support the possibility of applying multiple concurrent actions at a single transition, and be able to give a definition of plan that provides for parallel actions, the γ function takes as argument a set of actions.

Definition 4 (State Transition System)**.** A state transition system based on a planning domain extended with the knowledge-level representation $\mathcal{PD}' = \langle V, Par, Act\rangle$ is a triple $\Sigma = \langle S, Act, \gamma\rangle$, such that:

- S is a set of states $S = \{s = \{(x_1, D_s^{x_1}), \ldots, (x_n, D_s^{x_n})\}$, with $x \in V \cup Par$ and $D_s^{x_i} \subseteq D^{x_i}$.
- $\gamma : S \times \wp(Act) \to \wp(S)$ (where $\wp$ denotes the powerset) is the transition function: given a state s and a set of actions $A = \{a_1, \ldots, a_n\} \subset Act$ so that *precond*(a_i) hold at s for all $a_i \in A$, the application of all *effects*(a_i) on s leads to a set of successor states S_s. If for some a_i, *precond*(a_i) do not hold at s, or if $A = \emptyset$, $\gamma(s, A) = \emptyset$.

Generalizing on sets of states S, we define $\hat{\Gamma}(S, A) = \bigcup_{s\in S} \gamma(s, A)$. Having provided the syntax of the goal and the notions related to the state transition system modeling the planning domain, we can now proceed to the definition of the planning problem and plan.

Definition 5 (Planning Problem)**.** A *planning problem* is a triple $P = \langle \varSigma, S_0, g\rangle$, where Σ is a transition system as in Definition 4, S_0 is the set of all states which satisfy a conjunction of propositions *prop_init*$_i$, and g is a goal.

Definition 6 (Plan)**.** A *plan* consists of a sequence of action sets $\pi = \langle A_0, \ldots, A_{k-1}\rangle$, where k is the length of the plan, and a sequence *inPars* of assignment relations *inPars*$_i$ for each $A_i \in \pi$. *inPars*$_i$ is defined as
$\{(p := c_p) \mid \forall p \in in(a)\ \forall a \in A_i\}$, where $c_p \in D^p$.

Note that many A_is may refer to empty sets, since k is set to some number greater than the expected maximum plan (see also Section 3.7). We extend the $\hat{\Gamma}$ function to capture the set of states that are brought forth by applying the actions in π, starting from $S_0[inPars_0]$, where $S_0[inPars_0]$ is S_0 with the domains of input parameters updated according to $inPars_0$. Given an action sequence $\pi = \langle A_0, \ldots, A_{k-1} \rangle$, and an $inPars = \langle inPars_0, \ldots, inPars_{k-1} \rangle$ we use the notation:

$$\Gamma(S) = \hat{\Gamma}(S[inPars_0], A_0),\ \Gamma^2(S) = \hat{\Gamma}(\Gamma(S), A_0)[inPars_1], A_1),$$

and similarly for $\Gamma^3(S), \ldots, \Gamma^k(S)$. Thus, a plan consisting of π and $inPars$ imposes a sequence of state sets

$$\tilde{S} = \langle S_0, \Gamma(S_0), \Gamma^2(S_0), \ldots, \Gamma^k(S_0) \rangle.$$

We call $\tilde{S}$ the offline *execution path*. Note that the transition function is applied on a subset of the state set resulting from the previous transition, as induced by the sequence of input parameter assignments in the plan. In the next section, we formally describe when a plan π has the potential to solve the planning problem P, i.e., when the application of π yields an $\tilde{S}$ that satisfies the goal g.

3.5.1 Semantics of the goal

The notion of goal satisfaction is defined in terms of the execution path $\tilde{S} = \langle S_0[inPars_0], \ldots, S_k \rangle$ induced by a planning problem $P = \langle \Sigma, S_0, g \rangle$, an input parameters assignment $inPars$, and a sequence of action sets $\pi = \langle A^0, \ldots, A^{k-1} \rangle$. As described in Section 3.4.1, the goal g amounts to a conjunction of condition goals and subgoals, and eventually its elemental constituents are propositions. In the followings, we use the notation $S \supseteq props$ if there is at least one state $s \in S$ that satisfies the propositional formula $props$. We say that a plan has the potential to solve the planning problem, if it corresponds to an execution path which subsumes some sequence of states that satisfy the propositions inferred by the goal. We denote the index of the last state set in an execution path as $last(\tilde{S})$. We first introduce the notion of the minimal execution path in reference with a set of propositions, which is used for defining the formal semantics of the goal.

Definition 7 (Minimal execution path). $min(\tilde{S}, props) = \langle S_0, \ldots, S_n \rangle$ is a subsequence of $\tilde{S}$, so that

$$(S_n \supseteq props)\ \wedge (\forall i,\ 0 \leq i \leq n-1\ :\ \neg(S_i \supseteq props)).$$

Thus, $min(\tilde{S}, props)$ represents the execution path whose final state set S_n is the first one in the sequence that contains a state that satisfies $props$.

We say that an execution path $\tilde{S} = \langle S_0, \Gamma(S_0), \ldots, \Gamma^k(S_0) \rangle$ *has the potential to solve* the planning problem P given a set of initial states S_0 and a goal g, and we write $\tilde{S} \models g$ if:

$\tilde{S} \models \texttt{final}(props)$: $S_k \supseteq props\_and\_known$
where $props\_and\_known = props \ \wedge \ \bigwedge_{var_i \in props} var_i\_known = true$

$\tilde{S} \models \texttt{all\_states}(props)$:
$\forall S_j \in \tilde{S} : S_j \supseteq props\_and\_known$

$\tilde{S} \models \texttt{achieve}(props)$:
$\exists \ S_j \in \tilde{S}$ such that $S_j \supseteq props\_and\_known$

$\tilde{S} \models \texttt{achieve-maint}(props)$:
$\tilde{S} \models \texttt{achieve}(props) \wedge (\forall j, \ k \geq j \geq last(min(\tilde{S}, props\_and\_known)))$:
$S_j \supseteq props\_and\_known$

$\tilde{S} \models \texttt{find\_out}(props)$:
$\tilde{S} \models \texttt{achieve}(props \ \wedge$
$\bigwedge_{var_i \in props \text{ and appear in world-altering effects}} var_i\_changed = false)$

$\tilde{S} \models \texttt{find\_out-maint}(props)$:
$\tilde{S} \models \texttt{find\_out}(props) \ \wedge$
$\forall j, \ k \geq j \geq last(min(\tilde{S}, props\_and\_known))$: $S_j \supseteq (props \ \wedge$
$\bigwedge_{var_i \in props \text{ and appear in world-altering effects}} var_i\_changed = false)$

$\tilde{S} \models (condition\text{-}goal = sg$ `under_condition` $goal$:
$(\tilde{S} \models sg \ \wedge \ (min(\tilde{S}, props\_and\_known(sg)) \models goal)$
where $props\_and\_known(sg)$ are the propositions corresponding to sg plus the requirement that all variables involved in them are known

$\tilde{S} \models (condition\_or\_not\text{-}goal = sg$ `under_condition_or_not` $goal$:
$(\tilde{S} \models goal) \Rightarrow \tilde{S} \models (sg$ `under_condition` $goal)$

$\tilde{S} \models \bigwedge_i \ goal_i \ : \ \bigwedge_i(\tilde{S} \models goal_i)$

As in the case of the constraints entailed by the preconditions presented in Section 3.3, an extra requirement that all variables involved in *props* should be known is added. This implies that, setting aside uncertainty stemming from sensing effects, a plan has the potential to solve the planning problem if the goal is satisfied for all possible assignments to variables allowed by *prop_init*. Thus, given some *prop_init* that imply $(1 \leq v \leq 2)$, an empty action set and the goal g=`final`$(v = 1)$, g is not satisfiable. If, on the other hand, there is a sensing action with effect $sense(v)$, there is a plan that has the potential to satisfy the goal. However, this extra requirement that all variables in the goal should be known may exclude plans that would otherwise be considered acceptable. For example, given the same *prop_init* and an empty action set, the goal `final`$(v < 3)$ is also not satisfiable, despite the fact that it holds for all possible assignments to v. This strong restriction is necessary to prevent the constraint solver from presenting trivial assignments as acceptable solutions, despite the fact that we will never be sure if this assignment is indeed a solution. If, however, we have the opportunity to sense the actual state of the environment we will be able to check the validity of the solution during execution time. Thus, the term *potential to solve* refers to the uncertainty of outcomes during sensing, but not to the uncertainty due to the incomplete knowledge about the initial state.

3.6 Translating the goal into constraints

The goal is translated into a set of constraints on the CSP-level state variables, which are added to the set of constraints formulating the planning domain as described in Section 3.3. The process of transformation of the goal into constraints is presented in Algorithm 1. TRANSLATE_GOAL(g, 0, k) returns the set of constraints that model the goal given a planning problem with bound k. The conjunction of subgoals and condition goals comprising the goal is traversed, and each one is transformed accordingly to a set of constraints on the state variables. If the goal is a subgoal, the conjunction of propositions *props* included in it is extracted. The generated constraints follow from the semantics described in Section 3.5. In case of conditional goals, the goal constraints are generated recursively regarding the minimal execution path which satisfies the heading subgoal. Given a goal *sg* `under_condition` g, CONDITION_CONSTRAINTS(sg, g, 0, k) recursively calls TRANSLATE_GOAL(g, 0, j), where j is the index of the first state where the propositions corresponding to subgoal *sg* are satisfied (j is the state for which props_and_known(sg) hold, but do not hold at the previous state $j-1$).

Algorithm 1 Translate goal into constraints on CSP variables

```
function TRANSLATE_GOAL($g$, $m$, $n$)
  for $g_i \leftarrow$ get_next_in_conjunction($g$) do
    match type($g_i$)
      case subgoal:
        $constr[i] \leftarrow$ TRANSLATE_SUBGOAL($g_i$, $m$, $n$))
      case condition-goal $\vee$ condition_or_not-goal:
        $constr[i] \leftarrow$ TRANSLATE_COND_GOAL($g_i$, $m$, $n$))
  end for
  return $\bigwedge_i constr[i]$
end function
```

```
function TRANSLATE_SUBGOAL($sg$, $m$, $n$)
    $props \leftarrow$ extract_propositions($sg$)
    $props\_and\_known \leftarrow props \ \wedge \ \bigwedge_{var_i \in props} var_i\_known = true$
    match type($sg$)
        case $final$:  return $props\_and\_known[n]$
        case $all\_states$:  return $\bigwedge_{i=m}^{n} props\_and\_known[i]$
        case $achieve$:  return $\bigvee_{i=m}^{n} props\_and\_known[i]$
        case $achieve\text{-}maint$:  return $props\_and\_known[n]$
            for $i \leftarrow m, n-1$ do
                return $props\_and\_known[i] \ \Rightarrow \ \bigwedge_{j=i+1}^{n} props\_and\_known[j]$
            end for
        case $find\_out$:
            return $\bigvee_{i=m}^{n}(props\_and\_known[i] \wedge$
      $\bigwedge_{var_l \in props \text{ in world-altering effects}} var_l\_changed[i] = false)$
        case $find\_out\text{-}maint$:
            return $props\_and\_known[n] \wedge$
      $\bigwedge_{var_l \in props \text{ in world-altering effects}} var_l\_changed[n] = false \wedge$
      $\bigwedge_{i=m}^{n-1} props\_and\_known[i] \ \Rightarrow \ \bigwedge_{j=i+1}^{n} props\_and\_known[j]$
end function

function TRANSLATE_COND_GOAL($g$, $m$, $n$)
    $sg \leftarrow$ get_head($g$)
    $cg \leftarrow$ get_tail($g$)
    match type($g$)
        case condition-goal:
            return TRANSLATE_SUBGOAL($sg$, $m$, $n$) $\wedge$
            CONDITION_CONSTRAINTS($sg$, $cg$, $m$, $n$)
        case condition_or_not-goal:
            return $\neg$TRANSLATE_GOAL($cg$, $m$, $n$) $\vee$
          TRANSLATE_COND_GOAL($sg$ under_condition $cg$, $m$, $n$)
end function

function CONDITION_CONSTRAINTS($sg_1$, $g2$, $m$, $n$)
    $props_1 \leftarrow$ extract_propositions($sg_1$)
    $props\_and\_known_1 \leftarrow props_1 \ \wedge \ \bigwedge_{var_i \in props_1} var_i\_known = true$
    return $\bigwedge_{i=m+1}^{k} (props\_and\_known_1[i] \wedge$
      $\neg props\_and\_known_1[i-1]) \Rightarrow$
          TRANSLATE_GOAL($g_2$, $m$, $i$-$m$-1))
end function
```

3.6.1 Translating a goal example into constraints

In the followings, we provide the constraints generated for the encoding of *Goal 1* of Section 3.4.1:

```
/*achieve-maint subgoal*/
$bookedConcert[k] \wedge bookedConcert\_known[k]$
for $i \leftarrow 0, k-1$
    $(bookedConcert[i] \wedge bookedConcert\_known[i]) \Rightarrow$
        $\bigwedge_{j=i+1}^{k}(bookedConcert[j] \wedge bookedConcert\_known[j])$
/*under_condition constraints*/
for $i \leftarrow 1, k$
    $(bookedConcert[i] \wedge bookedConcert\_known[i] \wedge$
    $\neg(bookedConcert[i-1] \wedge bookedConcert\_known[i-1])) \Rightarrow$
    /*find_out-maint subgoal*/
    $((tempr[i-1] > 0 \wedge tempr\_known[i-1] \wedge \neg tempr\_changed[i-1])$
      $\wedge \bigwedge_{j=0}^{i-2}((tempr[j] > 0 \wedge tempr\_known[j]) \Rightarrow$
          $\bigwedge_{n=j+1}^{i-1}(tempr[n] > 0 \wedge tempr\_known[n]))$
```

3.7 Solving the CSP

The set of constraints resulting from the translation of the planning domain, the propositions referring to the initial state and the goal form the CSP which is passed to the constraint solver. The constraint solver computes a valid assignment to the CSP variables that model the planning actions, and this assignment corresponds to an optimistic plan that has the potential to solve the planning problem. We use the Choco v2.1.1 constraint programming library [*Choco library documentation*, 2012], which provides a large choice of implemented constraints, as well as a variety of predefined but also custom search methods. Moreover, it allows the dynamic addition and removal of constraints, a feature which is necessary for efficiently incorporating the new information acquired at execution time, and keeping track of environment evolution as explained in Chapter 5.

Prior to calling the solver, the planner first prunes from its search space the actions about which it knows in advance they have no potential to contribute to the goal. This preliminary process identifies all actions a_i that include at least one of the goal variables in their effects, and then recursively finds all actions which include in their effects variables that are involved in the preconditions of the actions a_i that are directly related to the goal. The search for applicable actions during solving is thus limited to this set of possible candidates, an effect that may considerably facilitate the solver's work in situations where there are many actions available, but few are relevant to the goal—as is usually the case of large domains which offer diverse functionalities. Along with this preliminary pruning, a value selection strategy that first tries to assign false values to the action variables is employed. This way, the inclusion of redundant sensing or even unwanted world-altering actions in the produced plans is usually avoided. Yet, it does not guarantee that the computed plans are optimal, i.e., that they include the least possible number of actions which fulfill the goal. It should be noted that the standard methodology for shortest-plan search, which starts with trying to find a plan of length $k = 1$, and continues in case of failure with a plan of length $k = 2$ etc., would not result in optimal plans due to the fact that parallel actions are allowed (one variable per action). Although the RuG planner does not ensure optimality, the tests on diverse domains confirm that the produced plans are "good" and in most cases optimal. Moreover, the plans usually exhibit a high degree of parallelism.

In the current implementation, the supported types for state variables are enumerations and integers. For computing a valid plan it is enough to only instantiate the variables modeling the actions, since, once these are known, the values of the relevant state variables can be inferred by propagation. Therefore, only the action variables are indicated as "decision" variables. The solving process proceeds through a combination of consistency techniques and search (branching) algorithms. Choco allows the specification of different levels of consistency to be enforced on different kinds of constraints. By default, constraints are propagated using the GAC3rm algorithm [*Choco library documentation*, 2012]. Nested constraints of high arity, i.e., which involve a high number of variables, or constraints with at least one variable with a very large domain are represented by decomposition, and are thus automatically reified [Gent et al., 2007] at the solver level through the introduction of intermediate variables. Decomposition is necessary especially for dealing with the complex constraints that model the maintainability and conditional goals, however at the cost of a decrease of the level of filtering. The way that a planning domain is modeled also affects the structure of the resulting CSP and thus the performance of the solver. For example, opting for an encoding which keeps the number of planning operators as low as possible is beneficial for propagation, since it results in less constraints.

The branching strategies are defined on the decision variables. They instruct the selection of some uninstantiated variable, using some variable ordering heuristic, and the assignment of some value from its domain to it, by employing some value ordering heuristic. Several branching strategies have been investigated and tested, and depending on the domain instance the combination which provides the optimal results varies. Usually, search strategies that yield good-quality plans have worse performance than strategies which lead to plans that include redundant actions, e.g., by applying a random value assignment. In the rest of the thesis, a "most constrained" variable selection heuristic and an "increasing domain" value iteration strategy is employed in the testing process, unless stated otherwise. Most constrained implies selecting the variable involved in the largest number of constraints. Variables modeling virtual KB actions are selected before all others. Then an iteration over values in increasing order takes place.

Regarding the choice of k, this is selected depending on the planning domain. It could be restricted by the number of grounded action instances, however since this can be very high (given the potentially large domains of the input parameters), k is set by the domain designer, based on his anticipation of a maximum size of expected plans and his knowledge of the domain. For example, given a domain, where a robot has to move between some locations, k could be set to 3 times the number of locations. Note that due to the high degree of parallelism that characterizes the produced plans, many solutions which require considerably more than k actions will be found.

Exploiting properties characteristic to the planning problem, such as causal dependencies, that are used in modern heuristics becomes difficult because the resulting CSP encoding is quite detached from the structure of the original planning problem, and the encoding is not propositional. As already mentioned, the enhancements presented in [Barták and Toropila, 2008, 2009] are not applicable in a planning domain as expressive as the one supported by the RuG planner. The model reformulations from logical to combinatorial encodings are dependent on a classical STRIPS representation, and cannot be applied because of the use of numeric-valued variables and input parameters, arithmetic preconditions, and effects beyond mere assignments. Moreover, the main proposals for improved search strategies rely on a formulation that only allows sequential plans. However, including parallel actions when possible pays off at execution time, since many actions and especially sensing ones, take a long time to respond. Due to these reasons, the performance of the RuG planner has less inferential power in complex combinatorial propositional reasoning compared with state-of-the-art planners. However, it provides for an expressive domain representation, with extensive support for numbers, can efficiently deal with uncertainty about numeric information, and supports a rich language for expressing extended goals. These characteristics are indispensable to model and handle pragmatic scenarios in WS application domains.

3.7.1 A planning example

Let us now consider a planning problem which models the scenario described at an abstract level in Section 1.3.1. The planning domain consists 30 different planning actions, 23 of which have sensing effects. The planning operators simulate the functionality of services that reside in the Web, derived from different business areas related to making online appointments, shopping, shipping, traveling, learning about entertainment events, maps, calendar, and weather services. The actions can be mapped to the APIs of real services, as presented in [Kaldeli et al., 2011]. For example the Yahoo! weather service can be used as a source of information regarding temperature or weather conditions, Google maps for geographical information such as distances, Google calendar for checking the marked occupations for some given date, and the `eventful.com` for collecting useful data about a number of cultural events, e.g., browse through the list of concerts for a given band. The responses of the actual services are in these cases XML documents, which are parsed to extract the respective information, and intermediate transformations may be needed to translate the derived data to some format that can be used at the planning level, and vice versa. For example, because the Yahoo! weather-related services require WOEIDs (Where on Earth IDentifier) as the form of their location-related input parameters, locations have to be mapped to this format, while dates are also transformed between different formats, depending on the specification of each service. The process of translating the combination of planning actions and input parameters into concrete invocations and appropriately parsing the responses is taken care by some separate executor, as e.g., implemented in [Westra, 2010].

Such a domain that encompasses a broad diversity of services can serve a large variety of different user needs, from arranging some entertainment activity to purchasing some item or making an appointment with a doctor. For example, let us consider a user who lives in Groningen, NL, and wants to book a ticket and a hotel room for the nearest upcoming concert of the band "Neutral Milk Hotel" whose date and location meet some criteria referring to the weather conditions, the distance from Groningen, his availability on the performance day according to his agenda, as well as about the price he is willing to pay for his overnight stay. This wish is captured by the following nested goal:

Entertainment goal

`achieve-maint`$((bookedHotel = TRUE \wedge hotelPrice < 80)$
    `withParams` $(hPlacePar = eventPlace \wedge hDatePar = eventDate \wedge$
      $numbOfNightsPar = 1 \wedge roomTypePar = \text{“}single\text{”}))$
`under_condition`

$\texttt{achieve-maint}((\mathit{bookedTicket} = \mathit{TRUE})\ \texttt{withParams}$
$\quad(\mathit{bandNamePar} = \text{“}\mathit{NeutralMilkHotel}\text{”} \wedge$
$\qquad \mathit{concDatePar} = \mathit{eventDate}))$
$\texttt{under\_condition}$
$\ \texttt{find\_out}((\mathit{temperature} > 0)\ \texttt{withParams}$
$\quad(\mathit{wPlacePar} = \mathit{eventPlace} \wedge \mathit{wDatePar} = \mathit{eventDate})$
$\quad \wedge\ \mathit{busy} = \mathit{FALSE}\ \texttt{withParams}\ (\mathit{cDatePar} = \mathit{eventDate})$
$\quad \wedge\ (\mathit{distance} < 200)\ \texttt{withParams}$
$\qquad(\mathit{mapsOriginPar} = \text{“}\mathit{Groningen, NL}\text{”} \wedge$
$\qquad \mathit{mapsDestinPar} = \mathit{eventPlace}))$

The variables *eventPlace* and *eventDate* on which the performance will take place are unknown offline, and it is up to some knowledge gathering service (namely the `eventful.com` service) to provide them. In the initial optimistic plan these are assigned some convenient value by the solver, however the respective knowledge-level variables indicate that this value is not a valid one. An assignment to some variable $\mathit{var} = \mathit{value}$ for which $\mathit{var\_known} = \mathit{false}$ is signaled in the optimistic plan by a “defaultVar” mark.

By employing the conservative combination of most constrained and increasing domain selection strategies, the following plan is generated offline for the entertainment goal:

{*getEventsList*(Neutral Milk Hotel)},
{*getNextEvent*},
{*checkCalendarAvail*(defaultDate),
getDistance(Groningen, defaultPlace),
getTemperature(defaultPlace, defaultDate),
getAvailHotels(defaultDate, defaultPlace, 1, 1)},
{*bookConcertTicket*(Neutral Milk Hotel, defaultDate)},
{getNextHotelInfo},
{*bookHotel*(defaultHotel, defaultDate, 1, 1)}

For readability reasons we put the assignment to the input parameters together with the actions. The values “defaultPlace”, “defaultDate” and “defaultHotel” all correspond to the same values, i.e., to the yet unknown *eventPlace*, *eventDate* and *hotelId* sensed by *getNextEvent* and *getNextHotelInfo*. *getEventsList* computes the list of performances for a given band ordered by date. The service for dealing with hotels provides aggregated searching and booking facilities over a wide range of hotel providers (such as services as `booking.com` do), and orders the results according to some criterion (e.g., price). *getNextHotelInfo* returns the information (price, hotel id) of the next hotel in the formed list. The time for transforming the planning domain and goal into a CSP takes 3.4 sec on an Intel Core i5 Core i5 @2.50Ghz computer with 4GB of RAM. The time for computing the solution-goal is 3.7 sec. More results regarding evaluation of different scenarios are provided in Chapters 4, 5 and 6.

Depending on the information returned at runtime, there are many different possible ways for the plan execution to evolve. For example, it may turn out that the place of the first upcoming concert is too far, or that there is no hotel available on that date within his budget, etc. In such cases, the original plan has to be interrupted and revised, so that the conditions regarding the whereabouts and date of the next concert are looked up. To further complicate things, at any moment a service may fail. So, if e.g., the booking service of the first selected hotel that meets the users criteria happens to be in a permanent failure state, an alternative hotel has to be searched, and depending on the result, the goal may finally be satisfied or not. In Chapter 5, we pick up again this example, and discuss the behavior of an orchestrating algorithm that takes care of contingencies that come up after the exposure of the offline plan to the environment.

3.8 Goal editor

A graphical goal editor, which is designed to assist the user in specifying a goal given planning domain, has been implemented as part of a web-based user interface for smart homes. Detailed information about the implementation of the UI and its features can be found in [Yumatov, 2011]. The UI is platform-independent and can operate on a broad range of devices ranging from mobile phones to portable computers and traditional PCs. The goal editor is one of the features offered by the UI, along with a user-friendly explorer of the available services, a graphical overview that keeps track of the current states of the devices-services, and support for directly commanding services if possible.

The goal editor presents the user with the constructs of the language and the planning-level variables, and guides him in specifying a goal, by suggesting the allowed combinations of constructs and expressions through pop-up lists. Meta-information, such as the location of devices in the case that actions correspond to operations offered by devices, is used to group state variables offered by the services in the pop-up lists. Services and variables are annotated with icons, if these are available, so that the user can conveniently detect the desired properties. The domains of the variables are taken into account to restrict the options for variables and values selection depending on the kind of the chosen propositional formula. Thus, variables incompatible with some expression operator as well as unacceptable values are filtered out. Figure 3.2 depicts a screenshot of an expression editing window. The *Val* button opens up a value picker menu, while *Srv* opens a dialog with all devices which include variables with a domain compatible with the light variable.

Once the goal is specified through the graphical contacts, it is stored into an XML format, that respects an XML schema which models the goal syntax. The planning domain, parsed by the UI to exclude all necessary information and produce the graphical overview, is also represented in XML format. Figure 3.3 depicts how the goal

$\texttt{achieve-maint}(main\_livRoom\_lamp : light = ON)$
    $\texttt{under\_condition\_or\_not}$
        $\texttt{find\_out}(natural\_light = DARK) \wedge$
$\texttt{achieve-maint}(chair\_livRoom\_lamp : light = OFF \wedge$
    $tv : tv\_state = ON \ \wedge \ tv : tv\_channel = 2 \wedge$
    $chair : chair\_state = MIDDLE)$

is specified and stored in the goal editor. Note that `under_condition_or_not` is indicated as "optional condition", while a variable identifier includes the device name (before the separator ":"). The variable values are represented visually through appropriate icons, e.g., the state "ON" for a lamp is presented by an image which depicts a lamp that is on.

Support for managing stored goals, such as editing, organizing them into groups according to their functionality or other properties or bookmarking them, is pro-

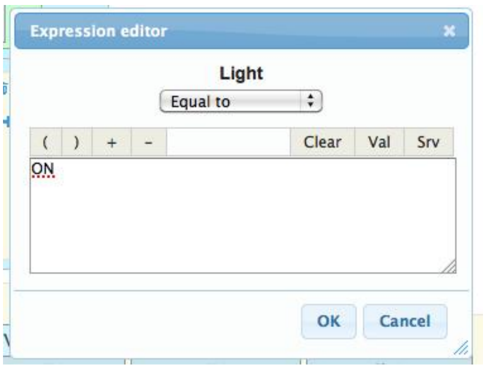


Figure 3.2: Goal editor: an example of an expression editing window

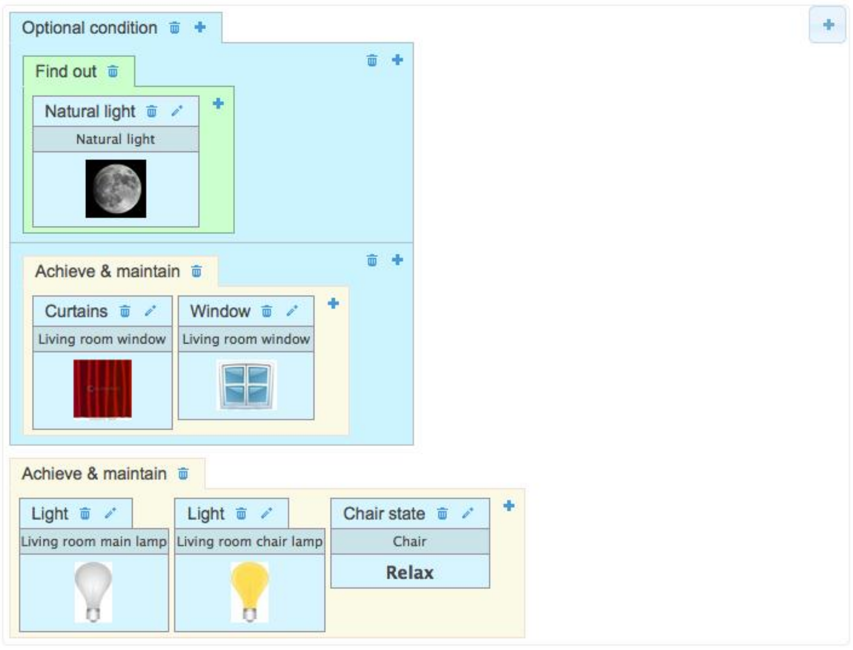
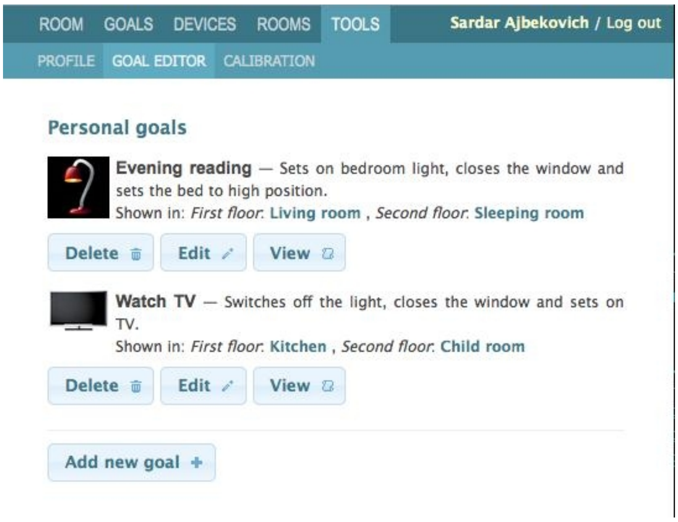


Figure 3.3: Goal editor: an example of an conditional goal representation



Personal goals

Evening reading — Sets on bedroom light, closes the window and sets the bed to high position.
Shown in: *First floor.* **Living room** , *Second floor.* **Sleeping room**

Delete Edit View

Watch TV — Switches off the light, closes the window and sets on TV.
Shown in: *First floor.* **Kitchen** , *Second floor.* **Child room**

Delete Edit View

Add new goal

Figure 3.4: Goal editor: an example of an conditional goal representation

vided to the user. The user can only edit or delete these goals that are allowed by his user permissions, and the viewing of available goals can also be filtered according to the user profile, including e.g., his preferences or possible disabilities. A screenshot of some goals viewable by a user is shown in Figure 3.4.

In the current implementation, the UI communicates with the planner and the repository which keeps the domain representation (see also Section 4 for the UI as part of a domotics architecture) through the REST (Representational State Transfer) interface. Once a request for satisfying a goal is issued, the XML representation of the goal is passed as input to the planner, which parses it, and translates it into constraints which are dynamically added to the set of constraints that model the planning domain. Then, after taking into account the current state of the world, the resulting CSP is passed to the constraint solver, which computes a valid assignment to the action variables, i.e., a plan that has the potential to satisfy the goal.

Chapter 4

Planning in a Smart Home

The use of planning techniques for service composition has so far mainly focused either on the public Web, where services are distributed on the internet, being registered, for example, on a UDDI (Universal Description, Discovery and Integration) registry, or on corporate IT scenarios, where services are kept to some infrastructure accessible only to a limited number of stakeholders, e.g., a private cloud. However, the evolution of web technologies has also highly affected other open and heterogeneous networks of autonomous entities. A prominent example is the *Web of Things*, which is concerned with the interoperation of everyday embedded devices. The Web of Things paradigm is widely used in the field of *domotics*, which aims at the realization of highly automated intelligent home environments, so as to enhance the feeling of comfort and safety of the its inhabitants. The similarities between the two sorts of Web include the loose coupling of the computational hosts, the high heterogeneity in terms of hardware and software running on the connected nodes, the importance of communication protocols, and the need to effectively coordinate and integrate the different components, in order to deliver to the end-user a transparent and satisfactory access to the system.

Given these requirements, infrastructures that base on the concept of service constitute a natural proposal to the realization of next-generation homes [Aiello and Dustdar, 2008]. Indeed, there are a number of service-oriented platforms for pervasive applications, such as the Java-based Jini infrastructure [Apache, n.d.], the Universal Plug and Play multimedia standards [UPnP Forum, 2008], and the Open Services Gateway initiative (OSGi) [OSGi Alliance, 2009]. However, these platforms focus mainly on aspects related to basic device interoperation and spontaneous networking, without providing for dynamic coordination and more complex and intelligent functionalities that can be built at a higher application level. One should take full advantage of the capabilities offered by a well-designed Service-Oriented Architecture for the home, and, by automatically composing the available autonomous device operations, enable the delivery of added-value services which will be perceived as *smart* by the user.

4.1 Smartness via service composition at home

To satisfy the wishes of the user and guarantee his comfort and safety, the house has to be able to exhibit quite complex functionalities rather than just triggering some single service or a pre-designed sequence of fixed services. A trivial operation such as turning on a light in a corridor can be achieved with a switch or a passive infrared sensor. However, the coordination of the home so as to effectively deal with a gas leak detection is far more demanding, especially when considering the many possible contextual states the house and the user can be in, and each of which may require several possible handlings to achieve the same ultimate safety goal. Moreover, developing rigid solutions that are tailored to a specific home instance and user needs is not an efficient approach, given the considerable effort that is required to adapt them for new customers.

Designing and predicting all possible service compositions is thus not a viable solution given 1) the large variety of different user requirements and home instances, and 2) the lifecycle of a specific home: devices evolve over time, with new functionalities constantly appearing or disappearing, the state of the devices constantly changes, users move around, and thus the number of possible contextual states can be very high. Therefore, approaches to service coordination in such a dynamic setting have to be easily customizable to different home instances and user needs, be able to support home evolution, and perform complex reasoning about contextual information rather than relying on hardwired sets of service instances. These remarks are in line with the challenges that should be addressed by service composition mechanisms in pervasive environments as identified by Bronsted et al. in their survey about service composition issues in pervasive computing [Bronsted et al., 2010]:

- *Context awareness:* "A composition is context aware if it is sensitive to context changes, its constituent services are sensitive to context changes, or the set of composed services varies with context changes."

- *Managing contingencies:* "In a pervasive computing environment, devices—and thus services—often have unpredictable availability [...] A service in a given composition might become unavailable and need replacement, so the logic for discovering and inserting a replacement service shouldn't reside in the constituent services."

- *Leveraging heterogeneous devices:* "Pervasive computing systems use a variety of devices, from Internet servers to networked sensors and actuators."

- *Empowering users:* “Pervasive computing applications require new interaction models because document-centric, desktop-based computer interfaces are often unavailable or impractical.”

In this chapter we show how these challenges can be addressed through an architecture which bases on the notion of service abstraction, and has at its core the RuG planner presented in Section 3. In such a context, the objective to be achieved is described in the form of a declarative goal, while the services published by the available devices are selected and combined during runtime. This way, different compositions can be be computed for the same goal depending on the current state of the devices, thus meeting the first challenge about context awareness. Regarding the second requirement, the system supports dynamic service availability, and devices can enter or leave the network. Since the composition is computed at runtime rather than at design time, the reasoning is performed on the most up-to-date set of services, which may change over time. The third challenge is realized by an open and dynamic pervasive layer which supports a number of different communication protocols, and offers a standard interface for controlling all devices to the upper layers, thus ensuring interoperability. Finally, the infrastructure is user-centric and can be easily adapted to match new user needs through the specification of goals which can be inserted either by the designer or the home inhabitants themselves, as well as the support for different user interfaces, such as a touch screen, voice-based or Brain Computer Interfaces (BCI).

We think that Smart Homes constitute an environment that is particularly expedient for the application of AI planning techniques. In fact, the applicability of elaborate automated discovery and composition of services available online in the Web is limited by the lack of machine-interpretable and standardized semantic mark-ups, as well as the very limited meaningful correlation and compatibility among operations of different services, with the vast majority of public services being mere data sources, as concluded by the findings presented in [Fan and Kambhampati, 2005]. The environment of a Smart Home, on the other hand, is more structured, well-defined and controllable, thus making the added value gained by non-trivial automated composition and monitoring of services a feasible and realistic task. In this case, one can rely on consistent descriptions of service operations, with proper syntactic and semantic mark-ups provided by the home domain designer, to perform powerful reasoning for complex tasks which considerably advance the level of home intelligence.

As far as we are aware, this is the first attempt to fully integrate a domain-independent planning component in an event-driven service-oriented prototype for pervasive applications in a domotic environment. In such a setting, the planning module has to continuously interact through asynchronous message passing with the other components, such as the context awareness and the service repository, so that it seamlessly adjusts the planning domain instance to reflect environmental changes, and reacts accordingly at runtime. Although the implementation presented herein relies on the OSGi-UPnP platform for exposing devices as Web Services, the architectural components are loosely coupled with each other and independent of the particularities of the specific architecture (e.g. SOAP, OSGi-UPnP). Thus, they can be easily adapted so that they inter-work in a different setting (e.g., see [Caruso et al., 2012] for an implementation using the Representational State Transfer (REST) for inter-communication, and custom proxies for a variety of real hardware devices with different protocols). The platform has been fully implemented, evaluated, and tested with real users on a simulated home environment.

The adopted planning domain modeling relies solely on individual descriptions of de-coupled device operations, described and implemented using existing protocol standards, such as UPnP-OSGi. The multi-valued variable-based encoding of the planning domain maintains a close relation to the actual way that device operations are realized, e.g., adhering to a direct mapping between UPnP- and planning-level variables (see Section 9). These characteristics contribute towards reducing the manual effort, and making more intuitive the task of converting the pervasive-level domain and context into their planning-level equivalent, as well as the users' goal specification. Contextual changes are propagated asynchronously to the planner and incorporated into the planning instance representation in form of constraints, while the generated plans are characterized by a high degree of parallelism, which can be exploited at execution time for achieving better overall response times. Straightforward replanning is applied for simple failure recovery.

The architecture we present in this chapter has been designed, implemented and evaluated in the context of the European project Smart Homes for All (SM4ALL) [SM4All, 2008-2011]. The service-orientation principles of the architecture are not limited to the lower levels of the pervasive layer, but also at the application layer. By building on established standards such as OSGi and UPnP, the platform abstracts to higher layers that support complex reasoning on the set of available services and their state, as well as the interaction with state-of-the-art user interfaces such as BCIs. Most importantly, the architecture supports the performance of runtime service composition. A fully-working prototype has been implemented and evaluated with respect to the efficiency of the composition layer, but also the acceptability and effectiveness from the side of end-users from diverse backgrounds, namely a group of elderly and disabled people, and a group of younger technologically experienced users. The manuscript extends the

The remainder of the chapter is organized as follows. Section 4.2 describes some scenarios that seek to demonstrate what kind of problems and situations a ‘smart’ home equipped with the SM4ALL architecture can deal with. The main aspects of the SM4ALL project and the software architecture we propose are introduced in Section 4.3. Section 4.4 describes how service composition through planning can achieve a smart home behavior. The prototype we built to test the validity of the approach is presented in Section 4.5, while the engineering process to deliver a smart home is illustrated in Section 4.6. The technical evaluation of the system and the user evaluation of it are presented in Sections 4.7 and 4.8, respectively.

4.2 A day in the Smart Home

Let us now describe how a Smart Home equipped with the SM4ALL architecture behaves over a possible course of events that happen throughout a rather adventurous day in the house, including both conventional user requests and reactions to emergency situations. The following scenarios play the role of demonstrative examples throughout the paper, and have been tested in a simulated environment, as described in Section 4.7. We consider that the home inhabitant is a disabled person who can move around on an electric wheel-chair, while a nurse pays a visit for some hours every day. A location component keeps track of the location of the users to the level of some predefined areas.

At 8 pm the waking-up goal prescribed by the user is automatically triggered: the alarm clock rings, the curtains in the bedroom are opened, the lights may be turned on depending on the amount of daylight detected by a natural light sensor, and the motorized bed is elevated. After taking a shower, the user wants to move to the sitting room and watch some TV. Such a goal dictates that the TV is set to the user’s channel of preference, the lights are adjusted depending on the indication of the natural light sensor, and the curtains are also shut accordingly. The air-conditioner is turned on if the temperature sensor in the living room indicates that the temperature is too high, while the necessary doors are opened to facilitate the user moving to the sitting room.

At noon, the user goes to the kitchen to prepare something to eat. While being there, the smoke detector in the kitchen identifies a potentially dangerous smoke leak—but fortunately not due to fire. As a result, a predefined home goal for dealing with this situation is automatically triggered: after having ensured that the user has safely moved out of the kitchen (let’s say to the adjacent sitting room), the door leading to the kitchen is closed to isolate the smoke in a single room. The ventilator is turned on and the kitchen window is also opened, so that the foul air is expelled, while an alarm notification appears on the TV screen. While waiting in the sitting room, the user wants to move back to the kitchen, but only after having assured that the environment there is safe, and the smoke has been eliminated. This wish implies resorting to sensing to identify the current situation in the kitchen. Let's assume that after some time the smoke is eliminated, causing the alarm on the TV and the ventilator to automatically turn off.

After verifying that no serious damage has been caused, the user moves to the sofa in the sitting room and wishes to have a cold beer in his hands. Assuming that the house is equipped with a housekeeping robot (similar to the cooking assistant described in [Gravot et al., 2006]) able of performing basic recognition and manipulation tasks, such as moving around, getting and putting items at particular places, sensing their temperature etc., then the request of the user can be fulfilled by the robot. Let's say that there are no beers in the fridge, however the system finds out that there are some beers left on the storage shelf —the assumption is that items in the house are labeled by RFID tags, and a smart fridge and smart shelves keep track of them. Having this information in hand, the robot will move to the storage room and get a beer from there. In order to satisfy the requirement that the beer should be cold, it will proceed in placing the beer it has taken in the fridge, and leave it there for two minutes to cool. Then it will take it out again and bring it to the sofa.

Later in the afternoon, while the user is taking a bath, and the nurse has gone out for some shopping, a fall is identified by the fall detector attached to him [Aiello and Dustdar, 2008], and an emergency goal is automatically triggered: the health center is notified and an informative message is sent to the nurse's PDA or mobile phone, while the robot is moved to the bathroom in case the user wants to ask for some additional assistance. Given that the fall has not caused any serious trouble, the night finds the user lying in his bed reading a book, and after some time he decides that it's time for going to sleep. He thus issues a goal that prepares the bedroom conditions for sleeping, which involves setting the alarm clock to some preferred wake-up time, lowering the motorized bed position, turning off the lights and closing the curtains.

It should be emphasized that user goals as well as the description of the device functionalities are kept as de-coupled as possible from the particular setting of a home instance, and the set of desired service invocations is reasoned at runtime, depending on the capabilities of the particular house and its current context. Thus, the functionality for sending a message to the nurse for example is specified in a generic way, so that it may be taken care by different atomic device instances or a combination of them. This depends on which particular devices that can offer the semantically prescribed unified messaging possibility are available in the specific pervasive system, e.g., a smart phone, PDA, mobile etc. Moreover, depending on what is inferred about the current state of the house, the same goal may lead to quite different compositions of activities. For example, regarding the goal about getting a cold beer, if there exists a beer already in the fridge, the composition will instruct the robot to directly get it from there, or, in the case of the fall detection goal, if the nurse happens to be at home, all that has to be done is to turn on a local alarm to notify him, so that he can take care.

4.2.1 Replanning for basic failure recovery

The scenarios mentioned above assume that no contingencies occur during execution, and that all service invocations complete successfully. What if, however, a service is out of order, and responds with a failure or if a timeout occurs? In such cases, the system will first try to re-invoke the erroneous service, and if again a failure or timeout is observed, it will perform replanning. This means that the composition engine will attempt to achieve the goal by computing an alternative plan, which does not include the faulty service.

Considering the scenario with the beer described above, let us assume that the door that leads to the kitchen is blocked, e.g., because in the meantime someone has put an obstacle which hinders its opening, and therefore the respective automatic door opening invocation reports an error (or times out). As a result, the composition engine looks whether there is an alternative route to the kitchen, which does not go through the problematic door. It should be noted that the new plan will take into account the contextual situation that has resulted after executing all actions that preceded the attempt for opening the kitchen door, which means that the robot may need to go back in order to follow the right route. If no alternative plan can be found, then the system responds that the goal is not satisfiable under the given contextual circumstances. Similarly, let us assume that, when executing the plan that prepares the living room for watching TV, the automatic turning on operation of the TV service responds with a failure. Assuming that the robot assistant has also the ability of turning on the TV by manually pressing the button on the device, the composition engine will compute an alternative plan which involves moving the robot in front of the TV so that it can switch it on.

4.3 Architecture

The key idea underpinning the SM4ALL architecture is that the software infrastructure is entirely based on the abstraction of a service providing for an open, dynamic and flexible sensing and control infrastructure. Figure 4.1 provides a schematization of the systems' main components and their basic interactions. One can distinguish three macro layers: the pervasive layer, where the home devices live; the composition layer, which is responsible for collecting information about the environment, interpreting it and coordinating the available services; and the user layer which provides the interface to issuing commands to the home.

4.3.1 The pervasive layer

The role of the pervasive layer is to discover and interconnect networked devices, and provide a common mechanism for accessing the services they offer for the rest of the middleware layers and applications. It bases on standard device-level service-oriented technologies, such as the UPnP and OSGi. This way, all types of devices are described in a standardized programmatic manner, and are controlled in accordance with this description. The pervasive platform enables heterogeneous components to be integrated independently of their interconnectivity protocol through the use of an appropriate proxy for each communication technology. The platform is extensible, so that new device instances can be integrated to it in an efficient and dynamic way, without requiring a reboot of the system. Discovery of new devices is performed

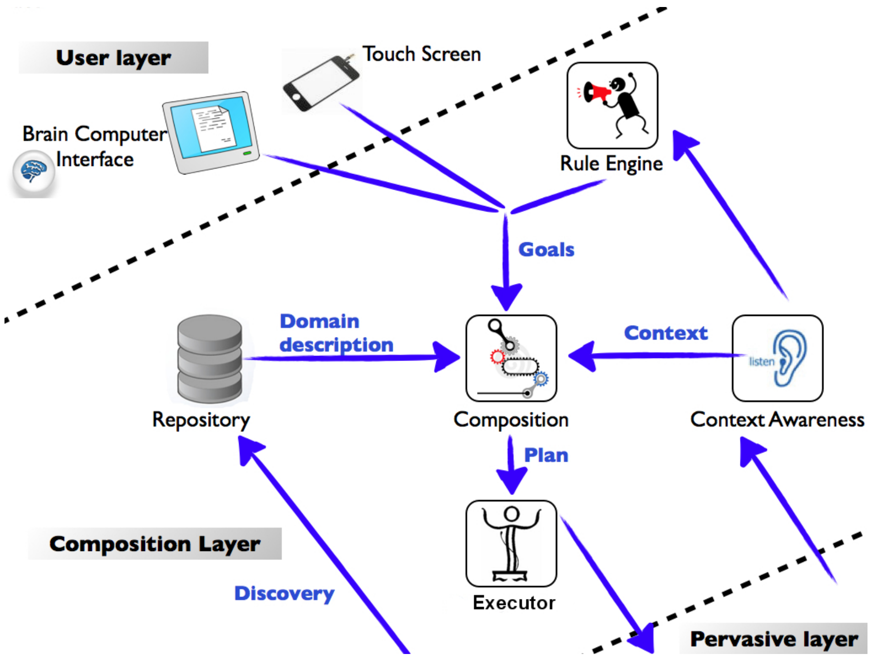


Figure 4.1: Architectural overview.

automatically, and depending on the type of the detected device, the pervasive platform checks whether it can find the respective control software in the drivers repositories it has access to. If a driver prescribing the functionalities provided by the discovered device can be found, then no manual configuration is needed, otherwise an appropriate description of the new device type has to be added into the system.

The layer is based on an asynchronous publish and subscribe architecture so that interested parties are notified about the appearance and disappearance of services, as well as about state changes. Several clients can connect to the pervasive layer, such as a BPEL engine, a context-awareness component, a visualization software tool (see Section 4.5.3) or a user interface [Aloise et al., 2011]. These clients subscribe to event types they are interested in, i.e., concerning the change of some variable of interest. More details about the technologies and standards adopted for satisfying the requirements for the pervasive layer are provided in Section 4.5.

4.3.2 The service composition layer

Central to the SM4ALL architecture is the composition layer, which is further abstracted into five components. The *repository* keeps the descriptions of the set of supported service types, including appropriate semantic markups about the operations offered, as well as the registry with the actual device instances that are active at any given moment. This is kept up-to-date according to the notifications received from the pervasive layer. A map representing the layout of the house (e.g., the rooms that comprise it, and how they are arranged) is also stored in the repository. Whenever a new device registers itself to the pervasive layer, it also publishes itself to the repository as an instance of its associated abstract type, specifying its functionalities in terms of action preconditions and effects. The *context awareness* component seamlessly monitors the status of the devices and the users' location, collects and aggregates information, and via a publish-subscribe mechanism notifies the interested parties. The *rule engine* uses information about context changes and, if certain conditions hold, takes action (e.g., a fire is detected, and an emergency plan should be put into practice) by directly invoking the composition module.

The *composition* module receives high-level complex goals issued either by the user layer (e.g., a request for a beer) or the rule engine (e.g., an emergency goal for combating some dangerous gas that has been detected), and tries to fulfill them by generating appropriate compositions of the available services. The compositions are computed automatically and on the fly by the RuG planner. Whenever a goal is issued, the planner generates a plan, whose execution changes the state of the environment in accordance with the properties prescribed by the goal. The plan is then passed to the *executor*, which translates the composition into lower-level service invocations and executes them step-by-step, in a synchronous manner. In case that a service operation returns a permanent failure, the plan execution is terminated, the erroneous service is removed from the registry of currently active devices, and the composition module is asked to compute a new alternative plan for the same goal.

4.3.3 The user layer

The user layer provides the means for the final users to interact with the middleware and instruct the home. The basic module of the user layer is the Abstract Adaptive Interface (AAI) [Catarci et al., 2011], which acts as a proxy that provides services to the particular user interface. Through a unique adaptable algorithm, the AAI is able to manage many different user interface models, such as a touch screen or a brain computer interface, by changing its behavior on the basis of the concrete UI characteristics.

The AAI collects information about the available service operations of active devices and the goals kept in the repository, and forwards them to the concrete UIs. The information collected from the repository includes visual data (icons) associated with the service operations offered by the devices, as well as information about their location, so that they can be organized accordingly, depending on the capabilities of the concrete UIs. Moreover, a set of icons representing complex goals, such as preparing the bedroom for sleeping, are also made available. The AAI is seamlessly updated to reflect the most recent status of the devices, as delivered by the context awareness component, and notifies the concrete UIs connected to the system accordingly. Whenever an icon is selected, the respective instruction is sent either directly to the executor, if it represents a single operation, or to the composition module, if it corresponds to a complex goal.

4.4 The home as a planning domain

4.4.1 The OSGi UPnP-level home domain

All devices that participate in the home domain must have an interface description in accordance with the OSGi UPnP Device Service specification, so that they can be automatically discovered by the base driver, and added to the OSGi registry. Each device exposes its functionalities as one or more UPnP services, which provide a collection of method calls that constitute the *UPnP actions*, and is associated with a set of public variables, called *state variables*. State variables are typed, and can be posted as events, which means that a notification will be generated whenever their value changes. A UPnP action can have multiple input and output arguments, which according to the OSGi UPnP specification are also represented as state variables. An action may have access to state variables that are associated to other services, and may perform computations on them or actively change them, e.g., a robot may be able to manually control external devices. UPnP actions can be distinguished into sensors, which just sense the value of a state variable, and actuators, which change the value of one or more state variables. The home domain can thus ultimately be conceived as a set of UPnP actions, which belong to several UPnP services, that in turn are provided by UPnP devices, and can be defined at this low level of the UPnP hierarchy as follows.

Definition 8 (Home Domain)**.** A Home Domain (at the UPnP level) is a tuple $\mathcal{HD} = \langle \mathit{UVar}, \mathit{UPar}, \mathit{USetAct}, \mathit{UGetAct} \rangle$ where:

- *UVar* is a set of variables that reflect some attribute of a service. Each $v \in \mathit{UVar}$ ranges over a finite domain D^v.
- *UPar* is a set of variables that play the role of input arguments to actions. Each $p \in \mathit{UPar}$ ranges over a finite domain D^p.
- *USetAct* are UPnP actions that change the value of one or more variables and *UGetAct* are purely sensing actions that return the value of a variable. We assume that there is a sensing action for every variable of interest $v \in \mathit{UVar}$. These two sets form together the set of all available UPnP actions $\mathit{UAct} = \mathit{USetAct} \cup \mathit{UGetAct}$. Each $ua \in \mathit{UAct}$ has an identifier $id(ua)$ and optionally a set of input arguments $in(ua) \in \mathit{UPar}$. The identifier of the action has the form $id(ua) =$*DeviceId:ServiceId:ActionId*, where *DeviceId* and *ServiceId* are the identifiers of the device and the respective service the action belongs to. Each device is assigned a unique identifier.

The OSGi UPnP actions describe in a syntactic way the operations that can be performed on the state variables. For example, the OSGi UPnP action "Close-Curtains" sets the value of the Boolean state variable "Curtains" to false. Usually, at this level, the description of the way actions perform is rather primitive, and does not include any checks about conditions that must hold for the action to be invoked in a safe and correct way. For example, given a window that opens inwards, if the "CloseCurtains" action is invoked while the window is open, its casements will interfere with the curtains. Similarly, the action for setting the TV channel will fail its goal if the TV is off, or the action that is responsible for moving the robot may lead to an unfortunate situation if it is performed towards a closed door. This higher degree of reasoning, which is essential for coordinating more complex tasks, is captured by the planning-level semantics.

4.4.2 The planning-level home domain

In order to automate the task of composition, the OSGi UPnP services have to be enriched with additional semantic annotations, which are necessary for the formalization of the available activities, as well as the description of the goal that has to be fulfilled upon a user request or a triggering event. To this end, the service operations must be annotated by the domain designer with appropriate semantic mark-ups, in terms of preconditions and effects as described in Definition 1. This set of semantically annotated activities constitute the actions that form the planning domain, which is formally defined as follows:

Definition 9 (Planning Home Domain)**.** Given a UPnP-level Home Domain $\mathcal{HD} = \langle UVar, UPar, UGetAct, USetAct\rangle$, a Home Planning Domain is a Planning Domain $\mathcal{PHD} = \langle Var,\ Par,\ Act\rangle$ (see Definition 1), with:

- $Var = UVar$
- $Par = SPar$
- For each $ua \in USetAct$, there is an action $a \in Act$, with $id(a) = id(ua)$ and $in(a) = in(ua)$

The preconditions and effects of the planning actions are defined on top of the UPnP-level syntactic descriptions.

It should be noted that the set of sensing actions *UGetAct* are not represented as planning actions, since their values are updated upon the receipt of the events that are continuously generated by device state changes or by the available sensors as shown in 4.4.3.The Planning Home Domain is transformed into a CSP following the methodology described in Section 3.3. On top of the constraints corresponding to the Planning Home Domain, constraints stating some conditions that should hold at every state, or should never be violated at any state, may also be added to the CSP by the domain designer. These constraints for example may reflect the layout of the house, stating e.g., $adjacent\_rooms(KITCHEN, LIVING\_ROOM)$.

4.4.3 Incorporating context changes

The current value of a state variable may become known either asynchronously via a change event that originates from the invocation of some actuator kind of UPnP action (*USetAct*), or synchronously from the call of some UPnP action of sensing type (*UGetAct*). A sensing action is usually called internally by the respective sensor device, either periodically or when a specific condition in the environment is detected, depending on the type of sensor. It may be also called by any external client that can control the sensor device. A state variable event change or an output argument conveys a tuple *(v, value)*, where $v \in UVar$ and $value \in D^v$. The new knowledge contained in the tuple is incorporated as a constraint into the CSP as follows:

- When bootstrapping, all sensing actions that are accessible by the orchestrator component are called. Thus, for each variable $v \in Var$ which can be sensed, the retrieved pairs *(v, inValue)* are kept in a structure *mapVarVal*.
- For each v that is included in the bootstrapping phase, a constraint $v[0] = inValue$ is added to the *CSP*, reflecting the current knowledge at the initial planning state.
- Whenever the context awareness component receives a change event, or the orchestrator calls a sensing action, the respective tuple *(v, value)* is processed: if v is included in the *mapVarVal* structure, and has a current value *inValue* and $inValue \neq value$, then the constraint $v[0] = inValue$ is removed from the *CSP*, and the constraint $v[0] = value$ is added.

Besides changes in the values of variables, a contextual change may reflect the
Besides changes in the values of variables, a contextual change may reflect the realization that a service has become unavailable, if the response after a synchronous call of the UPnP action *ua* by the orchestrator indicates a permanent failure. In such a case, the semantic repository is notified that the respective operation is not functioning properly anymore, and removes it from the registry of available services. The repository thus publishes an event which indicates that the action *ua* has become unavailable, and in turn, the following constraints are added to the *CSP*: for all $0 \leq i < k$, $a[i] = false$, where $a[i]$ is the CSP-level boolean variable modeling the planning action that corresponds to the UPnP action *ua*, with $id(a) = id(ua)$. This way, subsequent plans are not allowed to include action *a* in any step. If the services become available again, then the above constraints are dynamically removed from the CSP, upon the appropriate notification received from the repository. We remark that the connection and disconnection of constraints is postponed if the constraint solver is currently searching for a valid assignment. Therefore, under certain circumstances, the solution-plan computed by the solver may rely on assumptions about the contextual state that have become out-of-date.

4.4.4 User and home goals

A set of predefined goals depending on the user's routine and needs are made available through the set of buttons modeling complex activities that appear in the control panel of the supported UI. If the goal issued can be satisfied, the generated plan is executed and the home devices change state accordingly. If the goal is not satisfiable under the current context, a message is shown on the user interface. Table 4.1 shows how the goals described in natural language in the scenarios of Section 4.2 are expressed in the goal language presented in Section 3.4.1. In Goal 5, the `achieve-final` construct allows *robotLocation* to change many times while the robot is moving around to find and get the beers, as opposed to the `achieve-maint`($\bigwedge_i prop_i$) subgoals which ensure that once $\bigwedge_i prop_i$ become true at some state, they will stay true till the final state of the produced plan. The construct $goal_1$ `under_condition` $goal_0$ in Goal 4 ensures that only if $kitchSmoke = OFF$ holds will the rest of the goal about moving to the kitchen be carried on. In contrast, the `under_condition_or_not` structure in Goal 2 ensures that the subgoal $sitrAirCond = ON$ will be satisfied if the temperature is higher than 30 degrees, while if the temperature is lower than that, then only the rest of the conjunctions of subgoals will be looked after. Note that in the case of Goal 8, $healthEm + \text{'\_ALARM'}$ refers to a concatenation of strings, taken care by an external method call.

Given a goal, the composition module may come up with completely different plans, depending on the domain instance and initial state. The cause that triggers an event can also be taken into account: in the case of the health emergency goal, the notification message sent to the nurse's mobile phone and to the hospital incorporates the cause of the failure. If the Rule Engine triggered the goal because it recognized a fall, then $healthEm = \text{'FALL'}$, if the context conditions indicated a heart attack then $healthEm = \text{'HEART\_ATTACK'}$ etc. The heath emergency recognition can be based on complex computations on several sensed context variables, like e.g., presented in [Li et al., 2009]. For example, a fall could be detected based on the measurements delivered by wearable sensors, such as gyroscopes and accelerators, e.g., $angle_2 \div angle_1 > v1 \ \wedge \ acc_x > v2 \ \wedge \ acc_y > v3 \rightarrow healthEm = \text{'FALL'}$.

In Section 4.7, we present the plans generated by the planner module for each of the goals in Table 4.1 for a specific smart home domain and for the particular initial states we have used for testing our scenarios.

Goal 1: wake up (by Rule Engine)	`achieve-maint`($alarmClock = ON \ \wedge\ bedrCurtains = OPEN$ $\wedge\ bedLevel = HIGH\ )\ \wedge$ `achieve-maint`($bedrLight = ON$) `under_condition_or_not` `find_out-maint`($bedrNatLight = LOW$)

Goal 2:
watch TV
(by UI)
	`achieve-maint`($TvChannel = CH5\ \ \wedge\ personRoom = SITR$)
	$\wedge\ sitrLight = MEDIUM)\ \wedge$
	`achieve-maint`($sitrCurtains = CLOSED$) `under_condition_or_not`
	`find_out-maint`($sitrNatLight = LIGHT$) $\wedge$
	`achieve-maint`($sitrAirCond = ON$) `under_condition_or_not`
	`find_out-maint`($sitrTemperature > 30$)

Goal 3: deal with smoke leak (by Rule engine)	`achieve-maint`($kitchVentilator = ON\ \wedge$ $TvState = ALARM\ \ \wedge kitchWindow = OPEN)\ \wedge$ `achieve-final`($doorsLeadTo(KITCHEN) = CLOSED$) `under_condition_or_not` `achieve-maint`($personRoom \neq KITCHEN$)
Goal 4: smoke eliminated (by Rule engine)	`achieve-maint`($kitchVentilator = OFF\ \ \wedge\ \ TV = OFF$)
Goal 5 go to kitchen if safe (by UI)	`achieve-maint`($userLocation = AT\_OVEN$) `under_condition` `find_out-maint`($kitchSmoke = OFF$)
Goal 6: bring cold beer (by UI)	`achieve-final`($robotLocation = userLocation\ \wedge$ $robotHolds = BEER\ \ \wedge\ \ beerTaken = COLD$)
Goal 7: health emergency (by Rule Engine)	`achieve-maint`($nurseNotif = healthEm + `\_ALARM'\ \wedge$ $hospitalNotif = healthEm + `\_ALARM'\ \wedge$ $robotLocation = userLocation$) `under_condition_or_not` `find_out-maint`($nurseLocation = OUTSIDE$)
Goal 8: go to sleep (by UI)	`achieve-maint`($bedLevel = LOW\ \ \wedge\ \ alarmClockTime = 08{:}00$ $bedrCurtains = CLOSED\ \ \wedge\ \ bedrLight = OFF$)

Table 4.1: Goals for the smart home.

4.5 The prototype

The SM4ALL architecture is fully implemented, so as to test its technical properties, but also the experience of users with it. In the followings, we illustrate the prototype built based on the design presented in Section 4.3.

4.5.1 Pervasive and composition layers

We use the UPnP protocol to control the hardware devices, HTTP to enable access to remote clients, and the OSGi Service Platform as the intermediary between the physical UPnP layer and the service endpoints. The implementation is based on the

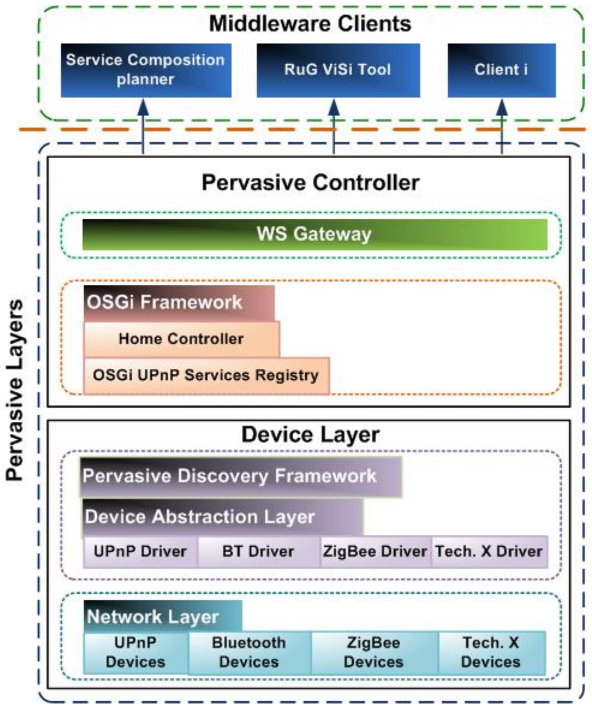


Figure 4.2: Architecture of the pervasive layer

Apache Felix project[1], which is a framework for writing devices exposed as UPnP (conforming with the OSGi UPnP specification version 1.1) and integrating them into the OSGi bundle repository. The interface of the services is written in Java. Figure 4.2 provides an overview of the internal structure of the pervasive layer, and the standards it uses. At the bottom sits the *network layer*, where physical devices with different networking protocols are located. The *device abstraction* layer abstracts away the underlying device technology by offering a driver for each of the technologies that the pervasive middleware supports. UPnP is used as the device-neutral technology to which all devices are wrapped by the respective driver, so that they can be then registered as OSGi services.

Besides UPnP, the prototype is able to automatically discover and support Bluetooth and ZigBee[2] devices, but it can be easily extended by adding drivers for other technologies as well. All devices' provided functionalities, independently of

[1]`felix.apache.org`
[2]`www.zigbee.org`

their network protocol, are described in compliance with the format prescribed by the OSGi UPnP specification, based on two types of elements: actions, which describe the operations a service supports, and *state variables* which represent the current state of an UPnP service. Whenever the value of a state variable changes, the respective event is published and propagated to the upper layers, notifying all subscribed parties.

OSGi is used as the platform to expose the devices' functionalities as services to the application layers. All components participating in the OSGi framework are deployed as so-called 'bundles'. The *Controller* is a special OSGi bundle that is responsible for handling events and controlling the services available in the framework, functioning as a bridge between the OSGi layer and the *WS gateway*, which executes a lightweight HTTP server that provides a standardized API to external components. Several clients can be registered to the server running on top of the OSGi framework, and call the exposed operations, such as getting the list of available services, subscribing to state variable events, or invoking an action offered by a service. Clients can be a BPEL engine, a home visualization application [Warriach et al., 2010], or the SM4ALL executor component.

The context awareness module is registered as a client to the WS endpoint on top of the pervasive layer, and subscribes to all change events of the variables involved in the service descriptions. The executor is also a client to the pervasive layer, without however subscribing to any variable change events: all it has to do is to be able to invoke services through the respective operation exposed by the WS server, as instructed by the composition component or directly by a simple command coming from the user layer. The semantic repository is yet another client of the pervasive layer, which is notified about the registration and de-registration of services, so that it adds a new instance of the abstract description of the associated service type. The pervasive layer and the clients registered to it, interact through the exchange of XML messages.

The context awareness acts as a listener to UPnP change variable events, which are further processed by the planner, so that they are ultimately incorporated at the initial state of the evolving CSP instance. Whenever a goal is issued, the RuG planner computes a solution which amounts to a valid plan (the implementation details of the RuG planner are provided in Section 3.7). The invocations of operations by the executor take place in a synchronous way, so that the next action in a totally ordered plan is invoked after a success return value is received by the previous action invocation.

4.5.2 The user layer

The Abstract Adaptive Interface (AAI) [Catarci et al., 2011] is registered as a client of the server on top of the OSGi framework, and whatever commands are issued via the concrete UIs are passed through it to the lower levels of the architecture. Its implementation is based on Apache Tomcat and Apache Axis. Two kind of UIs, a standard Web-based and a Brain Computer Interface, are connected to the AAI proxy, with which they interact via the exchange of XML messages. The Web Interface (or alternatively referred to as control panel) consists of dynamic and responsive web pages developed in JSP (Java Server Pages). The web pages provide different views of the virtual home. A global view presents all commands available to the user, in the form of clickable icons along with some descriptive text, corresponding either to atomic service operations, such as "turn on the light in the bedroom", or to complex goals (discussed in the following section) stored in the repository. The icons represent the current state of the devices, e.g. the icon indicating "Kitchen light ON" reflects the fact that the current state of the respective light is on, and clicking on it entails turning the light off, and thus refreshing the web page accordingly, after the notification originated by the respective UPnP device is received. A page depicting the rooms of a virtual apartment allows the user to move to the view of a particular room, from which only the devices whose

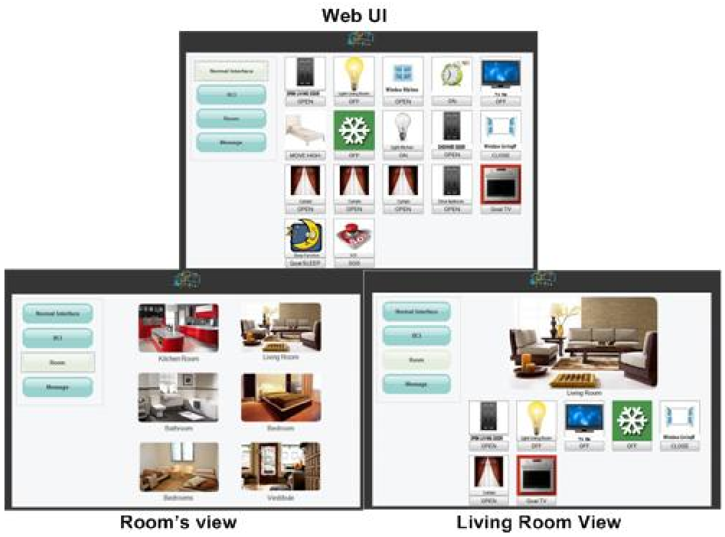


Figure 4.3: Views available via the Web interface.

location matches this room can be seen and controlled, as depicted in Figure 4.3. An extra Web page is reserved to reflect the current state of the devices at the user layer when the BCI is used.

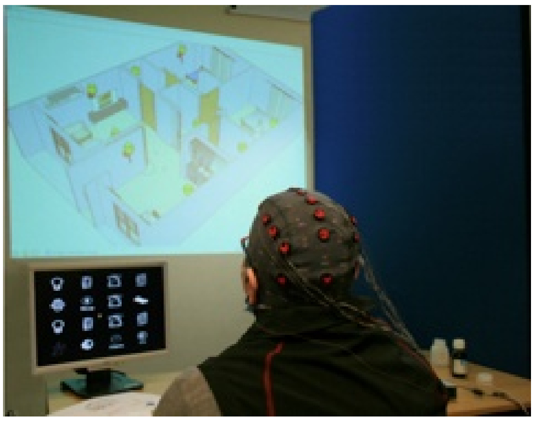


Figure 4.4: A BCI user interacting with the virtual home.

The BCI [Guger et al., 2009] is intended for users who have lost part of their motor ability due to aging or chronic neurological disorders, and are therefore unable or find it difficult to control the system via the standard Web interface. The BCI used in the test sessions is a portable asynchronous P300-based one, which translates the users' voluntary electroencephalographic (EEG) modulations into a control signal sent to some external device. The set of available commands modeling devices and goals are presented on a computer monitor in a form of a 4 by 4 matrix of flashing icons, which are flashing in a random order. The user wearing the EEG cap has to concentrate on a specific symbol, and whenever this is highlighted, a particular component is recognized in the measured EEG data. As a result, the identified command is transmitted to the AAI proxy. Because the flashing icons have to be static, i.e., they represent a device rather than its current state, the effect of the commands issued through the BCI are reflected via the web page of the Web Interface reserved for this purpose, which is updated whenever the state of a device changes. Figure 4.4 shows a user wearing an EEG cap, having the list of the screen with the flashing icons-commands on her left side, and viewing a projected simulation of a Smart Home. More details about the technology and testing results concerning the BCI can be found in [Aloise et al., 2010].

4.5.3 Simulation and visualization

Setting up an actual physical home or lab facility, furnished with modern sensors and actuators, is particularly expensive and effort-demanding, and performing tests with end-users in it can be inefficient. Therefore, it is instrumental to be able to acquire feedback from users before moving to the actual home and installing the real hardware devices, so that their requirements are taken into account early in the development process. To this end, we have implemented a virtual home environment which mimics as closely as possible an actual home setting, with simulated home services substituting physical hardware.

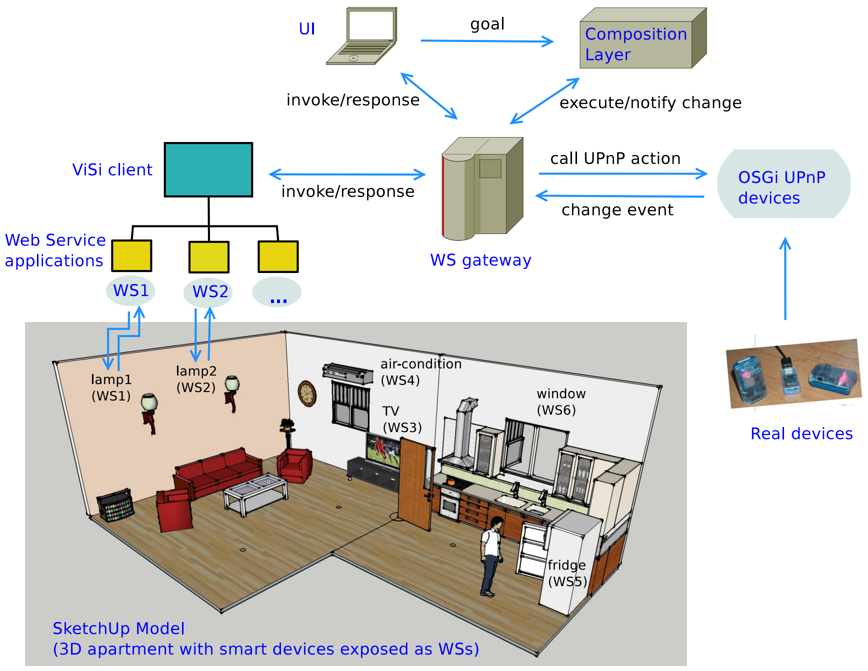


Figure 4.5: A screenshot of the home simulation.

The implementation of the simulation and visualization platform –the RuG ViSi tool– is based on Google SketchUp and has been demonstrated [Lazovik et al., 2009; Warriach et al., 2010]. It is integrated in the framework as a client of the WS endpoints at the pervasive layer. The apartment modeled is equipped with virtual devices implemented in Ruby[3], which are coupled with the Web Services exposed by the devices in the pervasive layer. In this way, whenever a device

[3]www.ruby-lang.org

state is changed, the result is replicated in real time in the visualized home. For instance, to model a reaction to fall detection, we have coupled a virtual alarm in the simulated house with a Sentilla mote[4] equipped with an accelerometer. The device uses the ZigBee communication protocol, and is wrapped in the OGSi layer. When shaken, the virtual alarm is turned to red, indicating a warning about the fall. The position of a user in the house is also shown, by coupling a user virtual service with a location detector that provides information about the user's location. Conversely, one can also control the devices at the pervasive layer through their virtual equivalents, so that there is a one-to-one mapping between the state of the OSGi UPnP-level environment and its visual reproduction. Figure 4.5 depicts a screenshot of a virtual house, and shows how visualized devices at the RuG ViSi level, such as lamps or the TV, interact with OSGi UPnP devices. In case of a composition, the series of effects entailed by the executed UPnP-level operations are reflected in the appropriate sequence at the visualization level.

4.5.4 Sample interaction flow

In order to demonstrate how the different components of the SM4ALL prototype are integrated and cooperate with each other, we go through a simple scenario and describe the control and information flow which realizes the desired behavior. Let us consider an example with a single physical device of type *Lamp*. The description of the *Lamp* type includes one published Boolean-valued UPnP variable, *lightStatus*, and three UPnP actions, *turnOn*, *turnOff* and a sensing one that returns the current value of *lightStatus*. It is also annotated by an appropriate semantic representation of the two actuator operations in terms of preconditions and effects. This description is stored in the Semantic Repository, as an XML file. During bootstrapping, the device is automatically discovered thanks to the OSGi-UPnP platform, and the Semantic Repository is notified about its subscription. As an effect, the Semantic Repository produces an instance-specific semantic description of the operations offered by the particular device, by adding the device's unique identifier (*lamp1*) as a prefix to the variables and actions that are declared in the abstract type *Lamp* description. All subscribed components are notified about the lamp availability and its description, so that based on that, the planner produces a domain consisting of one variable (*lamp1*::*status*) and two planning actions (*lamp1*::*turnOn* and *lamp1*::*turnOff*). To inform all interested parties about the current state of all devices, the executor invokes all available sensing actions. As an effect, the lamp device publishes an event which contains the current value of *lamp1*::*status*. The

[4]`www.sentilla.com`

context awareness component forwards this event to the UI, the Rule engine, and the planner, which sets accordingly its initial state (see Section 4.4.3).

Let us also assume that a trivial goal, which specifies that *lamp1* : *status* should be *TRUE*, is also stored in the repository. Thus, the UI along with the subscribed services is also notified about the existing goals, and thus presents in the Web Interface the appropriate icon. When the user selects this icon, a request for producing a composition that satisfies the respective goal is forwarded by the AAI proxy to the planner. Assuming that at the current state *lamp1*::*status* = *FALSE*, the planner computes a plan consisting of a single action *lamp1*::*turnOn*. The plan is passed to the executor, and ultimately the UPnP action called *turnOn* that belongs to device *lamp1* is called at the pervasive layer. The call is synchronous, and if successfully fulfilled a "success" reply is returned to the executor. Moreover, a change event concerning the *lamp1*::*status* variable of *lamp1* is published by the pervasive layer, and is ultimately received by all subscribed clients: the planner updates its initial state, so that it reflects the most latest values of the UPnP variables, and similarly the UI changes the icon which indicates the state of the lamp. If the response received by the executor indicates that the lamp is broken, the executor notifies the Semantic Repository, which takes the initiative to unsubscribe the service from the OSGi-UPnP platform, and asynchronously notify about this removal all interested parties. Thus, if the goal for turning on the light is issued again, the planner will respond that no plan can be found.

4.6 Practically engineering a Smart Home

The SM4ALL architecture can in principle be fitted to any existing home. In the followings, we describe the actual phases that such a fitting process, necessary to make a home smart with respect to the SM4ALL approach, would have to go through. Clearly, the average home user does not want to bother with technical details, and is willing at most to provide some input on what are the goals he wishes to regularly perform in the home. In Figure 4.6, we provide a schematization of the process where we show from left to right the state in which the home goes through, and we distinguish the engineering phases (top) and the stakeholders who are actually responsible for successfully completing each phase of the process (bottom).

The first phase of the SM4ALL fitting process requires to make an inventory of the devices present in the home and identify which additional hardware is required to cover the user needs e.g., door motors, smart meters, smart fridge, etc. Then the new devices have to be physically installed in the home. In this initial phase, it is mostly to the SM4ALL expert to do the requirement engineering and to the carpenter to fit the hardware in the home. The second phase consists of making the hardware interoperate, which relates to the pervasive layer of the SM4ALL architecture. An internetwork expert has to make sure that all devices are connected to the network and can exchange messages. This is in principle effortless for the devices which adhere to UPnP standards, and should be easily achieved also for other devices based on known protocols.

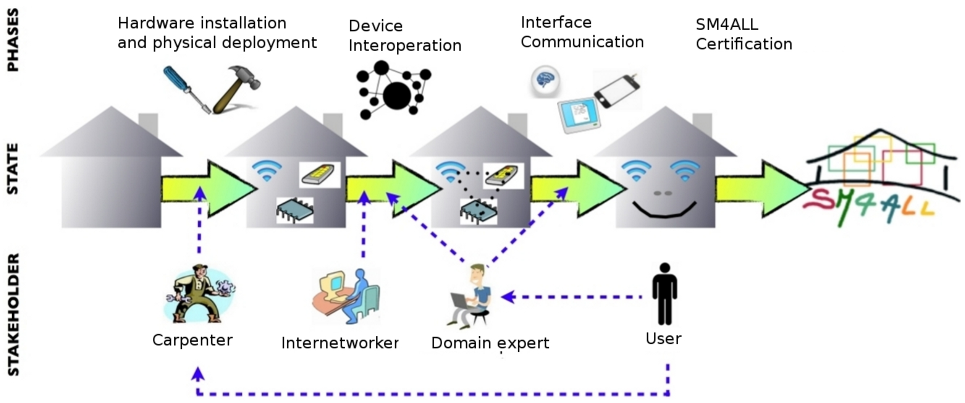


Figure 4.6: The process of fitting the SM4ALL architecture into an existing home.

If a new type of device is introduced into the SM4ALL system, i.e., the functionalities it offers have never been described before in terms of preconditions and effects, it is also necessary to add these extra semantics. If the new device, e.g., a particular lamp, is an instance of a known service type, e.g., the lamp-A type, then all that has to be done is to declare the type of the device. This task is performed by the domain designer. The effort of this stakeholder is thus considerable at the beginning, when the behavior of the supported device types have to be semantically specified, and gradually diminishes as more and more devices are added to the semantic repository to be used by the composition layer. The last technical phase of the process consists in customizing the interface to the home for the user. This means identifying the appropriate type of interface hardware (e.g., BCI, touchscreen, voice interface), and also the complex requests according to the users' needs and routine. The requests are formulated by the domain designer or the experienced user in conformity with the declarative fashion of the extended goal language, and are tied to an icon that appears in the user interface. Emergency goals are also formulated along with the conditions that enact them and added to the rule engine. Again, the specification of the goals requires more effort in the first installations, and becomes less demanding as reuse of already formulated goals becomes the norm. Finally, one could imagine a final certification phase where an expert or standardization body may certificate the home to be SM4ALL compliant, thus allowing for interoperation with new SM4ALL certified hardware.

Considering the shift towards interoperable and service-oriented home setting, OSGi-UPnP is a good candidate for constituting a common standard for home appliances, especially since it can easily support different network protocols. However, the vendors who offer devices with a ready-to-use OSGi interface, certified as "OSGi Compliant", are still limited (these include Samsung, 4DhomeNet, IBM, Connected Systems and others). Given that the reality in home appliances is still far from the adoption of some common standard, the task of enabling compliance with the OSGi platform falls on the home designer. Depending on the specifications of each device, this task may vary from easy to difficult or impossible. For some frameworks it is possible to wrap them directly as OSGi bundles, for some implementing an adaption layer is necessary, while for others some patching of their sources is unavoidable. To give an example from our own experience, the task of representing a Sentilla accelerometer in terms of the UPnP standards is a matter of less than an hour, however implementing an adapter for Zigbee, the network protocol used by the device, required a couple of weeks development time. However, an adapter for a given network protocol needs to be developed only once, and as long as well-known protocol discovery plugins and adapters are available at the OSGi home gateway, new devices that use these protocols can be automatically integrated.

4.7 Technical evaluation

We provide both a technical evaluation of the system to assess whether the architecture is effective and the used techniques have adequate performance; and a user evaluation in Section 4.8 to give an initial assessment of the acceptability and usability of the solution. The major focus of the technical evaluation is on the composition component, and is based on the scenario described in Section 4.2. The tests have been run on a 1.83 Ghz computer running Debian lenny and Java 1.6.0_12. The service components are simulated with accordance to the OSGI UPnP Device Specification and are exposed as OSGi bundles. Each device offers one or more services, each of which involves a number of actions and state variables.

The composition layer subscribes as a client to the Web Service server: the semantic repository gets the list of active devices and provides the respective action descriptions, the context awareness component subscribes to the events regarding all domain variables, and the executor is also connected, ready to receive invocation instructions. For the evaluation purposes we model a home with 5 rooms, and 14 simulated UPnP devices. For simplicity, we have simulated one aggregated device for managing all doors by passing the specific door which the open/close operations affect as an input argument, thus having one device for controlling 4 doors, and a similar case holds for the lights, window, and curtains devices. The total number of UPnP actions implemented by the devices is 28 (plus the sensing actions that are defined for each state variable in the domain) and involve 37 different state variables. These actions model the getting and setting of the declared state variables. For example, the air condition device comprises two state variables, $AirConditionState \in Boolean$ and $AirCondTemperature \in Integer$. It also offers two UPnP actions of activator type, each one defined in a separate service: a *SetEnumStateVar* for turning on and off *AirCondState*, and *SetIntStateVar* for controlling the desired *AirCondTemperature*. For the purposes of simulation, the user himself is represented as one of the services at the pervasive layer. One can think of a person on a wheel chair, which can be controlled automatically, and its position is being tracked by a localization component. The robot device refers to a robot that, without loss of generality, is dedicated in the testing scenario to the task of bringing beers to his master: it can move around the house (as described in Section 3.3.1), get a beer from the fridge or the storage, sense if it is cold or not, and cool it if necessary by putting it in the fridge and waiting for some time. A state variable can be involved in more than one services, possibly belonging to different devices, like the *FridgeDoor* variable, which can be controlled automatically or directly by the *Robot* device.

Initial state	Plan
Goal 1 (wake up)	
`[1a]` : *bedLevel=LOW, bedrNaturalLight=DARK, bedrCurtains=CLOSED, bedrLight=OFF*	*{set_bedLevel(MEDIUM), ring_alarmClock, open_bedrCurtains turnOn_bedrLight}, set_bedLevel(HIGH)*
`[1b]` : Same as above, but with *bedrNaturalLight=LIGHT*	*{set_bedLevel(MEDIUM), ring_alarmClock, open_bedrCurtains}, set_bedLevel(HIGH)*
Goal 2 (watch TV)	
`[2a]`: *TV=OFF, sitrLight=LIGHT, sitrTemperature=32, sitrNaturalLight=LIGHT, sitrCurtains=OPEN,* $\forall i\ door_i$*=CLOSED*	*{turnOn_lightSitr(MEDIUM), close_sitrCurtains, turnOn_sitrAirCond, set_TV(ON)}, set_TVChannel(CH5)*

[2b]: Same as above, but with *sitrTemperature=20, sitrNaturalLight=DARK*	{*turn_lightSitr(MEDIUM), set_TV(ON)*}, *set_TVChannel(CH5)*
Goal 3 (deal with smoke leak)	
[3a]: *userLocation=AT_OVEN, TV=ON, kitchWindow=CLOSED, ventilator=OFF, kitchSitrDoor=OPEN*	{*turn_on_ventilator, open_kitchWindow, open_kitchSitrDoor, set_TV(ALARM)*}, *moveUser_to(AT_KITCH_DOOR), moveUser_to(AT_TV), close_kitchSitrDoor*
[3b]: Same as above with *userLocation=AT_TV*	{*turnOn_ventilator, open_kitchWindow, close_kitchSitrDoor, set_TV(ALARM)*}
Goal 4 (smoke eliminated)	
[4]: *kitchSmoke=OFF, TV=ALARM, kitchVentilator=ON*	{*turnOff_kitchVentilator, set_TV(OFF)*}
Goal 5 (go to kitchen if safe)	
[5a]: *kitchenSmoke=ON*	The goal cannot be satisfied
[5b]: Same as above but with *kitchenSmoke=OFF, userLocation=AT_STOR_ENTR*	*open_kitchStorDoor, moveUser_to(AT_FRIDGE), moveUser_to(AT_OVEN), close_kitchStorDoor*
Goal 6 (get cold beer)	
[6a]: *robotLocation=AT_START, userLocation=AT_SOFA, kitchStorDoor=CLOSED, sitrKitchDoor = CLOSED, numOfBeersInFridge=0, numOfBeersInStorage=8, robotHolds=EMPTY, fridgeDoor=CLOSED*	{*open_sitrKitchDoor, open_kitchStorDoor*}, *moveRobot_to(AT_OVEN), moveRobot_to(AT_STOR_SHELF), robotGetsBeerFromStorage, open_fridgeDoor, moveRobot_to(AT_FRIDGE), robotCoolsBeer*, {*open_fridgeDoor, close_kitchStorDoor*}, *robotGetsBeerFromFridge*, {*moveRobot_to(AT_SOFA), close_fridgeDoor*}

`[6b]`: Same as above but with *numOfBeersInFridge=1*	*open_sitrKitchDoor, moveRobot_to(AT_OVEN), open_fridgeDoor, moveRobot_to(AT_FRIDGE),* {*robotGetsBeerFromFridge, open_kitchStorDoor, open_bedrBathrDoor, open_sitrBedrDoor*}, {*moveRobot_to(AT_SOFA), close_fridgeDoor*}
Goal 7 (health emergency)	
`[7]`: *nurseLocation=OUTSIDE, cause=FALL,* $\forall i$ *door*$_i$*=CLOSED, robotLocation=AT_TV*	{*sendMsg_NrsPDA(FALL_ALARM), notifyHospital(FALL_ALARM), open_sitrBedrDoor, open_bedrBathrDoor*}, *moveRobot_to(AT_BED), moveRobot_to(AT_BATH)*
Goal 8 (go to sleep)	
`[8]`: *bedLevel=UP, bedrCurtains=OPEN, bedrWindow=OPEN, bedrLight=ON*	{*set_bedLevel(MEDIUM), close_bedrWindow, set_alarmClock(08:00), turn_on_bedrLight=OFF*}, {*close_bedrCurtains, set_bedLevel(LOW)*}

Table 4.2: The plans generated for the goals in Table 4.1, for different initial states (only the initial values that are of interest to the goal are mentioned).

Table 4.2 shows the sequence of actions generated by the planner for each of the goals in Table 4.1 (because of space issues, only the initial values that are of interest to the goal are mentioned in the table, rather than the full initial context of the home and user). Each plan is represented as a partially ordered set of actions, with comma-separated actions $a1, a2$ indicating that action a_1 has to be performed before a_2, while the actions included in the same set $\{a_1, \ldots, a_n\}$ can be executed in parallel. One can see that depending on the current contextual state, the plans which satisfy a given goal may be radically different. The planner produces plans with a high degree of parallelism. However, in the current implementation, the executor does not support parallel action executions, and thus the actual invocations are performed in a serialized manner. Because of the use of a random search

Test	Number of actions in plan	Plan Composition time (in sec)	Plan execution time (in sec)
`[1a]` (wake up, no natural light)	5	1.1	0.5
`[1b]` (wake up, natural light)	4	1	0.3
`[2a]` (watch TV, temperature too high, natural light)	5	1.5	0.7
`[2b]` (watch TV, temperature ok, no natural light)	3	1.4	0.6
`[3a]` (deal with smoke, user in kitchen)	7–9	1.2	1
`[3b]` (deal with smoke, user not in kitchen)	4	0.6	0.4
`[4]` (smoke eliminated)	2	0.7	0.4
`[5a]` (go to kitchen, smoke on)	0	0.1	-
`[5b]` (go to kitchen, smoke off)	4–5	0.7	0.4
`[6a]` (get beer, fridge empty)	12–15	2.6	0.8
`[6b]` (get beer, fridge full)	7–10	2.2	0.7
`[7]` (health emergency)	6	1.4	0.5
`[8]` (go to sleep, bedrWindow open)	6	1.2	0.6

Table 4.3: Time required for composition and execution.

strategy, the plans returned may slightly vary between different runs: the order of some actions may be different or some extra actions may be included. The latter is due to the fact that the planner does not generate optimal plans, i.e., the ones comprising the minimum possible number of actions. Thus, it may occur that a plan includes unnecessary actions or useless repetitions of actions, as e.g., in plan `[5a]`, where some doors are opened for no reason without that being necessary for the goal's satisfaction.

The time required by the planner to subscribe to the available UPnP services, build the planning-level domain description, and sense the first initial state, by invoking the UPnP sensing actions for all state variables, is 10.8 sec. After that, it is ready to generate plans for the goals that are issued by the user or the rule engine, while it is notified about any changes by the context module, and updates its current initial state accordingly. We have measured the time the planner takes to compute each of the plans, as well as the time needed for each plan to be actually executed by invoking the respective UPnP actions. These results are summarized in Table 4.3. We have used a random branching strategy during constraint solving, by resting the search after a maximum number of backtracks. The reported times both for composition and execution are the average over 5 separate runs. Of course, if we consider real rather than simulated devices, the execution times especially for motion-related actions would be much longer. The most demanding goal is Goal 6 (getting a cold beer), especially in the case where there are no beers already stored in the fridge, mainly due to the substantial backtracking required to find a solution (up to 478 backtracks, compared to 47 backtracks in the worst case concerning the other goals). The invocation time per operation call is a up to a few msec for all devices, since these are simulated. The execution time amounts to the time required for the interaction between the composition module and the executor, i.e., the time for mapping the planning actions to UPnP actions, calling them, and getting the response, while at the same time a listener parses the UPnP change variable events and updates the CSP.

An evaluation of the performance of the pervasive layer with respect to the number of clients it can support in association with the number of connected devices is beyond the scope of the current presentation. Results with respect to such a parameter are presented in [Kaldeli et al., 2010].

4.7.1 Replanning scenarios

For the purpose of simulating failure handling scenarios, we only consider two basic kinds of UPnP action invocation responses: "success" and "failure". The policy upon a failure response is first to try to invoke the erroneous operation once more, and if a failure occurs for a second time, then to attempt to replan. The application of different policies depending on the kind and severity of contingencies that are detecting during execution may be possible if a more subtle distinction of the cause of failure is available (e.g., attempt to re-invoke the same service several times, or directly remove the service if the response indicates a permanent failure). Timeout conditions may differ depending on the type of action (e.g., a service operation for closing/opening a door should respond within a second, while closing the curtains takes longer). Timeouts are handled the same way as failures.

Table 4.4 summarizes the behavior and performance of the system under certain circumstances concerning the scenarios described in Section 4.2.1, where an error occurs during the execution of the initial plan. The services used for the tests are the same as before, with the addition of 3 extra service actions. For the purpose of scenario 1, which refers to the goal for watching TV, a "switch" operation is added to the robot device, which models its ability to turn on the TV if its location is in front of it. To simulate the different scenarios regarding scenario 2, two extra door services are added, to simulate the possibilities for the robot to follow alternative

Scenario 1: Re-planning for Goal 2 (watch TV)

Initial state:	as in Table4.2`[2a]` and robotLocation=AT_BED
Initial plan:	as in Table4.2`[2a]`
Execute plan:	*set_TV(ON)* responds with failure twice, re-planning
New plan:	*open_sitrBedrDoor, moveRobot_to(AT_SOFA) , moveRobot_to(AT_TV), robotSetTV(ON), set_TVChannel(CH5)*
Execute plan:	Completed successfully

Planning attempts: 2
Total planning time: 3.3 sec

Scenario 2: Re-planning for Goal 6 (bring cold beer)

Initial state:	as in Table4.2`[6b]`
Initial plan:	as in Table4.2`[6b]`
Execute plan:	*open_sitrKitchDoor* responds with failure twice, re-planning
New plan:	{*open_sitrStorDoor, open_kitchStorDoor*}, *moveRobot_to(AT_STOR_SHELF), open_fridgeDoor, moveRobot_to(AT_FRIDGE), moveRobot_to(AT_SOFA)*
Execute plan:	*open_kitchStorDoor* times out twice, re-planning
New plan:	{*open_sitrBedrDoor, open_bedrKitchDoor*}, *moveRobot_to(AT_BED), open_fridgeDoor, moveRobot_to(AT_FRIDGE), moveRobot_to(AT_SOFA)*
Execute plan:	*open_sitrBedrDoor* times out twice, re-planning
New Plan:	The goal cannot be satisfied

Planning attempts: 4
Total planning time: 12.3 sec

Table 4.4: Behavior and time results of the planner for two possible re-planning scenarios depending on execution circumstances

routes to reach the kitchen.

In scenario 1, after the invocation to remotely turn on the TV fails, a second attempt is made, and after the service responds with a failure again, the erroneous action is removed from the constraint network, through the addition of the appropriate prohibitive constraint (see Section 4.4.3). The planner is called again, and the new composition instructs the robot to move from the bed where it currently is to the TV and switch it on. In scenario 2, the robot cannot move from the sitting room to the kitchen directly, because the door that connects the two rooms is blocked. After pruning the faulty door from the search space, an plan that leads the robot to the kitchen through the storage room is generated. However, the door that connects the living room with the storage room also proves to be defect, and the invocation for opening it times out. As a result, the planner will try to find an alternative route through the bedroom. Due to bad luck though, it turns out that the door to the bedroom is also out of order, and the planner will make a fourth attempt to compute a plan which does not include any of the malfunctioning doors. Since no alternative plan can be found, the plan reports that the requested goal cannot be satisfied given the current circumstances.

4.8 User evaluation

According to the ISO 9241-11 standard, usability refers to "the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use". In the case of the SM4ALL framework, the context of use is determined by the diverse requirements, abilities and technological knowledge of the intended users. To perform a fair test, we identify two antithetic groups: the first group comprises elderly people, some of whom suffer from severe motor disability, and will be referred to in the followings as the Elderly and Disabled (E/D) group; the second group consists of young people experienced with computer innovations, who will from now on be referred to as the Technological Savvies (TS). The focus of the testing methodology is to assess whether the architectural design and implementation of the SM4ALL framework is useful and usable by users with diverse capabilities and aspirations, without the need of personalized reconfigurations.

The user evaluation methodology bases on a quantitative analysis regarding a number of basic dimensions, which are determined by connecting the established usability components in the literature [Nielsen, 1994*b*,*a*; Kim et al., 2003] with the context of domotic environments. Each of the dimensions takes into account some specific metric parameters, about which users are asked to give a score (usually in a scale from 0 to 4). The main dimensions' scores are calculated by taking the average of the parameters' values. In the followings we list the main usability features along with their relevant metric components:

- *Acceptability* of domotic solutions in general captures the opinion of users towards the importance of domotics technology, their eagerness to delegate tasks to a computer, and their extent of concerns towards privacy intrusion.

- *Learnability* assesses how easy it is for the user to get familiarized with the system. It refers to the amount of effort the users have to make in order to comprehend the functionalities of the system, and to be able to control it in accordance with the tasks he wants to accomplish.

- *Aggregate system effectiveness* measures how satisfied the users are from the system, by taking into account a number of aspects referring to different components. Virtual apartment effectiveness refers to the extent to which the design of the home and the optical effects at the visualization level give the feeling of a real home. Two metrics are used with respect to the control panel's usability, assessing how clear and attractive the Web Interface is, and how convenient to use it is. One more metric is used to assess the satisfaction of users with respect to the support of complex goals. Finally, the extent of difficulties or irritation resulting from some unexpected behavior or missing feature, and from low performance is also taken into account.

- *Efficiency* is concerned with the speed at which the system performs certain tasks. Time efficiency is measured with respect to the user's assessment of the time required to complete atomic operations and complex goals.

4.8.1 Experimental setup

Demographics

The E/D group consists of 31 elderly people (12 males and 19 females), between 47 and 91 years old, and an average age of 71 years. Eight persons out of this group suffer from chronic neurological disorders and make use of the BCI (5 males, 2 females; mean age=64.85 ± 5.81 years). All users of this group are clients of the Frisian health care institution in the Netherlands "Thuiszorg Het Friese Land" (THFL). 13 of the users in this group have experience with computers, and 9 of them make use of some kind of domotic devices at their house (e.g., automatic shutters, motorized armchair, lights etc). The testing took place in the months of October and November 2010. The TS group consists of 30 students who are doing their MSc in Computer Science at the University of Groningen, and attend a course on ubiquitous computing in the spring of 2011. Their age ranges between 21 and 30 years old, with an average age of 25 years. 10% of them are female and 90%

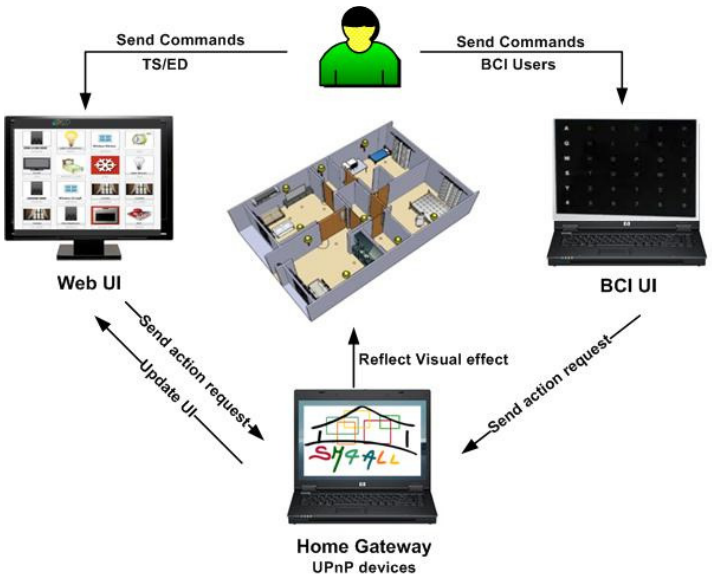


Figure 4.7: Basic interactions between the components at the experimental setup.

male. Naturally, all members of this group are advanced users of computers, and 3 of them have used actual domotic devices.

Testing sessions

Figure 4.7 provides a high-level overview of the essential components of the experimental setup, and the basic sequence of events that take place between them. The user issues his commands via the Web Interface or the BCI panel, depending on whether she suffers from motor disabilities or not. The instructions are passed to the lower levels of the SM4ALL platform, and the results are reflected at the visualization level, projected on a separate screen, while the Web Interface view is updated accordingly. The home instance visualized and controlled by the users is based on a virtual reconstruction of a real apartment built at the premises of the Fondazione Santa Lucia in Rome. The apartment consists of four rooms (two bedrooms, a kitchen and a living room), equipped with 32 simulated devices (lights, doors, motorized bed, curtains, windows, TV, air condition etc). The Web Interface provides icons for controlling individually all available devices, organized per room view, and additional icons for modeling two complex goals, one for adjusting the living room conditions for watching TV, and one for preparing the bedroom for sleeping. The BCI offers control capabilities for only a subset of the devices and goals, since it supports up to 16 icons at a time. Users are asked to follow an instructive scenario, i.e., a predetermined set of actions specified by the experimenters, which includes achieving certain conditions, e.g., preparing the house for the night, by issuing individual commands, such as “turn off kitchen light”, and the complex goals that are made available to them. The user can then freely interact with the system. At the end, the user is asked to fill in a questionnaire, where she is required to provide a score for each of the usability components already described. Along with the metric values, users are encouraged to provide a short explanation of their assessment. The questionnaire addressed to the TS group includes some extra questions with respect to the one addressed to the E/D group, requesting more details about the assessment of time efficiency and the effectiveness of complex goals.

The testing sessions with the E/D group have been arranged and conducted in cooperation with staff members of the THFL. The testing sessions which do not involve use of the BCI took place separately for each user, at his home of residence. Each individual testing session, including the platforms setup in the users environment, lasted one hour in average. The BCI testing sessions took place at the THFL premises, conducted in two consecutive days. The first day was dedicated to the training of the BCI system, and making a profile of the brain activity of each participant (the BCI training requires 30 minutes on average per person). The second day the users were ready to interact with the actual SM4ALL platform. The tests with the TS group were conducted during three different sessions in a university lecture room.

4.8.2 Usability evaluation results

Elderly people

Table 4.5 summarizes the quantitative findings of the usability tests with the members of the E/D group. All quantitative factors included in the questionnaires are mapped to a scale from 0 to 1. Results of time efficiency assessment or particular to the complex goals are not presented, as this was intended specifically for the TS group. A natural observation is that the amount of effort reported by the participants of the BCI tests is higher than the effort assessment of the users who did not have to learn how to use the BCI equipment. The findings indicate that satisfaction from the Web Interface effectiveness is particularly high, while satisfaction from the virtual experience delivered by the visual effects is quite lower. The data are further analyzed in the comparative evaluation presented in at the end of this section.

	Mean		
	Non-BCI	BCI	Overall
Acceptance of privacy disclosure [0 (negative) - 1 (positive)]	0.79	0.91	0.82
Acceptance of tasks automation	0.7	0.82	0.72
Aggregate domotics acceptance	0.78	0.88	0.8
Learnability effort [0(low) - 1 (high)]	0.15	0.37	0.26
Control panel effectiveness [0 (negative) - 1 (positive)]	0.88	0.93	0.89
Virtual home effectiveness	0.78	0.69	0.76
Aggregate framework effectiveness	0.83	0.79	0.82

Table 4.5: Aggregated results of the usability tests with the E/D group.

Technological savvies

The quantitative results of the testing sessions conducted with the members of the TS group are presented in Table 4.6. Similarly to the findings from the E/D group, the technological savvies gave a high score to the effectiveness of the control panel and a lower score to the satisfaction from the virtual home feeling. It is worth mentioning the particularly high assessment of the complex goals effectiveness, which the members of this group highlighted as particularly interesting and useful.

Satisfaction from the time performance of tasks associated to complex goals is rather smaller with regard to efficiency of performing atomic operations. The diagram Figure 4.8 shows how satisfaction from time efficiency is distributed in the TS group. The results indicate that 76% of the users give a time efficiency rank of over 0.7, while only 3% of the users give an assessment lower than 0.5. Further analysis of the rest of the dimensions is provided in the next section.

Comparative analysis

The comparison between the results of the two groups can lead to some interesting conclusions. Regarding the acceptability of domotic technologies, the E/D group is more reluctant to have a computer overtaking tasks, while the TS group is more positive towards the automation of domotic routines. 30% of the E/D group gives a score below 0.5 to acceptance of domotic tasks automation. On the other hand, members of the E/D group express less concerns about privacy in comparison with

	Mean
Acceptance of privacy disclosure [0 (negative) - 1 (positive)]	0.48
Acceptance of tasks automation	0.81
Aggregate domotics acceptance	0.74
Learnability effort [0(low) - 1 (high)]	0.4
Complex goals effectiveness [0 (negative) - 1 (positive)]	0.89
Virtual home effectiveness	0.7
Control panel effectiveness	0.84
Aggregate framework effectiveness	0.78
Time efficiency complex goals [0 (slow) - 1 (fast)]	0.79
Time efficiency for atomic actions	0.9
Aggregate time efficiency	0.83

Table 4.6: Aggregated results of the usability tests with the TS group.

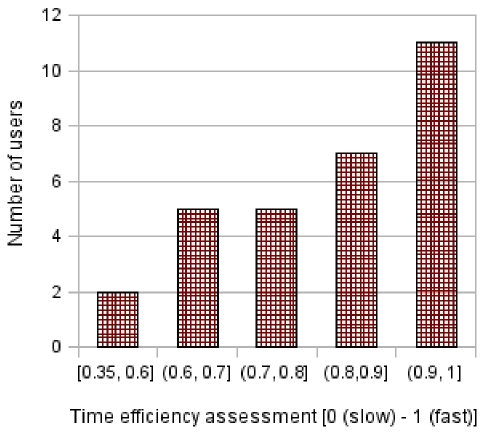


Figure 4.8: Overall time efficiency assessment by the TS group.

the TS group. Many of the elderly, and especially the ones suffering from serious disabilities or health problems, are quite used to being surveilled and looked after by specialized personnel, such as nurses or household assistants, and therefore 50% of them do not express any considerable worries about privacy intrusion. On the contrary, 40% of the young technological savvies, are seriously concerned about invasion of privacy and violation of personal space.

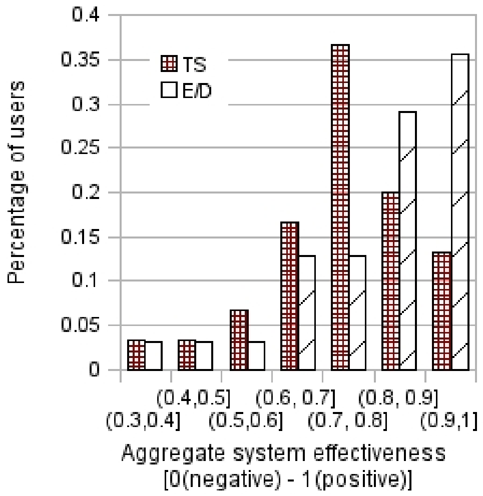


Figure 4.9: Comparative distribution of aggregate assessment with respect to system's effectiveness.

With respect to the amount of time required for understanding and learning how to use the framework, members of the E/D group needed 10 min on average, while the technological savvies 2 min on average. For the members of the E/D group who had not used a computer before, considerable time was required to get familiarized with the use of a mouse. Although members of the E/D group presumably needed more time to learn the system, both groups assessed that the system was easy to perceive and control: 73% of the E/D and 83% of the TS users put the amount of the learnability effort between 0 and 0.25.

Regarding aggregated satisfaction from the system's effectiveness, it should be noted that in the case of the E/D group the average is in most cases calculated with respect to less constituting parameters, because the members of the E/D group left many questions unanswered. As can be seen in the distributions plotted in Figure 4.9, the findings regarding the TS group can be approximated by a Gaussian distribution, with a mean of $\mu = 0.78$ and $\sigma = 0.15$. In the case of the E/D group, the population is concentrated around higher values of aggregate effectiveness assessment, with 65% giving a value of over 0.8. In both groups, for 77% of the users, the aggregate assessment for effectiveness is over 0.7.

Chapter 5

Plan Orchestration via Altering the CSP

Due to the conditions of incomplete knowledge which has to be sensed, as well as other sources of contingency such as failures or unexpected changes in the state of the world, the problem of service composition cannot be tackled in detachment from the actual context of execution. In Chapter 3 we have presented an intuitive knowledge-level encoding which accommodates for proactive information seeking as part of the planning process, based on the agent's knowledge and the way that this evolves via the application of sensing actions. We have described how the functioning of the RuG planner leads to the construction of plans which rely on certain optimistic assumptions about the actual state of the world. Due to the large amount of possible service outcomes, which are often numeric, and the range of unforeseen contingencies at runtime, computing offline contingent plans for all possible configurations, as e.g., in [To et al., 2011; Bryce et al., 2006; Pistore, Marconi, Bertoli and Traverso, 2005; Hoffmann et al., 2010], becomes particularly expensive or even infeasible. Moreover, most planning approaches to WSC completely disregard recovery from failures and, with some notable exceptions such as [Au et al., 2005; Bertoli et al., 2009; Klusch and Renner, 2006], they rely on the assumption of a static environment. Information is expected to persist till the end of execution, while it is taken for granted that the world changes only as a result of the actions of the planning agent, and in conformance with their functional description.

In order to overcome these limitations without compromising the requirement for domain-independence, an approach that integrates planning, monitoring and execution is opted for, so that findings about the environment are directly integrated into the planning process. To this end, the computation of alternative plans is delayed until this is called for by the new information acquired during execution. Continual planning is performed, so that the upcoming plan steps anticipated offline can be revised as execution progresses, in face of inconsistencies that arise either from the newly acquired information, from services' inconsistent behaviors or from the actions of exogenous agents that interfere with the plan. In such a setting, the goal is considered to have been accomplished, if all individual actions in this sequence of updated plans are executed successfully. We call this process that starts from an initial plan and moves on by interweaving action invocations with plan revisions an *orchestration*. The concept of orchestration in that context is in line with the intended meaning of the term in Service Oriented Computing, in the broad sense of realizing a conductor (planning agent) who decides which instruments (service operations) should play which notes in a piece of music (goal or plan). In practice, orchestration usually refers to the process of concretizing synthesis and making it executable. In our approach, synthesis and execution-time coordination are blended together, and orchestration refers to that integrated process.

The orchestration approach adopted herein shares many concerns with the frameworks for planning in the environment of Unix operating systems presented in [Golden and Weld, 1996; Golden, 1998; Knoblock, 1995], including the modifications incurred to the current plan depending on the bindings of the runtime variables (analogous to the RuG planner's "var_response" variables) and the world state changes. However, these approaches are still dependent on well-crafted domain-specific knowledge regarding e.g., failure handlers, causal links between producers and consumers or search control. An approach for interleaving planning and execution, where parts of the plan are postponed depending on a number of replanning assertions is described in [Brenner and Nebel, 2009].

In the followings, we show how orchestration can be performed by applying gradual modifications to the CSP instance which models the planning domain, the goal and the constantly changing contextual state. The modifications correspond to the incorporation of new facts about the state of the world or the removal of obsolete ones, to refinements of the sequence of actions included in the already computed plan, or to useful information about the behavior of services that is collected by inspecting how they operate. The orchestration algorithm is characterized by the following traits:

- It exploits the high degree of parallelism in the plan, by performing concurrent invocations and handling the responses in a non-blocking way. Since the execution time of some service operations may be very long, the orchestrator is able to continue planning and proceed to the execution of subsequent actions if this is allowed by the domain and goal restrictions, while waiting for the response of some service invocation.

- It continuously incorporates asynchronous context updates, if such a mechanism is available.

- Provides the means to recover from failure-indicating responses and timeouts. Other arbitrary service outcomes that contradict its expected behavior can also be tolerated under the assumption of a consistent and timely publish-subscribe mechanism.

- Can cope with possible discrepancies due to the activity of exogenous agents, which may act in parallel with the plan execution and interfere with it.
- Seeks to keep a balance between the effort spent in computing a new plan from scratch and in refining the existing one.
- It takes care of the data flow by instantiating numeric-valued input parameters to the actually sensed output parameters. This is performed through plan refinement, by considering the history of known facts in the form of constraints and the goal requirements.

5.1 Architectural overview

Figure 5.1 provides a high-level overview of the main interactions of the orchestrator with the other components which interweave planning, monitoring and execution. The orchestrator is realized in accordance with the *actor model* [Greif, 1975], which enables a high degree of concurrent computations, through the asynchronous exchange of messages, the creation of new concurrent entities on the spot, and the designation of certain tasks to them, so that they are carried out in parallel.

Whenever a request about computing a plan for a new goal is issued, the orchestrator asks the planner to compute an initial optimistic plan as described in Chapter 3. This entails adding dynamically the constraints that follow from the goal (see Section 3.6) to the *model* kept by the RuG planner, i.e., the constraint network that models the planning domain (see Section 3.3). The *solver* takes into consideration the current model along with some assignments to CSP-level variables that reflect the initial planning state, as delivered by the current context. The solution to the CSP which amounts to an offline plan (if one exists) is passed to the orchestrator, whose task is to gradually execute and update it, according to the information it acquires through its interaction with the environment. Every step of the plan involves a set of parallel actions (see Definition 6), which are mapped to a set of respective concrete service operations that are executed concurrently, as described in Section 5.3.2.

Since the offline plan has no way to anticipate any values that are to be observed, whenever new information is sensed, some revision is needed. For example, if the user wants to send some mail to a particular address, which is unknown and has to be supplied by some address-providing service, then at the point when the address becomes known, re-solving is required to instantiate the right input arguments of the sensing actions that depend on that information. Other sources of contingencies, such as failures or timeouts, may also call for a refinement of the plan, as discussed in extent in Section 5.3.3. For example, if some actions respond with a permanent failure or persistently do not respond within some expected timeout, then an alternative plan which does not include these actions has to be sought for. While waiting for actions that take long to respond, the orchestration process goes on with continuously planning new actions to be executed, depending on the latest view it has about the world state and the history of execution so far. These conditions also determine whether it is preferable to refine the existing plan or to plan from scratch. Requests for refinement are addressed directly at the solver level of the RuG planner, as long as the basic model remains the same (i.e., changes refer only to the initial state). This way, the search process can start from an already propagated instance of the model, and proceed from a state where some variables are already instantiated according to the most up-to-date context. The details of how the plan refinement process works are discussed in Section 5.3.1.

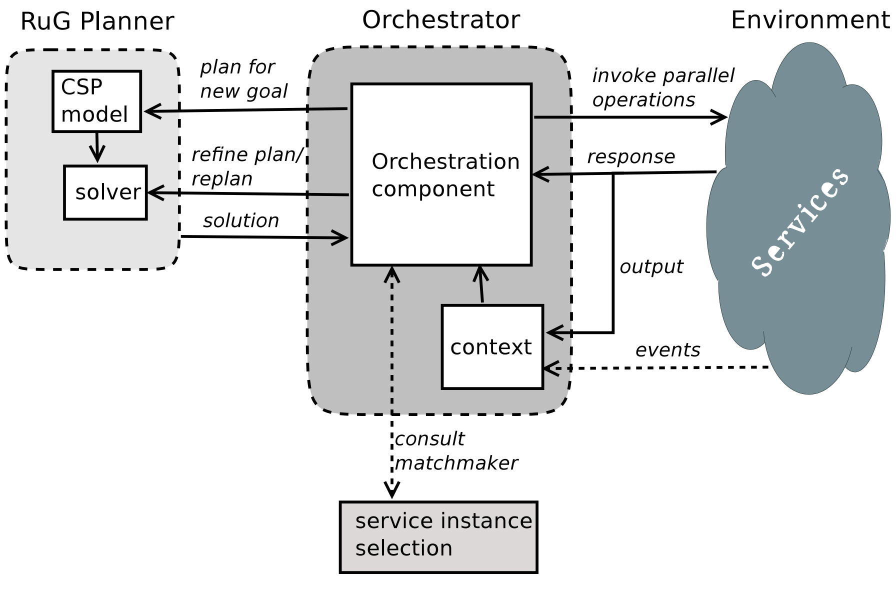


Figure 5.1: Architectural overview of the orchestrator and the basic interactions which enable the interchange between planning, monitoring and execution.

Dashed arrows in Figure 5.1 correspond to interactions that are only available under certain assumptions. Change events can be received asynchronously if this is supported by some publish/subscribe mechanism, such as the one provided by OSGi-UPnP presented in Chapter 4. The instantiation of an action to the operation offered by a specific service provider is taken care by some external component, which is responsible for discovering and selecting the appropriate service instances. The process of service discovery and selection is an interesting and difficult problem by itself, e.g., see [Skoutas et al., 2008; Pilioura and Tsalgatidou, 2009], and is out of the scope of the current thesis.

5.2 Execution-time transition system and orchestration path

Recalling Definition 4, the STS Σ has to be extended in order to capture observations and external events. For simplicity reasons, we make some simplifying assumptions about the situation that this extended STS models: (i) if an action is applied, all of its effects take place as prescribed (ii) observations and events refer to disjoint sets of variables, and (iii) all observations corresponding to a set of actions are retrieved timely, i.e., before the application of the next sets of actions in a plan. Failed actions and byzantine behaviors can be modeled indirectly, by introducing events which assign certain variables after the application of some action. We show that for such a model, the orchestration algorithm can be trapped in dead-ends.

Definition 10 (Execution-level State Transition System)**.** An execution-level state transition system based on a planning domain $\mathcal{PD}' = \langle V, Par, Act \rangle$ (where $V = Var \cup Kb \cup Cv \cup Rv$) is a tuple $\Sigma_e = \langle S, Act, Ev, Obv, \zeta \rangle$, where:

- S is a set of states.
- Act is a set of actions.
- Ev is a set of events. An event is an assignment to some variable ($var := val$), where $var \in Var$ does not participate in any observational effect, and $val \in D^{var_i}$.
- Obv is a set of observations. An observation is an assignment to some response variable ($var\_response =: val$), where $var\_response \in Rv$ and $val \in D^{var}$.
- $\zeta : S \times \wp(Act) \times \wp(Obv) \times \wp(Ev) \to \wp(S)$ is the execution-level state transition function $\zeta(s, A, O, E) = \{\delta(s', O, E) \mid \forall s' \in \gamma(s, A)\}$, where $A \subset Act$, $O \subset Obv$, $E \subset Ev$, and:
    - all assignments in E and O refer to different variables.

- γ is defined in Definition 4.
- $\delta : S \times \wp(Obv) \times \wp(Ev) \rightarrow S$ is a function which updates a state$s$ by applying the assignments in E and O.
- For every *var* which appears in a sensing effect $sense(var)$ of some $a_i \in A$, *var_response* is part of some observation $o_i \in O$.

Generalizing on sets of states S, we define: $\hat{Z}(S, A, O, E) = \bigcup_{s \in S} \zeta(s, A, O, E)$.

Definition 11 (Orchestration Problem)**.** An *orchestration problem* is a triple $OP = \langle \Sigma_e, S_0, g \rangle$, where Σ_e is an execution-level state transition system, S_0 is the set of all states that satisfy a conjunction of propositions $\bigwedge_i prop\_init\_i$, and g is a goal as specified in Section 3.4.1.

Definition 12 (Orchestration)**.** An *orchestration* is a sequence of triples of sets of actions, events and observations
$orch = \langle (A_0, O_0, E_0), \ldots, (A_{k-1}, O_{k-1}, E_{k-1}) \rangle$, and a sequence *inPars* of assignment relations $inPars_i$ as defined in Definition 6 for each A_i that appears in *orch*.

In an orchestration, the sequence of events and observations is uncontrollable, and the sequence of actions and input parameters is selected by the planner. We call the sequence of actions $\langle A_0, \ldots, A_{k-1} \rangle$ in *orch* along with *inPars* the execution-level plan π_e. Similarly to Section 3.5, we extend the $\hat{Z}$ function to capture the sequence of set of states that are brought forth by *orch* and *inPars*, starting from S_0. Given an orchestration

$$orch = \langle (A_0, O_0, E_0), \ldots, (A_{k-1}, O_{k-1}, E_{k-1}) \rangle,$$

we use the notation:
$Z(S) = \hat{Z}(S[inPars_0], A_0, O_0, E_0)$, $Z^2(S) = \hat{Z}(Z(S)[inPars_1], A_1, O_1, E_1)$ etc. Thus, an orchestration comprising π_e induces a sequence of state sets

$$\tilde{S_e} = \langle S_0, Z(S_0), Z^2(S_0), \ldots, Z^k(S_0) \rangle.$$

We call $\tilde{S_e}$ the *orchestration path* or run.

An orchestration path $\tilde{S_e} = \langle S_0, Z(S_0), \ldots, Z^k(S_0) \rangle$ is a solution to the orchestration problem $OP = \langle \Sigma_e, S_0, g \rangle$, and we write $\tilde{S_e} \models g$, if $\tilde{S_e}$ satisfies the properties described in Section 3.5. We say that an execution-level plan π_e is a *weak* or *optimistic* solution to the orchestration problem OP, if there is some sequence $\{(O_0, E_0), \ldots, (O_{k-1}, E_{k-1})\}$ where $E_0 = \ldots = E_{k-1} = \emptyset$, which leads to an orchestration path that is a solution. That is, if no events occur during the orchestration and there is some convenient assignment to response variables that satisfies the goal. We say that π_e is a *strong* plan, if it leads to an orchestration path that is a solution for any sequence $\{(O_0, E_0), \ldots, (O_{k-1}, E_{k-1})\}$. Since the sequence of events and observations is unknown at planning time, we cannot say whether a plan is a solution or not before all transitions actually take place. Strong plans, i.e., plans that are a solution no matter how the execution behaves, do not exist in the systems we are interested in, since any event may occur at any transition point.

Due to the uncontrollable nature of events, dead-ends cannot be avoided, i.e., the orchestration may bring the world to a state from which the goal is no longer satisfiable. Given a partially executed plan $\{A_0, \ldots, A_{i-1}\}$, considering the current set of states S_i, an event e_i may lead to a set of states $S'_i = \{\delta(s, \emptyset, \{e_i\}) \mid \forall s \in S_i\}$, from which no plan $\{A_i, \ldots, A_{k-1}\}$ that is an optimistic solution to the problem $OP = \langle \Sigma_e, S'_i, g \rangle$ can be computed. In Section 5.3, we describe the practical aspects of an orchestration algorithm, which constructs an execution plan incrementally, by taking the first set of actions of each offline optimistic plan computed by the RuG planner. Each optimistic plan is computed considering the states that result from the application of the δ function at each step. At each revision step, the planner considers a new CSP instance following from the current version of the knowledge base, which incorporates the information included in the latest sets of observations and events.

5.3 Main policies of the orchestration algorithm

In the most common scenarios concerning marketplaces of services on the public Web or in inter- and intra-corporate domains, the planning agent communicates with the execution environment in a synchronous manner, through a sequence of requests and responses that follow from the plan steps. This means that the planning agent has no other way to get informed about any changes that occur in the environment except by resorting to explicit service invocations. Due to this situation, the planner will comprehend that some actions it scheduled for invocation are no longer feasible only after attempting to invoke them. E.g., after a hotel room that fits the user criteria has been found, there is a chance that this becomes unavailable before the plan proceeds to the booking process. The unavailability will only be realized by the planner when the response from the booking service is received, indicating that the room has already been reserved by some external agent. Therefore, in such domains, the reasonable information persistence assumption is made, i.e., it is taken for granted that the knowledge collected by the actions at runtime remains valid till the end of the plan's execution. This implies that activities of external agents that also affect the environmental state are assumed not to interfere with the plan execution, otherwise the plan may end up in undesirable situations. For example, it may proceed in buying some item based on some information about its price that has in the meantime been changed, but this change passed unnoticed. Given these assumptions, the offline plan has to be checked for possible inconsistencies and revised only when new information is sensed, or in case of failures or timeout of service invocations. This is enough under an additional premise that is tacitly made: that services are characterized by clean failures, i.e., during an invocation either all of the action's effects are materialized as modeled in the planning domain, or none of them is (in which case the service responds with a failure or timeouts).

On the other hand, in settings such as the SM4ALL architecture described in Chapter 4 or the Business Process Execution framework presented in Chapter 6 which reacts to interference, the planning agent is asynchronously notified about environmental changes, by subscribing to events it is interested in. For example, in the smart home setting, events are published whenever a service changes its state, and the planner continuously listens to these events. This way, the planning agent is kept up-to-date about the evolution of the environmental state, and can exploit this knowledge to escape undesirable outcomes by timely turning aside from its scheduled route. For example, recalling the scenario about fetching the cold beer (see Section 4.2), if someone closes the storage door while the robot is about to take the beer from the storage shelf, the planning agent has to revise the remaining plan steps to avoid stumbling upon a closed door. Thus, consistency has to be ensured every time a context change due to some external agent is perceived by the planner, and this change may pose a "threat" to the remaining plan actions. However, whenever a planner receives some service variable change event, there is no way to distinguish whether this change is the result of some invocation instructed by the planner itself or by some independent actor. Therefore, violation checks have to be enforced whenever the context changes, either due to the progress of the plan or the activity of exogenous agents. These consecutive inspections for inconsistency are of course more time consuming than having to verify the plan only towards sensing outcomes and failures. However, it prevents unsafe developments due to a particular kind of byzantine behavior, when services report successful completion but in fact behave in a way that contradicts their expected effects. No matter what arbitrary results a service invocation brings about, as long as the asynchronous notification mechanism works correctly and timely, these will be taken into account as part of the new context when planning for the next steps. The assumptions concerning the publish-subscribe mechanism are that change events are not lost and that they are received in the order they occur in the environment (FIFO is respected).

5.3.1 Plan repair vs. replanning

Due to the above observations, it becomes clear that the time spent in consistency checks and plan updates becomes dominant in the overall planning and execution time till the goal is satisfied (see e.g., the evaluation results in [Kaldeli et al., 2011]). Even in the most common case when some output is being sensed, the time for instantiating subsequent input arguments which depend on this newly acquired information and inspecting whether this leads to some conflict may be considerable, especially since the domain includes numeric relations. One way to perform the necessary plan updates is to completely disregard the previous solution-plan, and perform replanning from scratch (this is the approach adopted in [Kaldeli et al., 2011]). Another way is to try to reuse parts of the existing plan as the building blocks for constructing a new plan, as in the strategies adopted e.g., by [Fox et al., 2006; Wallace et al., 2009; van der Krogt and de Weerdt, 2005]. However, under certain circumstances the effort spent on modifying a plan can be larger than the effort required to generate a new one [Nebel and Koehler, 1995]. For example, if the context change is such that a drastically different plan is required, then investing too much time in adapting the previous plan is not a good strategy. It should be noted that in the scenarios we are interested in, maintaining minimal perturbation or plan stability [Fox et al., 2006] of the original plan is not a concern per se. We are rather interested in computing good quality plans in short time from the current state onwards. Depending on the situation, namely the domain structure and goal in combination with the type of context change, sticking to the existing plan structure may offer bad guidance.

In the orchestration approach adopted herein, we try to establish a middle-ground for the tradeoff between investing too much effort in attempting to adjust the current plan suffix, and directly proceeding into computing a new plan from scratch. If the plan revision process takes too long, this is probably an indication that the new situation calls for a plan that looks quite different from the original one, and should therefore give up in favor of replanning from scratch (see example in Section 5.5.1). We therefore try within some time limit, usually a fraction of the time required to compute the original plan, to perform some fast search for refining the plan. In terms of the CSP representation, plan repair amounts to dealing with the dynamic CSP problem, where a CSP is subject to a sequence of alterations, i.e., additions and removals of value assignments and constraints. There are several methods which rely on CSP solution reuse to speed up the task of finding a consistent assignment to the altered CSP, e.g., [van der Krogt and de Weerdt, 2005] which exploits no-good learning. These methods, however, are beneficial under certain assumptions, e.g., when context changes correspond to constraints/value additions and deletions that pertain to a few variables (scope), or when the changes to the CSP are monotonic (relaxation through removal of a constraint or restriction through addition of some constraint). The heuristic reasoning approach based on reusing information about certain stable problem features presented in [Wallace et al., 2009; Wallace and Grimes, 2010] may be helpful for our purposes, and it would be interesting to test it in the future.

The refining process adopted herein attempts to construct within some time limit a new plan by adding extra actions at the beginning, the end or in parallel with actions of the existing plan, and allows input parameters of the actions in the current plan to take different values. This way, output collected at the last step of execution is taken into account, so that input arguments to subsequent actions which depend on that output can be instantiated accordingly in the updated plan. At the CSP level, this approach amounts to constructing a partial assignment consisting of the action variables participating in the original plan, while leaving the rest of the variables to be assigned by the search strategy. Performing un-refinement, i.e., determining certain actions in the original plan as being redundant or even hinder the goal fulfillment given the new initial state, can be particularly difficult and time-consuming (repeatedly reserving old assignments can lead to tremendous thrashing [Wallace et al., 2009]), and therefore we directly resort to replanning from scratch if no augmented plan can be found.

Every time a bundle of concurrent actions in the current plan is executed, and all respective responses are received or some average expected response time elapses, it may be necessary to check whether the remainder of the plan is still valid under the new context. Consistency checks are performed at every step if a notification mechanism is available, giving the opportunity to the planning agent to compare the expected planning state with the actual state of the world as delivered by the latest change events. In such a case, the context encapsulates all the world-altering results of the services invoked by the planning agent independently of whether these are in conformance with the expected effects or not, as well as the world-altering behavior of probable exogenous agents. Consistency inspection is very quick, since it amounts to passing to the solver a complete assignment. After the phase of the parallel execution of some actions at step i completes, the solver is passed a full assignment to variables and parameters, which is the same as the assignment corresponding to the current plan, except that variables at state $i + 1$ are assigned the values delivered by the current environmental state. If there is no notification mechanism, then all world-altering effects are materialized as prescribed in the planning domain, and no consistency check is necessary. In such a case, validity has to be checked only when new information is sensed.

Regarding the new information accumulated by possible sensed outputs, this is incorporated into the knowledge base *knowlBase* (see Section 3.3.2). At each plan revision, the respective virtual KB actions are extracted as described in Section 3.3.2, and the respective constraints are added to the CSP, after removing the ones referring to the obsolete knowledge base. This way, predicates are modeled indirectly by keeping the history of persistent information collected so far for different input parameters. The constraints induced by the knowledge base ensure that the refinement process leads to the appropriate output-to-input assignments, as instructed by the goal (see example in Section 5.5.1).

If the plan suffix is found to be invalid with respect to the current environmental state, then an attempt to extend the plan is made. More specifically, let us consider a plan $\pi = \langle A'_0, A'_1, \ldots, A'_{k-1} \rangle$ (see Definition 6), and define $\hat{\pi} = \langle A_0, A_1, \ldots, A_{n-1} \rangle$, $n \leq k$, as the sequence of non-empty action sets in π, respecting order. Assuming that k is always selected to be quite larger than the maximum number of plan steps (see Section 3.3), after the phase of parallel execution of A_0 completes, and depending on the collected outcomes, $\hat{\pi}$ can be augmented by adding extra actions before, after or in parallel with the actions in the suffix $\langle A_1, \ldots, A_{n-1} \rangle$. The refining process shifts the plan suffix leaving $d = \lceil (k-n)/2 \rceil$ "empty" places before it, and the rest after it. To do so, it constructs a *partial assignment* at the solver level: $\{a[d+i-1] = 1 \mid \forall a \in A_i,\ 1 \leq i < n\}$, where $a[j]$ is the CSP-level action variable representing a at state j. Input parameters are left open to be assigned by the solver, depending on the updated initial state, which includes the most up-to-date variables about the world. Any new observation (value of some *var_response*) returned after the completion of A_0 is added to *knowlBase*, predicated on the specific input parameter values with which A_0 were called. This entails a modification to the CSP model, through the addition of the respective virtual KB actions. The updated model and partial solution are passed to the solver, and the search process attempts to find an assignment for the remaining variables. If this process fails to find a valid solution within some limited time, then the instantiations to the action variables are removed, the search retracts to the generic model instance, and a new solution is sought with a partial assignment reflecting only the initial state.

5.3.2 Executing parallel actions and dealing with timeouts

One of the advantages of the RuG planner is that the produced plans are distinguished by a high degree of parallelism. This property can prove highly beneficial at execution time, especially since service compositions are likely to involve many sensing operations that can be performed in parallel. Moreover, even if only a subset of a bundle of concurrent actions complete successfully and within some time frame, this may be enough to enable the invocation of subsequent actions in the plan. This is an eager and optimistic strategy based on the assumption that fulfilling part of the goal is desirable, even if the goal as a whole is not satisfiable. Since many services are characterized by long response times, it would be inefficient to suspend plan execution entirely while awaiting the results of slow or problematic services. Only actions whose preconditions are activated by effects that have been materialized will be scheduled for invocation at the next step, and only given that this is not against the goal specifications. It is one thing for a user to wish to purchase a CD *and* a book, and another to state that he wants to reserve a hotel room at some place *under the condition* that he has found a way to reach this place. In the first case, the user would be probably pleased if he managed to purchase at least one of the desired items, while in the second place he would be reluctant to pay for the hotel room without being able to arrive at the place.

Actions which according to the domain and the goal depend on operations that are still pending are postponed till the effects of the slow operations on which these depend have been substantiated. If it turns out that there is no way (through invoking alternative service providers or by pursuing a different plan) to achieve the effects which are necessary for proceeding to the reliant actions, the orchestrator returns with an infeasibility notification, but the tasks that are independent of these unattainable effects have already been fulfilled. Such a behavior of eager execution is similar in spirit with the ENQUIRER algorithm presented in [Kuter et al., 2005; Au et al., 2005], although the latter refers exclusively to long-lasting observational queries, which are external to the plan (the HTN-based algorithm used in this approach does not *plan* for knowledge gathering), and relies on the assumption that the information-gathering and execution task is disconnected from planning, which is only about altering the environment. The orchestration algorithm presented herein, on the other hand, puts no restriction on whether the effects of the pending service interfere with the rest of the plan or not.

This eager-to-execute policy works in the following way. Considering a sequence $\hat{\pi} = \{A_0, A_1, \ldots, A_{n-1}\}$, all actions in A_i are invoked in parallel, by generating an equal number of concurrent *futures* , i.e., containers which act as proxies for the yet unknown result of the respective action (see Section 5.4 for implementation details). The futures complete either when a response is received (indicating success or failure) or when some predefined timeout period expires. The timeout reflects some short delay within which an average, reasonably fast service is expected to respond. Services which justifiably need longer time for searching or processing data, are kept in a list of pending actors until they respond successfully or they expire, i.e., the respective expected long response time passes. In case an asynchronous mechanism for receiving notifications about environmental changes is available, the assumption is that this mechanism is reliable and timely, so that notifications regarding the effects of the completed futures are received within some milliseconds after the short timeout period.

After the short timeout elapses, the collected information and context changes are processed to decide on plan refinement or replanning from scratch. The updated plan (refined or new) is computed starting from an initial state that reflects the new context, and *assumes* the successful completion of any pending actions. Thus, considering the invocation of the parallel actions A_i at state i, and some actions $A_p \subseteq A_i$ which do not respond within the short time limit, the new initial state is formed by assigning all variables that participate in the effects of A_p to the values they have at the solver state $i + 1$. The values of these variables at state i are stored for bookkeeping so that they can be later recovered in case of a failure response or timeout. An action a which is part of the updated plan is executed only if *precond*(a) does not include any variable which is part of the effects *pendEffects* of any of the pending actions. Moreover, if (i) the goal comprises a goal of type $g1$ `under_condition` $g2$, (ii) some variable in *precond*(a) or *effects*(a) is involved in any of the propositions within $g1$, and (iii) some variable which is part of *pendEffects* appears in $g2$, then a is prevented from being executed. In such a case, the orchestration process waits until the respective pending action either responds or expires, i.e., the expected delay time elapses. These restrictions for allowing an action invocation are overly strict, and may end up putting some actions on hold unnecessarily. For example, if an action's a preconditions share some common variable with the effects of some pending action pa, a's invocation will be suspended until pa expires or responds, no matter if these preconditions are actually satisfied independently of the outcome of pa.

Expiration is interpreted as an indication of erroneous behavior and can be dealt with as described in Section 5.3.3. Whenever a pending action expires, its assumed effects which had been incorporated in the initial planning state have to be retracted. This is done by assigning the variables which participate in its effects to the stored values that correspond to the state before the action's invocation. This way, any future plans will be computed as if the pending had failed. If however in the meantime the variables involved in the expired action's effects have been modified by some other actor, these changes will be overwritten by the stored values, and the new plans will rely on an obsolete state of the world. To deal with this issue, whenever a change event concerning a variable that participates in a pending action effects is received, the stored values for retracting are updated accordingly.

5.3.3 Dealing with erratic behaviors, constraint violations, and persistent information

Erroneous responses and expirations are handled depending on the type of the faulty service, the availability of alternative service implementations, and on the severity of reported failure if some response signifying a failure is returned. Accordingly, a second invocation attempt may be made, some different service provider may be looked for, or an alternative plans which can lead to the same results may be computed. In many domains, a certain functionality modeled by some planning action may be realized by more than one service providers, e.g., different weather information APIs, flights search and booking services etc. In such cases, a planning action corresponds to some "abstract" service or service *type*, which can be translated to different concrete service operations at execution time. Thus, if some erroneous behavior is observed regarding a specific physical service instance, there may be some alternative concrete service that matches the same logical action. We assume that this matchmaking process is undertaken by some special-purpose discovery and selection component, e.g., [Skoutas et al., 2008; Pilioura and Tsalgatidou, 2009], which returns the set of functionally equivalent services and selects the next appropriate instance to invoke according to some criteria. These criteria may consider Quality of Service metrics, or also take into account user-specific preferences, and is not the focus of this work.

In cases where some sensed output leads to a constraint violation, whether it is advisable to sample some other concrete instance, give it a try with the same provider or make a functionally different choice at the planning level depends on the nature of the service. For services whose output may differ depending on the selected provider, such as stores returning the availability or price of a requested item, it makes sense to try alternative instances. This is not the case for services that provide information that is not instance-specific, such as the weather, map etc.

If the received faulty response indicates a permanent failure, then there is no use in trying to invoke the same service instance again. The respective service implementation entry in the list of candidate services is marked accordingly, so that the specific instance is restrained from future selections. If no alternative implementation for the same action can be found, a constraint which forbids the action in question to be chosen by subsequent plans is added to the CSP. Depending on the policy and the type of the service, the ban may concern the action in general, i.e., for all $0 \leq i < k$, $a[i] = 0$, or the action in combination with the input parameters values $\overline{v_p}$ it was invoked, i.e., for all $0 \leq i < k$, $\bigwedge_{p \in in(a)} p[i] = v_p \Rightarrow a[i] = \mathit{0}$. Timeouts that surpass the expected delay for a given service are treated as a symptom of a service being in problematic state.

Byzantine services which indicate successful completion without delivering the expected results can be indirectly tolerated without threatening the consistency of the plan, if the orchestrator is consistently and timely informed about the actual state of the world. Consistency checks are performed on the basis of the latest change events, and thus, every time the preconditions of the next actions are checked towards the actual and not the expected environmental state. This way, for example, before trying to move through some door, the orchestrator will wait until receiving the change event that the door has been opened. However, spotting *which* service is the abnormal one, in order to take proper action and isolate it from future steps, is a much more difficult task. Since it is impossible to identify the actor who invoked a service operation, it remains unclear whether some inspected world-altering effect is the result of invoking a faulty service or of the activity of some external actor. Separate dedicated monitoring techniques are required to decide whether a service is byzantine or not, by inspecting the behavior of all services and infer unusual patterns, as e.g., in [Murugan and Ramachandran, 2012]. If the orchestrator is not notified that a service has been compromised so as to forbid it from subsequent plans, it may keep on invoking the service, as long as it receives a reply that indicates success. To prevent the orchestrator from repeating invocations to such malicious services, an upper limit is set to the number of times that a certain operation can be consecutively called with the same input for the same planning state. This way, consecutive calls of healthy services instructed by the plan itself are allowed, as far as consistency checks return successfully. This is the case as long as we are not dealing with services which are expected to return different output each time they are called (and thus should be justifiably included in subsequent plans until they provide the right result). If there is some external actor which is responsible for the repeated executions (e.g., someone consecutively turns off some light right after the orchestrator turns it on), then banning the service perceived wrongly as byzantine is probably a good idea in that case as well.

Undesirable situations in the general case can be effectively resolved only if actions with potentially severe world-altering effects, which e.g., involve a payment, are reversible. It may be the case that no solution can be found from the current execution stage, because some actions which have been performed as part of a previous plan or by some exogenous agent have brought the world to a state from which the goal is no longer satisfiable. Such situations can only result due to environmental uncertainty and evolution, and not due to incomplete offline search, which chose to follow some wrong path. If there is access to undoing activities, then world-altering actions which have been executed as part of the plan can be retracted one by one, to seek if the goal is satisfiable from prior states.

5.3.4 The orchestration algorithm

Algorithm 2 summarizes the behavior of the orchestrator, which encompasses the aforementioned policies. The orchestrator is realized as an actor [Greif, 1975; Hewitt et al., 1973], i.e., an object which seamlessly reacts in a concurrent manner to messages it receives asynchronously. The message newModel($domain$) is related to the construction of a new CSP model which encodes the planning domain $domain$ as described in the translation process in Section 3.3. The resulting constraints are propagated, and this generic world propagated instance is stored as a starting point for all subsequent solving requests (as long as no service with new functionalities is installed, in which case the model has to be re-constructed). Each time a request for satisfying a new goal is issued (newPlan($goal$)), the constraints modeling the goal are added to the constraint network (after removing any constraints modeling previously handled goals), and the CSP is propagated and stored as the goal-specific world instance from which search will start in all refinement and replanning attempts, till the goal is satisfied or no solution can be found.

Other messages concern the receipt of some change event , an indication that some service type has become unavailable, and the appearance of a new service type (contextChange(var, val), removeAction(a), and addAction(a) respectively). In the latter case, if the description of the respective action a already exists in the planning domain, i.e., the constraints which represent its preconditions and effects are part of the CSP, then it is enough to remove the constraint that had banned it at the solver level. Otherwise, if the new service type offers new functionalities, which have not been encountered before, the addition cannot take place in a dynamic manner: the whole CSP has to be reconstructed, so that the state variables and frame axioms associated with the new action are taken into account.

The message monitor($plan, s_i$) refers to the core part of the orchestrating process. The pseudocode pieces together and codifies all the steps of parallel execution through futures, plan refinement or replanning, handling of failures and timeouts, and selection of alternative physical services that have been described in detail in Section 5.3. The algorithm represents the most general case, i.e., it considers asynchronous context changes, accommodates for a particular kind of byzantine behavior, and maintains an evolving set of alternative service instances for the same action. The pieces of code which are within curly brackets ({}) indicate critical sections. Since all concurrent futures work on the same CSP and context, only one such entity at a time is allowed to access the CSP or context, and any other critical requests from other actors are suspended until the lock on the respective object is released.

Algorithm 2 Orchestrating Actor: asynchronous context changes, concurrent execution and continual refinement of plan or replan depending on latest context

```
function RECEIVE
  case newModel($domain$):
    FORM_CSP($domain$)
  case contextChange($var$, $val$):
    {UPDATE_CONTEXT($var$, $val$)}
    if $var$ is part of the effects of some $f \in$ pendingActors then
      BOOKKEEPVALUES($f$, $var$, $val$)
  case removeAction($a$):
    Add constraint $a[i] = 0$ for all $0 \leq i < k$ to model level
  case addAction($a$):
    if $a$ exists in CSP then
      Remove constraints $a[i] = 0$ for all $0 \leq i < k$
    else
      Create new CSP model including new action description $a$
  case newPlan($goal$):
    SET_GOAL_CONSTRAINT($goal$)
    return SOLVE_CSP
  case monitor($plan$, $s$):
    $A$ := GETNEXT_PARALLEL_ACTS($plan$, $s$)
    $F$ := EXECUTE_PAR_ACTS($A$, $s$) // Form futures sequence
    for $f[a, serv] \leftarrow F$ do
      // $f$ concerning $a$ and service instance $serv$
      ON_FUTURE_COMPLETE($f$, $a$, $s$)
    end for
    $F$ onAllComplete
       // short timeout has expired or all futures responded
      CHECK_PLAN($plan$, $s$)
end function

function ON_FUTURE_COMPLETE($f$, $a$, $s$)
  $f$ onComplete      // Future return or timeout: run in parallel
    case success:
      if response contains sensed output then
          {UPDATE_CONTEXT($out$)}
          if persistent info then
              {BOOKKEEPBEHAVIOR($f$, $inParams(f, s)$, $out$)}
      if $f \in$ pendingActors then // $f$ with long timeout returned
          Add $f$ to pendingActors
    case short timeout:
        if service($f$) has justifiably longTimeout then
             // $f$ in pendingActors until response or longTimeout expires
            Add $f$ to pendingActors
            // Proceed as if $a$'s effects have been materialized
            {UPDATE_CONTEXT($effectVars(a)[succ(s)]$)}
            BOOKKEEPVALUES($f$, $effectVars(a)[s]$)
    case failure:
        if $f \in$ pendingActors then //$f$ with long timeout returned
            Remove $f$ from pendingActors
        BOOKKEEPBEHAVIOR($f$, $inParams(f, s)$, $failure$)
end function

function CHECK_PLAN($plan$, $s$)
    // Partial assignment at solver level, complete solution
    $updPlan$ := REFINE_PLAN($plan$, $s$)
    if $updPlan \neq \varnothing$ then
        send monitor($updPlan$, $0$)
    else    // Refinement failed, replan from scratch
        $newPlan$ :=REPLAN
        if $newPlan \neq \varnothing$ then
            send monitor($newPlan$, $0$)
        else
            notify client "Goal is not satisfiable"
end function
```

```
function REFINE_PLAN($plan, s$)
    // Instantiation of known variables at the solver level
    COPY_CONTEXT_TO_INIT_STATE
    $planSuffix$ := CONSISTENT($plan, s$)
    if empty($planSuffix$) then
         // CSP-level action variables in plan suffix after $s$ are set to 1
         // Input parameters and rest of action variables left unassigned
        ASSIGN_PARTIAL_SOLUTION($plan, s$)
        return SOLVE_CSP
    else
        return $planSuffix$
end function

function EXECUTE_PAR_ACTS($A, s$)      // Returns a set of futures $F$
    for $pf \leftarrow$ pendingActors do
        if $longTimeout(pf)$ has expired then
            Remove $pf$ from pendingActors
            UNDO_EFFECTS($pf$)
            CHECK_PLAN($plan, s$); return
        else
            $F$.add($pf$)
    end for
    for $a \leftarrow A$ do
        if $a$ not dependent on any other action in pendingActors AND
         not scheduled before at $s$ then
            $serv$ := GET_NEXT_INSTANCE($a, inParams(a, s)$)
            $F$.add(EXECUTE($serv, inParams(a, s)$))
        if $a$ has been scheduled before at $s$ then
             // Detected some malicious action
            BOOKKEEPBEHAVIOR($f, inParams(f, s), failure$)
    end for
    return $F$
end function
```

5.4 Implementation

The framework for interleaving planning, monitoring and execution has been implemented by using the *akka* library in Scala[1], which builds upon the theory of actor models [Greif, 1975; Hewitt et al., 1973]. Akka provides the means for building scalable and fault-tolerant concurrent applications at a high abstraction level, based on an asynchronous, non-blocking and lightweight event-driven programming model. The orchestration component, the RuG planner and the service environment correspond to different actors, which are independent entities that operate concurrently and communicate with each other by asynchronously exchanging messages. Each component-actor may supervise a set of smaller actors-children, each of which represents some lower-level constituent entity and is responsible for certain functions that are assigned to it.

All services that participate in the service environment are modeled as individual actors, which are overseen by the parent-environment actor. The parent actor delegates the requests for service calls it receives from the orchestration component to the respective child-actor, which simulates the requested service. The service-level actor processes the message and reacts accordingly, depending on the behavior that we wish to simulate, e.g., by replying with a message that includes some output values or indicates some failure, by raising an exception, taking too long to respond or sending no message at all. Each child is treated separately, and different fault handling directives (e.g., stop, resume, restart, escalate) can be implemented by the parent actor, depending on the type of failure and the failing service actor. Service-level actors can be connected to real services, e.g., some OSGi bundle realizing some intelligent device, or the API of some actual Web Service (see [Westra, 2010]).

An important tool for dealing with concurrency is futures. Futures are used to retrieve the results of parallel invocations in a non-blocking way, as described in Algorithm 2. Each future's lifecycle is treated by some separate callback, which amounts to some generated special-purpose actor that waits and reacts to the future's completion. These callbacks, which may be executed in any order or in parallel depending on the service behavior, entail certain modifications to the CSP, which is shared among them. Atomicity on the operations on the CSP is ensured through the Scala STM (Software Transactional Memory) library[2], which takes care of coordinating access to shared data from concurrent threads.

[1] `akka.io`

[2] `nbronson.github.com/scala-stm`

5.5 Running examples

In the followings, we present how the orchestrating algorithm operates given some simulated service behaviors in different planning domains and environmental circumstances. All scenarios presented in the followings were performed on a Core i5 @2.50Ghz computer with 4GB of RAM, running Ubuntu 12.10.

5.5.1 Entertainment WS marketplace

Let us recall the example for attending a concert presented in Section 3.7.1, and show a running instance of the orchestrating algorithm, given the initial plan. In such a setting, the only source of information about the context changes are the responses of the service invocations, and it can be assumed that no external actor interferes with the plan. The scenario involves many sensing actions (about the place, date, weather etc.) whose output has to be passed as input to subsequent actions, and thus the plan refinement process has to repeatedly take care of the appropriate instantiation of input parameters. For example, when the information about the upcoming band's performance is acquired, the input parameters of all actions which depend on the concert's date and place have to be instantiated to the outputs of the "getNextEvent" action (instead of the random convenient values they were assigned offline).

A run using real WSs has been presented in [Kaldeli et al., 2011], with an orchestration algorithm that plans from scratch at every step. The run simulates a situation where the place of the first upcoming concert turns out to be too far. This information, as well as the information gathered by the other knowledge-providing actions, i.e., the distance, weather etc., is added to *knowlBase*. Due to violation of the goal constraints caused by the distance variable, an attempt for refinement is made. The respective virtual KB actions are constructed, and the CSP is updated accordingly. At the initial state, all variables referring to the weather, distance etc., are unknown, since they are not included in *varKnown* (their knowledge is predicated on certain input parameter values as stated in *knowlBase*).

Since retrieving the facts included in the knowledge base does not lead to a solution, the planner adds the following sensing actions to the plan: first the retrieval of the next performance, and then the respective actions for sensing the new distance, weather, hotel list etc. Refinement is performed once more, after the information about the next performance is instantiated. Then the information about the weather, calendar availability, distance and hotel rooms regarding the new whereabouts is collected in parallel. Since the new information does not violate the goal requirements, the existing plan is found to be consistent, and a ticket is booked. However, when proceeding to hotel room reservation, the hotel provider first in the list (the default order is by increasing price) returns a permanent failure. The policy for dealing with a *bookHotel*s failure response in this case is to forbid it to be called again with the same input arguments. Thus, the refinement process enforces the investigation of the next provider in the list of available hotels.

The sequence of steps taken by the orchestrator when the monitor(*initialPlan*, 0) message is received are summarized in the followings:

getEventsList(Neutral Milk Hotel) $\rightsquigarrow$ *evList*=known
getNextEvent $\rightsquigarrow$ eventDate=2012-02-05, *eventPlace*=Brussels
Refine plan (assignment of outputs to inputs of actions in plan suffix)
In parallel: {
checkCalendarAvail(2012-02-05) $\rightsquigarrow$ *calendarAvail*=true
getTemperature(Brussels, 05 Feb 2012) $\rightsquigarrow$ *temperature*=11
getDistance(Groningen, Brussels) $\rightsquigarrow$ *distance*=360
getAvailHotels(2012-02-05, defaultPl, 1, single) $\rightsquigarrow$ *hList*=known
}
Sensed value 360 for distance violates constraints, *Refine plan*
Updated plan found by adding extra actions to plan suffix
getNextEvent $\rightsquigarrow$ *eventDate*=2012-02-08, *eventPlace*=Amsterdam
Refine plan (information regarding date and place has changed)
In parallel: {
getDistance(Groningen, Amsterdam) $\rightsquigarrow$ *distance*=182
checkCalendarAvail(2012-02-08) $\rightsquigarrow$ *calendarAvail*=true
getTemperature(Groningen, 08 Feb 2012) $\rightsquigarrow$ *temperature*=11
getAvailHotels(2012-02-08, Amsterdam, 1, single) $\rightsquigarrow$ *hList*=known
}
bookConcertTicket(Neutral Milk Hotel, 2012-02-08) $\rightsquigarrow$ **ok**
getNextHotelInfo $\rightsquigarrow$ *hotelWS*=Chancellor Hotel, *hotelPrice*=60
Refine plan (assignment of outputs to input parameters of actions in plan suffix)
bookHotel(Chancellor Hotel, 2012-02-08, 1, single) $\rightsquigarrow$ **null**
A failure occurred, *Refine plan* (after banning bookHotel with same input parameters)
getNextHotelInfo $\rightsquigarrow$ *hotelWS*=Fairmont Hotel, *hotelPrice*=75
Refine plan (assignment of outputs to input parameters of actions in plan suffix)
bookHotel(Fairmont Hotel, 2012-02-08, 1, single) $\rightsquigarrow$ *hBooked*=true

As we see, the plan is refined whenever the response of an action invocation includes some newly sensed output, and the consistency check of the existing plan suffix fails. This happens either because the input parameters of subsequent actions have to be updated, or because the new information violates some constraint. If the response indicates success (**ok**) and the action does not include any knowledge-providing effects, the process proceeds directly to executing the next action(s), assuming that the service indeed behaved as expected. A response which indicates a failure (**null**) also calls for plan refinement. The run includes 6 calls to the solver for plan refinement which take 3.5 sec in total, and 9 consistency checks which require 2.8 sec.

5.5.2 "Moving in grid" scenario

In the followings we investigate a simulation which is designed to combine in a single run of the orchestrator three types of contingencies which occur during execution: an exogenous action which interferes with the plan, a malicious service with byzantine behavior, and an operation which requires a long time to respond. The scenario concerns a robot moving around in a house setting as indicated in Figure 5.2. The rooms of the house are connected through doors, which can be either open, closed or locked. The robot can open a door only if it stands in front of it (on either of the two sides), and only if the door is unlocked (i.e., it cannot unlock doors). Let us also assume that especially for opening the two doors which lead to $R22$ the robot needs to have at hand a special password, which it can retrieve by invoking a slow sensing operation that requires 40 sec to respond. The robot can move as instructed by the action described in Section 3.3.1, that is, between locations which are adjacent to each other (connected through the dashed lines in Figure 5.2), with the additional requirement that if the locations belong to separate rooms, the intermediate door should be open.

At the initial state the robot resides at location $R00\_R01$, and the goal is to reach location $R22\_R21$ at the final state (where RX_RY is the location at room RX and is connected with a location in room RY). All doors are initially closed and unlocked. A possible run of the orchestration, including three different types of contingencies, is presented in Appendix A.1. The initial plan guides the robot to $R22$ through $R01$, $R11$ and $R21$, and instructs calling the sensing action to retrieve the necessary password at the second state of the plan. Since the sensing action takes long to respond, it is checked whether there are any actions in the plan suffix that do not depend on the expected password and can thus be executed. Indeed, the robot can keep on moving until the door leading to $R22$. However, while the robot is still in $R01$, someone locks the door which leads from $R11$ to $R21$. Such a contingency at this state requires a drastic change in the robot's planned route, and cannot be dealt with by just augmenting the plan. Thus the refinement process

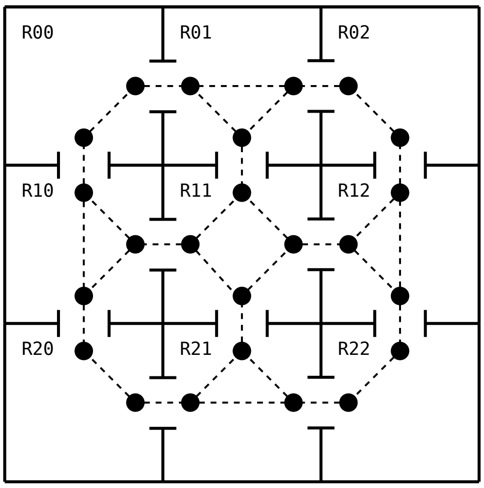


Figure 5.2: Graphical representation of a planning domain modeling the movement in a 3x3 grid. A robot can move between adjacent locations that belong to the same room (connected through lines) and between rooms if the interconnecting door is open.

fails, and replanning from scratch is performed.

The new plan leads the robot through $R02$ and $R12$. Due to bad lack though, it turns out that the door that connects $R02$ and $R12$ behaves in a corrupt way: although the opening operation responds with a success, the door is actually not opened. As a result, an augmented plan which includes a second attempt is computed. The door behaves the same way for a second time, and thus a new refined plan is produced. However, the opening door operation is not invoked again, since already two invocations corresponding to the same planning state have been attempted, and the action is in turn forbidden from any subsequent plans. Since no refined plan can be found given the prohibition of the specific door opening operation, a new plan is computed from scratch. The alternative route directs the robot back to $R01$, then to $R11$ and then to $R12$. However, as the robot stands before the door leading to $R22$, the password sensing action expires (it has not responded within the expected maximum timeout), and is thus treated as a failed action. Since there is no other way to collect the password, it is impossible to satisfy the goal. An alternative policy could be to try re-invoking the expired actions, with the hope that they would provide the required output.

During the above orchestration run, there are 14 validation checks and 4 attempts to refine the plan (including the failed ones), taking 6.6 sec in total, while

3 plans are computed from scratch (including the initial one), taking 17.9 sec altogether. It should be noted that the way the orchestration run actually evolves highly depends on the structure of the generated plans. For example, in the above example, assuming that doors can be opened remotely, independently of the location of the robot, it makes a difference whether the opening of the doors is scheduled in parallel at the beginning of the plan, or delayed till later steps. Although both plans consist of the same number of actions (i.e., are equally "good"), it is desirable to push actions as early as possible in the plan, since this way any erroneous services are detected earlier, thus avoiding invocations which prove to be needless or misguiding (e.g., avoid moving on to some door, then detect it is erroneous, and have to go back again). Moreover, operations which are expected to take a long time to complete should be preferably scheduled at an early stage of the plan.

5.6 Empirical evaluation

We now present some scenarios especially designed to demonstrate the behavior of the orchestrator under different "interesting" circumstances.

5.6.1 Refinement towards replanning from scratch

The test cases presented in Table 5.1 seek to investigate the tradeoff between attempting a plan refinement or directly proceeding to planning from scratch. All tests are variations of a planning domain which represents a robot moving between adjacent rooms connected by doors, which can be open, closed or locked. In all tests, the goal is for the robot to move from the top leftmost room to the bottom rightmost room, starting from an initial state where all doors are closed. The numbers 3–5 indicate the dimension of the grid, i.e., refer to a 3x3, 4x4 and a 5x5 grid respectively. In tests annotated by "*" no attempt for plan refinement is made, and all updated plans at every step of the orchestration process are computed by resorting directly to replanning from scratch. The upper time limit that the refinement process may take is set to half the time required by the last planning from scratch invocation.

In all simulations, as the robot proceeds, an external actor repeatedly hinders its route: just before the robot is about to cross an opened door, the troublesome agent locks that door. All operations are simulated so as they successfully respond within 4 sec, and 2 sec is the extra waiting time, within which the change events implied by the invocations are expected to be received. The conservative most-constrained/increasing domain strategy (see Section 3.7) is used for planning from scratch in the 3x3 and 4x4 grid tests, and for the refinement process in all cases. For

Test	Refine time (♯ attempts)	Plan from scratch time (♯ attempts)	Consistency check time (♯ attempts)	Total orchestration time
unlock3	0.9 (3)	0.3 (1)	0.4 (11)	24
unlock3*	−−	1.0 (4)	0.5 (13)	25
unlock4	2.9 (5)	3.3 (1)	3.5 (26)	62
unlock4*	−−	12.8 (6)	3.2 (24)	69
unlock5	6.0 (7)	39.2 (1)	10.3 (24)	102
unlock5*	−−	202.6 (10)	12.4 (33)	327
unlock-notInRoom3	0.8 (3)	0.6 (1)	1.2 (18)	39
unlock-notInRoom3*	−−	2.4 (4)	1.3 (18)	41
unlock-notInRoom4	3.9 (5)	3.2 (1)	4.1 (21)	53
unlock-notInRoom4*	−−	19.5 (6)	5.2 (26)	79
unlock-notInRoom5	13.1 (7)	59.3 (1)	22.8 (33)	158
unlock-notInRoom5*	−−	176.8 (8)	23.0 (33)	268
permanent-lock3	0.2 (3)	0.3 (4)	0.3 (8)	18
permanent-lock3*	−−	0.4 (4)	0.3 (8)	18
permanent-lock4	13.4 (8)	8.1 (9)	5.2 (23)	61
permanent-lock4*	−−	8.3 (9)	4.9 (22)	49
permanent-lock5	81.5 (12)	147.1 (13)	13.8 (27)	317
permanent-lock5*	−−	135.6 (13)	14.3 (31)	243

Table 5.1: Results for different simulated tests where some external actor intervenes with the plan (time in sec). Total orchestration time counts the time elapsed between issuing the goal and its satisfaction or failure. Plan from scratch time includes the time for computing the initial plan.

the tests in the 5x5 grid, the initial plan takes more than 15 minutes to complete with the default search strategy, and therefore a random values assignment approach is employed whenever planning from scratch, with the search restarting every time some increasing node limit is reached. The resulting plans are not always the optimal ones, and may include redundant actions.

In the tests signified by "unlock", the robot can open as well as unlock rooms. Thus, every time the robot unexpectedly encounters a locked door, it has to update its plan by first unlocking and then opening the door targeted by the exogenous agent. In these cases, attempting a plan refinement instead of directly planning from scratch saves a considerable amount of time, at least for the 4x4 and 5x5 grids, as is shown in Table 5.1. As expected, the longer time plan generation takes, the more benefit is gained by employing plan extension, since in these simulations the updated plan should just involve the addition of two more actions at the beginning, which is very fast to compute. The two approaches also lead to slightly different overall sequence of steps, since some redundant actions may be included at times, especially when employing the random value assignment strategy.

To investigate the performance of the orchestration in situations where a more elaborate plan refinement is required, let us assume that the robot can unlock a door only if it resides in some room other than the ones the door connects. This entails that whenever the exogenous agent locks a door that the robot is about to cross, the robot has to first move to some adjacent room, unlock the door, then move back, open it and go on with its route. This situation is reflected by the "unlock-notInRoom" tests in Table 5.1. Also in these tests, resorting to plan refinement leads to considerably better performance. However, in cases where there is no way to augment the plan to tackle the new contextual situation, the time devoted to attempting plan refinement is wasted. This case is simulated by the "permanent-lock" tests, which concern the same planning domain as in "unlock", except that the robot has no capability to unlock doors. Thus, every time the robot has to deal with an unexpectedly locked door, the refinement attempt fails, since a drastically different plan has to be found. In this case, resorting directly to replanning from scratch actually saves time.

5.6.2 Timeout of sensing actions

The following tests simulate situations where some actions take justifiably long time to respond, and aim at demonstrating how different plan structures affect the overall orchestration process. The planning domain concerns again a robot moving between adjacent rooms in a square of increasing dimension. To open a door the robot should know a 3-digits password specific for that door, which it can retrieve by invoking a sensing action from any location. The robot has to move from the uppermost left room to the bottom rightmost one, starting from an initial state where all doors are closed and all passwords are unknown to the robot. In all tests simulating the execution behavior, the password sensing actions take 40 sec to respond, moving actions take 8 sec, opening the door takes 1 sec, and the average waiting time for a bundle of parallel actions to respond is 10 sec.

Depending on which planning states sensing actions are scheduled, the robot may end up waiting for shorter or longer time for some sensing action to respond before being able to open the respective door. Table 5.2 summarizes the results for three different structures of initial plans which are passed for execution and monitoring to the orchestration algorithm. All plans consist of the same number of

Test	Refine time (♯ attempts)	Consistency check time (♯ attempts)	Total orchestration time
pswdOpen3-opt	0.5 (1)	0.6 (7)	76
pswdOpen3-subOpt	0.8 (2)	0.9 (9)	88
pswdOpen3-worst	4.7 (4)	1.1 (11)	121
pswdOpen4-opt	4.1 (1)	10.8 (8)	113
pswdOpen4-subOpt	11.0 (3)	18.3 (13)	155
pswdOpen4-worst	12.1 (6)	13.7 (22)	275
pswdOpen5-opt	12.8 (1)	58.5 (11)	144
pswdOpen5-subOpt	33.5 (3)	60.9 (14)	281
pswdOpen5-worst	63.4 (8)	71.6 (20)	437

Table 5.2: Results for simulated tests with actions that take long to respond for three different plan structures (time in sec). Total orchestration time counts the time elapsed between issuing the goal and its satisfaction or failure.

actions, however the state at which actions are placed varies: "opt" refers to the optimal situation where all the password sensing actions in the plan are concentrated at the first state, "subOpt" to the plan actually generated by the RuG planner by employing the random values assignment with restarts searching strategy, and "worst" to a plan where each password sensing action is scheduled just before the respective door opening. After the invocation of some parallel actions sets, a validation check and probably a refinement attempt is performed when all actions respond or the short timeout expires. Given the validated or updated plan, it is checked whether it is possible to proceed with any plan actions which do not require the knowledge of the specific password. Thus, the robot can move further if possible, while the password is being sensed (this happens with the "subOpt" simulations). Whenever a pending action responds, the orchestration goes on with executing the updated plan suffix. The results in Table 5.2 demonstrate the large gain in time when the actions which take long to respond are scheduled as early as possible in the plan. The larger the domain, i.e., the more slow actions are involved, the larger the difference between the optimal and the worst approach is.

5.7 Discussion

Given the non-determinism following from observations, external events and erroneous or byzantine action behavior, the orchestration algorithm may be trapped in some dead-end. Dead-ends cannot be avoided, because the planner cannot predict the actual state and the evolution of the environment, and during the orchestration any event and observation may occur. If there are alternative offline plans, there is no bias in favor of a specific plan based on some model of the execution-level and environmental behavior. The orchestration algorithm may end up in a dead-end even if one assumes that there are no external events and that an action's execution leads to a set of states which satisfy the propositions entailed by the action's effects. To give an example, let us consider a goal `final`$(var = true)$, and two actions a_2, a_3 having an effect $assign(var, true)$. a_2 has a precondition $v_2 = 1$ and a_3 a precondition $v_3 = true$. Let us also assume that there is an action a_0 with no preconditions and with effects $sense(v_2)$ and $assign(vb, false)$, and some other action a_1 with effect $sense(v_3)$ and precondition $vb = true$. Starting from a situation where v_2, v_3 are unknown and $vb = true$, there are two offline plans (of equal length) that have the potential to satisfy the goal: $\langle a_0, a_2\rangle$, and $\langle a_1, a_3\rangle$. If in the actual world, which is sensed at execution time, $v_2 = false$ and $v_3 = true$, then the execution of the first plan ends up in a dead-end, since after the execution of a_0, a_1 is not applicable anymore. Similar situations can result from problems where actions have either exclusively observational or exclusively world-altering effects.

As has been shown in Section 5.5.2, producing the shortest plan is not enough to lead to the optimal orchestration runs. The plans that are opted for should be the ones which include actions as much in parallel and as early as possible. However, the RuG planner does not always produce the desired structure of plans, and in some cases it even computes suboptimal ones, thus the orchestration process may not follow the shortest run. Moreover, the orchestrator may be trapped into repeating a sequence of steps without managing to reach the goal. This "stuck in a loop" situation may arise in cases where suboptimal plans of a certain pattern are repeatedly produced, or due to the malicious behavior of external actions which may lead a certain execution to a deadlock. To give an example of such a problematic situation, let us assume a robot which wants to move from $R00$ to $R02$ of Figure 5.1, and that the door leading from $R01$ to $R02$ can be opened only if the robot resides in $R00$. If an external actor closes the door leading from $R01$ to $R02$ every time before the robot tries to pass through it, a revised plan instructing the robot to go back to $R00$, reopen the door and move forth again will be repeatedly generated. The orchestrator has no way to identify that it is trapped in a loop due to the consistent behavior of some exogenous actor, so as to make the decision to follow an alternative plan, e.g., reach $R02$ through rooms $R10$, $R11$, and $R12$, with the hope that the doors in that route will not be blocked. A similar deadlock may result in any case in which there is need for replanning when the robot resides in $R01$ and the planner generates a suboptimal plan, which directs the robot first back to $R00$, and then forward to $R01$ and further. To avoid such situations, some randomness should be inserted in cases where the orchestrator repeats the same sequence of actions, and this repetition is not included in the current plan. For example, the orchestrator may try to randomly choose from the set of actions applicable at the current state, with the hope that from that new state the planner will escape the deadlock situation.

It should also be noted that the algorithm's behavior highly depends on the selection of concrete service instances for some given planning-level action. The selection process in the current orchestrating algorithm only addresses simple situations, like trying an alternative provider if some instance returns a failure or timeouts. However, more advanced policies regarding the instance selection process and its implications to the planning level are required to address more tricky situations. For example, in some cases, selecting a specific provider at some early step of the plan may affect later selection possibilities.

Because of the separation between the instance-level selection process and the planning (action) selection level, the orchestrator may not exhaust all possible combinations of concretized syntheses, and thus be unable to find a solution, even if there exists one. In the current implementation, information acquired by some service invocation is incorporated at the planning level (in the form of implication constraints to the CSP), i.e., it is assumed that this information is independent of the service provider selection and refers to the abstract action itself. If at some later stage it turns out that no plan can be found from the state onwards, it should be possible to remove the known facts kept in the *knowlBase* map which are dependent on some specific service instance. This way, the planner can include in the updated plan the same actions with the same input parameters, hoping for some different, more appropriate output. The selection process employed by the orchestrator chooses a different service instance each time that the same knowledge-providing action with the same input is requested in a given orchestration run. However, depending on the circumstances (e.g., for information which is known to change very quickly) it may be desirable to re-invoke the same instance for the same action request. The identification of the right policies depending on the type of the service and the history of the orchestration run is left as future work.

It should be emphasized that the orchestration algorithm is *memoryless*, in the sense that at each stage where a plan refinement or replanning is requested it only considers the current state, but not the *history* of the execution path (sequence of state states) since the orchestration run started. This implies that goals over state trajectories may never be satisfied, and the orchestration process may needlessly re-invoke the same actions or wrongly report that the goal is not satisfiable, because it fails to realize that parts of the goal have been fulfilled in the past. For example, consider Goal 3 from 3.4.1 requesting that a robot should visit all rooms ($\bigwedge_i$`achieve`$(robotLocation = room_i)$). If at some point during the plan's execution a need for plan update appears, then the new plan will re-schedule moving to rooms that have already been visited. Even worse, if in the meantime some doors leading to these already visited rooms have been blocked and pruned from the search space, the orchestrator will fail to find a valid solution. Thus, for certain combinations of goals and circumstances, correctness may be violated. Checking whether a goal holds over the execution path induced by the complete orchestration run is far from straightforward. It requires a separate modeling of the sequence of state sets which has occurred so far independently from actions, which has to be combined with the planning problem.

The architecture and methodologies presented in this chapter is the result of joint work with Nick van Beest, and the process executor presented in Section 6.5.2 is a contribution by Pavel Bulanov.

Chapter 6
Automatic Runtime Business Process Repair

Current organizations are characterized by long-running distributed Business Processes (BPs) involving many different stakeholders [Li et al., 2010; Gomaa et al., 2010]. The application of service orientation in the field of Business Process Management enables the integration of interoperable, local or remote services within a BP, thus realizing complex composite functionalities, while aiming at adaptability and reuse. The application of AI planning techniques as a means to infuse flexibility and and adaptability to the field of Business Process Management Systems (BPMS) is not new, e.g., [Beckstein and Klausner, 1999; Ferreira and Ferreira, 2006; de Leoni et al., 2007]. In this chapter we focus on the problem of process interference, which arises when multiple BPs that share common resources are running simultaneously. In such a dynamic setting, data modifications by external actors and other concurrent processes may lead to inconsistencies and undesirable business outcomes [Xiao and Urban, 2007; van Beest, Szirbik and Wortmann, 2010]. Such outcomes may refer to repetition of activities that have already been fulfilled by some other process (e.g., multiple orders or invoices), proceeding to activities based on obsolete information (e.g., delivering a product to some address that is not valid anymore), or disregarding events that call for compensation activities or the process to hold (e.g., some necessary condition ceases to hold). In [van Beest, Bulanov, Wortmann and Lazovik, 2010], the authors propose a runtime approach to address the problems caused by interference in highly dynamic service-oriented environments. The approach bases on annotating the BP with dependency scopes and triggering predefined intervention processes to recover from inconsistencies that are discovered during execution. However, this involves a lot of manual configuration, which even for simple BPs can become particularly time-consuming and error-prone, especially since one has to ensure that all important intervention cases are correctly taken into account and designed.

The adoption of domain-independent AI planning appears a natural assistant on this limitation. One can view intervention processes as plans, which can be synthesized dynamically on the fly, by combining activities from the BP and available compensation operations, based on how the BP's knowledge about the world evolves during execution, and how this knowledge affects the next tasks in its workflow. The workload of the domain designer can be reduced through the specification of some high-level goals, which describe in a declarative way the desired properties that has to be fulfilled in case of interference. However, in order to enable this automatic computation of intervention processes, one has to enrich the BP with appropriate semantics annotations so that it can be transformed to a planning domain. One of the questions that arise in that process is how the semantics of workflow constructs such as XORs, repeat structures etc. can be captured in terms of preconditions and effects. Following our main requirement for sustaining domain-independence, the idea is to maintain a common repository of atomic service descriptions, which can be reused as the building blocks of different BPs. The particular restrictions that are imposed by the specific control flow of each BP should be inferred automatically, by parsing the syntactic BP specification. Dealing with incomplete knowledge is also essential, since workflows often involve XOR-splits where the condition depends on the value of a variable which is unknown offline, and becomes known only at runtime, after an interaction with the operating environment (e.g., requesting some approval or advice). Moreover, numeric-valued fluents and logical conditions on them are commonly used in BPs. Given these requirements, the RuG planner comes forward as a good candidate for enabling the automatic generation of intervention processes.

6.1 The problem of process interference

Modern private and public organizations are moving from traditional, proprietary and locally managed Business Process Management Systems to BPMSs where more and more tasks are outsourced to third party providers and resources are shared among different stakeholders [Li et al., 2010; Gomaa et al., 2010]. In such a setting, as realized by the emergent paradigms of Service Oriented Architecture and cloud computing, BPs can no longer be considered in isolation, since disregarding their interdependencies with external actors and other processes may lead to inconsistent situations. The situation where data can be simultaneously accessed and modified by different processes that are running in parallel is referred to as *process interference [Xiao and Urban, 2007; van Beest, Bulanov, Wortmann and Lazovik, 2010]*. Process interference occurs far more often than most people realize. BPs are developed under the assumption that data are stable, which is generally not true. In a distributed setting where data are accessed and modified by a number of different parties, changes on important data, that the process relies upon, may pass unnoticed. These changes often yield wrong results, however, because no immediate software error occurs, there is no direct sign that something wrong is happening during the process execution. Therefore, the consequences are often realized only by end customers [van Beest, Szirbik and Wortmann, 2010], by erroneous orders or invoices, customer requests that are never handled, etc. Consider, for example, a business process for approving and delivering a wheelchair for disabled people in the Netherlands. It takes up to 6 weeks from sending the initial request to receiving an actual wheel chair. What if in the meantime the person, which requested the wheel chair, has moved to a different place, e.g., to a care home with nursing support? The original process has to be adjusted, either by forwarding the wheel chair to a new place, or by canceling the request, depending on what is more suitable in the new concrete situation.

Traditional verification techniques for workflow and data-flow (e.g., [Trčka et al., 2009]) are not sufficient for ensuring the correctness of such BPs, since they assume a closed environment where no other process can use a service that affects the data used by that organization. In addition, most work about resolving process interference refers to failing processes or concerns design-time solutions [Xiao and Urban, 2008; Urban et al., 2011]. Consequently, neither of these solutions is suitable for a highly-dynamic and distributed environments. In [van Beest, Bulanov, Wortmann and Lazovik, 2010; van Beest et al., 2012], a runtime mechanism is proposed, where vulnerable parts of the process are monitored in order to manage interferences by employing intervention processes. *Dependency scopes (DS)* are used to specify *critical sections* of the BP, whose correct execution relies on the accuracy of a *volatile* process variable, i.e., a variable that can be changed externally during the execution of the process. If a volatile variable is modified by some exogenous factor during execution of the activities in the respective DS, an *intervention process (IP)* is triggered, with the purpose of resolving the potential execution problems stemming from this change event.

By using DSs, testing for unforeseen data interactions at each activity can be avoided. As a result, the process designer does not need to know all potential process interactions in advance. However, a significant amount of manual specification of the intervention patterns is required, since the appropriate IPs may differ considerably depending on the current execution state at which modification of a volatile variable occurred. For complex processes with numerous activities, it is very difficult and time-consuming to define IPs at design-time, as the amount of potential IPs may be particularly high. In addition to that, it is difficult to ensure that *all* important intervention cases are taken into account. Moreover, as the same BP may be deployed and used by more than one organization, different intervention processes have to be specified for each potential interference case at each organization.

The workload due to extensive manual configuration can be significantly reduced by employing the RuG planner to automate the task of IP generation. In that way, the manual work required by the domain designer is reduced to the specification of a high-level goal, which describes in a declarative way the desired consistent state that has to be reached in case of interference. To realize such a level of automation, additional semantic annotations are required, which capture the functional aspects of the activities participating in the business domain in terms of preconditions and effects, in spirit with existing process ontologies such as OWL-S [W3C, 2004]. However, as is shown in Section 6.4.2, the semantic annotations reflecting the BP-specific restrictions and relations between the activities can be derived in an automatic way. Although the focus of the work presented in this chapter is to address process recovery from inconsistencies that result from process interference, the overall approach of using domain-independent AI planning for BP reconfiguration is more general, and can be used to react to any kind of events.

6.1.1 Process interference in e-Government: a test case

E-Government is a typical area characterized by multiple concurrently executing knowledge-intensive processes, which access and modify commonly shared resources such as citizen data, information reported by external contracted parties, etc. In such a context, addressing process interference becomes a critical issue, since important data used by subsequent tasks may become obsolete, and conditions on which the process relies may not hold anymore. Therefore, a BP has to be continuously informed about changes concerning that data, reason about them, and react accordingly in order to be able to ensure its consistency with the new state of the world.

In order to illustrate the effects of process interference and the potential ways to overcome them, let us consider a real case-study from the Dutch e-Government, which concerns the process realizing the Dutch Law for Societal Support, known as the *WMO law*[1]. This law is intended to offer support for people with a chronic disease or a disability, by providing facilities (usually by external parties) such as domestic care, transportation, a wheelchair or a home modification. Naturally, several different instances of the WMO process can be executed concurrently, together with other governmental processes, which may access and modify the same data. For example, during the execution of the WMO process, the citizen may move to a

[1]`http://nl.wikipedia.org/wiki/Wet_maatschappelijke_ondersteuning`

different address, the medical status of the citizen may alter, the eligibility criteria may change because of some new directive etc. The BP which models the WMO law, referred to as the WMO process, concerns the handling of the requests from citizens at one of the 418 municipalities in the Netherlands. Figure 6.1 represents the WMO process as used by one of the municipalities and is annotated with the required dependency scopes.

6.1.2 WMO process description

The WMO process starts with the submission of an application for a provision by a citizen. After receiving the application at the municipality office, a home visit is executed by an officer, in order to gather a detailed understanding of the situation. After the home visit, additional information on the citizen's health may still be required, which can be obtained via a medical advice provided by e.g., a general practitioner. Based on this information, a decision is made by the municipality to determine whether the citizen is eligible to receive the requested provision or not. In case of a negative decision, the citizen has the possibility for appeal. In case of a positive decision, the process continues and the requested provision is provided. For domestic help, the citizen has the choice between "Personal Budget" and "Care in Kind". In case of a "Personal Budget", the citizen periodically receives a certain amount of money for the granted provision, and in case of "Care In Kind" suppliers who can take care of the provision are contacted. For obtaining a wheelchair, first the detailed requirements are acquired before sending the order to the supplier. The home modification involves a tender procedure to select a supplier that provides the best offer. If the selected tender is approved by the municipality, the order is sent to the selected supplier. After delivery of the provision, an invoice is sent by the supplier to the municipality. Finally, the invoice is checked and paid.

In case of a negative decision (i.e., the application is rejected or the granted provision is less than the citizen requested), the citizen has the possibility for appeal. In case of a legitimate appeal, the provision is either granted, or the process is restarted. In case of a positive decision, the appropriate activities are executed, depending on the requested provision. For domestic help, the citizen has the choice between "Personal Budget" and "Care in Kind". In case of a "Personal Budget", the citizen periodically receives a certain amount of money for the granted provision to pay for workers or supervisors, and decide where the money is spent. In case of "Care In Kind" suppliers who can take care of the provision are contacted. A home modification involves a tender procedure to select a supplier, prior to execution of the actual home modification. A wheelchair is usually provided using a contracted supplier. After acquiring the detailed requirements, the order is sent to the selected

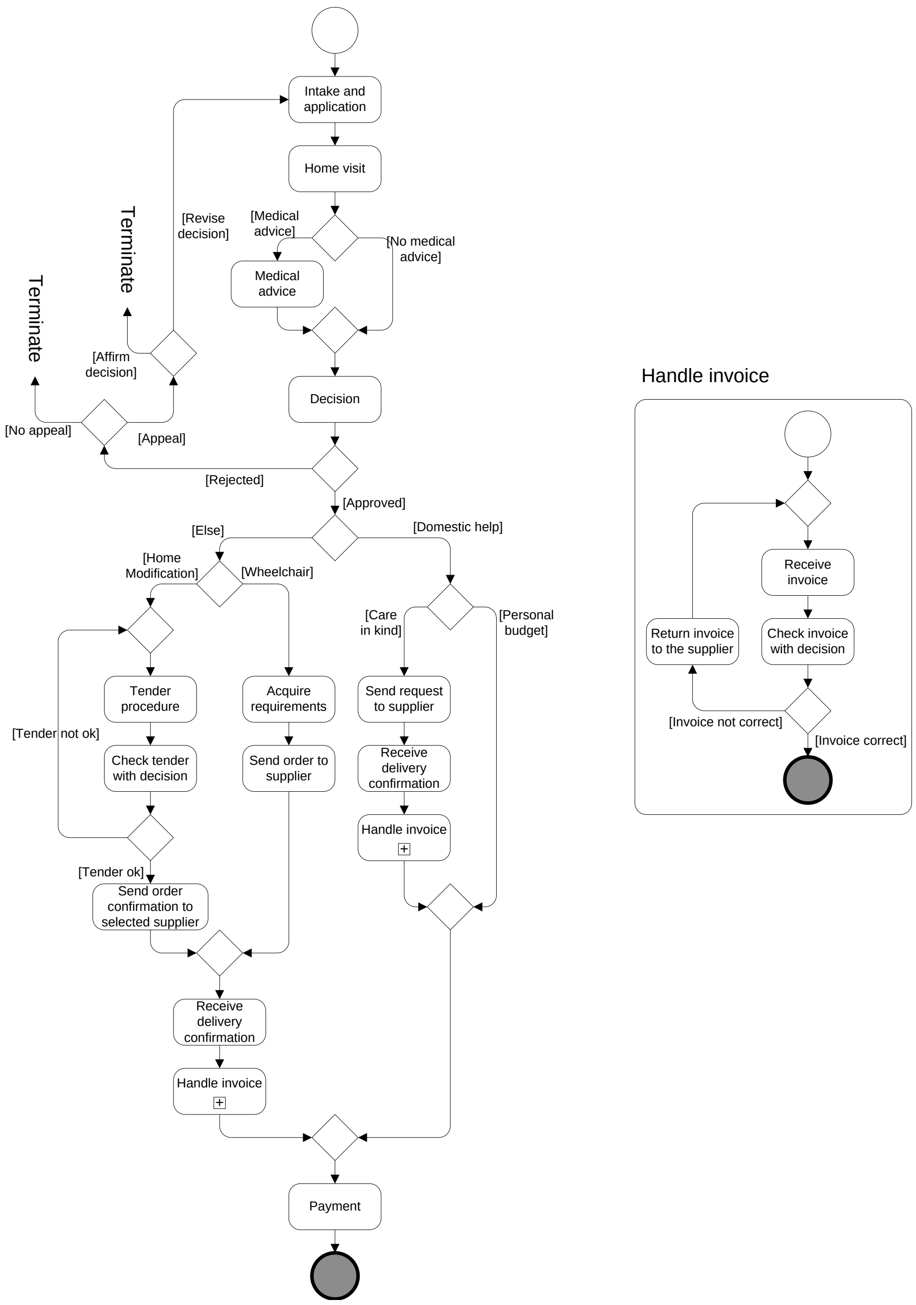


Figure 6.1: WMO process model

supplier, who delivers the provision. After that point, the process is identical for all provisions. The order is sent to the selected supplier, who delivers the provision and sends an invoice to the municipality. Finally, the invoice is checked and paid.

6.1.3 Interference examples

The request for a wheelchair or a home modification may take up to 6 weeks until the delivery of the provision. Evidently, this process depends on other stakeholders and their processes as well: the processes executed by the suppliers of the provisions or the citizens requesting the provision can affect the process as executed by the municipalities, due to their mutual dependence on certain process variables (e.g., the address or provision specifications). The WMO process depends on the correctness of some process variables as well. However, these process variables may be changed by another process running in parallel independently of the WMO process and are, therefore, volatile. Regardless of whether the WMO process is designed to be proprietary to the municipality, a change in either of these volatile process variables is entirely beyond the span of control of the municipality and may potentially have negative consequences for the WMO process. That is, due to its dependencies on those variables, these changes may result in undesirable business outcomes. Consequently, changes in these variables pose a potential risk of interference.

For instance, the activities after the decision until delivery strongly depend on the accuracy of the citizen's address. That is, the requirements of the wheelchair not only depend on the citizen, but also on the residence as this may pose some constraints to e.g., the width of the wheelchair. Consequently, an address change after "Acquire requirements" might result in a wheelchair that does not fit the actual requirements. Similarly, if the citizen moves to a nursing home after "Check tender with decision", the home modification is not necessary anymore. However, the supplier is not notified of this address change and the municipality is notified through a different process, which is external to the WMO process. Unless some action is taken to cancel or update the order, the WMO process proceeds with the home modification. In addition to the address, the process depends on the medical condition of the citizen, after executing the home visit and obtaining the medical advice. If the condition of citizen deteriorates, potentially the provision needs to be adjusted. If, on the other hand, the condition improves, the provision may no longer be necessary.

In order to guard for changes of the volatile process variables, Dependency Scopes can be defined covering a section of the process for which such a change poses a potential risk of interference. In Figure 6.2, a part of the process is annotated with the appropriate DSs. The section covered by DS1 relies on the accuracy of the

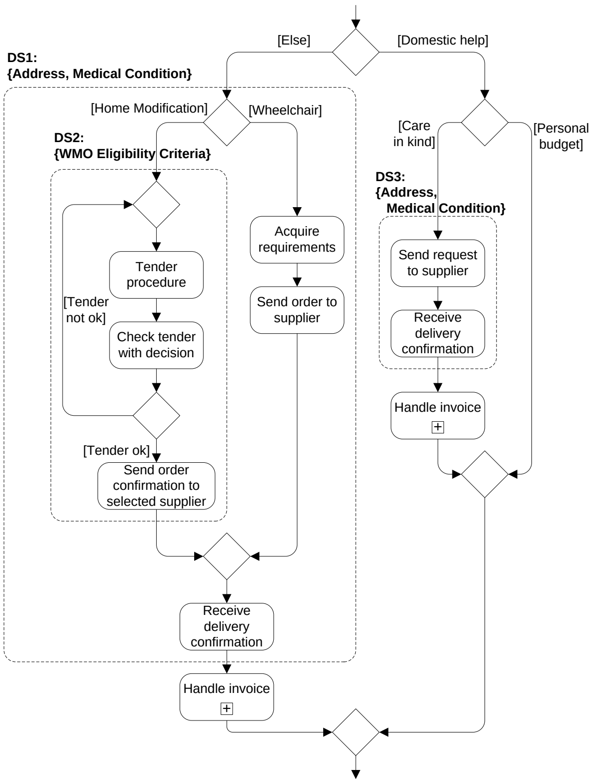


Figure 6.2: Dependency scopes in the WMO process.

address as well as the medical condition of the citizen, while the section covered by DS2 relies on the accuracy of the WMO eligibility criteria. That is, if the legal criteria that are relevant for the used contract have changed, this might affect the order itself, or the potential suppliers that are participating in the tender procedure. Finally, the section within DS3 depends on the address and the medical condition of the citizen as well. However, it is separate from DS1, because they comprise different branches of the BP. If a DS is triggered by an external change on some of the volatile variables it covers, some recovery activities may need to be executed to restore consistency. This leads to intervention processes, discussed in the next subsection.

6.1.4 Required intervention processes

The required IPs may differ for each situation. Let us consider for example the DS1 of Figure 6.2. If the provision concerns the delivery of a wheelchair, and the address change is detected before the order for the wheelchair is sent to the supplier, the following actions have to be performed. First, a new home visit to the new address has to take place in order to check the new residence and living conditions, which are important for the advice provided by the external consultant. Subsequently, the medical expert has to provide an updated advice, taking into account the characteristics of the new residence, and then a new decision has to be made by the municipality considering the newly acquired information (for example, if the user has moved to a nursery home, then the citizen may no longer be eligible for a wheelchair). If the municipality still approves the citizen's request, the requirements concerning the wheelchair have to be updated, and the respective order has to be sent to the supplier, as shown in Figure 6.3a. However, if the order was already sent to the supplier before the new information became available, this order has to be canceled prior to proceeding with the new one (Figure 6.3b). Please note that if there is an operation that offers the possibility to update the contents of an order that has already been issued (given that it has not already been delivered), the IP would include this operation rather than can celling the existing order and re-issuing a new one. After the execution of the appropriate IP, the process proceeds from the state just after the DS.

Similarly, in case of a home modification, the form of the appropriate IP depends on the state at which the address change has occurred as well. If the address changes before the order is sent, it is sufficient to execute the IP as represented in Figure 6.3c. Since the specifications on the order for a home modification directly rely on the physical properties of the house, a change of address implies a cancellation of the order if an order has already been sent, as shown in Figure 6.3d. However, these examples assume that the citizen moves *within* the municipality (in our example this is 'Groningen'). If the citizen has moved to another municipality, the order should be canceled and a notification sent to the city hall. Then, the entire process should be aborted, regardless of the requested provision, as each municipality has its own policies and procedures (Figure 6.3e).

It becomes evident from the example that even for a small DS, the complexity and workload required for specifying the IPs is high. Addressing the consequences of an address change on a small part of the process requires 5 distinct IPs. Anticipating and manually specifying the appropriate IP is difficult, time-consuming, and prone to oversights of possible situations that may arise: different IPs are required not only depending on the current state, but also on the actual value of the modified

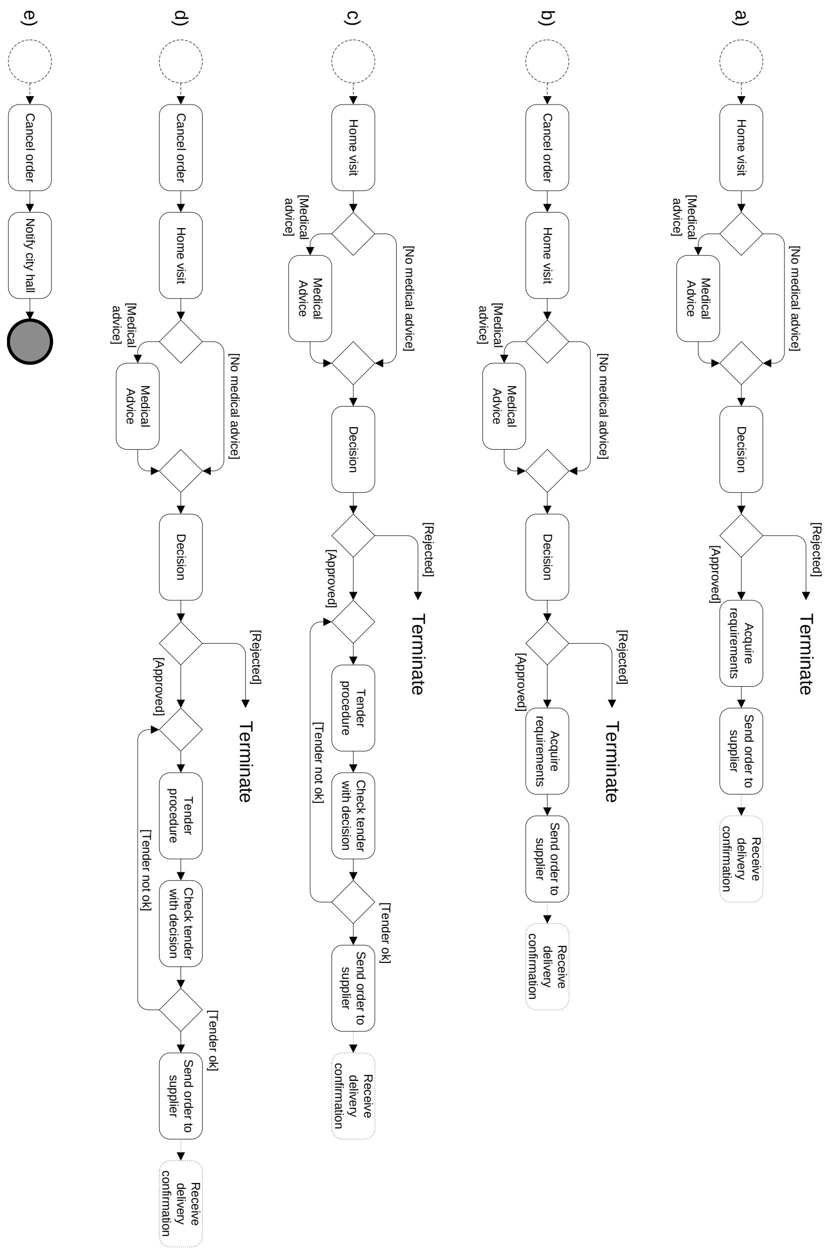


Figure 6.3: Required intervention processes corresponding to DS1, in case of an address change

variable. As a result, for each possible state in a DS and type of change to the modified variable, a different IP may be required. Moreover, since the same BP may be used by more than one municipality, different IPs have to be specified for each of the different cases, as they may have access to different compensation services or comply with different rules.

6.1.5 Automatic intervention process generation

Instead of relying on a procedural specification of IPs (or equivalently a composite IP with a huge number of conditional branches to take into account all possible combination of situations), we propose to assign the task of computing the appropriate IP to an AI domain-independent planner. The task of the BP designer in this case is reduced to a declarative specification of the properties that have to be fulfilled at a higher level of abstraction, considering some general, more intuitive cases. This way, it is not necessary to specify explicitly how these properties can be achieved under all possible combinations of environmental conditions and execution states of the BP. In this case the desired properties are captured by a *goal*. The goal represents the desired conditions that should hold in the state after the end of the DS, along with some (optional) features to be achieved. For example, considering the case of DS1 where the address change indicates that the user has moved to a new home within the range of the municipality, all that has to be captured by the goal is that at the end "the order of the citizen has to be delivered". The goals accompanying DS1 are presented in Section 6.3.3. In this respect, the BP designer does not have to be concerned with which service operations are available, what provision is requested, whether the order has already been sent or not, which activities have to be performed and in what order.

The approach adopted herein, leaves it to the domain-independent RuG planner to automatically generate the IP, whenever possible, based on semantically enriched services and BP specifications, the current state and the value of the volatile process variable. The assumption is that appropriate semantic annotations are available: the semantics accompanying the pool of services used by the BP have to be specified once and are reused by different BPs, while the BP-specific semantics represent the BP structure in a BPEL-like way, along with the direct dependency of some variables on the validity of some other variables (see Section 6.3). In the next section, the architecture of the framework supporting the automatic generation of IPs is presented, and the interplay between the different constituent modules is explained.

6.2 Architectural overview

Figure 6.4 provides an overview of the main components of our framework, along with their basic interactions. A *Process Modeler* (PM) is used to assist with the task of the graphical modeling of the BP, providing a selection of standard control blocks like sequence, flow, XOR etc., and design tools for modeling DSs, in accordance with their definition provided in Section 6.3. DSs include the specification of some high-level *goals* of declarative nature, which have to be fulfilled by the respective intervention process in case the conditions indicating an inconsistency are fired.

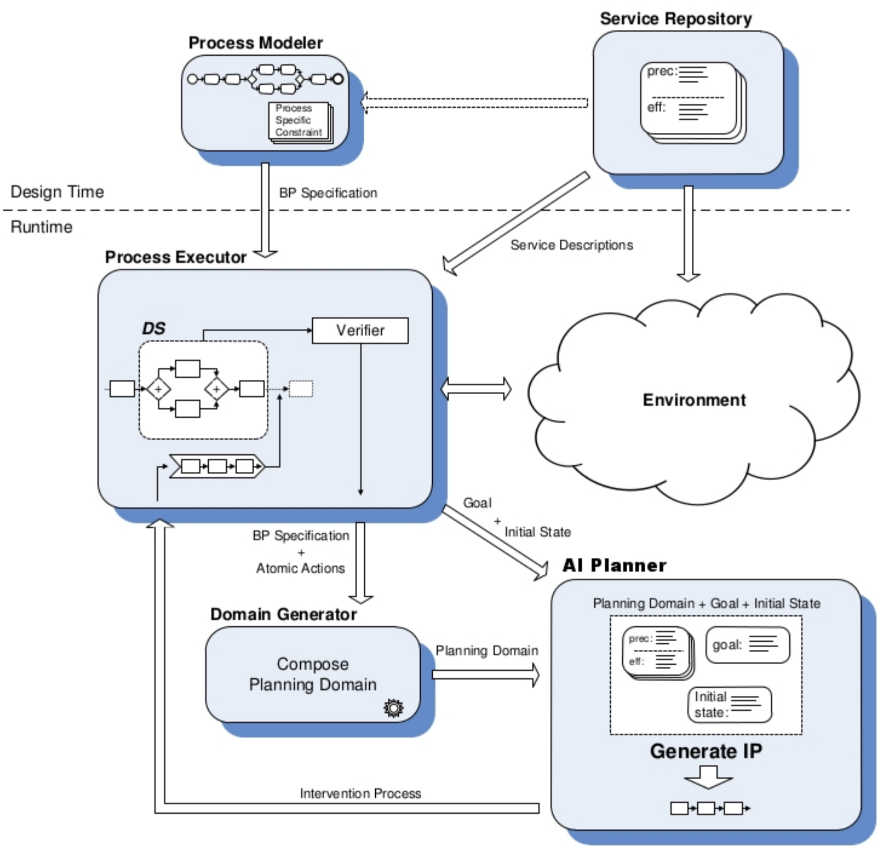


Figure 6.4: Main components of the framework and their basic interactions

The BP modeled by the PM uses activities that are available in the *Service Repository* (SR) by means of service operations. The SR keeps a list of service instances (providers) that offer a set of service operations. Each service instance implements a service description, which specifies the interface of the service annotated by some extra semantics. These semantics allow each service operation to be represented as a planning *action*, reflecting its functional behavior in terms of preconditions and effects, which are necessary for enabling the automatic generation of intervention processes. A subset of the service operations are referenced by the BP specification, whereas operations offered by other service instances can be marked as pertinent compensation actions, and can become part of an IP if necessary.

The *Process Executor* (PE) is responsible for executing the BP step by step (i.e., the normal course of events as specified during design-time), and takes care of discovering, binding and invoking the respective service operations residing in the *Environment*, according to their specification as included in the SR. Some of the variables describing the state of the environment can be directly changed by the process being executed by the PE, through the invocation of services it has access to, or can be modified by some external process. In the latter case, the PE receives a modification event, and updates its current internal state accordingly. In addition to process execution, the PE supports the use of DSs. Before execution of each activity, the PE checks whether the current state indicates a modification of the volatile variables that are guarded by a DS that covers this activity. If so, it verifies whether any of the conditions specified in the DS hold. If a condition holds (e.g., the new address is outside the current municipality), then the PE interrupts the execution and invokes the RuG planner. The RuG Planner requires as input (i) the Planning Domain (ii) the initial planning state (i.e., the values of all process variables at the current execution step and a set of variable interdependencies), and (iii) the goal describing the desired properties to it be achieved (e.g., a notification should be sent to the city hall).

The Planning Domain is computed by the *Domain Generator* (DG) only once for a certain process instance, namely the first time that the PE identifies the need for automatic IP generation. In order to form the Planning Domain, the PE passes the *Atomic Actions (AA)* and the BP specification (provided as output by the PM) to the DG. The AA represent the BP-pertinent action descriptions as kept in the SR, i.e. the ones referenced by the BP along with the available compensation operations. Given these two inputs, the DG can generate the encoding of the Planning Domain, by enriching the generic action descriptions of the *AA* with extra preconditions and effects that reflect the BP-specific interdependencies between the actions (e.g., sequence, flow and switch).

Given the Planning Domain, the initial state and the goal, the RuG planner generates the appropriate IP that achieves the associated goal. The generated IP is then returned to the PE. After the execution of the IP, the PE either proceeds with the execution of the original BP, starting from the state right after the triggered DS (as in Figures 6.3a-d, where the original BP execution resumes after "Delivery"), or aborts if the IP leads to a state that indicates the termination of the BP (as in Figure 6.3e). If the former is the case, potential branches that were running in parallel are also resumed from the point they were interrupted, otherwise the entire process is interrupted. In case of nested DSs, as for example DS1 and DS2 of Figure 6.2, the PE checks first whether the conditions specified by the outermost DS are true, and if not, it proceeds by checking the inner DS. The generated IP is executed within the scope of the DS it was triggered from and the parent DSs, i.e., variable modifications that are received during the execution of an IP are covered by the same set of DSs that covered the action before which the planner was fired. If no plan can be found, i.e., there is no way to overcome the inconsistencies caused by the volatile variable modification using the activities it has access to, then the BP is canceled, and a request for manual inspection is issued.

6.3 Basic concepts

In order to automate the task of intervention process generation, the original BP should be represented in a format which embraces the appropriate semantic annotations, namely the demarcation of the dependency scopes along with their accompanying goals, and the formalization of the participating activities in terms of preconditions and effects. The BP-specific information, concerning its structural constituent elements, is kept separate from the generic, BP-independent service descriptions, which are maintained in a separate repository, and can be referenced by different BPs. The basic syntactic structure of the BP builds upon the standardized executable language for describing BPs with Web Services, WS-BPEL[OASIS, 2007].

6.3.1 Service Repository

In the followings, we first define the Service Repository, which plays the role of a pool of services that are used as the building elements of different process specifications. The repository consists of a set of service descriptions and a set of service instances. Service descriptions model the logic of some abstract functionality (e.g., delivering some product), while service instances this realize some well-defined functionality by grounding it to some specific provider (e.g., a specific delivery company). Thus, the service descriptions comprise semantics which specify the operations provided by a service type, while the service instances specify the way to invoke a certain service in conformance with a service description.

Definition 13 (Service Type (st))**.** A service type is a tuple $st = (stid, O, SV)$, where *stid* is a unique identifier, O is a set of service operations, and SV is a list of variables, each ranging over a finite domain. These variables correspond to state variables internal to the service, whose value can be changed by the operations of the service. Each service operation $o \in O$ is a tuple $o = (id(o), in(o), out(o), prec(o), eff(o))$ where:

- $id(o)$ is the identifier of the operation.
- $in(o)$ is a list of variables that play the role of input parameters to o, ranging over finite domains.
- $out(o)$ is a list of variables that play the role of output parameters to o, ranging over finite domains.
- $prec(o)$ is a set of preconditions and $eff(o)$ a set of effects (as defined in Definition 1 with $Var = in(o) \cup out(o) \cup SV$).

For example, the service type related to home modification services has $stid = hm$, three operations (*TenderProcedure*, *CheckTender*, *SendOrderToSelSupplier*), and two state variables (*orderId* and *orderContents*). Each of the three operations has its own input and output parameters. E.g., *TenderProcedure* has two input (*tpIn_cid* and *tpIn_homeInfo*) and one output parameter (*tpOut_tenderSelected*). More details about the specifications of the service operation can be found in Appendix A.2.2.

Definition 14 (Service Instance (si))**.** A service instance is a tuple $si = (iid(si), st(si))$, where:

- $st(si)$ refers to the identifier of the service type $st \in ST$ this instance is compliant with.
- $iid(si)$ is the instance's unique identifier. For each pair of service instances si_1, si_2 that have the same service type identifier $st(si_1) = st(si_2)$, $iid(si_1) \neq iid(si_2)$.

Different service instances realizing the same service type have their own local variables and operations, which conform with the declarations specified in the respective service type. For example, service instance with $iid =$ "*prov1*", which implements *HomeModification*, keeps its own local variables *orderId* and *orderContents*, with the same names and domains as in the respective service type.

As is later shown in Section 6.3.2, BP activities refer to operations and variables of specific service instances through a unique identifier, consisting of the *stid*, the *iid* and the respective operation or variable name as defined in the service type. We can give the definition of the Service Repository:

Definition 15 (Service Repository (SR))**.** A Service Repository $SR = (ST, SI)$ is a storage, which keeps a set of Service Types ST and a set of Service Instances SI.

6.3.2 Business Process

In the followings, we provide the definition of a Business Process, which includes the basic activities and control structures such as sequence, flow and switch, and refers to services included in the SR. The BP is enriched with DSs, which also constitute parts of the process. Although the WMO process in Figure 6.1 is represented in BPMN-notation for readability reasons, the BP specification used in this paper is block-structured [Ouvans et al., 2006; Kopp et al., 2008], and is based on the basic constructs of BPEL. The BP specification can thus be directly parsed and executed by the Process Executor (see Section 6.5.2), and allows for its automatic transformation to a representation usable by the planner. The BP syntax is ultimately a tree structure where a block can have other blocks as children, and for each block its parent can be obtained. The definition is recursive, so that control structures and DSs can be nested within each other.

Definition 16 (Business Process (BP))**.** Given a Service Repository $SR = (ST, SI)$, a Business process is a tuple $BP = (PV, E)$, with E being a process element $E = ACT \mid SEQUENCE \mid FLOW \mid SWITCH \mid REPEAT \mid WHILE \mid DS$, where:

- $PV = PV_i \cup PV_e$ is a set of variables ranging over finite domains.
  - PV_i is a set of internal variables, which are BP-specific. A subset of these variables are passed as input parameters to the entire BP, and can be initialized with specific values at execution time.
  - PV_e is a set of external variables, which refer to state variables declared in the SR. An external variable $v \in PV_e$ is a reference $stid.iid.vid$, where $stid$ is the identifier of a service type $st = (stid, O, SV) \in ST$, iid is the identifier of a service instance $si = (iid, stid) \in SI$, and vid is the identifier of some state variable $v \in SV$.
- ACT is a process activity, as defined in Definition 17.

- *SEQUENCE* represents a totally ordered set of process elements, which are executed in sequence: $SEQUENCE\{e_1 \ldots e_n\}$, where $e_k \in E$.
- *FLOW* represents a set of process elements, which are executed in parallel: $FLOW\{e_1 \ldots e_n\}$, where $e_k \in E$.
- *SWITCH* is a set of tuples $\{(c_1, e_1), \ldots, (c_n, e_n)\}$, where $e_k \in E$ and c_k is a propositional formula, with all variables $\in PV$. All c_k participating in a *SWITCH* are mutually exclusive, i.e. for any given assignment to PV, only a single c_k evaluates to true, and e_k will be executed if c_k evaluates to true.
- *REPEAT* represents a loop structure, and is defined as a tuple $(pe, c\{pe_k\})$, where c is a propositional formula (see Definition 1), and $pe, pe_k \in E$. c is evaluated just after the end of pe, and if it holds, then pe is repeated, after the execution of the optional pe_k.
- *WHILE* is similar to *REPEAT*, with c being evaluated before pe starts.
- DS is a dependency scope as defined in Definition 18.

Definition 17 (Activity (ACT))**.** Given a Service Repository $SR = (ST, SI)$, an activity is a process element E which represents one of the following constructs:

- the invocation of a service instance, with $act = (id(act),\ in(act), out(act))$, where:
  - $id(act)$ is a reference *stid.iid.oid*, with *stid* being an identifier of a service type $st = (stid, O, SV) \in ST$, *iid* the identifier of a service instance $si = (iid, stid) \in SI$, and *oid* is the identifier of some operation $o \in O$.
  - $in(act) = in(oid)$.
  - $out(act) = out(oid)$.

  In BPEL, it may correspond to an *invoke*, *receive*, *reply*, etc.
- the idle activity *no-op*, which corresponds to *empty* in BPEL.
- the special activity *exit*, whose execution causes the entire BP to halt (corresponding to *exit* in BPEL).

The syntax we use to represent the BP is in line with the XML-based BPEL, and is presented by example in Appendix A.2.1.

For example, an activity *act* referring to operation *SendOrderToSelSupplier* provided by the service instance with $iid = prov1$ is characterized by the identifier $id(act) = hm.prov1.SendOrderToSelSupplier$. State variables of service instances, which form the PV_e set, are accessed from the BP through the variable's unique reference, e.g., *hm.prov1.orderId*. These variables can be accessed and changed by different BPs through the respective service instance operation calls, e.g., *hm.prov1.orderId*'s value is modified through the invocation of the *hm.prov1.SendOrderToSelSupplier*. The input and output parameters of activities, on the other hand, are local to each different BP instance, and can be assigned with constant values or other process variables: $id(act)(ipar_1 := v_1, \ldots, ipar_n := v_n)$, where $ipar_k \in in(act)$, and $v_k \in PV$ or v_k is a value compliant with $ipar_k$'s domain. The activity outputs can be stored in some process variable $pv_k \in PV$: $pv_k := opar_k$, where $opar_k \in out(a)$. the ultimate set of variables, including the input and output parameters to the operations of the service instances used by the planner, is described in detail in Section 6.4.1.

6.3.3 Dependency scope

A DS is a guard-verify structure, where the critical part of the BP is included in the *guard* block, while the *verify* block specifies the types of events that require intervention. Whenever such an event occurs, the control flow is transferred to the *verify* block, and the respective goal is activated. Once the resulting IP finishes its execution in the updated environment, the control flow of the BP continues from the point following the *guard-verify* structure, unless it is explicitly forced to terminate.

Definition 18 (Dependency Scope (DS))**.** Given a $SR = (ST, SI)$ and a $BP = (PV_i \cup PV_e, E)$, a Dependency Scope is defined as a tuple $DS = \langle guard(VV)\{CS\}, verify(\{(case(C_k) : G_k \mid BP_{ip} \mid terminate(G_k) \mid terminate(BP_{ip}))\})\rangle$, where:

- $guard(VV)$ indicates the set of volatile variables $VV \subset PV_e$ whose modification triggers the verification of the DS, and CS a process element of E, which is called the *Critical Section*. Whenever during the execution of CS an event is received indicating a change in the value of a volatile variable $vv \in VV$, the *verify* part of the DS is triggered, and the execution of the BP is interrupted.

- $verify(\{(case(c_k) : G_k \mid BP_{ip})\})$ comprises a set of tuples consisting of a case-condition c_k and a goal G_k or pre-specified intervention process BP_{ip} to be pursued if c_k holds.

  - c_k is a propositional formula (see Definition 1). Providing a case condition is optional, with the default interpretation being $c_k = \mathit{TRUE}$.
  - G_k specifies a goal, which ensures the satisfaction of the properties that reflect the state right after the final activity of *CS*. G_k is specified in the goal language supported by the RuG planner. After interrupting the BP execution, the plan that satisfies the respective G_k is executed. When the execution of the plan is completed, the *BP* is resumed at the state after *CS* and from any other parallel branches of the *BP* that were interrupted.
  - If a BP_{ip} is pre-specified to be executed in case C_k holds, then the execution of *BP* is interrupted, BP_{ip} is executed. After its completion, *BP* resumes from the end of *CS*.

- $\mathit{terminate}(G_k)$ ($\mathit{terminate}(BP_{ip})$) forces the process to terminate, i.e. abort the rest of the execution of *BP*, after fulfilling G_k (completing the execution of BP_{ip}).

The complete specification of the full WMO process, annotated with all DSs, is provided in A.2.1. Following Definition 18, the DS specification representing DS_1 of Figure 6.2 is the following:

```
<DS>
  <guard>
    <variables>
      <variable name="bpAddress"/>
      <variable name="bpMedCond"/>
    </variables>

    <!-- Subprocess covered by DS1 as in Figure 2 -->

  </guard>
  <verify>
    <case condition="bpAddress.county!='Groningen'">
      <terminate>
        <achieve-maint>
          <eq-val var="notifiedCityHall" value="TRUE"/>
          <eq-val var="messagePar" value="countyChange"/>
          <invalid var="orderId"/>
        </achieve-maint>
      </terminate>
    </case>
    <case condition="bpAddress.county='Groningen'␣AND␣bpMedCond!='deceased'">
      <achieve-maint>
        <known variable="dlOut_conf"/>
      </achieve-maint>
    </case>
    <case condition="bpMedCond='deceased'">
```

```
      <terminate>
        <achieve-maint>
          <invalid variable="orderId"/>
        </achieve-maint>
      </terminate>
    </case>
  </verify>
</DS>
```

According to *DS₁*, if a modification in the address or the medical condition occurs within the scope of the guarded subprocess, the following goals are pursued:

- If the address change indicates that the citizen has moved outside of the municipality, the goal ensures that the intervention plan leads to a state, where the order for a wheelchair or home modification (depending on the value of the "provision" variable, which is determined by the activity "Intake and Application") has been canceled, and a respective notification is sent to the city hall. The plan is equivalent to IP (e) of Figure 6.3.
- If the new address of the customer is still within the range of the municipality or/and the medical condition has changed to some new value that does not indicate "deceased", the final desired goal is that the delivery of wheelchair or home modification is performed by taking into account the new situation (the new medical condition and/or address). Depending on the state at which the modification occurs and the kind of the modification, the generated plan is one of the IPs (a) to (d) of Figure 6.3. After the plan's execution the BP execution resumes to handle the invoice.
- If the new value of medical condition indicates "deceased", then the goal specifies that the order should be invalidated.

Depending on the state of the DS in the original BP, at which the relevant volatile variable modification was identified, the generated plan may vary considerably for the same goal. This way, one DS definition covers all forms of IPs specified in Figure 6.3, which are generated automatically by the AI Planner. The domain designer just prescribes in the goal what properties have to be satisfied during recovery, but is not required to know the combinations of actions that can achieve the goal. The conservative action selection strategy used by the RuG planner promotes shorter plans. Considering, for example, an address change after an order has been sent in DS1 in Figure 6.2, if the supplier service offers an *updateOrder* operation, the planner will advocate an update in the order address information, instead of canceling the existing order and sending a new one.

Interdependencies between variables are also defined on top of the BP specification, prescribing the direct dependency of some variables on the validity of some other variable. The *dependsOn* relation is used for this purpose: $dependsOn(v) = \{v_1, \dots, v_n\}$. Whenever a change in variable v is discovered or whenever v is invalidated (by transitivity, as an effect of some other variable interdependency) by the PE, the direct invalidation of the current values of $v_1, \dots, v_n$ is automatically implied, without the need of some special-purpose process to take care of that. For example, *dependsOn(bpAddress_address)* = {*hvOut_homeInfo*}, since *hvOut_homeInfo* refers to the information retrieved for the specific *hvIn_address*. Thus, if the person moves to some other address, the collected information is not valid anymore. In turn, a set of variables, like *arOut_requirements* reflecting the acquired requirements concerning the wheelchair, are directly dependent on *hvOut_homeInfo*. On the other hand, an *orderId* is not directly dependent on the address, since it remains valid after these variables change, unless some other course of interaction actively cancels it. These additional statements are of particular relevance when the change of a volatile variable is discovered, so that all information directly dependent on the consistency of the volatile variable also becomes obsolete, as shown in Section 6.4.3. The full set of variable interdependencies that accompany the WMO process specifications are provided in A.2.1.

6.4 The BP as a planning domain

Given a *BP* specification and the BP-independent semantics of atomic services defined in the *SR*, the PE constructs a planning domain in conformance with Definition 1. The service operations stored in the *SR* in many cases do not include any preconditions, since many of them can be invoked individually at any time. On the other hand, the majority of the atomic-level effects are sensing effects, representing the outputs that the respective operation produces. For example, the result of the “Decision” action in Figure 6.1 is captured via an effect of the form *sense*(*dcOut_confirm*). To model the behavior of some compensation actions, the use of the *invalidate* type of effects is necessary (see Definition 1), in order to indicate that the value of a variable is not valid anymore. For instance, the action *cancelOrder*(*orderId*) has as an *invalidate*(*orderId*) effect, which entails that the *orderId* of an order that was processed is no longer valid.

To construct the *PD* which corresponds to a specific *BP*, the atomic-level semantics kept in the *SR* are enriched with extra preconditions and effects that are derived depending on the interrelations between the activities as specified in the *BP* structure. Different constructs in the *BP* description lead to different preconditions and effects. For example, conditional effects of the type *sense-cond_effect* are added to model XOR-splits, so that different effects are materialized depending on the outcome of some knowledge-providing action. e.g., a negative effect of the "Check Tender with Decision" activity (if the tender selection is not approved by the municipality) entails the invalidation of the "Tender Procedure" outcome for selecting the company to undertake the home modification process (because of the loop, as shown in Figure 6.1). As a result, the effect $\textit{sense-cond\_effect}(\textit{tsOut\_tenderSelOK} = \textit{false}, \textit{invalidate}(\textit{ctOut\_tenderSel}))$ is automatically generated and added to the effects of *CheckTender*, as shown in Appendix A.2.1.

6.4.1 Formation of the atomic actions

The semantic specifications stored in the Service Repository are process-independent, and capture the generic functionality of the respective service operations in terms of preconditions and effects, so that they can be used in the context of various BPs. Usually these preconditions and effects concern the set of inputs and outputs of the respective operations and some additional aspects that are internal to the particular service.

For each BP, the operations of a subset of service instances in the Service Repository are marked as pertinent compensation methods. These methods can be part of the intervention processes for repairing the BP, and are annotated by the domain designer. If a permissive approach is adopted, the entire set of service instances in the SI part of the SR is allowed to be used by the IP. These compensation methods, along with the invocation methods referenced by the activities in the BP, form the *BP-Pertinent Methods (BPPM)* set. For each method $\textit{stid.iid.oid} \in \textit{BPPM}$ of a service instance $si = (\textit{iid}, \textit{stid}) \in SI$, whose service description includes an operation o with $id(o) = oid$, the PE generates some instance-level variables, preconditions, and effects, based on its iid and the operation description o this method realizes. The resulting set of instance-level method descriptions forms the *Atomic Actions.*

Atomic Actions (AA) Given a Service Repository $SR = (SD, SI)$, a BP, and a set of BP-pertinent Methods *BPPM*, the Atomic Actions (AA) are formed as follows:

- When the PE receives a request to execute the BP, a unique instance reference $\textit{bp-iid}$ is assigned.
- For each method $\textit{bpo} = \textit{sdid.iid.oid} \in \textit{BPPM}$, the service description $sd = (\textit{sdid}, O, SV) \in SD$ is found, and the operation $o = (id(o), in(o), out(o), prec(o), \textit{eff}(o)) \in O$ with $id(o) = oid$ is retrieved.
- For each input parameter $ip_i \in in(o)$, a new input variable is created for $\textit{sdid.iid.oid}$, with name $\textit{bp-iid.sd.iid.oid.ip}_i$ and a domain identical to ip_i.

Similarly, for each output parameter $op_i \in out(o)$, a new output variable is created, with name $bp\text{-}iid.sd.iid.oid.op_i$ and a domain identical to op_i. The resulting instance-level input and output parameters form the sets $in(bpo)$ and $out(bpo)$ respectively.

- Based on the preconditions and effects of o, the sets $prec(bpo)$ and $eff(bpo)$ are generated, by substituting each input and output parameter with name v appearing in $prec(o)$ and $eff(o)$ by the reference $bp\text{-}iid.sdid.iid.oid.v$. In case of a service state variable $var \in SV$ with local name v, the reference is substituted with the universal name $sdid.iid.v$, which is BP independent. If $sdid.iid.v$ has not been met before, the respective variable with name $sdid.iid.v$ and with domain identical to var is created.

Atomic Actions (AA) Given a Service Repository $SR = (ST, SI)$, a BP, and a set of BP-pertinent Methods $BPPM$, the Atomic Actions (AA) are formed as follows:

- When the PE receives a request to execute the BP, a unique instance reference $bp\text{-}iid$ is assigned.
- For each method $bpo = stid.iid.oid \in BPPM$, the service type $st = (stid, O, SV) \in ST$ is found, and the operation $o = (id(o), in(o), out(o), prec(o), eff(o)) \in O$ with $id(o) = oid$ is retrieved.
- For each input parameter $ipar_j \in in(o)$, a new input variable is created for $stid.iid.oid$, with name $bp\text{-}iid.stid.iid.oid.ipar_j$ and a domain identical to $ipar_j$. Similarly, for each output parameter $opar_j \in out(o)$, a new output variable is created, with name $bp\text{-}iid.stid.iid.oid.opar_j$ and a domain identical to $opar_j$. The resulting instance-level input and output parameters form the sets $in(bpo)$ and $out(bpo)$ respectively.
- Based on the preconditions and effects of o, the sets $prec(bpo)$ and $eff(bpo)$ are generated, by substituting each input and output parameter with name v appearing in $prec(o)$ and $eff(o)$ by the reference $bp\text{-}iid.stid.iid.oid.v$. In case of a service state variable $var \in SV$ with local name v, the reference is substituted with the universal name $stid.iid.v$, which is BP independent.

This way, the invocation method description tuple $imd = (bp\text{-}iid.stid.iid.oid$, $in(act)$, $out(act)$, $prec(act)$, $eff(act))$ is created by the PE for each action $act = stid.iid.oid \in BPPM$. Each imd is converted to a planning action (see in Definition 1) $a = (id(a) = (bp\text{-}iid.stid.iid.oid$, $in(a_i) = in(act))$, $prec(a) = prec(act)$,

$\mathit{eff}(a) = \mathit{eff}(act)$). These actions form the AA. The set of the instance-level inputs and outputs of all $bpo \in BPPM$ form the $AtomicInputs(AI)$ and the $AtomicOutputs$ (AO) respectively, while the service state variables involved in the preconditions or effects of the service types of all $bpo \in BPPM$ form the *Atomic Service Variables (ASV)*.

The AA together with the set of variables AI, AO, ASV formed as described in the definition above, reflect only the atomic-level semantics of the actions. In the context of a certain BP structure, the universal action descriptions in the AA have to be enriched with extra preconditions and/or effects, which reflect the process-specific interdependencies, and which can be automatically inferred from the structure of the BP.

6.4.2 Generation of the planning domain

The Domain Generator is responsible for transforming the AA to a Planning Domain that comprises a process-specific representation of actions participating in the particular BP, which restricts their use according to the BP structure, as well as the compensation activities that are allowed to be used by the respective IPs. The DG is called only once by the PE, the first time it needs to call the RuG Planner. In this section, it is explained how these additional semantics are added to the atomic descriptions of the actions, in order to capture process-specific constraints.

Some additional assumptions regarding the BP definition given in Definition 16 have to be made, which allow us to derive all process-specific preconditions and effects in an automatic way from the BP specification. Given a repeat structure $repeat = (pe, c\{pe_i\})$, if the optional intermediate pe_i is empty, it is assumed that in case c holds, the outcomes of the activities in pe are automatically invalidated, in order to enforce the repetition of $\mathit{firstAct}(pe)$. For example, if the outcome of "Check tender with decision" is negative, the previous supplier selection made by the "Tender Procedure" becomes directly invalid, and another tender has to be selected. On the other hand, in case a pe_i is intervened before pe, some activity in pe_i should take care of the invalidation of the relevant outcomes of the actions in pe (as e.g., is the case with "Return invoice to the supplier"). These additional restrictive assumptions are not necessary if the extra preconditions and effects are added explicitly by the domain designer.

Algorithm 3 takes as input the BP specification, and the set of atomic actions AA (which comprise the activities participating in the BP plus the allowed compensation actions). By parsing the BP, it constructs the appropriate preconditions and effects for each activity that is part of the BP. These preconditions and effects are added on top of the atomic functional preconditions and effects of the respective action in the AA.

Algorithm 3 Automatic addition of BP-specific preconditions and effects given a BP specification and a set of atomic actions AA. The resulting set of BP-specific action descriptions constitutes the Planning Domain.

```
procedure PD($BP, AA$)
  while hasNext(BP) do
    $e$ = getNextElement($BP$)   //depth-first parsing of the BP tree
    match $type(e)$
      case $activity$:
        while hasNextInput($e$) do   //parse input assignments
          ($ipar_i := v$) = parseNextInput($e$)
          addPrec(getAction($id(e), AA$), '$ipar_i = v$')
        end while
        while hasNextOutAssign($e$) do   //parse possible assigns of outputs to vars
          ($bpVar := eOut\_v$) = parseNextOutAssign($e$)
          addEffect(getAction($id(e), AA$), '$assign(bpVar, eOut\_v)$')
        end while
        addPrec(getAction($id(e), AA$), SEQPREC(PREVELEM($e$), $BP$))
      case $switch\{(c_1, e_1), \ldots, (c_n, e_n)\}$:
        while hasNextBranch($e$) do   //parse all branches of the switch
          ($c_i, e_i$) = getNextBranch($e$)   //precs for all actions at beginning of switch
          $\forall a_i \in$ FIRSTACT($e_i$): addPrec(getAction($id(a_i), AA$), '$c_i$')
        end while
      case $repeat(pe, c)$:   //$e$ is a repeat without an intermediate $pe_i$
        $\forall a_i \in$ LASTACT($pe$):   //effects for all actions after the loop $pe$:
          //invalidate the outputs of all actions in the repeat loop
          addEffect(getAction(id($a_i$), AA), '$c \Rightarrow \wedge_{a_j \in pe, o_k \in out(a_j)} invalidate(o_k)$')
      case otherwise: continue
  end while
end procedure
```

The BP is treated as a tree (represented as an XML tree), where the root is the outer-most element in the specification, and the leaves are the activities. For each element, its parent can be obtained, and given an element one can reach its children. The parsing starts from the root and gets the next element in a depth-first way. If the element is an activity a, first its inputs are parsed: for each assignment to an input parameter, the respective equality proposition is added to a's preconditions. Next, possible assignments of the outputs of a to BP variables of the form $bpVar := eOut\_v$ are parsed, and the respective *assign* effect is added to the effects of a.

Algorithm 4 Functions for computing preconditions capturing sequence relations. The computed preconditions are added to the action that follows in the BP.

```
function seqPrec($e, BP$): Precondition
  match type(e)
    case $activity$:
      return '$\wedge_{o_j \in out(e)}$ $known(o_j)$'   //action's outputs are valid
    case $seq\{e_1, \ldots, e_n\}$:  seqPrec($e_n, BP$)
    case $repeat\{pe, c\{e_i\}\}$:  return $\neg c \wedge$ seqPrec($pe, BP$)
    case $switch\{(c_1, e_1), \ldots, (c_n, e_n)\}$:     //$e_i$ of switch-branch $c_i$ is valid if $c_i$
      return '$\wedge_i$ ($\neg c_i \vee$ seqPrec($e_i, BP$))'
    case $flow\{e_1, \ldots, e_n\}$:     //all parallel $e_i$s are valid
      return '$\wedge_i$ seqPrec($e_i$)'
    case $empty$:
      if prevAct($e$) $\neq \varnothing$ then
        seqPrec(prevAct($e, BP$))
      else
        return true
end function

function prevElem($e, BP$): Element   //Returns the previous element of $e$
  match type(parent($e, BP$))
    case $seq\{e_1, \ldots, e_n\}$:
      if $e = e_i \ \wedge \ i \neq 1$ then
        return $e_{i-1}$   //if $e = e_i$ not last in seq, return $e_{i-1}$
      else   //if last, the previous is the previous of the parent
        prevElem(parent($e, BP$))
    case otherwise:
      if parent($e, BP$)=$\varnothing$ then    //if root
        return $\varnothing$
      else   //in all other cases, previous is the previous of the parent
        prevElem(parent($e, BP$))
end function
```

The preconditions enforcing a's sequence relation with respect to its preceding process element e, as computed by the prevElem function in Algorithm 4, are returned by the function seqPrec. These preconditions ensure that the appropriate preceding actions are executed prior to a, depending on the type of e. More specifically, seqPrec obtains the preconditions corresponding to all execution paths that

Algorithm 5 Auxiliary functions used for adding switch and repeat conditions as preconditions.

```
function FIRSTACT($e, BP$): Set[Element]    //Find the first action(s) of an element
  match type(e)
    case $switch = \{(c_1, e_1), \ldots, (c_n, e_n)\}$:
      return FIRSTACT($e_1, BP$) $\cup \ldots \cup$ FIRSTACT($e_n, BP$)
    case $repeat = \{pe, c\{pe_i\}\}$:  return FIRSTACT($pe, BP$)
    case $flow\{e_1, \ldots, e_n)\}$:
      return FIRSTACT($e_1, BP$) $\cup \ldots \cup$ FIRSTACT($e_n, BP$)
    case $seq\{e_1, \ldots, e_n\}$:  return FIRSTACT($e_1, BP$)
    case $activity$:  return e
end function

function LASTACT($e, BP$): Set[Element]    //Find the last action(s) of an element
  match type(e)
    case $switch = \{(c_1, e_1), \ldots, (c_n, e_n)\}$:
      return LASTACT($e_1, BP$) $\cup \ldots \cup$ LASTACT($e_n, BP$)
    case $repeat = \{pe, c\{pe_i\}\}$:  return LASTACT($pe, BP$)
    case $flow\{e_1, \ldots, e_n)\}$:
      return LASTACT($e_1, BP$) $\cup \ldots \cup$ LASTACT($e_n, BP$)
    case $seq\{e_1, \ldots, e_n\}$:  return LASTACT($e_n, BP$)
    case $activity$:  return e
end function
```

may lead to a, by finding the last action(s) of the respective execution paths, and the possible respective conditions on which this path is depending. The function PREVELEM(a, BP) returns either the previous element of a in a sequence relation if such one exists. Otherwise it recursively goes back to the ancestors of a, until it reaches a sequence relation. If no sequence exists in its roots, there is no activity preceding a. If e=PREVELEM(a, BP) is an activity, the precondition states that the outputs of e have to be known. If e is a sequence, then SEQPREC is computed on the last element in that sequence. In case of a repeat, SEQPREC is called recursively on the loop element. Moreover, the negation of the condition at the end of the loop should hold for the control flow to proceed to a's execution. For multiple incoming branches in case of flow, the sequence preconditions modeling all elements in the flow are obtained. If the e is of type XOR=$\{(c_1, e_1), \ldots, (c_n, e_n))$, the preconditions state that the element e_i should be executed prior to a only *if* the respective branch was taken, i.e., if condition c_i holds. Finally, if e, i.e., the previous element with respect to the parent element of a, is the *empty* activity, and $parent(a)$ is not the root of the BP, then the algorithm proceeds recursively in computing the sequence preconditions entailed by the ancestors of e.

After taking care of the sequence preconditions, Algorithm 3 proceeds with checking the case where the current element in the tree is of type XOR. In this situation, for each branch(c_i, e_i) of the XOR the condition c_i is added as a precondition to the first activity(ies) of e_i. These first activities are computed by the function FIRSTACT in Algorithm 5. FIRSTACT recursively obtains the first element(s) of e_i, depending on the type of e_i, until this element is an activity. In the next step, if $e = \mathit{repeat}(\mathit{pe}, c)$, a conditional effect is added, which invalidates the results of all actions in the loop element pe, in case the repeat condition c holds, in order to compel their repetition. In Appendix A.2.1 the final planning domain representing the WMO BP, as produced by the application of Algorithm 3, is presented.

The outcome of the algorithm is a *BP-specific Actions Set (BPAS)*, which is the original AA enriched with the extra preconditions and effects. Together with the set of variables consisting of the variables AI, AO, ASV as described in Section 6.4.1 and the internal process variables PV_i declared in the BP, they constitute the planning domain considered by the planner. The BP-specific planning domain is thus defined as $PD = \langle \mathit{Var}, \mathit{Par}, \mathit{Act} \rangle$ (see Definition 1), with $\mathit{Var} = PV_i \cup AO \cup ASV$, $\mathit{Par} = AI$, and $\mathit{Act} = \mathit{BPAS}$.

6.4.3 Formation of the initial planning state

The initial planning state comprises the values of all variables at the current state of execution and the knowledge level with respect to the variables interdependency rules. Given the manually specified variable interdependencies in terms of the *dependsOn* sets, these are enriched during execution of the BP by the PE: if an action comprising an assignment effect $\mathit{assign}(v', v)$ or an increase(decrease) effect $\mathit{increase}(v', v)$ ($\mathit{decrease}(v', v)$), has been executed, variable v' is added automatically to the $\mathit{dependsOn}(v)$ set (if the set does not already exist, it is created). Each time the RuG planner is called by the PE, the initial planning state is formulated as follows.

- Each variable $\mathit{var} \in PV$ is equal to a value corresponding to the state of execution, i.e., considering the assignments to the BP input parameters, the outputs of the service invocations, the assignments to variables, and the received external events (for more details see Section 6.5).

- For each variable var, for which no specific value has been acquired yet, the respective knowledge variable $\mathit{known\_var}$ is set to false at the initial state ($\mathit{known\_var}(0) = \mathit{false}$).

- Given a change event on a volatile variable vv, the interdependency rules are parsed. For each $var \in dependsOn(vv)$, $known\_var(0) = false$, i.e., the value of var as reflected by the current state of execution is not valid. The same is done recursively for each $var' \in dependsOn(var)$, for all $var \in dependsOn(vv)$.

6.4.4 Generating the intervention process

By starting from the initial state as delivered by the PE, and depending on the goal, the IP can be generated by the RuG planner using the planning domain. This IP may include the re-invocation of activities with the up-to-date input parameters, if this is required to achieve the goal (e.g., pay a visit to the new address to acquire the informed requirements), or try to find a sequence of "undo" actions that actively lead to the invalidation of some variables (e.g., try to cancel an order that has been sent if possible). In case of deferred choices (i.e., XOR-constructs where the value of a variable participating in the respective condition is unknown offline) it has to be ensured that the right branch is followed at runtime. Given that sensing outcomes may well range over numeric-valued domains, we resort to a simple replanning from scratch mechanism to model deferred choices, when the value of the condition is acquired during runtime.

The plan originally returned by the RuG planner is optimistic, and the planner tends to choose the values which lead to the shortest plan. Thus, in case of the IP Figure 6.3c, it generates the plan that corresponds with the assumption that the output of "HomeVisit" $hvOut\_maRequired = false$, which indicates that the home inspection does not entail the need for a medical advice, that the decision is positive, and that the supplier selected by the customer is approved. Whenever a knowledge-providing activity is executed by the PE, and the initially unknown variable is instantiated, the outcome is compared with the value assumed by the plan. That is, it is checked whether the new knowledge incorporated in the CSP violates any constraint. If no violation is detected, then the execution of the IP may proceed according to the initial plan. Otherwise the planner is invoked again with the same goal and a new initial state, including the value of the sensed variable. As a result, a request for a Home Modification may require the following series of interactions when planning for Goal *achieve-maint(known(delOut_delId))* (see Section 6.3.3), in order to obtain the IP shown in Figure 6.3c (the input parameters are omitted for brevity):

Initial plan: $\{HomeVisit, Decision, TenderProcedure, CheckTender, SendOrder, Delivery\}$

Execute $HomeVisit$ Output: $hvOut\_maRequired = true$, **constraint violation, re-plan**

New plan: $\{MedicalAdvice, Decision, TenderProcedure, CheckTender, SendOrder, Delivery\}$

Execute: *MedicalAdvice*	$maOut\_medInfo$ =‘Document 12A’
Execute *Decision*	Output: $dcOut\_approvalCheck = true$
Execute *TenderProcedure*	Output: $tpOut\_tenderSelection$ =‘ACM Frizian Constructions’
Execute *CheckTender*	Output: $ctOut\_tenderOK = false$, **constraint violation, re-plan**

New plan: $\{TenderProcedure, CheckTender, SendOrder, Delivery\}$

Execute *TenderProcedure*	Output: $tpOut\_tenderSelection$ =‘van der Meer Elevators’
Execute *CheckTender*	Output $ctOut\_tenderOK = false$
Execute *SendOrderToSelSupplier*	Output: $soOut\_orderId$ =‘14578AS’
Execute *Delivery*	Output: $dlOut\_conf$ =‘Delivered’

If the output of “Decision” is negative, then no plan exists that satisfies the goal. In that case, the planner returns a message indicating that the goal is not satisfiable, causing the BP execution to be aborted. In total 9 service operations are invoked as part of the IP.

6.5 The prototype

The proposed approach for automatic process recovery upon data changes has been implemented in a prototype, comprising the components of the architecture outlined in Figure 6.4.

6.5.1 Process Modeler

The Process Modeler (PM) is implemented in Java, by the use of standard Java 2D graphical libraries. It supports all basic BP modeling constructs, including flows, XOR splits etc., with an added support for DS modeling. Furthermore, the PM provides for the declaration of the process variables, i.e., the definition of their name and type. However, the actual object creation is handled by the PE, which keeps and manages a local database as described in Section 6.5.2. The PM is connected to the Service Repository, so that the BP designer can use service operations that exist in the SR as activities in the BP being modeled.

Figure 6.5 presents a screenshot of the PM, showing the graphical representation of DS1 and DS2 of the WMO process from Figure 6.2. The DSs are saved along with the process specification itself. The final output of the PM is an XML representation of the BP, which conforms to Definition 16. This representation is passed to the PE for execution, as described in the next subsection.

6.5.2 The Process Executor

The Process Executor (PE) is responsible for executing a BP as specified by the PM. The PE takes as an input a BP specification in conformance with an XML

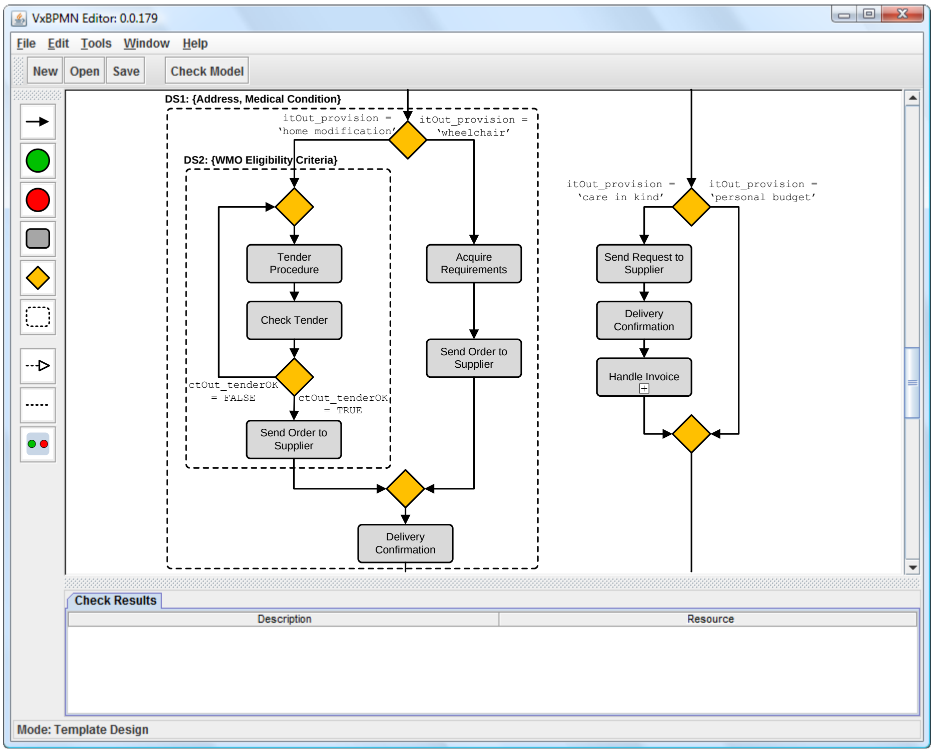


Figure 6.5: Screenshot of the Process Modeler.

schema that represents Definition 16, and with the BP input parameters instantiated to specific values. The PE works in cooperation with the Service Repository as described in Definition 15. The details of Service Instances implementation are out of scope of this work, and for the purposes of the testing presented in Section 6.6 the service invocations are simulated.

The activities included in the BP specification must refer to method invocations that can be retrieved from the SR. Given a fully qualified reference to an invocation method *stid.iid.oid* specified by an activity in the BP specification, the PE retrieves the respective description kept in the SR. For example, the activity "*Send Order*" in Figure 6.6 refers to "*HomeModification.iid.sendOrderToSel-Supplier*", which corresponds to the method "*sendOrderToSelSupplier*" of the "*HomeModification*" service type, and is provided by the service instance with identifier "*WMO_hm_GR*" (see Definition 15). The service type of "*HomeModification*" as well as the service instance (provider) "*WMO_hm_GR*" are kept in the *SR*, as shown in Figure 6.6. It

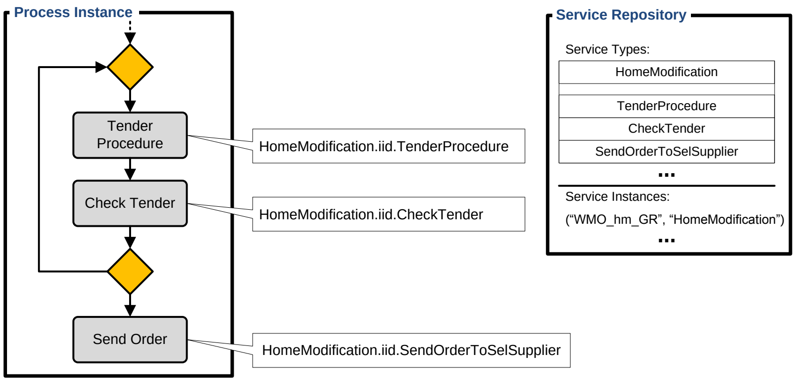


Figure 6.6: Example of a Service Description and a Service Instance.

should be noted that the value of the variable *iid* in the BP specification may be unknown before a process is actually started, and an assignment to another value $iid = iv$ can be used instead of a predefined value. The value of *iv* can be provided by the user at execution time, or retrieved by the PE as an output value of a service method call. In the example in Figure 6.6 the value "*WMO_hm_GR*" for the variable *iid* is provided at the time the process instance execution starts.

In the current implementation, an activity is executed by directly invoking the respective method, without checking whether the preconditions prescribed in the corresponding service instance description in the AA hold. Control flows are treated as by a typical execution engine. The data flow and knowledge about the environment are handled by a *local storage* (LS), which is maintained by the PE and reflects its knowledge about the environment and the state of the process instance execution. Some of these variables are specific to a particular BP running instance, and some are common to multiple BPs. During execution, the PE updates the LS according to the new information it receives from the environment (from service method invocations), and to the specifications included in the BP description (assignments to variables). When the PE receives a request for executing an instance of a BP specification $BP = (PV, E)$, it assigns a unique identifier *bp-iid* to the running instance, and constructs the AA along with the instance-level inputs and outputs $AI \cup AO$ (as described in Section 6.4.1), which are added to the LS. Each service state variable $sv \in ASV$ (see Section 6.4.1) is added to the LS if it does not already exist. This way, state variables of the AA are shared among running process instances, whereas instance-level input and output variables are unique to each process instance. Moreover, the PE constructs the instance-level internal variables declared in the BP (i.e. for each $var \in PV_i$) with name v a variable with name *bp-iid.v* and domain identical to *var*'s domain is added to the LS. The internal process variables are also unique to the process instance. The value of an instance-level variable cannot be changed by any other external factor other than the BP instance *bp-iid* it belongs to, while a shared variable can be modified by any other entity that calls the service operation which affects it.

The distinguishing feature of the PE with respect to other well-known BP execution engines is the support for dealing with the DSs specified in a BP. When a process execution runs into a DS, the PE turns into a special "DS mode". In that mode, the PE creates an event listener for each of the volatile variables specified in the DS. It is assumed that modification events can be captured by subscribing to specific variables of interest, and that external services that have the permission to change these variables publish an appropriate event that is caught by the subscribed clients (listeners). It should be mentioned that modern architectures for distributed and dynamic information systems provide strategies for scalable and reliable monitoring and notification of changes (e.g., Oracle JMS [Oracle, 2002] or [Vargas et al., 2005]). The details of event firing and catching are outside the scope of this paper. However, if there is no support for a publish-subscribe model, then the PE would have to adopt a "request/response" approach, by actively retrieving the current values of volatile variables before each execution step to check for changes.

The event handling is deferred until the activity currently being executed finishes, thus avoiding potential inconsistencies that may result from canceling an activity in the middle of execution. Therefore, the information conveyed by the data modification events is stored in a memory list that maintains tuples of the recently modified variables and their latest values. A new event on the same variable overwrites the old value of the variable kept in the memory list. This list of recent changes is checked prior to executing the next activity within a DS, and if it is not empty, the conditions in the *verify* block of the DS are checked towards the latest values kept in the list. If a condition evaluates to true, the respective goal or process element is fired, while the BP execution is suspended. In case of a flow, all parallel branches are put on hold. The list of recent changes is cleared, and the LS is updated accordingly, by incorporating the most up-to-date values to the respective variables.

In case a goal has to be pursued, the planner is invoked in order to create a plan which is then executed, while in case of a pre-specified element this is directly executed. After a plan or a pre-specified element is executed the initial process execution is resumed, starting from the activity which is immediately after the end of the current DS. In case parallel branches were suspended, these are resumed as well (the underlying assumption is that the execution of the generated IP does not introduce any inconsistencies in the suspended concurrent branches). The only exception is when there is a *terminate* annotation referring to the goal that is triggered (see Definition 18), in which case the original BP is terminated instead.

In case of nested DSs, the conditions are verified for all active dependency scopes starting from the most outer one and going inward. When the execution of a subprocess covered by some DS is finished, then the respective DS is removed from the list of active DSs, as well as all event listeners associated with it. If the list is empty, then PE leaves the "DS mode" and does not listen to any data modification events. Note that while executing an IP, the PE still remains in the same "DS mode", and thus treats the modification events it receives during the IP execution in the same way as it did during execution of the process element covered by the DS in the BP. This means that an IP "inherits" the DSs that covered the activity before which the planner was invoked. In case a DS condition is triggered, the current IP execution is interrupted, a new IP is generated, after whose completion, the PE returns to the state after the DS in the original BP.

In order to generate a plan, the RuG planner needs a planning domain representation (see Definition 1). To this end, the PE calls the Domain Generator, by passing to it the Atomic Actions (AA), built as described in Section 6.4.1 by including all service instances referenced in the BP and a set of eligible compensation services from the SR. The planning domain is constructed only once for a specific BP, the first time that a DS is triggered. The goal taken from the DS specification and the current state, i.e., the values of the variables that are part of the planning domain as reflected by the updated database, are handed over to the AI planner, which uses them along with the planning domain to compute a plan. This plan, which includes only sequence and flow structures, is then passed for execution to the PE. Loops in the plan are "flattened", i.e., the plans explicitly include all repetitions in sequence. Deferred choices (such as in the case of XORs) are addressed indirectly as already described in Section 6.4.4: whenever the PE executes an operation that returns a new value, the constraint solver is called to check whether this value leads to any inconsistencies with respect to the outcome anticipated by the plan, and if so, the planner is re-invoked with the current state of execution as the initial state (and the same goal).

6.6 Evaluation

Unfortunately, there is no commonly agreed benchmark for evaluating and comparing existing techniques for runtime BP reconfiguration. Different approaches depart from different starting points with respect to what is given and what is to be achieved (e.g., [Bucchiarone et al., 2011] assumes service descriptions and context properties as state transition systems); make different restrictive assumptions about the expected behavior of the available services (e.g., [de Leoni et al., 2009; Marrella and Mecella, 2011] assume deterministic service outcomes); and ultimately can resolve different kinds of inconsistencies (e.g., none of previous approaches addresses the problem of undesired business outcomes due to obsolete data). As a result, each approach considers its own test scenarios that fit the particular features it seeks to demonstrate, evaluation of performance is limited to a proof-of-concept use-case or not provided at all (e.g., [de Leoni et al., 2007, 2009; Marrella and Mecella, 2011] present only a theoretical framework). Due to these reasons, no direct comparison with previous approaches similar to ours in terms of some well-defined metrics can be made.

The aim of the evaluation presented in the followings is (i) to demonstrate the effectiveness of our approach with respect to our working example presented in Section 5.4 and (ii) to test the performance with respect to the time that is required to generate the necessary IPs. The specification of the desired goals and DSs has been conducted in close cooperation with WMO employees at the municipality of Groningen. Our experience confirmed that the translation of the requirements as expressed by non-technical employees to the representation required by our framework is rather intuitive, and is relatively easily understood when shown to non-experts for proof-checking.

In the tests presented in the followings, service invocations are simulated, and the methods provided by the service instances have a predefined behavior, simulated according to the different situations we want to test. The performance of the framework has been tested with respect to atomic action repositories of increasing size, since domains that comprise a large set of actions, may raise concerns of inefficiency. All tests presented thereafter were performed on a computer with an Intel® Core™2 Duo processor @2.83GHz running Java 1.6.0_24.

6.6.1 Tests on the WMO-law case study

In order to test the framework we have developed on a real case-study, the WMO process shown in Figure 6.1 was modeled, along with the DSs shown in Figure 6.2. The BP specification representing the case-study is as shown in Appendix A.2.1, while the Planning Domain used by the planner is the output of Algorithm 3, given this BP specification and the set of atomic actions descriptions.

Table 6.1 provides an overview of the times required to generate the initial plans for all IPs shown in Figure 6.3, corresponding to DS1 of Figure 6.2, in case of a change in the applicant's address. In all cases, the time for generating the respective initial IP is below one second. However all IPs in this example, except for case (e), comprise one or more deferred choices, which implies that replanning may be needed. As a result, after the execution of a knowledge-providing action, a violation check verifies whether the actual output differs from the expected value. If that is the case, the planner is invoked again with the same goal, but starting from the updated state corresponding to the newly sensed value(s).

IP	**Plan length**	**Time for planning (in sec)**
a	5	0.51
b	6	0.59
c	6	0.60
d	7	0.62
e	2	0.39

Table 6.1: Performance results for generating the IPs of Figure 6.3

.

Tables 6.2i and 6.2ii present the times for computing each updated plan in case of some possible environmental behavior for the IPs depicted in Figures 6.3b and 6.3c, which have 2 and 3 deferred choices respectively. Replanning is performed until the goal as specified in Section 6.3 is satisfied, or no solution can be found. The reported times are the average over 4 separate test runs.

The IP in Figure 6.3b corresponds to the situation where a change in address occurs when a wheelchair is already ordered but not yet delivered. The initial plan in Table 6.2i is generated assuming optimistic outcomes for the variables that are unknown at runtime. Consequently, it is assumed that no extra medical advice is required (*hvOut_medAdvReq=FALSE*) and that the decision is positive (*dcOut_-decision= 'Approved'*). During execution of the initial plan, the PE may find out that a medical advice is required, in which case it updates the plan accordingly by including an extra action. If the outcome of the decision is negative, a constraint violation is encountered by the PE. The new situation (with *dcOut_decision= 'Not Approved'*) is sent to the planner for replanning. In that case, however, no plan can be found that fulfills the goal, and the PE is informed accordingly.

The IP in Figure 6.3c covers the case where the address changes at the stage where a home modification is requested, but the request is not yet confirmed. Table 6.2ii presents the times for the initial plan (assuming no medical advice, a positive decision, and the selected tender to be approved), and the potential updates as a result of replanning. The actual service invocations may lead to the following discrepancies: the medical advice is actually required, and the plan is updated; the decision is negative, in which case no plan can be found that reaches the goal; the selected tender is not approved and a new plan is computed, asking the user to make a new selection (see also Section 6.4.4 for a possible execution behavior showing the exact service invocations that take place).

State when planner is called	**Plan length**	**Time for violation check and planning (in sec)**
Initial state	6	0.6 (optimistic plan)
“Medical Advice required”	5	0.3 (violation, new plan)
“Rejected”	- (no plan)	0.02 (violation, goal can’t be satisfied)

(i)

State when planner is called	**Plan length**	**Time for violation check and planning (in sec)**
Initial state	6	0.6 (optimistic plan)
“Medical Advice required”	6	0.3 (violation, new plan)
“TenderNotOK”	4	0.2 (violation, new plan)
“Rejected”	- (no plan)	0.02 (violation, goal can’t be satisfied)

(ii)

Table 6.2: Replanning times for (i) the IP of Figure 6.3b and (ii) the IP of Figure 6.3c

6.6.2 Scalability in a simulated domain

In case of the WMO process, the planning domain comprises 16 actions (i.e., the BP-pertinent methods including the actions that are part of the BP as well as the compensation actions), while the largest IP consists of 7 actions (note that if one adds up all actions that are executed as part of the replanning process, the total number of actions that are executed as part of an IP may be significantly larger). For most BPs, the length of the IPs for recovering from the most usual situations are relatively short. However, there are occasions where the length of the required IPs might be significantly larger than the examples presented for the WMO case. For example, since the planner cannot produce plans with structured loops, many repetitions of a set of actions may be required to represent the desired pattern.

In order to evaluate the scalability of our framework with respect to the size of the required IPs (i.e. the number of activities they comprise), a number of tests have been performed with different goals, whose fulfillment requires IPs with an increasing size from 20 to 100 activities. For the sake of these tests, a virtual set of 100 atomic actions has been created, comprising the search space of the planner. The actions in the domain are interconnected through trivial sequence relations, so that an action's preconditions are satisfied when the effects of all its preceding actions are materialized. The results of these tests are summarized in Table 6.3. They give an impression of how composition time is affected by the size of the required IP, for a given a business domain that consists only of sequence structures. The tests show that for a trivial domain, that about a minute is required to generate an IP consisting of as many as 100 activities.

	20 act	**40 act**	**60 act**	**80 act**	**100 act**
Planning time (in sec)	3.5	8.9	17.7	43.7	63.6

Table 6.3: Time for generating IPs of increasing size (domain size=100)

It should be noted that the simulated BP description imposes rather direct interdependencies between actions, thus leading to planning domains with quite a simple structure. Disjunctive propositions, which are known to add an extra burden to the constraint solver, result mainly from nested XORs (switches) with many branches (see Algorithm 4) and to a less extent from the repeat structures leading to a disjunctive effect (see Algorithm 3). However, given the predominance of the sequence construct, even in complex BPs the domain filtering during constraint solving proceeds fast. The experimental evaluation on a realistic BP taken from the Dutch e-government confirms that the time for generating IPs under different circumstances is a matter of a few seconds, which is an acceptable performance considering the average throughput time of long-running BPs (varying between 1 and 6 weeks for the WMO case). In case of a real time process, which requires a response within a few milliseconds, even a few seconds of planning time may not be acceptable. In such a situation, it is more efficient to specify DSs with predefined IPs, which can be directly passed for execution.

Chapter 7

Conclusions

The work presented in this thesis is driven by the interest to investigate the role that domain-independent automated planning can be reserved in service-oriented environments. Such environments pose many interesting problems to the AI planning community, whose tools and techniques can contribute towards leveraging the degree of adaptability and automation in a number of practical tasks, such as the coordination of smart devices and sensors in domotic settings and the recovery of Business Processes from undesirable situations. The common general objectives underlying these applications is the necessity to reason about the ways that functionalities offered by diverse services can be combined in a just-in-time and context-aware manner.

7.1 Recapitulation

In order to function in service domains, a planning system should be equipped with a number of special features that enable it to dispose of the uncertainty deriving from the open world assumption as well as from unexpected service behaviors, deal with the abundance of data-intensive operations, and overcome inconsistencies due to changes caused by exogenous factors. Most importantly, the applicability of the planning system should not be tailored to the specifics of some particular domain and task, but rather be in the position to fulfill a variety of diverse objectives with minimal manual reconfiguration.

To meet these requirements, we have described a planning framework which uses constraint satisfaction techniques and accommodates for complex goals, a knowledge-level representation to model lack of information, proactive sensing in the presence of variables that range over large domains, as well as an algorithm for monitoring execution and revising plans in a seamlessly changing environment. These features put together enhance the extent of scenarios that can be represented and dealt with compared to previous planning approaches to Web Service composition.

Recalling Table 2.1, which compares in summary the main planning approaches to WSC, it is time to position the capabilities of the RuG planner coupled with the orchestration algorithm. The assessment with respect to the different dimensions are illustrated in Table 7.1. The approach relies on a domain-independent representation, which consists of a set of loosely-coupled atomic service operations described as planning operators. Data-intensive domains which involve many operations that work with numeric-valued variables, including the case of numeric-valued sensing outputs, are effectively dealt with. Regarding goal expressivity, the RuG planner supports a number of constructs that impose constraints over the state traversal, but it does not accommodate for preferences. No restrictive assumptions about the interdependencies between sensing and word-altering actions in the composition are made, and sensing actions are proactively planned for. The approach performs continuous plan revisions to dispose of failure responses, long responses and timeouts, and with exogenous events. The orchestration algorithm can address effectively many problematic situations, under certain assumptions regarding the kind of goal and the point at which the contingency occurs during execution.

Domain independence	**Support for data**	**Goal expressivity**	**Sensing**	**Contingencies**
★★★ (atomic actions)	★★★ (numeric-valued outputs)	★★ (extended but no preferences)	★★★ (proactive, interleaved with world changes)	★★ (failures, timeouts, external events, in certain situations)

Table 7.1: Assessment of the RuG planner+orchestration framework with respect to the criteria of Table 2.1.

The practical use of the planning framework has been demonstrated on a number of scenarios and service platforms, exemplifying how the planning framework can be used to serve a wide range of objectives under various circumstances. Evaluation has been performed to assess the feasibility of the planning system as well as the overall service architecture within which it operates, including experiments about the performance of the planning techniques, the effectiveness of the produced solutions and usability tests. Although our work is inspired by applications in the field of Web Services, the essence of the planning methodologies we describe is more general, and touches upon issues that concern a series of problems where domain-independence, uncertainty and dynamicity are at stake.

7.2 Open issues and future directions

The issues explored in the current thesis are open to further investigation and point to many directions for additional research developments. From the application point of view, there is much space for improvement and extensions in order to make the proposed solutions more practical and user-friendly. To this end, automating the derivation of planning-level descriptions from a family of standard service semantics is an important step towards making the use of AI planning a feasible approach to real-world problems. An interesting follow-up issue concerns how background ontologies and associated hierarchies can be incorporated and exploited in the context of domain-independent planning.

Besides service composition, domain-independent AI planning can also prove helpful for other tasks that are of particular interest to service-oriented and pervasive platforms. For example, planning techniques for automatic plan recognition can be combined with current sensor- and vision-based recognition systems [Dominici et al., 2011] to advance the reasoning capabilities of human activity monitoring in modern ubiquitous environments. Similarly, planning can be used for diagnosis purposes [Sohrabi et al., 2011] along with standard failure detection mechanisms to identify and explain aberrant and problematic situations, e.g., to automate and advance the role of the rule engine discussed in Chapter 4. In the context of Business Process reconfiguration, planning can be used not only for repair purposes in case of some runtime data modifications, but also to automate process adaptation in case of changes in the business requirements and rules.

There is also much space for improving the performance of the RuG planner, as well as the quality of the generated plans. The application of some of the reformulation techniques proposed in [Barták and Toropila, 2008] for the parts of the domain representation where this is possible can probably prove useful to speed up search. However, what the planning system is mostly missing is planning-oriented rather than just CSP-based heuristics, which manage to extract additional constraints that reflect particular properties of the underlying planning problem, and are not restricted to propositional encodings.

Extending the planning system to deal with noisy data, i.e. sensing actions which return a set of possible values, is also an interesting direction for future work. To that end, it is worth investigating whether an approach similar to the interval-valued function described in [Petrick, 2011] and the performance of some sort of case-based reasoning can be adopted by the RuG planner. Regarding goal expressivity, the support for soft constraints would certainly add to the planner's power, however its implications on performance remain to be investigated. The capabilities of the orchestration framework can also be improved and extended in many ways. For example, exploiting techniques used in the context of dynamic CSP for intelligent solution reuse can probably benefit performance. How to validate an extended goal towards a complete orchestration run rather than just from the current state onwards is another interesting question to explore.

Appendix A

Appendix

A.1 Orchestration example of moving robot in grid

In the followings we present the sequence of steps taken by the orchestrator for the "moving in grid" scenario described in Section 5.5.2. The robot moves around the 3x3 grid depicted in Figure 5.2. At the initial state it resides at location $R00\_R01$, and the goal is to reach location $R22\_R21$ at the final state, while all doors are initially closed and unlocked. In order to open the doors leading to $R22$ it needs to have a password, which it can retrieve by issuing a *senseRoom22Code* sensing operation. During execution, three kinds of contingencies occur: some external actor locks a door, an opening door operation repeatedly reports success but fails to open the door, and the *senseRoom22Code* expires with no response. These situations ultimately require replanning from scratch, and at the end the goal cannot be satisfied since the required password cannot be retrieved.

open:doorR00_R01
Context change *doorR00_R01 = OPEN*
In parallel: {
senseRoom22Code set:robotPlace(R01_R00) }
Context change *robotPlace = R01_R00*
Short Timeout of senseRoom22Code2
In parallel: {
set:robotPlace(R01_R11)
open:doorR01_R11
}
Context change *doorR01_R11 = OPEN*
Context change *doorR11_R21 = LOCKED* //somebody locked doorR11_R21
Context change *robotPlace = R01_R11*
Refine plan: the current plan cannot be augmented
Replan from scratch //new plan guides robot through $R01, R02, R12, R22$
In parallel: {

set:robotPlace(R01_R02)
open:doorR01_R02
}
Context change *robotPlace* = *R01_R02*
Context change *doorR01_R02* = *OPEN*)
set:robotPlace(R02_R01)
Context change *robotPlace* = *R02_R01*
In parallel: {
set:robotPlace(R02_R12)
open:doorR02_R12
}
Context change *robotPlace* = *R02_R12*
Refine plan
//re-invoke open:doorR02_R12, since door not opened despite success response
open:doorR02_R12
Disable open:doorR02_R12
Refine plan: the current plan cannot be augmented
Replan from scratch //guides robot through $R02, R01, R11, R12$
set:robotPlace(R02_R01)
Context change *robotPlace* = *R02_R01*
set:robotPlace(R01_R02)
Context change *robotPlace* = *R01_R02*
set:robotPlace(R01_R11)
Context change *robotPlace* = *R01_R11*
set:robotPlace(R11_R01)
In parallel: {
set:robotPlace(R11_R12)
open:doorR11_R12
}
Context change *robotPlace* = *R11_R12*
Context change *doorR11_R12* = *OPEN*
set:robotPlace(R12_R11)
Context change *robotPlace* = *R12_R11*
set:robotPlace(R12_R22)
Context change *robotPlace* = *R12_R22*
Long timeout for senseRoom22Code2. Disable senseRoom22Code2.
Refine plan: the current plan cannot be augmented
Replan from scratch // not plan can be found

A.2 Representations of WMO BP and respective Planning Domain

A.2.1 BP Representation of the WMO Process

For brevity and clarity reasons, aliases are used instead of the full activity or variable identifiers, i.e., the complete references to service invocation methods, parameters and state variables which reside in the *SR*. For instance, the *decision* activity name is an alias for the full identifier *TenderWCSupplier.12CB.tender- Decision*. Moreover, the declaration of the local process variables that are used for storing the outputs of activities is omitted (e.g. *Temp_hvOut_homeInfo*).

```
<BusinessProcess name="WMO">
<input>
  <parameter name="bpAddress" type="dt:address"/>
  <parameter name="bpCid" type="dt:citInfo"/>
  <parameter name="bpEligCrit" type="dt:lawInfo"/>
  <parameter name="bpMedCond" type="dt:medInfo"/>
</input>

<execute name="intake"
         input="itIn_Cid=bpCid;itIn_address=bpAddress"
         output="tmp_itOut_prov:=itOut_prov"/>

<sequence>
  <repeatUntil>
    <sequence>
      <execute name="homeVisit"
               input="hvIn_Cid=bpCid;hvIn_address=bpAddress"
               output="tmp_hvOut_homeInfo:=hvOut_homeInfo;tmp_hvOut_maRequired:=
                    hvOut_maRequired"/>

      <DS name="DS0">
        <guard>
          <variables>
            <variable name="bpAddress"/>
            <variable name="bpMedCond"/>
          </variables>
          <sequence>
            <switch>
              <case condition="hvOut_maRequired=true">
                <execute name="medicalAdvice"
                         input="maIn_cid=bpCid"
                         output="tmp_maOut_medInfo:=maOut_medInfo"/>
              </case>
              <otherwise>
                <empty/>
              </otherwise>
            </switch>
            <execute name="Decision"
                     input="dcIn_cid=bpCid;dcIn_homeInfo=tmp_hvOut_homeInfo;
                          dcIn_eligCrit=bpEligCrit;dcIn_medInfo=tmp_maOut_medInfo
                          "
```

```
                   output="tmp_dcOut_approvalCheck:=dcOut_approvalCheck"/>
        </sequence>
      </guard>
      <verify>
        <case condition="bpAddress.county!='Groningen'">
          <terminate>
            <achieve-maint>
              <eq-val var="notifiedCityHall" value="TRUE"/>
              <eq-val var="messagePar" value="countyChange"/>
            </achieve-maint>
          </terminate>
        </case>
        <case condition="bpAddress.county='Groningen'">
          <achieve-maint>
            <known var="dcOut_approvalCheck"/>
          </achieve-maint>
        </case>
      </verify>
    </DS>

    <switch name="rejected">
      <case condition="dcOut_approved=false">
        <pick>
          <onMessage variable="appeal">
            <onMessage variable="granted">
              <empty/>
            </onMessage>
            <onMessage variable="notGranted">
              <exit/>
            </onMessage>
          </onMessage>
          <onAlarm><for>'PT14D'</for>
            <exit/>
          </onAlarm>
        </pick>
      </case>
      <otherwise>
        <empty/>
      </otherwise>
    </switch>

  </sequence>
  <condition>dcOut_approved=true</condition>
</repeatUntil>

<switch name="selectProvision">
  <case condition="itOut_prov='care␣in␣kind'">
    <sequence>
      <DS name="DS3">
        <guard>
          <variables>
            <variable name="bpAddress"/>
            <variable name="bpMedCond"/>
           </variables>
          <sequence>
            <execute name="sendOrder"
                     input="sdhrIn_cid=bpCid;sdhrIn_orderInfo=
                         tmp_hvOut_homeInfo;sdhrIn_address;bpAddress"
                     output="orderId:=sdhrOut_orderId;orderContents:=
                         sdhrIn_orderInfo"/>
            <execute name="receiveDeliveryConfirmation"
                     input="dlIn_cid=bpCid;dlIn_id=orderId;dlIn_address=
                         bpAddress;dlIn_delContents=orderContents"
                     output="tmp_dlOut_conf:=dlOut_conf"/>
          </sequence>
        </guard>
        <verify>
          <case condition="bpAddress.county!='Groningen'">
            <terminate>
              <achieve-maint>
                <eq-val var="notifiedCityHall" value="TRUE"/>
                <eq-val var="messagePar" value="countyChange"/>
                <invalid var="orderId"/>
              </achieve-maint>
            </terminate>
          </case>
          <case condition="bpAddress.county='Groningen'␣AND␣bpMedCond!='
              deceased'">
            <achieve-maint>
              <known var="tmp_dlOut_conf"/>
            </achieve-maint>
          </case>
          <case condition="bpMedCond='deceased'">
            <terminate>
              <achieve-maint>
                <invalid var="orderId"/>
              </achieve-maint>
            </terminate>
          </case>
        </verify>
      </DS>
      <execute name="handleInvoice"
               input="hiIn_cid=bpCid;riIn_id=orderId"
               output="tmp_hiOut_invId:=hiOut_invId"/>
    </sequence>
  </case>
  <case condition="itOut_prov='personal␣budget'">
    <empty/>
  </case>
  <otherwise>
    <sequence>
      <DS name="DS1">
        <guard>
          <variables>
            <variable name="bpAddress"/>
            <variable name="bpMedCond"/>
          </variables>
          <sequence>
            <switch>
              <case condition="itOut_prov='wheelchair'">
                <sequence>
                  <execute name="acquireRequirements"
                           input="arIn_cid=bpCid;adIn_homeInfo=
                               tmp_hvout_homeInfo"
                           output="tmp_arOut_requirements:=arOut_requirements"
                               />
                  <execute name="sendOrder"
                           input="soIn_cid=bpCid;soIn_orderInfo=
                                tmp_arOut_requirements;soIn_address=bpAddress"
                           output="orderId:=soOut_orderId;orderContents:=
                                soIn_orderInfo"/>
                </sequence>
              </case>
              <case condition="itOut_prov='home␣modification'">
                <DS name="DS2">
                  <guard>
                    <variables>
                       <variable name="bpEligCrit"/>
                     </variables>
                    <sequence>
                      <repeatUntil>
                        <execute name="tenderProcedure"
                                 input="tpIn_cid=bpCid;tpIn_homeInfo=
                                      tmp_hvOut_homeInfo"
                                 output="tmp_tpOut_tenderSelected:=
                                      tpOut_tenderSelected"/>
                        <execute name="checkTender"
                                 input="ctIn_cid=bpCid;ctIn_selTender=
                                      tmp_tpOut_tenderSelected;ctIn_eligCrit=
                                      bpEligCrit"
                                 output="tmp_ctOut_tenderOK:=ctOut_tenderOK"/>
                        <condition>ctOut_tenderOK=true</condition>
                      </repeatUntil>
                      <execute name="sendOrderConfirmation"
                               input="sosIn_cid=bpCid;sosIn_sid=
                                    tmp_tpOut_tenderSelected;sosIn_orderInfo=
                                    tmp_hvOut_homeInfo;sosIn_address=bpAddress"
                               output="orderId:=sosOut_orderId;orderContents:=
                                    sosIn_orderInfo"/>
                    </sequence>
                  </guard>
                  <verify>
                    <achieve-maint>
                      <known variable="orderId"/>
                    </achieve-maint>
                  </verify>
                </DS>
              </case>
            </switch>
            <execute name="receiveDeliveryConfirmation"
                     input="dlIn_cid=bpCid;dlIn_id=orderId;dlIn_address=
                          bpAddress;dlIn_delContents=orderContents"
                     output="tmp_dlOut_conf:=dlOut_conf"/>
          </sequence>
        </guard>
        <verify>
          <case condition="bpAddress.county!='Groningen'">
            <terminate>
              <achieve-maint>
                <eq-val var="notifiedCityHall" value="TRUE"/>
                <eq-val var="messagePar" value="countyChange"/>
                <invalid var="orderId"/>
              </achieve-maint>
            </terminate>
```

```
            </case>
            <case condition="bpAddress.county='Groningen'␣AND␣bpMedCond!='
                deceased'">
              <achieve-maint>
                <known variable="dlOut_conf"/>
              </achieve-maint>
            </case>
            <case condition="bpMedCond='deceased'">
              <terminate>
                <achieve-maint>
                  <invalid variable="orderId"/>
                </achieve-maint>
              </terminate>
            </case>
          </verify>
        </DS>
        <execute name="handleInvoice"
                 input="hiIn_cid=bpCid;riIn_id=orderId"
                 output="tmp_hiOut_invId:=hiOut_invId"/>
      </sequence>
    </otherwise>
  </switch>
  <execute name="payment"
           input="pmIn_invId=tmp_hiOut_invId"
           output="tmp_pmOut_conf:=pmOut_conf"/>
</sequence>
</BusinessProcess>
```

Variable interdependencies:

$dependsOn(bpAddress) = \{hvOut\_homeInfo\}$

$dependsOn(hvOut\_homeInfo) = \{maOut\_medInfo, dcOut\_approvalCheck,$
$\quad arOut\_requirements, tpOut\_tenderSelection\}$

$dependsOn(tpOut\_tenderSelection) = \{ctOut\_tenderOK\}$

$dependsOn(betokened) = \{maOut\_medInfo, dcOut\_approvalCheck,$
$\quad arOut\_requirements, ctOut\_tenderOK\}$

$dependsOn(bipolarity) = \{ctOut\_tenderOK\}$

A.2.2 Planning Domain modeling the WMO Process

Intake(itIn_cid, itIn_address)
Prec:
$\quad itIn\_cid = bpCid \wedge itIn\_address = bpAddress$
Eff:
$\quad sense(itOut\_prov)$

HomeVisit(hvIn_cid, hvIn_address)
Prec:
$\quad hvIn\_cid = bpCid \wedge hvIn\_address = bpAddress$
$\quad known(itOut\_prov)$
Eff:
$\quad sense(hvOut\_homeInfo) \wedge sense(hvOut\_maRequired)$

MedicalAdvice(maIn_cid)
Prec:
$maIn\_cid = bpCid \ \wedge \ known(hvOut\_maRequired) \wedge$
$hvOut\_maRequired = true \ \wedge \ known(hvOut\_homeInfo)$
Eff:
$sense(maOut\_medInfo)$

Decision(dcIn_cid, dcIn_homeInfo, dcIn_eligCrit, dcIn_medInfo)
Prec:
$dcIn\_homeInfo = hvOut\_homeInfo \ \wedge \ dcIn\_cid = bpCid \wedge$
$(\neg hvOut\_maRequired \ \vee \ known(maOut\_medInfo) \wedge$
$(hvOut\_maRequired \ \vee \ true) \ \wedge \ \neg known(dcOut\_approvalCheck) \wedge$
$(\neg hvOut\_maRequired \ \vee \ dcIn\_medInfo = maOut\_medInfo)$
Eff:
$sense(dcOut\_approvalCheck)$

AcquireRequirements(arIn_cid, arIn_homeInfo)
Prec:
$(itOut\_prov = 3 \ \vee \ itOut\_prov = 4) \ \wedge \ itOut\_prov = 3 \wedge$
$arIn\_cid = bpCid \wedge$
$arIn\_homeInfo = hvOut\_homeInfo \wedge$
$known(dcOut\_approvalCheck) \ \wedge \ dcOut\_approvalCheck = true$
Eff:
$sense(arOut\_requirements)$

TenderProcedure(tpIn_cid, tpIn_homeInfo)
Prec:
$(itOut\_prov = 3 \ \vee \ itOut\_prov = 4) \ \wedge \ itOut\_prov = 4) \wedge$
$tpIn\_cid = bpCid \ \wedge \ tpIn\_homeInfo = hvOut\_homeInfo \wedge$
$known(dcOut\_approvalCheck) \ \wedge \ dcOut\_approvalCheck = true$
Eff:
$sense(tpOut\_tenderSelected)$

CheckTender(ctIn_cid, ctIn_selTender, ctIn_eligCrit)
Prec:
$ctIn\_cid = bpCid \ \wedge \ ctIn\_selTender = tpOut\_tenderSelected, ctIn\_eligCrit = bpEligCrit$
Eff:
$sense(ctOut\_tenderOK) \wedge$
$(ctOut\_tenderOK = false) \ \Rightarrow \ invalidate(tpOut\_tenderSelection)$

SendOrder(soIn_cid, soIn_orderInfo, soIn_address)
Prec:
$soIn\_cid = bpCid \ \wedge \ soIn\_address = bpAddress \wedge$
$known(arOut\_requirements) \ \wedge \ soIn\_orderInfo = arOut\_requirements \wedge$
$\neg known(orderId)$
Eff:
$sense(soOut\_orderId) \ \wedge \ assign(orderId, soOut\_orderId) \wedge$
$assign(orderContents, soIn\_orderInfo)$

SendOrderToSelSupplier(sosIn_cid, sosIn_sid, sosIn_orderInfo, sosIn_address)
Prec:
$sosIn\_cid = bpCid \ \wedge \ sosIn\_sid = tpOut\_tenderSelected \wedge$
$known(ctOut\_tenderOK) \ \wedge \ ctOut\_tenderOK = true \wedge$
$sosIn\_address = bpAddress \ \wedge \ sosIn\_orderInfo = hvOut\_homeInfo \wedge$
$\neg known(orderId)$

Eff:
$sense(sosOut\_orderId) \wedge assign(orderId, sosOut\_orderId) \wedge$
$assign(orderContents, sosIn\_orderInfo)$

SendDHRequest(sdhrIn_cid, sdhrIn_orderInfo, sdhrIn_address)
Prec:
$(itOut\_prov = 1 \vee itOut\_prov = 2) \wedge itOut\_prov = 2) \wedge$
$sdhrIn\_cid = bpCid \wedge sdhrIn\_address = bpAddress \wedge$
$sdhrIn\_orderInfo = hvOut\_homeInfo \wedge known(dcOut\_approvalCheck) \wedge$
$dcOut\_approvalCheck = true \wedge \neg known(orderId)$
Eff:
$sense(sdhrOut\_orderId) \wedge assign(orderId, sdhrOut\_orderId) \wedge$
$assign(orderContents, sdhrIn\_orderInfo)$

DeliveryConfirmation(dlIn_cid, dlIn_id, dlIn_address, dlIn_delContents)
Prec:
$dlIn\_cid = bpCid \wedge dlIn\_id = orderId \wedge$
$dlIn\_delContents = orderContents$
Eff:
$sense(dlOut\_conf)$

ReceiveInvoice(riIn_cid, riIn_id)
Prec:
$riIn\_cid = bpCid \wedge riIn\_id = orderId \wedge$
$known(dlOut\_conf)$
Eff:
$sense(riOut\_invId)$

CheckInvoice(ciIn_invId)
Prec:
$known(riOut\_invId) \wedge ciIn\_invId = riOut\_invId \wedge$
$\neg known(ciOut\_invoiceOK)$
Eff:
$sense(ciOut\_invoiceOK)$

ReturnInvoice(rtiIn_invId)
Prec:
$known(riOut\_invId) \wedge riOut\_invId = rtiIn\_inveId$
$\wedge ciOut\_invoiceOK = false$
Eff:
$invalidate(riOut\_invId) \wedge invalidate(ciOut\_invoiceOK)$

Payment(pmIn_invId)
Prec:
$(\neg(itOut\_prov = 1 \vee itOut\_prov = 2) \vee$
$((\neg itOut\_prov = 1 \vee known(dcOut\_approvalCheck) \wedge$
$(\neg itOut\_prov = 2 \vee known(ciOut\_invoiceOK))))$
$\wedge (\neg(itOut\_prov = 3 \vee itOut\_prov = 4) \vee known(ciOut\_invoiceOK))$
$\wedge pmIn\_invId = riOut\_invId$
Eff:
$sense(pmOut\_conf)$

CancelOrder(coIn_orderId)
Prec:

$known(orderId) \ \wedge \ coIn\_orderId = orderId$

Eff:

$invalidate(orderId)$

notifyCityHall(nchIn_msg)

Prec:

$\varnothing$

Eff:

$sense(nchOut\_sent)$

Bibliography

Agarwal, V., Chafle, G., Dasgupta, K., Karnik, N. M., Kumar, A., Mittal, S. and Srivastava, B., (2005). Synthy: A system for end to end composition of web services, *Jouranl of Web Semantics* 3(4), 311–339.

Agarwal, V., Chafle, G., Mittal, S. and Srivastava, B., (2008). Understanding approaches for web service composition and execution, *in Proc. of the 1st Bangalore Annual Compute Conference*.

Agarwal, V., Dasgupta, K., Karnik, N., Kumar, A., Kundu, A., Mittal, S. and Srivastava, B., (2005). A service creation environment based on end to end composition of web services, *in Proc. of the 14th International Conference on World Wide Web (WWW)*, ACM, pp. 128–137.

Aiello, M., (2006). The role of web services at home, *in Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services (AICT/ICIW)*.

Aiello, M. and Dustdar, S., (2008). A domotic infrastructure based on the web service stack, *Pervasive and Mobile Computing* 4(4), 506–525.

Aiello, M., Papazoglou, M., Yang, J., Carman, M., Pistore, M., Serafini, L. and Traverso, P., (2002). A request language for web-services based on planning and constraint satisfaction, *in VLDB Workshop on Technologies for E-Services (TES)*, Vol. LNCS, Springer, pp. 76–85.

Aker, E., Erdogan, A., Erdem, E. and Patoglu, V., (2011). Causal reasoning for planning and coordination of multiple housekeeping robots, *in Proc. of 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR2011)*, pp. 311–316.

Akkiraju, R., Srivastava, B., Ivan, A.-A., Goodwin, R. and Syeda-Mahmood, T. F., (2006). SEMAPLAN: Combining planning with semantic matching to achieve web service composition, *in Proc. of the 21st National Conference on Artificial Intelligence and the 18th Innovative Applications of AI Conference (IAAI).*

Alamri, A., Eid, M. A. and El-Saddik, A., (2006). Classification of the state-of-the-art dynamic web services composition techniques, *International Journal of Web and Grid Services* 2(2).

Albore, A., Palacios, H. and Geffner, H., (2009). A translation-based approach to contingent planning, *in Proc. of the 21st International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1623–1628.

Albore, A., Ramírez, M. and Geffner, H., (2011). Effective heuristics and belief tracking for planning with incomplete information, *in Proc. of the 21st International Conference on Automated Planning and Scheduling (ICAPS).*

Aloise, F., Schettini, F., Aricò, P., Bianchi, L., Riccio, A., Mecella, M., Babiloni, F., Mattia, D. and Cincotti, F., (2010). Advanced brain computer interface for communication and control, *in Proc. of the International Conference on Advanced Visual Interfaces*, ACM, pp. 399–400.

Aloise, F., Schettini, F., Aricò, P., Salinari, S., Guger, C., Rinsma, J., Aiello, M., Mattia, D. and Cincotti, F., (2011). Asynchronous P300-based BCI to control a virtual environment: initial tests on end users, *Clinical EEG and Neuroscience* 42(4), 1–6.

Au, T., Kuter, U. and Nau, D., (2005). Web service composition with volatile information, *in 4th International Semantic Web Conference (ISWC)*, pp. 52–66.

Baligand, F., Rivierre, N. and Ledoux, T., (2007). A declarative approach for QoS-aware web service compositions, *in Proc. of the 5th International Conference in Service-Oriented Computing (ICSOC)*, pp. 422–428.

Barták, R., (2011). A novel constraint model for parallel planning, *in Proc. of the 24th International Florida Artificial Intelligence Research Society Conference (FLAIRS).*

Barták, R., Salido, M. A. and Rossi, F., (2010). New trends in constraint satisfaction, planning, and scheduling: a survey, *Knowledge Eng. Review* 25(3), 249–279.

Barták, R. and Toropila, D., (2008). Reformulating constraint models for classical planning, *in Proc. of the 21st International Florida Artificial Intelligence Research Society Conference (FLAIRS)*, pp. 525–530.

Barták, R. and Toropila, D., (2009). Enhancing constraint models for planning problems, *in Proc. of the 22nd International Florida Artificial Intelligence Research Society Conference (FLAIRS)*, AAAI Press.

Beauche, S. and Poizat, P., (2008). Automated service composition with adaptive planning, *in Proc. of the 6th International Conference on Service Oriented Computing*, pp. 530–537.

Beckstein, C. and Klausner, J., (1999). A meta level architecture for workflow management, *Journal of Integrated Design and Process Science* 3, 15–26.

Berardi, D., Calvanese, D., Giacomo, G. D., Hull, R. and Mecella, M., (2005). Automatic composition of transition-based semantic web services with messaging, *in Proc. of the 31st International Conference on Very Large Data Bases*, pp. 613–624.

Berardi, D., Calvanese, D., Giacomo, G. D., Lenzerini, M. and Mecella, M., (2003). Automatic composition of e-services that export their behavior, *in Proc. of the 1st International Conference on Service-Oriented Computing (ICSOC)*, pp. 43–58.

Berardi, D., Calvanese, D., Giacomo, G. D., Lenzerini, M. and Mecella, M., (2005). Automatic service composition based on behavioral descriptions, *International J. Cooperative Inf. Syst.* 14(4), 333–376.

Berardi, D., Calvanese, D., Giacomo, G. D. and Mecella, M., (2005). Composition of services with nondeterministic observable behavior, *in Proc. of the 3rd International Conference on Service-Oriented Computing (ICSOC)*, pp. 520–526.

Berardi, D., Cheikh, F., Giacomo, G. D. and Patrizi, F., (2008). Automatic service composition via simulation, *International Journal of Foundations of Computer Science* 19(2), 429–451.

Berardi, D., Giacomo, G. D., Lenzerini, M., Mecella, M. and Calvanese, D., (2004). Synthesis of underspecified composite *e*-services based on automated reasoning, *in Proc. of the 2nd International Conference on Service-Oriented Computing*, pp. 105–114.

Bertoli, P. and Cimatti, A., (2002). Improving heuristics for planning as search in belief space, *in Proc. of the 6th International Conference on Artificial Intelligence Planning Systems (AIPS)*, pp. 143–152.

Bertoli, P., Kazhamiakin, R., Paolucci, M., Pistore, M., Raik, H. and Wagner, M., (2009). Continuous orchestration of web services via planning, *in Proc. of the 19th International Conference on Automated Planning and Scheduling (ICAPS)*.

Bertoli, P., Pistore, M. and Traverso, P., (2006). Automated web service composition by on-the-fly belief space search, *in Proc. of the 16th International Conference on Automated Planning and Scheduling*, pp. 358–361.

Bertoli, P., Pistore, M. and Traverso, P., (2010). Automated composition of web services via planning in asynchronous domains, *Artificial Intelligence* 174, 316–361.

Borrajo, D. and Veloso, M., (2012). Probabilistically reusing plans in deterministic planning, *in Proc. of ICAPS-12 Workshop on on Heuristics and Search for Domain-Independent Planning*, pp. 17–25.

Brenner, M. and Nebel, B., (2009). Continual planning and acting in dynamic multiagent environments, *Autonomous Agents and Multi-Agent Systems* 19(3), 297–331.

Bronsted, J., Hansen, K. M. and Ingstrup, M., (2010). Service composition issues in pervasive computing, *IEEE Pervasive Computing* 9, 62–70.

Bryce, D., Kambhampati, S. and Smith, D. E., (2006). Planning graph heuristics for belief space search, *Journal of Artificial Intelligence Research (JAIR)* 26, 35–99.

Bucchiarone, A., Marconi, A., Pistore, M. and Raik, H., (2012). Dynamic adaptation of fragment-based and context-aware business processes, *in Proc. of the 19th IEEE International Conference on Web Services (ICWS)*, pp. 33–41.

Bucchiarone, A., Pistore, M., Raik, H. and Kazhamiakin, R., (2011). Adaptation of service-based business processes by context-aware replanning, *in Proc. of the 2011 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, IEEE, pp. 1–8.

Bultan, T., Fu, X., Hull, R. and Su, J., (2003). Conversation specification: a new approach to design and analysis of e-service composition, *in Proc. of the 12th International World Wide Web Conference (WWW)*, pp. 403–410.

Cabezas, P., Arrizabalaga, S., Salterain, A. and Legarda, J., (2008). An agent-based semantic OSGi service architecture, *in Computer and Information Science*, Vol. 131 of *Studies in Computational Intelligence*, Springer Berlin / Heidelberg, pp. 97–106.

Cardoso, J. and Sheth, A. P., eds, (2006). *Semantic Web Services, Processes and Applications*, Springer.

Caruso, M., Ciccio, C. D., Iacomussi, E., Kaldeli, E., Lazovik, A. and Mecella, M., (2012). Service ecologies for home/building automation, *in Poc. of the 10th IFAC Symposium on Robot Control*.

Catarci, T., Di Ciccio, C., Forte, V., Iacomussi, E., Mecella, M., Santucci, G. and Tino, G., (2011). Service composition and advanced user interfaces in the home of tomorrow: the SM4All approach, *in Proc. of 2nd International ICST Conference on Ambient Media and Systems (AMBI-SYS)*.

Chan, M., Bishop, J. and Baresi, L., (2007). Survey and comparison of planning techniques for web services composition. university of pretoria, Technical report, Polelo Research Group Department of Computer Science, University of Pretoria.

Cimatti, A., Pistore, M., Roveri, M. and Traverso, P., (2003). Weak, strong, and strong cyclic planning via symbolic model checking, *Artificial Intelligence* 147(1-2), 35–84.

Costa, P., Coulson, G., Mascolo, C., Mottola, L., Picco, G. and Zachariadis, S., (2007). Reconfigurable component-based middleware for networked embedded systems, *International Journal of Wireless Information Networks* 14, 149–162.

Dadam, P. and Reichert, M., (2009). The ADEPT project: a decade of research and development for robust and flexible process support, *Computer Science - - Research and Development* 23, 81–97.

Davidsson, P. and Boman, M., (2005). Distributed monitoring and control of office buildings by embedded agents, *Information Sciences* 171, 293–307.

de Leoni, M., De Giacomo, G., Lespèrance, Y. and Mecella, M., (2009). On-line adaptation of sequential mobile processes running concurrently, *in Proc. of the 2009 ACM Symposium on Applied Computing*, SAC'09, ACM, pp. 1345–1352.

de Leoni, M., Mecella, M. and De Giacomo, G., (2007). Highly dynamic adaptation in process management systems through execution monitoring, *BPM 2007* pp. 182–197.

Do, M. B. and Kambhampati, S., (2001). Planning as constraint satisfaction: Solving the planning-graph by compiling it into csp, *Artificial Intelligence* 132, 151–182.

Dominici, M., Frjus, M., Guibourdenche, J., Pietropaoli, B. and Weis, F., (2011). Towards a system architecture for recognizing domestic activity by leveraging a naturalistic human activity model, *in Workshop on Goal, Activity and Plan*

Recognition (GAPREC) at the International Conference on Automated Planning and Scheduling (ICAPS).

Duquennoy, S., Grimaud, G. and Vandewalle, J.-J., (2009). The web of things: Interconnecting devices with high usability and performance, *in Proc. of the 6th International Conference on Embedded Software and Systems (ICESS)*, IEEE Computer Society, pp. 323–330.

Dustdar, S. and Schreiner, W., (2005). A survey on web services composition, *International Journal of Web and Grid Services* 1(1), 1–30.

Edelkamp, S. and Hoffmann, J., (2004). PDDL2.2: The language for the classic part of the 4th international planning competition, Technical Report 195, Institut für Informatik, Freiburg, Germany.

Eid, M. A., Alamri, A. and El-Saddik, A., (2008). A reference model for dynamic web service composition systems, *International Journal of Web and Grid Services* 4(2), 149–168.

Eisenhauer, M., Rosengren, P. and Antolin, P., (2010). HYDRA: A development platform for integrating wireless devices and sensors into ambient intelligence systems, *in The Internet of Things*, The Internet of Things: 20th Tyrrhenian Workshop on Digital Communications, Springer, pp. 367–373.

Erol, K., Nau, D. S. and Subrahmanian, V. S., (1995). Complexity, decidability and undecidability results for domain-independent planning, *Artificial Intelligence* 76(1-2), 75–88.

Etzioni, Z., Keeney, J., Brennan, R. and Lewis, D., (2010). Supporting composite smart home services with semantic fault management, *in Proc. of the 5th International Conference on Future Information Technology (FutureTech)*, pp. 1 –8.

Fan, J. and Kambhampati, S., (2005). A snapshot of public web services, *SIGMOD Record* 34(1), 24–32.

Fargier, H., Lang, J., Lang, J. M. and Schiex, T., (1996). Mixed constraint satisfaction: A framework for decision problems under incomplete knowledge, *in Proc. of the 13th AAAI Conference on Artificial Intelligence*, pp. 175–180.

Ferreira, H. and Ferreira, D., (2006). An integrated life cycle for workflow management based on learning and planning, *International Journal of Cooperative Information Systems* 15, 485–505.

Fikes, R. and Nilsson, N. J., (1971). Strips: A new approach to the application of theorem proving to problem solving, *Artificial Intelligence* 2(3/4), 189–208.

Finger, J., (1987). Exploiting constraints in design synthesis, PhD thesis, Stanford University.

Foss, J., Onder, N. and Smith, D., (2007). Preventing unrecoverable failures through precautionary planning, *in Proc. of ICAPS-07 Workshop on Moving Planning and Scheduling Systems into the Real World*.

Fox, M., Gerevini, A., Long, D. and Serina, I., (2006). Plan stability: Replanning versus plan repair, *in Proc. of the 16th International Conference on Automated Planning and Scheduling*, pp. 212–221.

Fox, M. and Long, D., (2003). Pddl2.1: An extension to PDDL for expressing temporal planning domains, *Journal of Artificial Intelligence Research (JAIR)* 20, 61–124.

Friedrich, G., Fugini, M., Mussi, E., Pernici, B. and Tagni, G., (2010). Exception handling for repair in service-based processes, *IEEE Transactions on Software Engineering* 36(2), 198–215.

Gent, I. P., Miguel, I. and Rendl, A., (2007). Tailoring solver-independent constraint models: A case study with essence' and minion, *in Proc. of the 7th International Symposium in Abstraction, Reformulation, and Approximation (SARA)*.

Gerevini, A. and Long, D., (2006). Preferences and soft constraints in PDDL3, *in 16th ICAPS Workshop on Planning with Preferences and Soft Constraints*.

Gerevini, A., Saetti, A. and Serina, I., (2006). An approach to temporal planning and scheduling in domains with predictable exogenous events, *Journal of Artificial Intelligence Research (JAIR)* 25(1), 187–231.

Ghallab, M., Nau, D. and Traverso, P., (2004). *Automated Planning: Theory and Practice*, Morgan Kaufmann, Amsterdam.

Giacomo, G. D., Ciccio, C. D., Felli, P., Hu, Y. and Mecella, M., (2012). Goal-based composition of stateful services for smart homes, *in Proc. of 20th International Conference on Cooperative Information Systems (CoopIS)*.

Göbelbecker, M., Gretton, C. and Dearden, R., (2011). A switching planner for combined task and observation planning, *in Proc. of the 25th AAAI Conference on Artificial Intelligence*.

Golden, K., (1998). Leap before you look: Information gathering in the puccini planner, *in Proc. of the 6th International Conference on Artificial Intelligence Planning Systems (AIPS)*, pp. 70–77.

Golden, K., (2003). A domain description language for data processing, *in ICAPS Workshop on the Future of PDDL.*

Golden, K., Etzioni, O. and Weld, D., (1996). Planning with execution and incomplete information, Technical Report 97-11-05, UW CSE.

Golden, K. and Pang, W., (2004). A constraint-based planner applied to data processing domains, *in Proc. of the 10th International Conference on Principles and Practice of Constraint Programming (CP)*, p. 815.

Golden, K. and Weld, D. S., (1996). Representing sensing actions: The middle ground revisited, *in Proc. of the 5th International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pp. 174–185.

Gomaa, H., Hashimoto, K., Kim, M., Malek, S. and Menascé, D. A., (2010). Software adaptation patterns for service-oriented architectures, *in Proc. of the 25th Symposium on Applied Computing*, ACM, pp. 462–469.

Göser, K., Jurisch, M., Acker, H., Kreher, U., Lauer, M., Rinderle, S., Reichert, M. and Dadam, P., (2007). Next-generation process management with ADEPT2, *in Proc. of the Software Demonstrations of the 5th International Conference on Business Process Management (BPM).*

Gouvas, P., Bouras, T. and Mentzas, G., (2007). An OSGi-based semantic service-oriented device architecture, *in OTM Workshops - International Conference on On the move to meaningful internet systems*, pp. 773–782.

Grau, B. C., Parsia, B. and Sirin, E., (2004). Working with multiple ontologies on the semantic web, *in Proc. of the 3d International Semantic Web Conference*, pp. 620–634.

Gravot, F., Haneda, A., Okada, K. and Inaba, M., (2006). Cooking for humanoid robot, a task that needs symbolic and geometric reasonings, *in Proc. of the 2006 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 462 –467.

Gregory, P., Long, D. and Fox, M., (2010). Constraint based planning with composable substate graphs, *in Proc. of the 19th European Conference on Artificial Intelligence (ECAI)*, pp. 453–458.

Greif, I., (1975). Semantics of Communicating Parallel Processes, PhD thesis, MIT.

Guettier, C. and Yorke-Smith, N., (2005). Enhancing the anytime behaviour of mixed csp-based planning, *in Proc. of ICAPS Workshop on Planning under Uncertainty for Autonomous Systems*, pp. 29–38.

Guger, C., Daban, S., Sellers, E. Holzner, C., Krausz, G. Carabalona, R., Gramatica, F. and Edlinger, G., (2009). How many people are able to control a P300-based brain-computer interface (BCI)?, *Neuroscience Letters* 462, 94–98.

Guinard, D., Trifa, V., Mattern, F. and Wilde, E., (2011). From the internet of things to the web of things: Resource oriented architecture and best practices, *in Architecting the Internet of Things*, Springer, chapter 5, pp. 97–129.

Hassine, A. B., Matsubara, S. and Ishida, T., (2006). A constraint-based approach to horizontal web, *in Proc. of the 5th International Semantic Web Conference (ISWC2006)*, pp. 130–143.

Hatzi, O., Vrakas, D., Bassiliades, N., Anagnostopoulos, D. and Vlahavas, I. P., (2010). Semantic awareness in automated web service composition through planning, *in 6th Hellenic Conference on Artificial Intelligence (SETN)*, pp. 123–132.

Helmert, M., (2006). The fast downward planning system, *Jourlan of Artificial Intelligence Research (JAIR)* 26, 191–246.

Helmert, M., (2009). Concise finite-domain representations for PDDL planning tasks, *Artificial Intelligence* 173, 503–535.

Henneberger, M., Heinrich, B., Lautenbacher, F. and Bauer, B., (2008). Semantic-based planning of process models, *in Multikonferenz Wirtschaftsinformatik (MKWI)*.

Hewitt, C., Bishop, P. and Steiger, R., (1973). A universal modular actor formalism for artificial intelligence, *in Proc. of the 3rd International Joint Conference on Artificial Intelligence (IJCAI)*.

Hoffmann, J., (2008). Towards efficient belief update for planning-based web service composition, *in 18th European Conference on Artificial Intelligence (ECAI)*, pp. 558–562.

Hoffmann, J., Bertoli, P., Helmert, M. and Pistore, M., (2009). Message-based web service composition, integrity constraints, and planning under uncertainty: A new connection, *Journal of Artificial Intelligence Research (JAIR)* 35, 49–117.

Hoffmann, J., Bertoli, P. and Pistore, M., (2007). Web service composition as planning, revisited: In between background theories and initial state uncertainty, *in Proc. of the 21st AAAI Conference on Artificial Intelligence*, pp. 1013–1018.

Hoffmann, J. and Brafman, R. I., (2005). Contingent planning via heuristic forward search witn implicit belief states, *in Proc. of the 15th International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 71–80.

Hoffmann, J. and Nebel, B., (2001). The FF planning system: Fast plan generation through heuristic search, *Journal of Artificial Intelligence Research (JAIR)* 14, 253–302.

Hoffmann, J., Weber, I. and Kraft, F., (2010). SAP speaks PDDL, *in Proc. of the 4th AAAI Conference on Artificial Intelligence.*

Hull, R., Benedikt, M., Christophides, V. and Su, J., (2003). E-services: a look behind the curtain, *in Proc. of the 22nd ACM Symposium on Principles of Database Systems (PODS)*, pp. 1–14.

Hyafil, N. and Bacchus, F., (2003). Conformant probabilistic planning via CSPs, *in Proc. of the 13th International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 205–214.

Hyafil, N. and Bacchus, F., (2004). Utilizing structured representations and CSPs in conformant probabilistic planning, *in Proc. of the 16th Eureopean Conference on Artificial Intelligence (ECAI)*, pp. 1033–1034.

Iocchi, L., Lukasiewicz, T., Nardi, D. and Rosati, R., (2009). Reasoning about actions with sensing under qualitative and probabilistic uncertainty, *ACM Transactions on Computational Logic* 10(1).

Jarvis, P., Moore, J., Stader, J., Macintosh, A., Casson-du Mont, A. and Chung, P., (1999). Exploiting ai technologies to realise adaptive workflow systems, *in AAAI Workshop on Agent-Based Systems in the Business Context.*

Kaldeli, E., Lazovik, A. and Aiello, M., (2009*a*). Extended goals for composing services, *in Proc. of the 19th International Conference on Automated Planning and Scheduling (ICAPS 2009)*, AAAI Press.

Kaldeli, E., Lazovik, A. and Aiello, M., (2009*b*). Planning in a smart home: Visualization and simulation., *in Application Showcase Proc. of the 19th International Conference on Automated Planning and Scheduling (ICAPS).*

Kaldeli, E., Lazovik, A. and Aiello, M., (2011). Continual planning with sensing for web service composition, *in Proc. of the 25th AAAI Conference on Artificial Intelligence*, AAAI Press.

Kaldeli, E., Warriach, E., Lazovik, A. and Aiello, M., (2013). Coordinating the web of services for a smart home, *ACM Transactions on the Web* . To appear.

Kaldeli, E., Warriach, E. U., Bresser, J., Lazovik, A. and Aiello, M., (2010). Interoperation, composition and simulation of services at home, *in 8th International Conference on Service Oriented Computing (ICSOC)*, Vol. LNCS 6470, Springer, pp. 167–181.

Kambhampati, S., (2000). Planning graph as a (dynamic) CSP: Exploiting EBL, DDB and other CSP search techniques in graphplan, *Journal of Artificial Intelligence Research* 12, 1–34.

Kambhampati, S., (2007). Model-lite planning for the web age masses: The challenges of planning with incomplete and evolving domain models, *in Proc. of the 22nd AAAI Conference on Artificial Intelligence.*

Kastner, W., Kofler, M. J. and Reinisch, C., (2010). Using AI to realize energy efficient yet comfortable smart homes, *in Proc. of 8th IEEE International Workshop on Factory Communication Systems (WFCS '10)*, pp. 169–172.

Kim, S. H., Kim, S. W. and Park, H., (2003). Usability challenges in ubicomp environment, *in Proc. of the International Ergonomics Association (IEA).*

Kindberg, T., Barton, J., Morgan, J., Becker, G., Caswell, D., Debaty, P., Gopal, G., Frid, M., Krishnan, V., Morris, H., Schettino, J. and Serra, B., (2000). People, places, things: Web presence for the real world, *in 3rd IEEE Workshop on Mobile Computing Systems and Applications (WMSCA)*, pp. 365–376.

Klusch, M. and Gerber, A., (2005). Semantic web service composition planning with owls-xplan, *in Proc. of the 1st International AAAI Fall Symposium on Agents and the Semantic Web*, pp. 55–62.

Klusch, M. and Gerber, A., (2006). Fast composition planning of OWL-S services and application, *in 4th IEEE European Conference on Web Services (ECOWS)*, pp. 181–190.

Klusch, M. and Renner, K.-U., (2006). Fast dynamic re-planning of composite OWL-S services, *in International Conference on Intelligent Agent Technology (IAT) - Workshops*, pp. 134–137.

Knoblock, C. A., (1995). Planning, executing, sensing, and replanning for information gathering, *in International Joint Conference of Artificial Intelligence (IJ-CAI)*, pp. 1686–1693.

Kopp, O., Martin, D., Wutke, D. and Leymann, F., (2008). On the choice between graph-based and block-structured business process modeling languages, *in Modellierung betrieblicher Informationssysteme (MobIS 2008)*, Vol. 141 of *Lecture Notes in Informatics (LNI)*, Gesellschaft für Informatik e.V. (GI), pp. 59–72.

Krause, C., Maraikar, Z., Lazovik, A. and Arbab, F., (2011). Modeling dynamic reconfigurations in reo using high-level replacement systems, *Science of Computer Programming* 76(1), 23–36.

Kuter, U. and Golbeck, J., (2009). Semantic web service composition in social environments, *in International Semantic Web Conference*, pp. 344–358.

Kuter, U., Sirin, E., Parsia, B., Nau, D. S. and Hendler, J. A., (2005). Information gathering during planning for web service composition, *J. Web Sem.* 3(2-3), 183–205.

Kster, U., Stern, M. and Knig-Ries, B., (2005). A classification of issues and approaches in automatic service composition, *in First International Workshop on Engineering Service Compositions at ICSOC-05*.

Laborie, P., (2003). Algorithms for propagating resource constraints in AI planning and scheduling: existing approaches and new results, *Artificial Intelligence* 143(2), 151–188.

Lazovik, A., Aiello, M. and Gennari, R., (2005). Encoding requests to web service compositions as constraints, *in 11th International Conference on Principles and Practice of Constraint Programming (CP)*.

Lazovik, A., Aiello, M. and Papazoglou, M., (2003). Planning and monitoring the execution of web service requests, *in Proc. of the 1st International Conference on Service-Oriented Computing (ICSOC-03)*, Springer, p. 335350.

Lazovik, A., Aiello, M. and Papazoglou, M., (2006). Planning and monitoring the execution of web service requests, *Journal on Digital Libraries* .

Lazovik, A. and Arbab, F., (2007). Using reo for service coordination, *in Proc. of the 5th International Conference in Service-Oriented Computing (ICSOC)*, pp. 398–403.

Lazovik, E., den Dulk, P., de Groote, M., Lazovik, A. and Aiello, M., (2009). Services inside the smart home: A simulation and visualization tool, *in Demo Session of the 7th International Conference on Service-Oriented Computing (ICSOC-ServiceWave)*, Vol. 5900 of *LNCS*, Springer, pp. 651–652.

Lee, C., Ko, S., Kim, E. and Lee, W., (2009). Enriching OSGi service composition with web services, *IEICE Transactions on Information and Systems* E92.D(5), 1177–1180.

Li, G., Muthusamy, V. and Jacobsen, H.-A., (2010). A distributed service-oriented architecture for business process execution, *ACM Trans. on the Web* 4, 2:1–2:33.

Li, Q., Stankovic, J. A., Hanson, M. A., Barth, A. T., Lach, J. and Zhou, G., (2009). Accurate, fast fall detection using gyroscopes and accelerometer-derived posture information, *Wearable and Implantable Body Sensor Networks, International Workshop on* 0, 138–143.

Liaskos, S., Khan, S. M., Litoiu, M., Jungblut, M. D., Rogozhkin, V. and Mylopoulos, J., (2012). Behavioral adaptation of information systems through goal models, *Information Systems* 37(8), 767–783.

Lin, N., Kuter, U. and Hendler, J. A., (2007). Web service composition via problem decomposition across multiple ontologies, *in IEEE International Conference on Services Computing, 4th International Workshop on Semantic Web for Services and Processes (SWSP)*, pp. 65–72.

Lin, N., Kuter, U. and Sirin, E., (2008). Web service composition with user preferences, *in Proc. of the 5th European Semantic Web Conference (ESWC)*, pp. 629–643.

Lopez, A. and Bacchus, F., (2003). Generalizing graphplan by formulating planning as a CSP, *in Proc. of the 18th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 954–960.

Madhusudan, T. and Uttamsingh, N., (2006). A declarative approach to composing web services in dynamic environments, *Decision Support Systems* 41(2), 325–357.

Madhusudan, T., Zhao, J. L. and Marshall, B., (2004). A case-based reasoning framework for workflow model management, *Data and Knowledge Engineering* 50, 87–115.

Marconi, A., Pistore, M. and Traverso, P., (2006). Specifying data-flow requirements for the automated composition of web services, *in 4th IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, pp. 147–156.

Marrella, A. and Mecella, M., (2011). Continuous planning for solving business process adaptivity, *in 12th International Working Conference on Business Process Modeling, Development and Support (BPMDS).*

Martínez, E. and Lespérance, Y., (2004). Web service composition as a planning task: Experiments using knowledge-based planning, *in Proc. of the ICAPS Workshop on Planning and Scheduling for Web and Grid Services.*

Martínez, M., Ferníndez, F. and Borrajo, D., (2012). Variable resolution planning through predicate relaxation, *in Proc. of ICAPS-12 Workshop on Planning and Plan Execution for Real-World Systems: Principles and Practices (PlanEx).*

McDermott, D. and the AIPS-98 Planning Competition Committee, (1998). PDDL: the planning domain definition language, Technical Report CVC TR98003/DCS TR1165. URL: *www.cs.yale.edu/homes/dvm*

McDermott, D. V., (2002). Estimated-regression planning for interactions with web services, *in Proc. of the 6th International Conference on Artificial Intelligence Planning Systems*, pp. 204–211.

McIlraith, S. A., (2004). Towards declarative programming for web services, *in 11th International Symposium on Static Analysis (SAS)*, p. 21.

McIlraith, S. and Son, T. C., (2002). Adapting golog for composition of semantic web-services, *in* D. Fensel, F. Giunchiglia, D. McGuinness and M. Williams, eds, *Conference on Principles of Knowledge Representation (KR).*

Mediratta, A. and Srivastava, B., (2006). Applying planning in composition of web services with a user-driven contingent planner, Technical report, IBM Research Report RI 06002.

Müller, R., Greiner, U. and Rahm, E., (2004). Agentwork: a workflow system supporting rule-based workflow adaptation, *Data and Knowledge Engineering* 51, 223–256.

Murugan, S. and Ramachandran, V., (2012). Aspect oriented decision making model for byzantine agreement, *Journal of Computer Science* 8, 382–388.

Nebel, B. and Koehler, J., (1995). Plan reuse versus plan generation: A theoretical and empirical analysis, *Artificial Intelligence* 76(1-2), 427–454.

Nielsen, J., (1994*a*). Enhancing the explanatory power of usability heuristics, *in Proc. of the SIGCHI Conference on Human factors in computing systems: celebrating interdependence*, ACM, pp. 152–158.

Nielsen, J., (1994*b*). *Heuristic evaluation. Usability inspection methods*, John Wiley and Sons.

Oh, S.-C., Lee, D. and Kumara, S. R. T., (2007). Web service planner (WSPR): An effective and scalable web service composition algorithm, *International Journal of Web Services Research* 4(1), 1–22.

Orriëns, B. and Yang, J., (2006). A rule driven approach for developing adaptive service oriented business collaboration, *in Proc. of the 2007 IEEE International Conference on Services Computing*, pp. 182–189.

Ouvans, C., Dumas, M., ter Hofstede, A. and van der Aalst, W., (2006). From bpmn process models to bpel web services, *in International Conference on Web Services*, pp. 285–292.

Palacios, H. and Geffner, H., (2009). Compiling uncertainty away in conformant planning problems with bounded width, *Journal of Artificial Intelligence Research (JAIR)* 35, 623–675.

Papapanagiotou, P. and Fleuriot, J. D., (2011). A theorem proving framework for the formal verification of web services composition, *in 7th International Workshop on Automated Specification and Verification of Web Systems (WWV)*, pp. 1–16.

Papazoglou, M., Aiello, M., Pistore, M. and Yang, J., (2002). XSRL: A request language for web services, *IEEE Internet Computing* .

Pecora, F. and Cesta, A., (2007). DCOP for Smart Homes: a Case Study, *Computational Intelligence* 23(4), 395–419.

Peer, J., (2004). A PDDL based tool for automatic web service composition, *in 2nd International Workshop on the Principles and Practice of Semantic Web Reasoning*, Vol. 3208 of *Lecture Notes in Computer Science*, Springer.

Peer, J., (2005*a*). A POP-based replanning agent for automatic web service composition, *in 2nd European Semantic Web Conference (ESWC)*, pp. 47–61.

Peer, J., (2005*b*). Web service composition as AI planning - a survey, Technical report, University of St. Gallen, Switzerland.

Petrick, R. P. A., (2011). An extension of knowledge-level planning to interval-valued functions, *in AAAI Workshop on Generalized Planning.*

Petrick, R. P. A. and Bacchus, F., (2004). Extending the knowledge-based approach to planning with incomplete information and sensing, *in Proc. of the 14th International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 2–11.

Pilioura, T. and Tsalgatidou, A., (2009). Unified publication and discovery of semantic web services, *ACM Transactions on the Web* 3(3), 11:1–11:44.

Pistore, M., Marconi, A., Bertoli, P. and Traverso, P., (2005). Automated composition of web services by planning at the knowledge level, *in 19th International Joint Conference on Artificial Intelligence*, pp. 1252–1259.

Pistore, M., Spalazzi, L. and Traverso, P., (2006). A minimalist approach to semantic annotations for web processes compositions, *in Proc. of the 3rd European Semantic Web Conference (ESWC)*, pp. 620–634.

Pistore, M., Traverso, P. and Bertoli, P., (2005). Automated composition of web services by planning in asynchronous domains, *in Proc. of the 15th International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 2–11.

Pryor, L. and Collins, G., (1996). Planning for contingencies: A decision-based approach, *Journal of Artificial Intelligence Research (JAIR)* 4, 287–339.

Rao, J., Dimitrov, D., Hofmann, P. and Sadeh, N. M., (2006). A mixed initiative approach to semantic web service discovery and composition: SAP's guided procedures framework, *in Proc. of the 5th International Semantic Web Conference (ISWC)*, pp. 401–410.

Rao, J., Küngas, P. and Matskin, M., (2006). Composition of semantic web services using linear logic theorem proving, *Information Systems* 31(4-5), 340–360.

Rao, J. and Su, X., (2004). A survey of automated web service composition methods, *in 1st International Workshop on Semantic Web Services and Web Process Composition*, pp. 43–54.

Rao, S. P. and Cook, D. J., (2004). Predicting inhabitant action using action and task models with application to smart homes, *International Journal on Artificial Intelligence Tools* 13, 81–100.

Redondo, R. P. D., Fernandez, V. A., Cabrer, M. R., Arias, J. J. P., Duque, J. G. and Solla, A. G., (2008). Enhancing residential gateways: A semantic OSGi platform, *IEEE Intelligent Systems* 23(1), 32–40.

Reichert, M. and Dadam, P., (2009). Enabling adaptive process-aware information systems with ADEPT2, *in* J. Cardoso and W. Van Der Aalst, eds, *Handbook of Research on Business Process Modeling*, Information Science Reference, Hershey, New York, pp. 173–203.

Richter, S. and Westphal, M., (2010). The LAMA planner: Guiding cost-based anytime planning with landmarks, *Journal of Artificial Intelligence Research (JAIR)* 39, 127–177.

Rodríguez-Moreno, M. D., Borrajo, D., Cesta, A. and Oddi, A., (2007). Integrating planning and scheduling in workflow domains, *Expert Systems and Applications* 33(2), 389–406.

Rodríguez-Moreno, M. D. and Kearney, P., (2002). Integrating AI planning techniques with workflow management system, *Knowledge-Based Systems* 15(5-6), 285–291.

Ryu, S. H., Casati, F., Skogsrud, H., Benatallah, B. and Saint-Paul, R., (2008). Supporting the dynamic evolution of web service protocols in service-oriented architectures, *ACM Transactions on the Web* 2, 13:1–13:46.

Sardiña, S., Patrizi, F. and Giacomo, G. D., (2007). Automatic synthesis of a global behavior from multiple distributed behaviors, *in Proc. of the 22nd AAAI Conference on Artificial Intelligence*, pp. 1063–1069.

Shani, G. and Brafman, R. I., (2011). Replanning in domains with partial information and sensing actions, *in Proc. of the 22nd International Joint Conference on Artificial Intelligence*, pp. 2021–2026.

Shaparau, D., Pistore, M. and Traverso, P., (2006). Contingent planning with goal preferences, *in Proc. of the 21st AAAI Conference on Artificial intelligence*, AAAI Press, pp. 927–934.

Shaparau, D., Pistore, M. and Traverso, P., (2008). Fusing procedural and declarative planning goals for nondeterministic domains, *in Proc. of the 23d AAAI Conference on Artificial Intelligence*, pp. 983–990.

Sheshagiri, M., DesJardins, M. and Finin, T., (2003). A planner for composing services described in DAML-S, *in ICAPS Workshop on planning for web services.*

Simpson, R. C., Schreckenghost, D., LoPresti, E. F. and Kirsch, N., (2006). Plans and planning in smart homes, *in Designing Smart Homes*, pp. 71–84.

Sirbu, A. and Hoffmann, J., (2008). Towards scalable web service composition with partial matches, *in Proc. of the IEEE International Conference on Web Services*, pp. 29–36.

Sirin, E., Hendler, J. A. and Parsia, B., (2003). Semi-automatic composition of web services using semantic descriptions, *in Proc. of the 1st Workshop on Web Services: Modeling, Architecture and Infrastructure (WSMAI)*, pp. 17–24.

Sirin, E., Parsia, B., Wu, D., Hendler, J. A. and Nau, D. S., (2004). Htn planning for web service composition using shop2, *Journal of Web Semantics* 1(4), 377–396.

Skoutas, D., Sacharidis, D., Simitsis, A. and Sellis, T., (2008). Serving the Sky: Discovering and selecting semantic web services through dynamic skyline queries, *in 2nd IEEE International Conference on Semantic Computing*, pp. 222–229.

Sohrabi, S., Baier, J. A. and McIlraith, S. A., (2011). Preferred explanations: Theory and generation via planning, *in Proc. of the 25th AAAI Conference on Artificial Intelligence*.

Sohrabi, S. and McIlraith, S. A., (2010). Preference-based web service composition: A middle ground between execution and search, *in International Semantic Web Conference*, pp. 713–729.

Sohrabi, S., Prokoshyna, N. and Mcilraith, S. A., (2006). Web service composition via generic procedures and customizing user preferences, *in Proc. of the 5th International Semantic Web Conference (ISWC)*, pp. 597–611.

Sohrabi, S., Prokoshyna, N. and McIlraith, S. A., (2009). Web service composition via the customization of golog programs with user preferences, *in Conceptual Modeling: Foundations and Applications*, Vol. 5600.

Spiess, P., Karnouskos, S., Guinard, D., Savio, D., Baecker, O., de Souza, L. M. S. and Trifa, V., (2009). Soa-based integration of the internet of things in enterprise services, *in Proc. of the IEEE 7th International Conference on Web Services (ICWS)*, pp. 968–975.

Srivastava, B. and Koehler, J., (2003). Web service composition - current solutions and open problems, *in ICAPS Workshop on Planning for Web Services*, pp. 28–35.

To, S. T., Son, T. C. and Pontelli, E., (2011). Contingent planning as and/or forward search with disjunctive representation, *in Proc. of the 21st International Conference on Automated Planning and Schedulin (ICAPS)*.

Traverso, P. and Pistore, M., (2004). Automated composition of semantic web services into executable processes, *in 3d International Semantic Web Conference*, pp. 380–394.

Trifa, V., Guinard, D., Davidovski, V., Kamilaris, A. and Delchev, I., (2010). Web messaging for open and scalable distributed sensing applications, *in Proc. of the 10th International Conference on Web Engineering (ICWE)*, Springer-Verlag, pp. 129–143.

Trčka, N., van der Aalst, W. and Sidorova, N., (2009). Data-flow anti-patterns: Discovering data-flow errors in workflows, *in Adv. Inf. Systems Eng.*, Vol. 5565 of *LNCS*, Springer Verlag, pp. 425–439.

Urban, S., Gao, L., Shrestha, R. and Courter, A., (2011). The dynamics of process modeling: New directions for the use of events and rules in service-oriented computing, *in The Evolution of Conceptual Modeling*, Vol. 6520 of *LNCS*, Springer Verlag, pp. 205–224.

van Beest, N., Kaldeli, E., Bulanov, P., Wortmann, J. and Lazovik, A., (2012). Automatic detection of business process interference, *in 1st International Workshop on Knowledge-intensive Business Processes (KIBP), 13th International Conference on Principles of Knowledge Representation and Reasoning (KR).*

van Beest, N. R. T. P., Bulanov, P., Wortmann, J. and Lazovik, A., (2010). Resolving business process interference via dynamic reconfiguration, *in Proc. of 8th International Conference on Service Oriented Computing (ICSOC)*, pp. 47–60.

van Beest, N. R. T. P., Szirbik, N. B. and Wortmann, J. C., (2010). Assessing the interference in concurrent business processes, *in Proc. of 12th International Conference on Enterprise Information Systems (ICEIS)*, pp. 261–270.

van der Krogt, R. and de Weerdt, M., (2005). Plan repair as an extension of planning, *in Proc. of the 15th International Conference on Automated Planning and Scheduling*, pp. 161–170.

Vargas, L., Bacon, J. and Moody, K., (2005). Integrating databases with publish/-subscribe, *in Proc. of the 4th International Workshop on Distributed Event-Based Systems (DEBS), International Conference on Distributed Computing Systems Workshops (ICDCSW)*, IEEE Computer Society, pp. 392–397.

Vidal, V., (2004). Branching and pruning: An optimal temporal POCL planner based on constraint programming, *in Artificial Intelligence*, pp. 570–577.

Vukovic, M. and Robinson, P., (2005). Goalmorph: partial goal satisfaction for flexible service composition, *in International Conference on Next Generation Web Services Practices.*

Wallace, R. J. and Grimes, D., (2010). Problem-structure vs. solution-based methods for solving dynamic constraint satisfaction problems, *in Proc. of the 22nd International Conference on Tools with Artificial Intelligence.*

Wallace, R. J., Grimes, D. and Freuder, E. C., (2009). Solving dynamic constraint satisfaction problems by identifying stable features, *in Proc. of the 21st International Joint Conference on Artificial Intelligence.*

Wang, H., Tang, P. and Hung, P., (2008). RLPLA: A reinforcement learning algorithm of web service composition with preference consideration, *in IEEE World Congress on Services PArt II (SERVICES-2)*, pp. 163 –170.

Warriach, E. U., Kaldeli, E., Bresser, J., Lazovik, A. and Aiello, M., (2010). A tool for integrating pervasive services and simulating their composition, *in Demo Session of the 8th International Conference in Service-Oriented Computing (IC-SOC)*, Vol. LNCS, Springer, pp. 726–727.

Weber, I., Hoffmann, J. and Mendling, J., (2010). Beyond soundness: on the verification of semantic business process models, *Distributed and Parallel Databases* 27, 271–343.

Weske, M., (2001). Formal foundation and conceptual design of dynamic adaptations in a workflow management system, *in Proc. of the 34th Annual Hawaii Inte. Conference on System Sciences.*

Westra, K., (2010). Web service composition: connecting the web cloud, Bachelor's thesis, University of Groningen.

Wu, D., Parsia, B., Sirin, E., Hendler, J. A. and Nau, D. S., (2003). Automating DAML-S web services composition using SHOP2, *in Proc. of the 2nd International Semantic Web Conference*, pp. 195–210.

Xiao, Y. and Urban, S., (2007). Process dependencies and process interference rules for analyzing the impact of failure in a service composition environment, *in Business Inf. Systems*, Vol. 4439 of *LNCS*, Springer Verlag.

Xiao, Y. and Urban, S., (2008). Using data dependencies to support the recovery of concurrent processes in a service composition environment, *in Proc. of the 16th International Conference on Cooperative Inf. Systems.*

Yoon, S. W., Fern, A. and Givan, R., (2007). FF-Replan: A baseline for probabilistic planning, *in Proc. of the 17th International Conference on Automated Planning and Scheduling*, pp. 352–.

Yoon, S. W., Fern, A., Givan, R. and Kambhampati, S., (2008). Probabilistic planning via determinization in hindsight, *in Proc. of the 23rd AAAI Conference on Artificial Intelligence*, pp. 1010–1016.

Yu, T., Zhang, Y. and Lin, K.-J., (2007). Efficient algorithms for web services selection with end-to-end QoS constraints, *ACM Transactions on the Web* 1.

Yumatov, S., (2011). Web based interface for a smart home, Master's thesis, University of Groningen.

Zeadally, S. and Kubher, P., (2008). Internet access to heterogeneous home area network devices with an OSGi-based residential gateway, *International Journal of Ad Hoc and Ubiquitous Computing* 3, 48–56.

Zhang, W. and Hansen, K. M., (2008). Semantic web based self-management for a pervasive service middleware, *in Proc. of the 2nd IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, pp. 245–254.

Zou, G., Chen, Y., Xu, Y., Huang, R. and Xiang, Y., (2012). Towards automated choreographing of web services using planning, *in Proc. of the 26th AAAI Conference on Artificial Intelligence*.

Defense Advanced Research Projects Agency (DARPA), (2002). 'DAML-S 0.7 Draft Release'. URL: *www.daml.org/services/daml-s/0.7/*

Organization for the Advancement of Structured Information Standards (OASIS), (2007). 'Web services business process execution language'. URL: *www.oasis-open.org/committees/wsbpel*

World Wide Web Consortium (W3C), (2004). 'Semantic markup for web services: OWL-S'. URL: *www.w3.org/Submission/OWL-S*

World Wide Web Consortium (W3C), (2005*a*). 'Web service modeling ontology (WSMO)'. URL: *www.w3.org/Submission/WSMO*

World Wide Web Consortium (W3C), (2005*b*). 'Web service semantics: WSDL-S'. URL: *www.w3.org/Submission/WSDL-S*

Apache, (n.d.). 'Jini - apache river project'. URL: *river.apache.org*

Choco library documentation, (2012). URL: *www.emn.fr/z-info/choco-solver*

Oracle, (2002). 'Oracle JMS'. http://docs.oracle.com/javaee/1.3/jms/tutorial/.

OSGi Alliance, (2009). ‘OSGi service platform core specification release 4’. URL: *www.osgi.org*

SM4All, (2008-2011). ‘Smart hoMes for All’, STREP Project: FP7-224332. URL: *www.sm4art-project.eu*

UPnP Forum, (2008). ‘UPnP™ device architecture version 1.1’. URL: *www.upnp.org*

Samenvatting

Web Services kunnen worden gezien als de fundamentele elementen om gedistribueerde applicaties te bouwen, waarbij de interactie tussen heterogene software componenten wordt gefaciliteerd op een interoperabele manier. De mogelijkheid om bestaande services te selecteren en integreren opent nieuwe perspectieven voor de ontwikkeling van service-georiënteerde applicaties. Zelfs als er geen web service beschikbaar is om een bepaald doel te vervullen, zou het mogelijk moeten zijn om een compositie van verschillende services te genereren, die dat doel tezamen kunnen verwezelijken. Compositie van services is een zeer complexe taak en de automatisering daarvan blijft een belangrijke uitdaging. Onderzoek op het gebied van Artificial Intelligence planning kan een verder inzicht in het probleem verschaffen en bijdragen aan de geautomatiseerde compositie van services, die zich aan kunnen passen aan de veranderende behoefte van gebruikers en omgevingsvariabelen. In de afgelopen jaren zijn verschillende benaderingen ontwikkeld om het probleem van service compositie te formuleren als planningstaak. Het algemene uitgangspunt van deze benaderingen is dat services voorzien zijn van semantische annotaties. In dit opzicht worden de methoden, die worden aangeboden door services, gezien als planningsacties, welke zijn beschreven in termen van precondities en effecten, en de gebruikersdoelstelling wordt gezien als een planningsdoel.

Echter, de meeste bestaande planningsbenaderingen voor service compositie hebben een of meer van de volgende beperkingen: ze zijn niet domeinonafhankelijk, waardoor de toepasbaarheid van het domein wordt beperkt door een set voorgedefinieerde procedurele templates; ze zijn niet in staat om op een efficiënte manier om te gaan met numerieke variabelen, vooral wanneer deze betrekking hebben op observaties of gebruikersinvoer; en ze houden geen rekening met herstel van onvoorziene runtime omstandigheden als gevolg van foutief gedrag van services of externe gebeurtenissen, die interfereren met de uitvoer van het plan. Om deze tekortkomingen te ondervangen, introduceert dit proefschrift een nieuw domeinonafhankelijk planningsframework – genaamd RuG planner –, die gebaseerd is op het modelleren van een planningstaak als een Constraint Satisfaction Problem. Dit planningssysteem wordt gebruikt om te voldoen aan de eisen gesteld door drie verschillende service-georiënteerde platformen: een domein bestaande uit verschillende services die beschikbaar zijn via internet, een Smart Home voorzien van intelligente apparaten die als service beschikbaar zijn en een framework voor herstel van bedrijfsprocessen in het geval van procesinterferentie.

Om aan de eisen te voldoen, die door deze applicatiedomeinen worden gesteld, wordt de RuG planner voorzien van een aantal speciale eigenschappen, welke, wanneer ze worden samengevoegd, het aantal scenarios verhogen dat effectief kan worden ondervangen in vergelijking met voorgaande benaderingen. Deze eigenschappen omvatten een kennisrepresentatie om onzekerheid omtrent de beginstaat en het resultaat van de observationele acties te modelleren; efficinte omgang met numerieke variabelen, welke kunnen voorkomen als input voor acties of output van observationele effecten; generatie van plannen met een hoge graad van parallellisme; ondersteuning voor een declaratieve taal voor doelen met temporele extensie; en continue revisie van het plan om observationele uitkomsten, fouten, lange reactietijden en activiteiten van externe actoren te ondersteunen. Al deze eigenschappen zijn gerealiseerd op een manier zodanig dat de eisen van domeinonafhankelijkheid worden gerespecteerd.

Het RuG planningsframework is geëvalueerd op verschillende scenarios en service platformen, waaronder een aantal tests met betrekking tot de prestaties van de planningstechnieken, de effectiviteit van de gegenereerde oplossingen en gebruiksvriendelijkheid. De tests hebben aangetoond, dat de RuG planner kan worden gebruikt om verschillende doelen te bewerkstelligen onder uiteenlopende omstandigheden. Hoewel de planningsmethodologie focust op toepassingen met web services, is de essentie meer generiek en worden problemen ondervangen waar domeinonafhankelijkheid, onzekerheid en dynamiek een rol spelen.