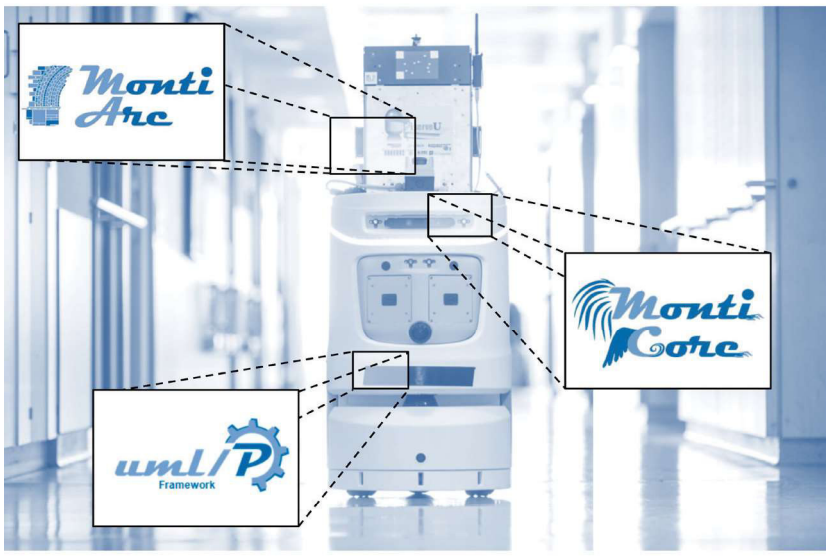
SE | RWTH AACHEN UNIVERSITY

Kai Adam, Arvid Butting, Robert Heim,
Oliver Kautz, Jérôme Pfeiffer,
Bernhard Rumpe, Andreas Wortmann

Modeling Robotics Tasks for Better Separation of Concerns, Platform-Independence, and Reuse

Results of the iserveU Federal Research Project



Aachener Informatik-Berichte,
Software Engineering Band 28

Hrsg: Prof. Dr. rer. nat. Bernhard Rumpe

RWTH Aachen University
Software Engineering

Modeling Robotics Tasks for Better Separation of Concerns, Platform-Independence, and Reuse

Results of the iserveU Federal Research Project

Kai Adam

Arvid Butting

Robert Heim

Oliver Kautz

Jérôme Pfeiffer

Bernhard Rumpe

Andreas Wortmann

Technical Report
Aachen, December 2017

[ABH+17] K. Adam, A. Butting, R. Heim, O. Kautz, J. Pfeiffer, B. Rumpe, A. Wortmann
Modeling Robotics Tasks for Better Separation of Concerns, Platform-Independence, and Reuse.
In: Shaker Verlag, ISBN 978-3-8440-5319-7. Aachener Informatik-Berichte, Software Engineering, Band 28. December 2017.
www.se-rwth.de/publications

This research has partly received funding from the German Federal Ministry for Education and Research under grant no. 01IM12008C. The responsibility for the content of this publication is with the authors.

Abstract

Deploying robotics applications requires expertise from multiple domains, including general software engineering and the application domain itself. Consequently, successful robotics applications are developed by teams of software experts, robotics experts, and application domain experts. The conceptual gap between application domain challenges and implementation domain solutions gives rise to accidental complexities from solving problem domain challenges with programming domain details. This complicates development and may lead to failure. Domain and robotics experts are rarely software engineering experts. Their involvement into the software engineering of reusable robotics applications requires that they become software experts or that implementation details can be abstracted.

Model-driven engineering reduces the conceptual gap by leveraging models to primary development artifacts. Models usually are more abstract than programming language artifacts and enable to use robotics vocabulary or application domain vocabulary. This supports domain experts in formulating solutions in established known vocabulary. Model transformations can embody the software engineering expertise required to translate such models into robust and reusable programming language solutions. Different target technologies can be addressed by different transformations, which decouples logical robotics tasks from heterogeneous platforms and ultimately produces deployable solutions. Proper separation of expert concerns is crucial to enable transformations and ultimately improve engineering of robotics applications. Component-based software engineering has proven useful for the integration of domain-specific solutions and system extensibility. Here, software components encapsulate functionality in reusable black boxes. That these components usually are programming language artifacts gives rise to accidental complexities again. Models of architecture description languages lift components to the model level. Component models can serve as building blocks of complex systems and can be translated into implementations for different target technologies as well. Combining model-driven engineering with component-based software engineering enables to provide robotics experts and domain experts with appropriate modeling languages, as well as software engineering experts with means to decouple their concerns while ensuring integration of their solutions.

We present a collection of domain-specific languages to describe service robotics applications. They enable formulating domains, tasks, actions, and properties of a robot and its environment free of GPL complexities. Their models are translated into component implementations of a MontiArcAutomaton [RRRW15] software architecture model and interpreted at system runtime. The architecture executes tasks via their transformation to Planning Domain Definition Language models, solves these, and executes the resulting plans with a robotics middleware. Leveraging separation of concerns and abstraction of modeling languages, this reduces the effort of describing robot tasks, facilitates extension

of the system with new components, and decouples logical task solving from the robot platform. This supports integration of domain experts and reuse of infrastructure constituents in different contexts and with different platforms.

Contents

Chapter 1

Introduction

As a result of cyber-physical systems permeating all areas of life, challenges for developing and integrating hardware and software rise. Robotic systems inherently are cyber-physical and their successful deployment requires integrating expertise of several domains, including motion, perception, software engineering, safety, usability, and many more. To handle the complex integration concerns software plays a key role. It enables, for example, checking a range of integration constraints automatically and fast. However, software itself has become so complex that the software development process requires decent investigation to circumvent failure and ensure high quality products. To this effect, modeling languages aim at better comprehensibility by abstracting from the accidental complexities [FR07] and notational noise [Wil01] of general purpose programming languages (GPLs) (such as Java or Python). Models describing system architecture and behavior produce better comprehensibility than GPL code [VSB+13], but they often are only used for communication among developers and for documentation [WHR14]. Thus, they easily become outdated and subvert the development process by providing misleading information. To counteract this effect, model-driven engineering (MDE) [FR07] raises models to first class development artifacts. Here, models do not only support communication and documentation but they serve as system specification from which code generators derive the actual software.

Approaches utilizing MDE often employ general-purpose modeling languages, such as the UML [OMG10, Rum11, Rum16]. While these languages abstract from the target platforms and GPLs, they are domain agnostic and, hence, often employ a wrong level of abstraction. For example, experts of participating domains might have different understandings of the same models since semantics may vary among the domains or the languages themselves do not match the actual problem domain. Consequently, development of perfectly fitting modeling languages for a specific domain seems promising. So called domain-specific modeling languages (DSMLs) [Fow10] enable domain experts to model solutions using their own terminology of the problem domain. Since DSMLs themselves are software, their development suffers from the same threats as traditional software engineering. For example, original specifications might no longer meet the actual requirements after some time and first prototypes should be available early to determine the actual suitability. To this effect, agile development of DSMLs and code generators that derive high quality software from DSML models has become the aim of modern language workbenches such as MontiCore [GKR+08]. Thereby, MDE enables development of software that fulfills today's complex requirements.

Many robotics solutions use specific languages addressing the concerns of specific parts of the robotics domain [NHWW16]. This raises challenges in integrating the languages and circumventing tight coupling between the solutions to enable better reuse in different configurations. Consequently, in component-based software engineering (CBSE) [BW98] software systems are built of components that employ well defined and stable interfaces for communication. Behind these interfaces components encapsulate functionality. Hence, CBSE decouples communication from computation and thereby prevents intermixing functionalities and separates concerns of participating experts. Furthermore, it enables reuse of components in different architectures. Component & connector architecture description languages (ADLs) [MT00, MLM+13] lift CBSE to model level. Here, software engineers describe software architectures using abstract component models, their composition, and their interaction. Experts of different domains provide component implementations and do not require knowledge about the overall system, its integration or implementation details of other components. Thereby, reuse and maintainability is supported. Experts still employ different domain-specific languages (DSLs) or GPLs that require integration, but an ADL based approach simplifies integration. Instead of integrating the specific languages with each other, integration efforts reduce to integrating each language with the ADL [HRW16]. In context of the 3-year iserveU research project funded by the German Federal Ministry for Education and Research (BMBF), we investigated the feasibility of pervasive MDE for complex service robotics applications in the domain of transport applications [HMSNR+15, ABH+16]. The concepts described in this report originate from this investigation.

In the following, Chapter 2 outlines the development methodology based on the iserveU DSLs. Chapter 3 motivates the modeling infrastructure by example. Chapter 4 describes the DSMLs in detail before Chapter 5 specifies the component and connector (C&C) software architecture that integrates the domain-specific solutions. Chapter 6 explains how transformations translate the domain-specific models to the planning domain definition language (PDDL) and back to employ an artificial intelligence planning software. Chapter 7 depicts human-robot interfaces (HRIs) and their integration with the software architecture. Chapter 8 gives a tutorial on deploying and executing the architecture. Chapter 9 discusses related work in terms of modeling languages, their integration with planning software and software architectures. Chapter 10 summarizes the main achievements, concludes the report, and gives an outlook on future work.

Chapter 2

Methodology

This report describes a collection of DSMLs and their integration with a reference C&C architecture for the efficient engineering of service robotics. Using transformations it employs standardized artificial intelligence planning software for task execution. Code generators deploy the system to different software platforms dedicated to robotics (e.g., SmartSoft [SSL11]). As sketched in [HMSNR+15] this approach enables modeling robot and world interfaces and tasks to gain separation of concerns and roles. Furthermore, its loose coupling of different model types empowers their reuse in different contexts and with different platforms.

Figure 2.1 illustrates the main layers and roles involved. On the model layer, infrastructure experts model the software C&C architecture while domain experts utilize DSMLs to specify solutions in their domain. For example, these models could specify tasks for item transportation in a hospital. Code generators derive the actual system software. Since generating the complete system is often neither possible nor useful generators enable integration of handwritten code [GHK+15]. They derive APIs from the components' stable interfaces against which robot experts and world experts can implement specific functionality by hand. Thereby, their solutions correctly integrate into the system.

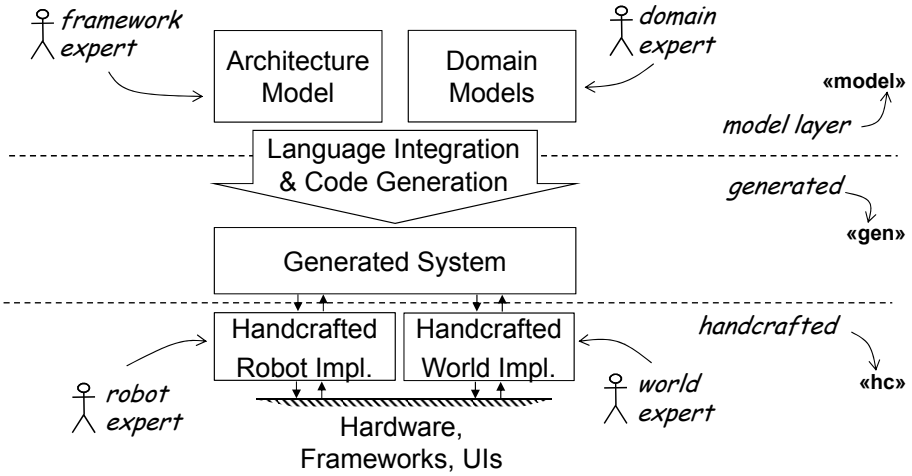


Figure 2.1: Combining CBSE and MDE decouples expert roles to facilitate reuse, maintainability, flexibility and the system integration of complex systems, here on the example of the service robotics domain.

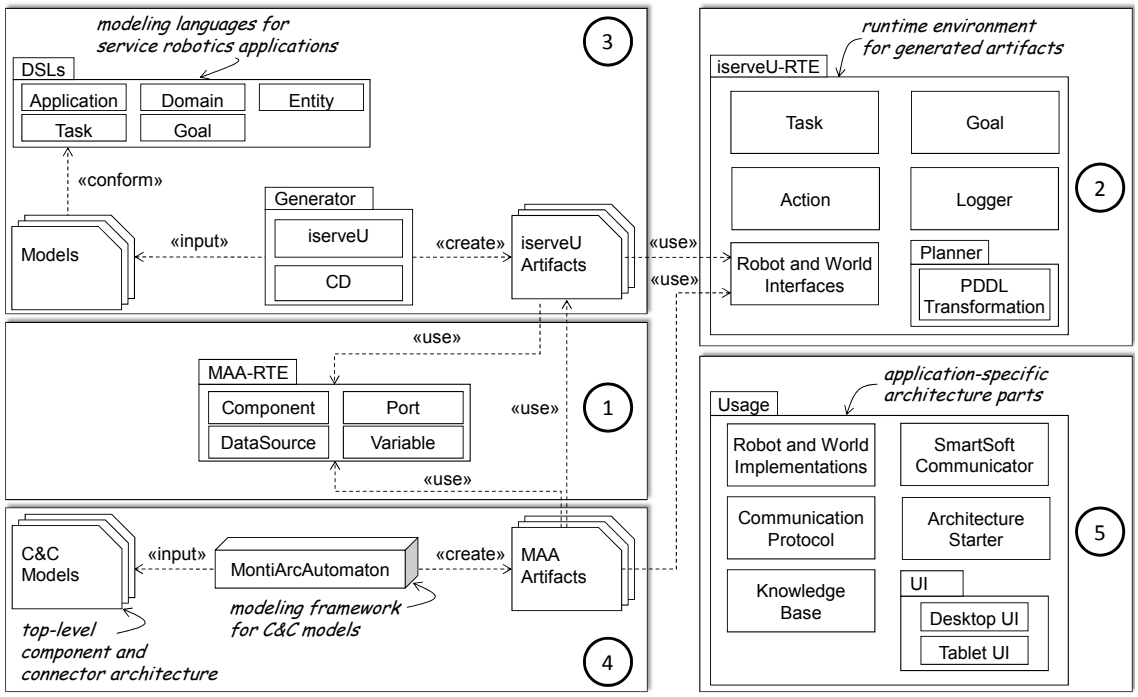


Figure 2.2: Overview of the most important components of the system architecture and the infrastructure for the MDE of service robotics applications.

Figure 2.2 sketches the methodology's main constituents in greater detail. The numbers denote the steps in the development process:

1. The systems builds on infrastructure provided by the MontiArcAutomaton runtime environment (RTE) [RRRW15].

2. Next, the iserveU RTE prescribes the structure of domain-specific artifacts (here, Tasks, Goals, Actions, etc.).

3. Since the artifacts are GPL-based DSMLs act as frontend for system specification. A generator translates their models to their GPL representations. The generated implementation is bound to the MontiArcAutomaton (MAA) infrastructure.

4. Using MAA, a C&C-reference architecture is modeled and translated to an implementation that handles the generated domain artifacts of step (3). It also makes use of the domain-specific RTE (2).

5. After creating the infrastructure in the previous steps, this one complements domain-specific implementations (for example, communication protocols to other system elements or user interfaces (UIs)).

To separate concerns of application domain experts and C&C experts (cf. Figure 2.1), models for the service robotics application (3) are separated from the C&C reference architecture (4). An overall benefit of the separation is an improved reusability of application and reference architecture specific models. The dedicated modeling languages

(cf. Chapter 4) further decouple the different application concerns and thereby support reuse. The domain models define the domain-specific types and their provided operations (e.g., rooms). Application models specify the participating entities (i.e., the actors of the scenario) consisting of a world entity representing the environment and a robot entity. Entities reside in their own models of the entity DSML. They explicate their properties and define actions in the context of the specific domain. Based on these, goal models describe specific situations using the entities' and domain's properties (e.g., that the robot carries a specific item). A task model defines sequences of goals to specify a reusable high level operation in the domain (e.g., fetching and delivering an item). The models are domain-specific and are not bound to any specific robotics platform which empowers their reuse on different platforms.

The MAA modeling infrastructure transforms a modeled C&C system architecture to a deployable software system (4). It builds on the MAA RTE (1) and uses the RTE developed for the DSMLs (2). The latter represents the models at runtime and provides additional infrastructure (e.g., logging and planning). Generators analyze the domain-specific models and translate them into runtime artifacts (3) for the C&C architecture. Handcrafted artifacts (5) provide domain specific additions. They consist of platform specific implementations (cf. Figure 2.1) for the application domain of the iserveU project. Here, the entities are bound to SmartSoft [SSL11]. Also, it integrates additional project specific features, such as dedicated UIs and knowledge base.

Chapter 3

Example

Giving an example, this chapter highlights challenges of modern robotics and motivates the modeling infrastructure showing how companies may benefit from it. A company, for example, may produce robots for transport services in different indoor contexts (such as warehousing and nursing (cf. Figure 3.1)). As the various contexts impose different requirements regarding the deployed robots' capabilities, the company produces multiple robot platforms. Although the contexts and platforms vary, the fundamental Indoor Transport Service (ITS) concepts remain invariant: The service robot has to receive orders, traverse rooms, and pick up and deploy items.

Figure 3.1: A Robotino3[1] robot used during the iserveU evaluation in the Katharinen Hospital in Stuttgart.

Instead of developing multiple infrastructures and code generators for different GPLs, the company employs model-driven techniques to describe the invariant transport services on an abstract, better reusable level. To this effect, they decompose a service robotics applications into the following parts:

- Domain model: the static context in which a transport service robot operates, such as rooms, items, and their properties.

[1] Robotino Website: `http://www.robotino.de`

- Entities: abstract descriptions of the employed robot and the context's environment (the "world") comprising actions and properties.

- Tasks: reusable, abstract descriptions of services that the robot provides in the given domain and world.

- Goals: atomic components of tasks, describing situations the robot must achieve at a certain point in time.

Separating these concerns by employing artifacts of different modeling languages allows the company's software engineers to decouple the actual robots via entities from tasks. Decomposing tasks into sequences of goals enables the reuse of goals in different tasks. Furthermore, the overall application logics are decoupled from underlying robotics issues, such as navigation, object recognition, or human-robot interaction.

For instance, the company's ITS applications must be able to transport items from one room to another. With the modeling languages presented in this report, this can be modeled as depicted in Figure 3.2. For each new deployment, an application model defines the available robot and world entities. Both relate to an unified modeling language (UML) class diagram (CD) modeling the domain in which they operate. Goals reference properties of the robot, the world, and the domain to describe achievements during task executions. Tasks compose sequences of goals which the robot fulfills one by one to achieve the complete task.

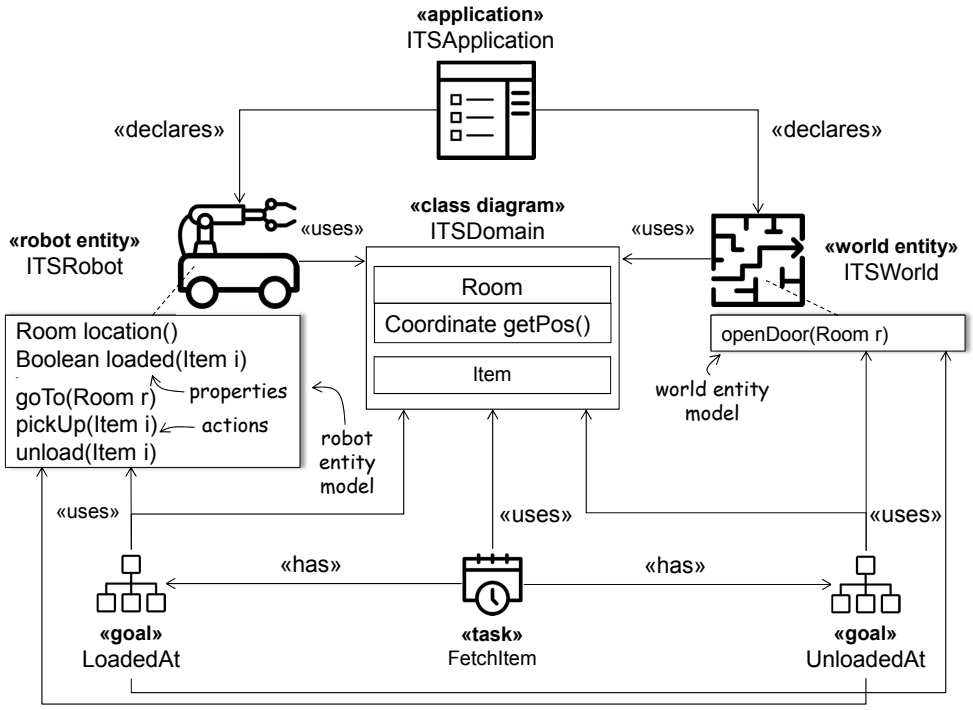


Figure 3.2: Illustration of models of the different modeling languages and their relations. They describe an application, a robot and a world, two goals, a task, and the domain.

Goal

```
1 domain ITSDomain;
2 robot  ITSRobot as rob;
3 world  ITSWorld;
4
5 goal LoadedAt(Room r, Item i) {
6   predicate {
7     rob.location() == r
8     && rob.loaded(i)
9   }
10 }
```

Listing 3.3: The goal `LoadedAt` defines a predicate over properties of the `ITSRobot`'s properties.

In this example, robots compatible with the company's ITS must provide at least two properties (`location` and `loaded`) and three actions (`goTo`, `pickUp`, and `unload`). The realization within the actually employed robot platform is left underspecified: While a robot with arms may pick up items on its own, a robot without arms might call for assistance of warehouse laborer instead. The same holds for the world entity. Here, it provides a single action (`openDoor`) which might open doors, for example, using an authorization code or it could require assistance. Actions and Properties refer to types of the domain model, which is depicted in the center of Figure 3.2. It contains the types `Room` and `Item`. Actions comprise a precondition that describes the circumstances in which their execution is possible, and a postcondition that must hold after execution. The planner (cf. Chapter 6) of the reference architecture (cf. Chapter 5) derives sequences of robot and world actions from goals, which then are executed in the, physical, real world.

Goals describe Boolean conditions over the entities' properties. At runtime, the planner tries to derive actions to achieve these properties and thereby fulfill the goal (cf. Chapter 6). Listing 3.3 shows the textual model of the goal `LoadedAt` (cf. bottom of Figure 3.2). After declaring its dependencies to entities and a domain (ll. 1-3), it defines its name and parameters (l. 5). Its body (ll. 5-10) conjuncts two expressions that reference properties of the robot and compare these to the passed arguments.

The parameters of goals refer to data types defined in the domain model. They configure the goal and thereby enable their reuse in different tasks. Tasks must specify corresponding parameters themselves to configure the involved goals at runtime (cf. Listing 3.4 l. 3). The textual model of the `FetchItem` task begins with importing the domain model data types (l. 1), but omits references to robot entities or world entities (they are transitively dependent via the referenced goals). After specifying its name (l. 3), the task declares its parameters: the source room, an item to fetch from the source room, and a destination room, where the item should be delivered to. The task's body (ll. 3-6) references two goals that must be achieved in sequential order for the task to be fulfilled: `LoadedAt` is instantiated with the arguments received via task parameters `src` and `i` and holds, if the robot was able to pick up the item `i` in the room `src`. The next goal, `UnloadedAt` holds

Task

```
1 domain ITSDomain;
2
3 task FetchItem(Room src, Item i, Room dst) {
4   LoadedAt(src,i);
5   UnloadedAt(dst,i);
6 }
```

Listing 3.4: The task `FetchItem` must be instantiated with three parameters and uses these to instantiate two goals. It is considered fulfilled if all goals are satisfied in the specified order.

when the robot is in room `dst` and delivered the item `i`. If the system finishes the task, the referenced item was successfully transported.

To execute tasks in the real world, engineers map actions and properties of the entity models to their implementations via concise interfaces. The reference architecture (cf. Chapter 5) solves tasks by handing their goal decomposition to the planner and executing the derived actions using the actually employed entity implementations. Being a C&C architecture, it also provides well defined extension points to add additional functionality.

Employing this modeling infrastructure, the company achieves a better separation of concerns between robotics experts (who map entities to actual GPL implementations), domain experts (who describe tasks, goals, and entities), users (who instantiate tasks), and software engineering experts (who integrate and deploy the reference architecture). Also, the company can reuse the different models for similar applications. In case the deployed robot platform changes, only the mapping of the robot model's actions and properties must be adapted. In case tasks or goals change while the robot's and world's properties and actions stay sufficient, neither the models of robot or world require adaptation and, hence, their implementation can be retained as well.

Chapter 4

Modeling Languages

Modeling service robotics applications such as the ITS (cf. Chapter 3) requires appropriate DSLs (cf. Chapter 1). To empower separation of concerns, several DSLs are combined as sketched in [HMSNR+15]. All languages are developed using the MontiCore language workbench [KRV10]. Section 4.1 presents the application modeling language. Applications consist of a robot and a world entity operating in a specific domain. CDs (Section 4.2) model the domain characteristics (e.g., rooms)and how they can be manipulated (e.g., openDoor). The entity modeling language (Section 4.3) enables explicating properties of a modeled entity and its capabilities in form of actions that may manipulate the domain and the employed entities. In this setup, tasks (Section 4.4) define sequences of goals (Section 4.5) that must be fulfilled in a given order. Goals are expressions over the properties of entities and the environment. At runtime, the employed planner (cf. Chapter 6) calculates action sequences for the entities to execute tasks by achieving their goals (cf. Chapter 5). Section 4.6 discusses the languages.

4.1 Application Language

An application model defines which entities participate in the modeled service robotics application. Each application model binds a world entity to a robot entity. This section presents the language elements and well-formedness rules to model applications.

4.1.1 Language Elements

Listing 4.1 illustrates the definition of a hospital application comprising a `Transport-Robot` robot entity and a world entity `RoomsWorld`. Each application is located in a package (l. 1). The application definition starts with the keyword `application` followed by the application's name (l. 3). An application definition has a body delimited by curly brackets (ll. 3-6). The body of an application must contain exactly one reference to a robot entity and exactly one reference to a world entity. References to robot entities start with the keyword `robot` (l. 4) and references to world entities with the keyword `world` (l. 5). The keyword is followed by the qualified name of the referenced entity.

Application

```
1 package hospital.applications;
2
3 application HospitalTransportApp {
4   robot hospital.entities.TransportRobot;
5   world hospital.entities.RoomsWorld;
6 }
```

Listing 4.1: The `HospitalTransportApp` application with the robot entity `TransportRobot` and the world entity `RoomsWorld`.

4.1.2 Well-formedness Rules

The well-formedness rules of the application language (Table 4.1) describe the language's static semantics and their constraints. They are separated into rules regarding conventions (**AC**) and referential integrity (**AR**). Violation of convention conditions raises warnings, while violation of referential integrity raises errors.

ID	Well-formedness Rule Description
AC-01	The application's name should start capitalized.
AR-01	All referenced entities exist.
AR-02	The referenced robot is a robot entity.
AR-03	The referenced world is a world entity.

Table 4.1: Well-formedness rules of the application language.

4.2 Domain Modeling

The domain of a robotics application is modeled using UML programmable (UML/P) CDs [Rum11, Rum12, Sch12]. They express the static properties of domain concerns as well as supported operations on these.

Figure 4.2 depicts a CD modeling the domain of the ITS application (cf. Chapter 3). It consists of the three classes `Room`, `Item`, and `Coordinate`. Each class provides methods to retrieve information and to manipulate its instances. Rooms have a name, a position in form of a Cartesian coordinate, can be open or closed, and contain a set of named items.

Using CDs to model the domain of a robotics application enables to describe domain types of arbitrary complexity. Since CDs are generally well-understood, modeling the domain types requires no additional effort for learning a new DSML.

4.3 Entity Language

Entities model the actors of an robotics application. This section presents the language elements and well-formedness rules of the entity language.

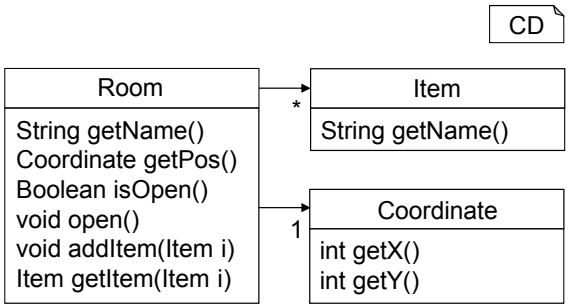


Figure 4.2: Exemplary UML/P CD modeling the domain types `Room`, `Item` and `Coordinate` for the ITS application.

4.3.1 Language Elements

An entity models either a robot or an environment (denoted *world*). They are very similar in their structure, but represent different parts of the application. We therefore distinguish between *robot entities* and *world entities*. Both kinds operate in the context of a domain and can reference the other entity.

Listing 4.3 depicts the model of a robot entity. Entity models start with a package declaration (l. 1). Each entity operates in the context of a domain (l. 2) and may reference another entity. For instance, the `TransportRobot` entity (Listing 4.3) references a world entity (l. 3). After declaring a domain and a potential other entity, the definition of a new entity begins either with keyword `robot` for robot entities or `world` for world entities (l. 5). The keyword is followed by the entity's name and a body delimited by curly brackets (ll. 5-18). A robot entity may only reference a single world entity and, vice versa.

The body of an entity can contain arbitrary many *property* and *action* definitions. The definition of a property starts with the keyword `property` followed by a return type, a name, and a comma separated list of typed parameters enclosed by parentheses. The return type of the property and the types of the parameters must be either defined in the imported domain model or must be a primitive type (e.g., `Boolean`, `Integer`,or `Character`). Here, the robot defines the two properties `robotLocation` and `hasLoaded` (ll. 6-7). The types `Room` and `Item` are imported via the entity's domain `LogisticsDomain` (l. 2) (cf. Figure 4.2). The property `hasLoaded` is parametrized with a single parameter of type `Item` and has the return type `Boolean`, whereas the property `robotLocation` is not parametrized and has the return type `Room`.

The actions of an entity specify its capabilities. Each action has a name, can be parametrized, can have a precondition, and must specify a postcondition. Entity actions resemble actions of STRIPS [FN72] and GOLOG [LRL+97]. The precondition of an action describes the robot and world states that have to be satisfied to be able to execute the action. The postcondition of an action, on the other hand, describes the modification of the robot and world states in response to the execution of the action. The planner component uses the actions' pre- and postconditions to determine which actions are applicable depending on the current system state and the goal to achieve (cf. Chapter 6). Later, implementations of the actions must guarantee the postconditions.

Entity

```
1 package hospital.entities;
2 domain  hospital.types.LogisticsDomain;
3 world   hospital.entities.World;
4
5 robot TransportRobot {
6   property Room robotLocation();
7   property Boolean hasLoaded(Item item);
8
9   action pickUp(Item item, Room room) {
10    pre: World.itemLocation(item, room) &&
11         robotLocation() == room;
12    post: hasLoaded(item) && !World.itemLocation(item, room);
13  }
14
15  action drop(Item item, Room room) {
16    pre: hasLoaded(item) && robotLocation() == room;
17    post: !hasLoaded(item) && World.itemLocation(item, room);
18  }
19 }
```

Listing 4.3: The robot entity `TransportRobot` operates in the context of the `LogisticsDomain` (l. 2) and relates to the world entity `World` (l. 3). It declares two properties (ll. 6-7) using domain data types and defines two actions `pickUp` (ll. 9-13) and `drop` (ll. 15-18).

As shown in Listing 4.3, the `TransportRobot` defines two actions `pickUp` (ll. 9-13) and `drop` (ll. 15-18). The definition of an action begins with the keyword `action`, followed by a name, a list of typed parameters enclosed by parentheses, and a body, which is surrounded by curly brackets. Similar to properties, types referenced in action parameters must be either defined in the imported domain or must be part of the Java package `java.lang`. The definition of an action's precondition (l. 10) begins with the token `pre:`, followed by a Boolean predicate over the properties of the entity, properties of the imported entity, and the domain data types passed to the action. If omitted, preconditions are automatically interpreted as to be always satisfied. The precondition of action `drop` (l. 15), for instance, requires the robot to actually have the item loaded and to be in the target room. In this case, the action can be executed and afterwards guarantees that the item remains no longer on the robot but is now available in the room. In general, preconditions act as guards whereas postconditions describe changes of properties.

4.3.2 Well-formedness Rules

Table 4.2 shows the well-formedness rules of the entity language. The well-formedness rules are separated into rules regarding referential integrity (**ER**), and uniqueness (**EU**) conditions. Violation raises errors.

ID	Well-formedness Rule Description
ER-01	Each data type referenced by a property is defined in the referenced domain model or is part of the Java package `java.lang`.
ER-02	Each data type referenced by an action is defined in the referenced domain model or is part of the Java package `java.lang`.
ER-03	Entity properties referenced by preconditions and postconditions must be defined within the respective entity.
ER-04	Methods of domain types referenced by preconditions and postcondition must be defined on the respective data type of the imported domain.
ER-05	Preconditions and postconditions must evaluate to Boolean.
ER-06	Postconditions do not contain disjunctions.
ER-07	Robot entities reference a single world entity only.
ER-08	World entities reference a single robot entity only.
EU-01	The names of properties and actions are unique.
EU-02	Names of parameters are unique within a property.
EU-03	Names of parameters are unique within an action.

Table 4.2: Well-formedness rules of the entity language.

4.4 Task Language

A task models an ordered sequence of desired situations that should occur one after another. A *situation* is the combination of a world state and a robot state. Each desired situation is modeled as a goal (cf. Section 4.5). Hence, a task model defines a sequence of goals. Tasks can be parametrized with types of the domain model. A task is assigned its arguments at runtime and hands them over to its goals that require arguments themselves. Sometimes, it is useful to denote that multiple goals have to be satisfied simultaneously. For this purpose, a task model can *lock* and *unlock* goals. If a goal is declared locked, the situation modeled by the locked goal has to be satisfied in addition to the situation modeled by the following goals until the locked goal becomes *unlocked*. Goals can become unlocked by unlock statements or by unlock-all statements. While each unlock statement only unlocks a single goal, unlock-all statements unlock all currently locked goals at once.

4.4.1 Language Elements

Listing 4.4 shows an example model of a task that fetches an item from a room and delivers it to another room. It first states its package (l. 1), imports goals (l. 3) and references the domain it relates to (l. 4). The keyword `task` introduces the actual task definition followed by the task's name `FetchItem` and a list of parameters enclosed by parenthesis (l. 6). The task defines the item to fetch as parameter of type `Item` and the involved rooms as parameters of type `Room`. The referenced data types must be defined in the referenced domain model or must be part of the Java package `java.lang`. The body of a task is enclosed by curly brackets (ll. 6-12) and consists of an arbitrary number of statements separated by semicolons. Each statement either directly references a goal (goal statement), starts with the keyword `lock` and is followed by a reference to a goal

Task

```
1 package hospital.tasks;
2
3 import hospital.goals.*;
4 domain hospital.types.LogisticsDomain;
5
6 task FetchItem(Room fromRoom, Item item, Room toRoom){
7   At(fromRoom);
8   lock Loaded(item);
9   At(toRoom);
10  unlock Loaded(item);
11  Unloaded(item) unlock-all;
12 }
```

Listing 4.4: The task model `FetchItem` fetches a given item from a room and delivers it to another room by defining a sequence of goals.

(lock statement), starts with the keyword `unlock` and is followed by a reference to a goal (unlock statement), or starts with a reference to a goal followed by the keyword `unlock-all` (unlock-all statement).

In the example (Listing 4.4), the task `FetchItem` defines five statements. The first statement (l. 7) requires the robot to be in the desired starting room (`fromRoom`). Next, the robot must load the required item (l. 8). To ensure that the robot does not drop the item, the goal is locked. Hence, the goal must be satisfied until it is unlocked. The next goal statement (l. 9) denotes that the robot is positioned in the target room (`toRoom`). Since the `Loaded` goal (l. 8) has been locked, it still must be satisfied in addition to the `At` goal. However, it becomes unlocked by the following unlock statement (l. 10). Finally, the unlock-all statement (l. 11) denotes that the robot does not carry the item anymore indicating that it has been dropped in the target room. In this case, the unlock-all statement does not unlock any goals, but ensures that no goal reference remains locked at the end of the task.

4.4.2 Well-formedness Rules

Table 4.3 presents well-formedness rules for task models. They are separated into rules regarding conventions (**TC**), type correctness (**TT**), uniqueness (**TU**), and referential integrity (**TR**). Violating **TC-02** causes warnings. Others raise errors.

4.5 Goal Language

A goal models a situation of a desired state within a robotics application. The latter consists of the state of the employed robot, the environmental world, and the specific application domain. Therefore, a goal references types defined by its corresponding domain and

ID	Well-formedness Rule Description
TC-01	The body of each task consists of at least one statement.
TC-02	All parameters of a task must be referenced by at least one statement in the task's body.
TC-03	Unlocking a goal requires that it has been locked before.
TC-04	Goals may not remain locked at the end of a task.
TT-01	The arguments of goal references match the referenced goal's parameters in type and number.
TU-01	The parameters of a task have unique names.
TR-01	The arguments of goal references must be names of the task's parameters.

Table 4.3: Well-formedness rules of the task language.

properties of its related entities. To support reuse of goals, they can be parametrized. Goals may query the properties of their corresponding entities (cf. Section 4.3).

4.5.1 Language Elements

Each goal model specifies a situation definition and may optionally relate to a domain, a robot, and a world. A *domain import* is introduced with the keyword `domain`, followed by a full qualified name referencing a domain model. If no domain is imported, the goal is unable to reference domain objects. A *robot import* starts with the keyword `robot`, followed by a full qualified name referencing a robot entity model. A *world import* is introduced with the keyword `world`, followed by a full qualified name referencing a world entity model. Importing robot or world entities is only required if the goal references their properties. After the optional world, robot, and domain imports, a goal model must contain a situation definition. A *situation definition* starts with the keyword `goal`, followed by a name, which defines the name of the goal, a comma separated list of typed parameters enclosed by parenthesis, and the situation definition's body, which is surrounded by curly brackets. The types of the parameters must be defined in the domain imported by the goal.

The body must contain exactly one predicate. A *predicate* is introduced by the keyword `predicate` followed by a situation description, which is enclosed by curly brackets. The predicate defines a Boolean expression over the properties of the imported robot and world entities as well as the types defined in the imported domain model. The Boolean expression may reference properties of the imported entities, which must be parametrized with constants or with parameters of the modeled goal. Also, the expression can employ parenthesis (`(` and `)`), conjunctions (operator `&&`), disjunctions (operator `||`), negations (operator `!`), and comparisons (operator ==).

For instance, Listing 4.5 depicts the model of the goal `LoadedAt`. The goal declares its package (l. 1), imports the domain `LogisticsDomain` (l. 3)), references the robot entity `TransportRobot` (l. 4), and references the world entity `RoomsWorld` (l. 5). The example assumes that the domain defines the data types `Room` and `Item`, whereas the robot entity defines the queries `robotLocation()` and `hasLoaded(Item)`. The world entity defines a query `isAccessible(Room)`, which returns `true` if the robot

Goal

```
1 package hospital.goals;
2
3 domain hospital.types.LogisticsDomain;
4 robot  hospital.entities.TransportRobot;
5 world  hospital.entities.RoomsWorld;
6
7 goal LoadedAt(Room r, Item i) {
8   predicate {
9     (TransportRobot.robotLocation() == r)
10    && RoomsWorld.isAccessible(r)
11    && TransportRobot.hasLoaded(i)
12  }
13 }
```

Listing 4.5: The goal model `LoadedAt` denotes the situation that the `TransportRobot` is at a specific room and carries a specific item.

is allowed to access a certain room. The goal has two parameters (l. 7): `r` of type `Room` and `i` of type `Item`. The predicate of the goal (ll. 8-12) checks whether the location of the robot `TransportRobot` is equal to the room `r` (l. 9), and whether the queries `hasLoaded(i)` and `isAccessible(r)` are `true` (ll. 10-11). In other words, the goal checks whether the robot is allowed at a specific room and carries a specific item.

4.5.2 Well-formedness Rules

Task models must satisfy the well-formedness rules described in Table 4.4. They consist of type correctness rules (**GT**), uniqueness rules (**GU**), and referential integrity rules (**GR**). They all raise errors if violated.

ID	Well-formedness Rule Description
GT-01	The predicate expression must evaluate to Boolean.
GU-01	The parameters of a goal must have unique names.
GR-01	Types referenced by goal parameters must be defined within the imported domain model or must be part of the Java package `java.lang`.

Table 4.4: Well-formedness rules of the goal language.

4.6 Discussion

The different modeling languages capture distinct aspects of a system and thereby support separation of concerns. None of the models described in any of the languages is bound to a specific robotics platform, environmental service, or even hardware. This supports binding the different languages to specific platforms in late stages of the development process and

supports reuse by using different bindings for the same model (e.g., Section 5.4 describes a mapping to a specific platform).

Application models (see Section 4.1) aggregate domain definitions as well as world and robot entities. Models of the other languages, however, reference their required world and robot entities. For example, a goal model often expresses a constraint on a robot's property. Hence, both cases state on which entities the artifacts depend. This is not redundant, but is meant as extension point: Introducing abstract entities or entity interfaces enables better reuse. For example, goal and task models could depend on specific robot capabilities defined in entity interfaces while the application model could define which entity implementations are used within the specific application. A new context condition could validate that the selected entities implement all entity interfaces used by other models. Hence, the application language enables better decoupling of models and thereby increases reuse and testability in future work.

Comparing tasks and goals, the most notable difference is that tasks describe linear sequential lists of situations that must be achieved one by one. On the other hand, goals are Boolean expression over properties of entities and a domain. Thus, tasks and goals only specify the intended objectives that a system has to achieve but do not describe *how* to fulfill them. An execution to accomplish their specifications is automatically derived by a planner. It makes use of actions defined in entity models (see Chapter 6) and calculates which actions lead to the specified situation of a given goal. After executing such a plan, the goal is achieved and the system may work on the next goal. A goal's intention is to describe a specific situation. Fulfilling a goal that the expression holds true and means that the expressed properties were achieved. Fulfilling a task means that each of the referenced goals must be achieved one after another. Hence, a goal is part of the solution that a task describes. This allows reuse and flexible variation to define new tasks using existing goals.

Also, when specifying task models, no knowledge about Boolean expressions is required since they are encapsulated by goals. This makes the task language usable for non-technical users. The goal language itself supports more advanced expressions and therefore its models should be defined by technical staff. Splitting up a task into different goals is also suitable in cases where the system fails to fulfill a goal. The system can start over again from the last fulfilled goal instead of planning the task from its beginning.

The task language is intentionally simple to enable non-technical users to define task models. To this effect, task models may only consist of simple parameters, references to goals and the lock mechanism (cf. Section 4.4). Consequently, the task language does not support conditions, loops, parallel execution, generics, or object instantiation.

The locking mechanism helps to avoid boilerplate code in goals because otherwise the following goals would require adjustments to also denote the expression of the former goal. Considering different combinations several goals would exist in multiple variants describing such combinations. Hence, the locking mechanism enables better reuse and provides flexibility in task definitions.

Tasks and goals refer to a domain, a world, and a robot entity. The system described in Chapter 5 interprets these references as references to specific entities. As mentioned, it is possible to abstract over entities and thereby interpret these references as references to

entity interfaces. This allows better reuse of tasks and goals, because this enables reuse of different specific robots and worlds that implement the required interface. The languages would not require any change for this, but the interpretation infrastructure would need to validate that the specific entities bound by an application model implement all interfaces used by any task or goal. Nevertheless, changing the entity language could make the relation between an abstract entity (or entity interface) and an implemented entity explicit.

Chapter 5

System Architecture

This chapter presents the reference architecture to execute tasks. An overview of the reference architecture is depicted in Figure 5.1. To this effect, it connects two UIs – one stationary, the other mounted to the robot – to a plan processor. The latter employs model transformations to translate goals of tasks into models of PDDL [MGH+98, FL03]. The Metric-FF [HN01] planner solves each goal and returns a sequence of actions to achieve that goal. The task processor translates these actions to commands for an employed middleware to achieve changes in the environment. The communication with the middleware can be realized in different ways. The communication with SmartSoft [SSL11], for instance, is realized as client/server communication. Despite that, the task processor is capable of scheduling different tasks that are executed consecutively.

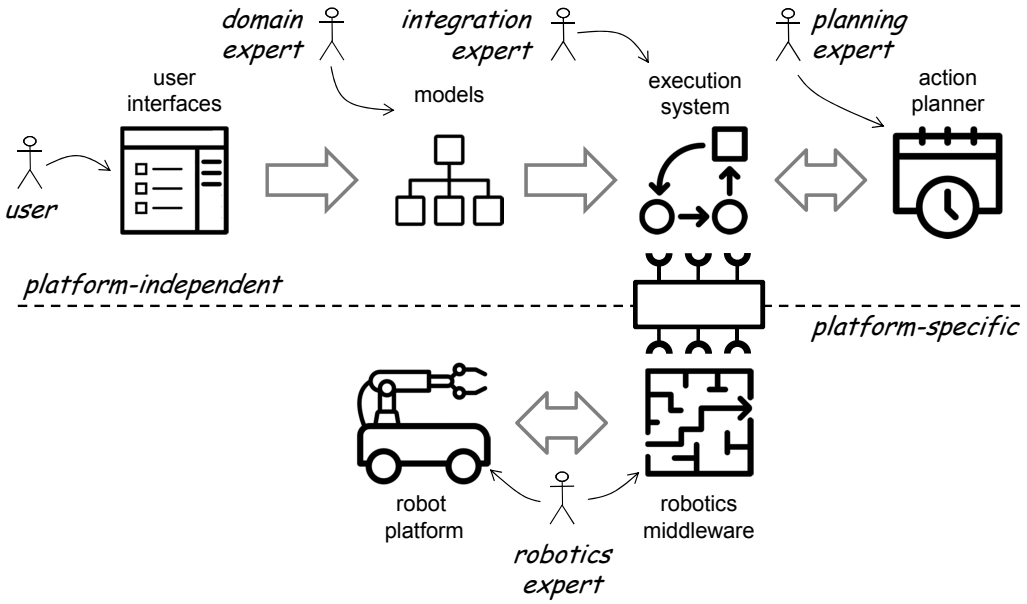


Figure 5.1: Illustration of the overall software architecture. User interfaces (left) instantiate tasks models (right) that the task processor divides into goals. The planner plans a sequence of actions for each goal. These are translated to commands for the robotics middleware, which then are executed on a robot platform.

This chapter presents the software component models employed in the reference architecture. First, Section 5.1 gives an introduction to C&C architectures and MAA, an infrastructure for the development of C&C architectures in context of cyber phyiscal

systems (CPS). Section 5.2 describes the overall architecture in terms of its component models and their implementations. Afterwards, Section 5.3 presents the architecture's RTE, which consists of the application-independent architecture parts required to execute an implementation of the reference architecture. Then, Section 5.4 explains the generating of implementations for component models of the reference architecture from models of the DSMLs described in Chapter 4. These implementations are specific to the models they are generated from. Section 5.5 explains the parts of the architecture that are specific to an application of the reference architecture. The parts include the implementation of the robot and world interfaces and a realization of the communication to the middleware SmartSoft. Section 5.6 demonstrates different excerpts of exemplary architecture executions in form of UML/P Sequence Diagrams. Section 5.7 concludes the chapter by discussing the changes required to use the reference architecture with another middleware or in a different application domain.

5.1 Component & Connector Architecture

C&C architectures enable software engineers to model the conceptual architecture of a software system, while keeping the specific implementation of the system in separation [MT00]. Functionality of a complex software system is thereby separated into system independent components that consist of interfaces for data exchange. A component is a unit that processes or stores information [MT00]. Each of them can be represented in models as well as reused in various contexts. In particular, an architecture is a composition of hierarchically structured and interconnected components, in which each component represents an abstraction of functionality. The interconnection between components is realized by means of connectors that represent data channels. They receive information from a source and deliver them to a destination. A set of participating components in an architecture and their corresponding connectors is called the configuration of an architecture. Each configuration describes the inner structure of the current component level. Related to the development process for software systems, C&C architectures might be either directly involved or passively used to support development phases such as analysis, design, implementation, and testing. A direct involvement, for instance, is the use of models for software architectures in conjunction with a suitable code generator, which is pursued in model-based software engineering (MBSE) [Rum11, Rum16].

MAA is a modeling infrastructure for C&C architectures [RRW14, RRW15b, RRRW15]. The modeling infrastructure encloses the MAA ADL, which supports encapsulation and composition of components, connectors, and ports. Components are treated as black boxes, which process or store information from the environment while hiding the internal structure and behavior. Therefore, a component consists of directed and typed interfaces, called ports. Those ports are used for communication with their environment, which is related to an inter-component communication. The semantics for inter-component communication is based on FOCUS [BDD+93, BS01], which is a infrastructure for stream based communication between components. Despite the interfaces, a MAA component is characterized as either

atomic, when it performs computations, or composed, when it is composed of other components. Therefore, atomic components consist of an explicitly defined behavior while the behavior of composed components is derived by the behavior of its incorporated components. The MAA language provides the opportunity of behavior embedding by means of DSMLs [RRW15a], such as the I/O$^{\omega}$ automaton language [Rum96]. In MAA, software engineers are able to define the behavior of atomic ADL models independently of the targeted platform by using DSMLs models or by supplying handwritten Java artifacts. Besides the MAA ADL, the modeling infrastructure consists of an infrastructure for model transformations and code generation [RRRW14]. An accommodated set of code generators facilitates the process of code generation by providing concepts to translate an internal representation of MAA models to artifacts in specific GPLs, such as Java.

5.2 Reference Architecture

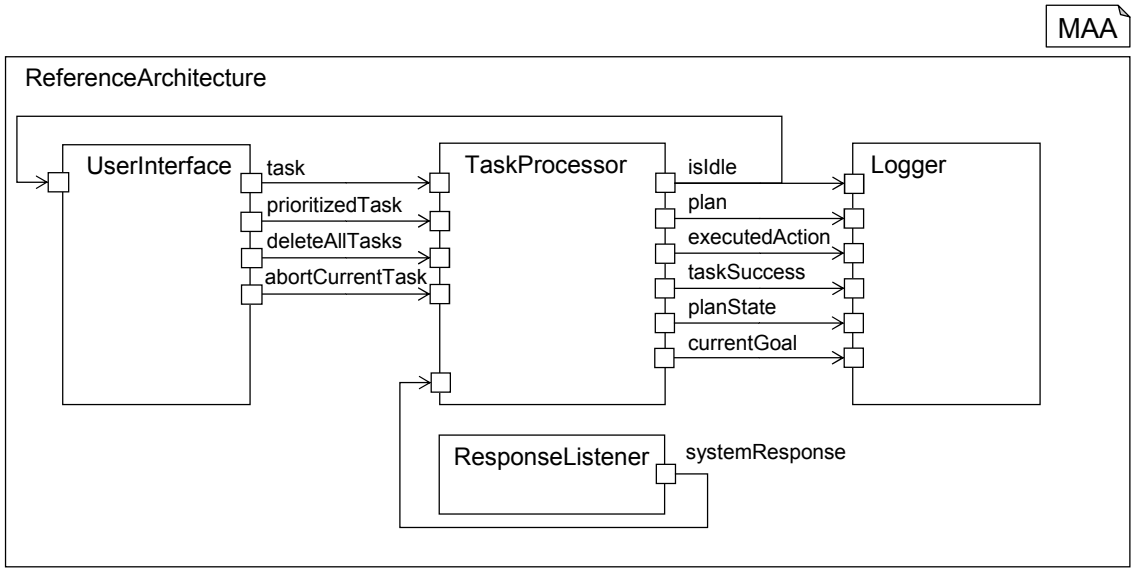


Figure 5.2: An overview of the reference architecture and its subcomponents `TaskProcessor`, `UserInterface`, `ResponseListener`, and `Logger`.

The reference architecture is the concept of the C&C architecture model illustrated in Figure 5.2. It contains four interconnected conceptual components. These components are not necessarily realized as MAA components, but represent software modules with well defined interfaces. The `TaskProcessor` component controls the execution of tasks and maintains a prioritized queue of pending tasks. After the translation of a task's goals into a list of actions using the `Planner` component (Section 5.2.5), it delegates the execution of these actions to an employed middleware. The middleware may confirm the execution with messages that are handled inside of the `ResponseListener` component. The latter translates the middleware messages into a middleware independent representations that can be handled by the `TaskProcessor`. The `UserInterface` component includes the human-robot interface, which instantiates tasks and sends them to the `TaskProcessor`

component. Further, it can send commands to the TaskProcessor, indicating an abort of a task execution or deletion of a task from the queue of pending tasks. The TaskProcessor component emits status messages that can be handled by the Logger component. The outgoing port isIdle of the TaskProcessor is additionally connected to the incoming port isIdle of the UI, to inject an *idle task* to the TaskProcessor once it is idle. An idle task is a predefined task that is automatically executed if the robot is idle. An idle task may, for instance, make the robot move to a charging station. This feature is optional. Our implementation uses the UI described in Chapter 7, the logger explained in Section 5.3.3, and the ResponseListener depicted in Section 5.5.2.

5.2.1 Remote Operator

The execution of a task decomposes to a consecutive execution of actions of a plan for that task. The execution of an action may either be successful or fail. Whereas successful action execution is straightforward, its failure can be caused by different temporary or permanent reasons. For instance, a robot moving from one location to another can be stopped due to a passing person or due to a locked door. While the passing person blocks the way for seconds only, a locked door can block it longer, or require human interaction to open it. If the execution of an action fails, the TaskProcessor aborts the execution of the current task and, after a short period of time, plans it again. Thereby, temporary causes of failure could be overcome. If the execution of an action fails another time, the robot is assumed to not be able to overcome the cause of failure on its own. In this case, the control of the robot is handed over to a person, the *remote operator*. Depending on the actual application, this person can have remote or local access to the robot's sensors and manipulators to control it. The remote operation again can be successful of fail. If it is successful, task replanning is performed. If it fails, the current task is discarded.

5.2.2 Task Processor

The composed component TaskProcessor (Figure 5.3) models the system that schedules and manages the execution of tasks. It comprises the subcomponents Controller, PropertyCalculator, ActionExecuter, PlanVerifier, Planner, and the InitialStateProvider. The ports task, prioritizedTask, systemResponse, as well as deleteAllTasks, and abortCurrentTask of component TaskProcessor are all connected to its subcomponent Controller.

Subcomponent Controller manages the administration and execution of incoming tasks. Tasks enter the system via the incoming ports task and prioritizedTask of component type TaskProcessor. The Controller maintains a prioritized queue of tasks that are planned to be executed. Prioritized tasks are added to the front of the queue, whereas tasks entering the architecture via the task port are enqueued at the end of the queue. The currently executed task is aborted if component TaskProcessor receives $true$ via port abortCurrentTask. On receipt of $true$ through incoming port deleteAllTasks, the system deletes all its buffered tasks. The architecture supports

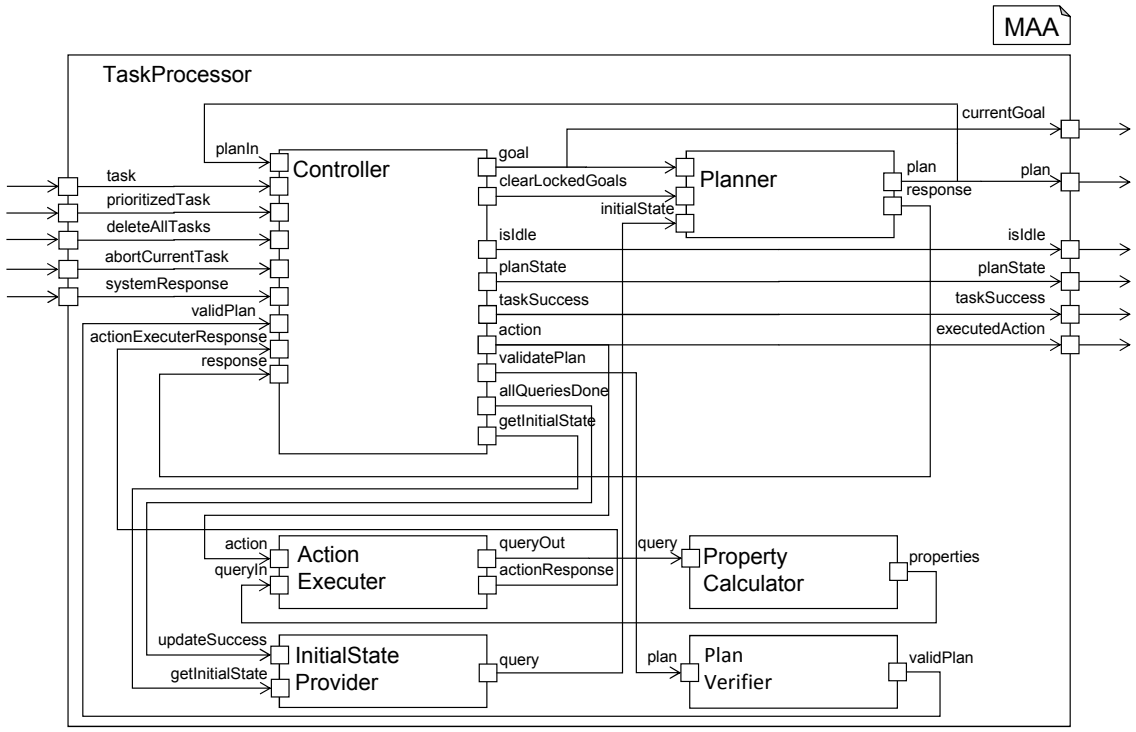


Figure 5.3: An overview of all subcomponents of the `TaskProcessor` component and their interconnections.

system interrupts, e.g., to execute an emergency stop when a physical button located on the robot is pressed. The system receives those interrupts via the incoming port `systemResponse`. A task consists of several goals that have to be satisfied sequentially (see Section 4.4). To satisfy a goal the robot must perform actions such as moving between locations or picking up items. To execute a task, the `Controller` component subdivides the task into its corresponding goals. To compute a plan, i.e., a sequence of actions to satisfy the goal, component `Controller` simultaneously emits the goal through its port `goal` and $true$ via its port `getInitialState`. The latter notifies the component `InitialStateProvider` to calculate the current state of dynamic domain types. Goals may become locked. If a goal is locked, it must be always satisfied by the planner (see Section 4.5). To unlock all locked goals, component `Controller` emits $true$ via its outgoing port `clearLockedGoals`. The component's outgoing port `goal` is connected to the incoming port `goal` of component `Planner`, whereas the outgoing port `getInitialState` is connected to the incoming port `getInitialState` of component `InitialStateProvider`. When component `Controller` receives a plan via its incoming port `planIn`, it forwards the plan via the outgoing port `validatePlan` and expects to receive an answer indicating whether the plan is valid via the incoming port `validPlan`. If a plan is valid, the component starts executing it. Otherwise, the component discards it. To execute a plan, the component decomposes the plan into the corresponding list of actions and tries to execute them in order. To execute an action, it sends a message through its outgoing port `action`. Afterwards it expects to receive a response indicating the result of executing the action via its incoming port

`actionExecuterResponse`. An action may be executed successfully or aborted. While planning, the `Controller` component informs its environment about the current planning state via its outgoing port `planState`. If the system is neither planning nor executing an action, component `Controller` sends $true$ through port `isIdle`.

Component `Planner` calculates and emits plans that can be used to satisfy the goals it receives. Component `Planner` uses the world state it receives via its incoming port `initialState` as the initial state for calculating a plan. The calculated plan satisfies the goal the `Planner` received most recently via its incoming port `goal`. After calculating a plan, the component outputs it via port `plan`. The component additionally sends information about the plan via its outgoing port `response`, e.g., information indicating if the component successfully calculated a plan or if the plan is empty (meaning that the goal is already fulfilled). Both outgoing ports are connected to component `Controller`, while the outgoing port `plan` is additionally connected to the same named outgoing port of component type `TaskProcessor`. When component `Planner` receives $true$ via port `clearLockedGoals`, it unlocks all currently locked goals.

Task Handling

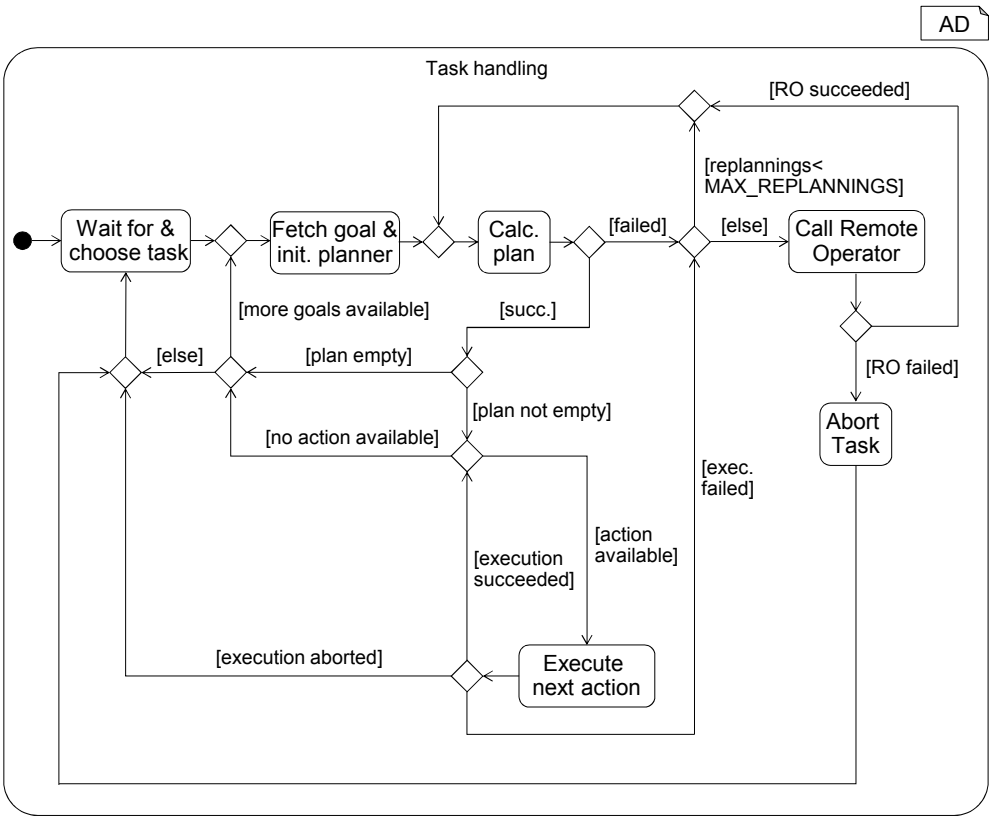


Figure 5.4: An activity diagram illustrating the activities that are performed by the task processor at runtime.

This section provides an overview of the activities performed by the task processor component (Figure 5.3) at runtime. Figure 5.4 depicts an abstract view on the activities

and their control flow. Once the task processor is initialized, it waits for tasks. As soon as tasks are present, the task processor starts executing the task with the highest priority. At first, the task is decomposed into its goals. Afterwards, the system tries to satisfy the goals one after another by performing the following activities: The system tries to calculate a plan to achieve the given goal. The planning process may fail or succeed. If it succeeds and the plan is empty, the task processor is ready to start a new goal. If it succeeds and the plan is not empty, the system tries to execute the plan. If the planning process fails, the task processor either tries to calculate a plan again, if the maximum number of replanning iterations is not reached yet, or calls the remote operator otherwise. In case of a successful remote operation, the system again starts a replanning process. Otherwise, it aborts the task execution and dismisses the current task. To execute a plan, its actions are performed in order of appearance. If the execution of an action fails, the system initiates a replanning process as described above. If the execution is aborted by a user, the system stops executing it and dismisses the current task. If an action is successfully executed and its corresponding plan contains more actions, the system executes the next action. If a successfully executed action is the last one of a plan, the corresponding goal is satisfied. In this case, the system tries to satisfy the next goal, if there is any, or starts the execution of a new task, otherwise.

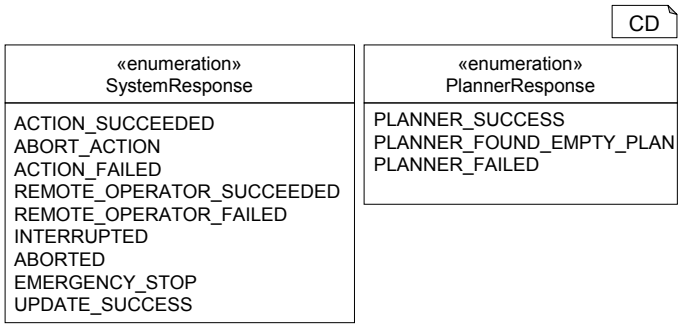


Figure 5.5: A class diagram depicting the types used in the reference architecture. The enumeration `SystemResponse` contains all actions and responses from the environment of the system that the reference architecture is able to respond to. The enumeration `PlannerResponse` contains all response types the `Planner` component responds with.

5.2.3 Data Types

We use a CD to model all types that serve as data types sent via component ports as depicted in Figure 5.5. The CD is comprised of two enumeration classes `SystemResponse` and `PlannerResponse`. `SystemResponse` enumerates all message types that are commands or feedback from the environment (e.g., the middleware) of the system the task processor is running on. `SystemResponse` contains all possible result types of the action execution. Success of the action execution is represented by `ACTION_SUCCEEDED`. The failure of the execution of an action can yield `ABORT_ACTION` or `ACTION_FAILED`. In case of `ACTION_FAILED`, the current goal is planned again. This is performed up to a configurable upper bound of consecutive replanning iterations for a specific goal. If that

bound is exceeded, the system assumes that the robot cannot execute the action on its own, e.g., because it is stuck, lost orientation, the path to a destination appears to be permanently blocked, or further similar scenarios (cf. Section 5.6). `ACTION_FAILED` should be used by robot and world experts for scenarios, which are either assumed to not permanently avoid a successful task execution or which may have alternatives (e.g., more than one path to a certain destination) that could be solved by using another plan. If the upper bound of replanning iterations is exceeded, the control is handed over to the remote operator who can manually maneuver the robot out of the situation that caused the failure and decides whether the failing task should be deleted or planned again. The activities of the remote operator again can have different outcomes. `REMOTE_OPERATOR_SUCCEEDED` denotes that the remote operator managed to solve the blocking event that caused the failure of the execution. In this case the task is planned and executed again. `REMOTE_OPERATOR_FAILED` denotes that the remote operator did not manage to solve the problem. In this case the failing task is discarded. If the outcome of the action execution is `ABORT_ACTION`, the entire task that contains the goal for the failing plan is discarded and no replanning iterations are performed. This appears in scenarios where the robot or world is assumed to impede a successful task execution permanently. For example, a task in which the robot should follow a person who is not registered in the knowledge base. If the `Controller` component has aborted the execution of a task, it sends the message `ABORTED` for further processing. Besides that, `SystemResponse` contains messages to immediately stop (`EMERGENCY_STOP`) or interrupt (`INTERRUPTED`) the execution of tasks on the robot independently of a task. The type `UPDATE_SUCCESS` is used internally for asynchronous message and event handling. It is sent when a response or an event that the system is waiting on has been received or has occurred.

The types of `PlannerResponse` reflect the different results of the planning process. `PLANNER_SUCCESS` denotes that the planner found a non-empty plan to solve a certain goal. `PLANNER_FOUND_EMPTY_PLAN` denotes that the desired situation in the goal that is being planned is already present, i.e., no actions have to be executed by the system to fulfill that goal. `PLANNER_FAILED` denotes that the planner was not able to find a valid order of actions that can solve the current goal.

5.2.4 Controller Component

The `Controller` component (Figure 5.6) is the central part of the task processor that interacts with all other components. It keeps track of all information passed through the task processor and stores a queue with all pending tasks. Internally, the controller is state based. Figure 5.7 depicts a simplified version of this state machine. Here, depending on the current state, different values are sent via the outgoing ports, controlling the behavior of all other components in the task processor. When a component is executed, the MAA infrastructure periodically computes values for the outgoing ports of all components. We refer to each of this computation steps as *tick*. One transition in the automaton depicted in this figure can fire during a single tick. For simplicity, this figure omits all looping transitions that have no explicit stimulus and condition (i.e., a tick that does not change any variable

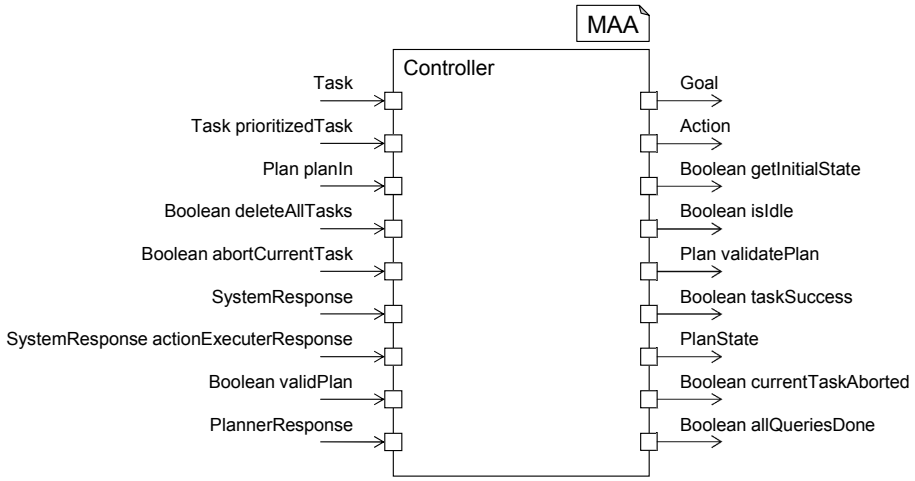


Figure 5.6: The `Controller` component controls the entire task processor. It manages a queue of pending tasks, controls the planning process, and manages the execution of actions.

or state). The handling of interrupts via the implementation of `InterruptHandler` (see p. 30), is also omitted in this automaton. An interrupt can occur in any state, and handling an interrupt can change the current state to any other state. The automaton has the hierarchically decomposed state `working` and the state `emergencyStop`. From any substate of the state `working`, the automaton can change to state `emergencyStop`, and from that state the automaton can change back into the state `working`, i.e., its substate `waitForNewTask`.

Besides controlling the task processor, the controller is responsible for interrupt handling and interacting with the user (e.g., confirming loading of an item on the tablet UI).

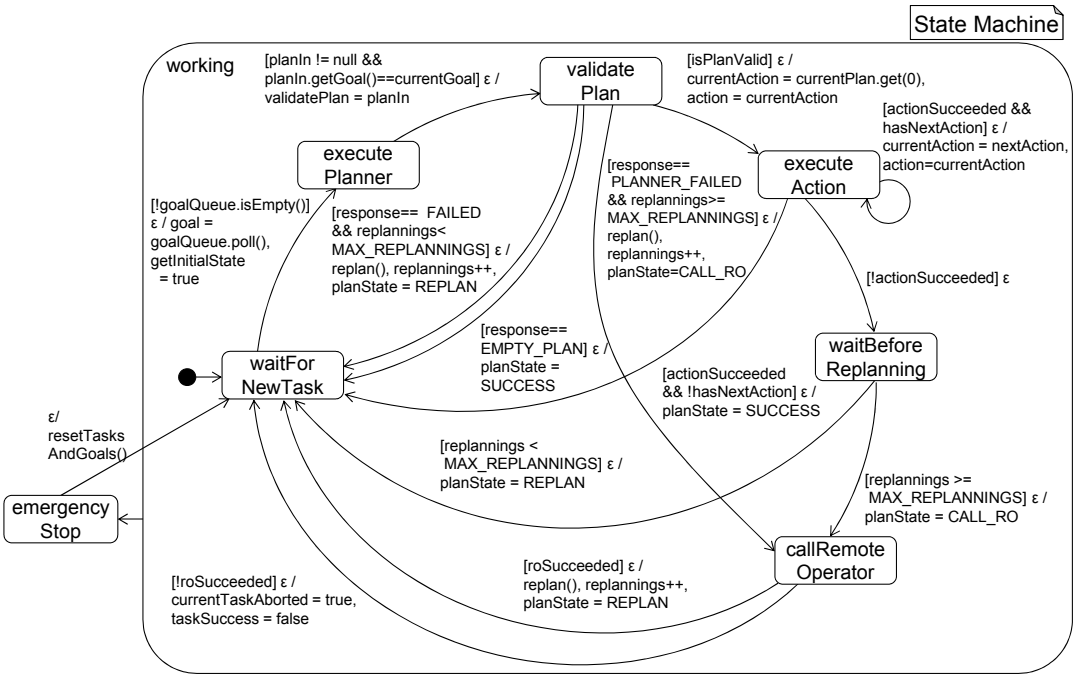


Figure 5.7: The `Controller` component is state-based internally.

A new task can be added to the controller in two different ways: Normally, the new task is enqueued to the task queue. A prioritized task can be directly added as first element of the queue. If the controller receives an interrupt or an abort message via the incoming port `abortCurrentTask`, it handles these events first. On start, the controller is in state `waitForNewTask`. If the task queue is not empty, it takes the first task of the queue and starts to process it. Then it changes to state `executePlanner`. In state `executePlanner`, it sends the current goal of the task to the `Planner` component. It changes to state `validatePlan` indicating that the controller waits for the planner to compute a plan. Next, there are three different cases: If the planner found an empty plan, which means that no actions need to be executed to fulfill the desired situation, it changes back to state `waitForNewTask` and processes the next goal of the current task, or starts a new task in case there is no next goal in the current task. If the planner failed finding a valid plan, the controller changes to state `waitForNewTask`, and increases the counter that counts the number of replanning iterations for the current goal. Then it executes the planner with the same goal again. If the number of replanning iterations exceeds an upper constant bound, it changes to state `callRemoteOperator` and hands over the control to the remote operator. If the planner found a valid, non-empty plan, the controller checks the validity of that plan with component `PlanVerifier`. In case of a valid plan, it changes to state `executeAction` and replans the goal otherwise. In state `executeAction`, all actions of the plan are executed consecutively, until either the end of the list of actions is reached or the execution of an action failed. In the latter case, the controller changes to state `waitBeforeReplanning`. In this state the controller waits a constant time before replanning. This waiting time can be set via the incoming port `waitTimeBeforeReplanning`. Without waiting time, the situation probably has not changed when the action is executed again in comparison to the failing situation. If all actions of a plan are executed successfully, the controller changes back to state `waitForNewTask` and starts to plan the next goal of the task, or a new task if the task was completely executed. The current `planState` and the `taskSuccess` are reported to the environment via outgoing ports of the component controller.

Interrupt Handling

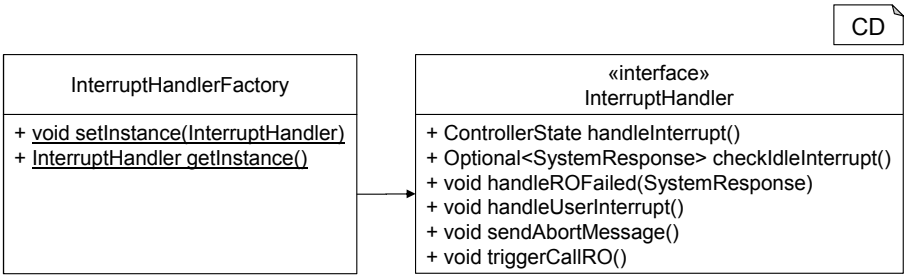


Figure 5.8: The `InterruptHandlerFactory` stores an instance of the interface `InterruptHandler`. An `InterruptHandler` provides several methods for different types of interrupts.

Some events that occur at runtime of the system require the architecture to react independently of its current state. To this effect, the `Controller` supports interrupts. At the beginning of each tick the controller checks whether a new interrupt occurred and handles these before performing anything else. The interface `InterruptHandler`, depicted in Figure 5.8, offers several methods to define the handling of different kinds of interrupts. The implementation of this interface is specific to the used middleware and should be independent of the reference architecture. The `InterruptHandlerFactory` of the reference architecture provides the instance of the `InterruptHandler`, because the implementation of the current interrupt handling method is not known in the reference architecture and, thus, is hidden to the controller. In combination with SmartSoft as middleware, the class `SmartSoftCommunicationProtocol` implements the `InterruptHandler`. The `InterruptHandler` provides the following methods:

- The method `handleInterrupt()` is invoked when an interrupt occurs outside of the architecture and its UIs. It returns the `ControllerState` that the controller changes to.
- `triggerCallRO()` is invoked if the control over the robot is taken over from outside the system.
- The implementation of `handleROFailed(SystemResponse)` determines the necessary operations to be executed in case the remote operator did not manage to solve the problem of robot or world. This could be, e.g., that the robot should return to its home station. However, the current implementation causes the controller to change to state `waitForNewTask` (cf. Figure 5.7) and start the execution of the next queued task.
- The method `checkIdleInterrupt()` checks whether an interrupt from the middleware is received while the reference architecture is in *idle* mode.
- The method `sendAbortMessage()` is used internally to send an abort message to all components of the task processor in case tasks are currently in execution when the control is handed over.

User Interaction

Some task types require interaction with a human, e.g., for loading an item to the robot. For interaction where the robot needs a person to do something, it asks to confirm the operation and the response determines the further behavior of the task processor. User response handling is integrated asynchronously, i.e., the task processor does not wait actively inside of a single tick for a user response.

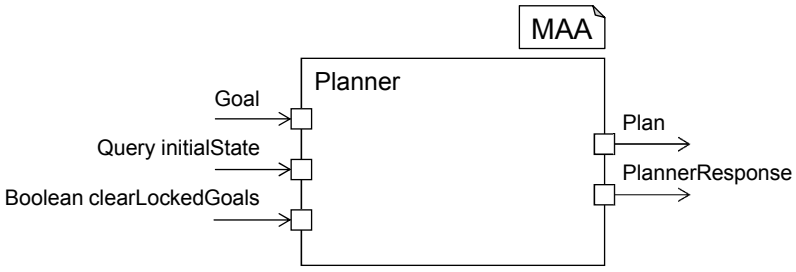


Figure 5.9: The `Planner` component obtains a goal and the current state of the robot and world. Given a set of actions with preconditions and effects, it tries to finds a valid sequence of actions that lead to the goal situation.

5.2.5 Planner Component

Figure 5.9 depicts the `Planner` component. For a given goal on the incoming `goal` port, it calculates and emits a plan on the outgoing port `plan` that satisfies the goal. It uses the world state it receives via its incoming port `initialState` from component `InitialStateProvider` (see Section 5.2.6) as the initial state for calculating a plan to satisfy the goal it received most recently via its incoming port `goal`. After calculating a plan, the component outputs it via port `plan`. The component additionally sends information about the plan via its outgoing port `response`, e.g., indicating if the component successfully calculated a plan or if the plan is empty (cf. Figure 5.5). Both outgoing ports are connected to component `Controller`, while the outgoing port `plan` is additionally connected to the same named outgoing port of component type `TaskProcessor`. When component `Planner` receives *true* via port `clearLockedGoals`, it unlocks all locked goals. Chapter 6 describes the interfaces, classes, and internal behavior of the planner in more detail.

5.2.6 InitialStateProvider Component

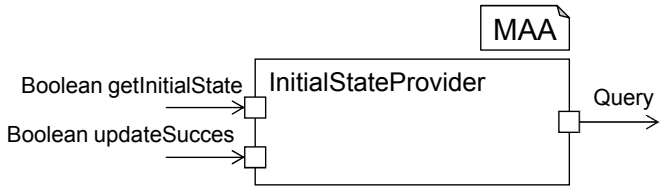


Figure 5.10: On receipt of a flag, the `InitialStateProvider` component provides the planner with the current robot and world state. To do so, it can query properties from the robot and world interface.

Component type `InitialStateProvider`, depicted in Figure 5.10, calculates and provides current robot and world states on request. When the component receives *true* via its incoming port `getInitialState`, it calculates the current world state and emits it via its outgoing port `query`, which is connected to port `initialState` of component `Planner`. The `InitialStateProvider` is implemented to perform the update asynchronously. This is necessary to decouple the update mechanism, which relies

on the middleware, from the execution of the architecture. If the component receives $true$ via the incoming port `getInitialState`, it updates the robot and world state by calling queries of the middleware. Once the responses of the queries have been handled, the controller component sends $true$ via its outgoing port `allQueriesDone`, which is connected to the incoming port `updateSuccess` of the initial state provider. Once this value has been received the response of the queries, in other words the initial state, is emitted via the outgoing port. The implementation of the `InitialStateProvider` is generated from the robot and world entity models (see Section 5.4).

5.2.7 PlanVerifier Component

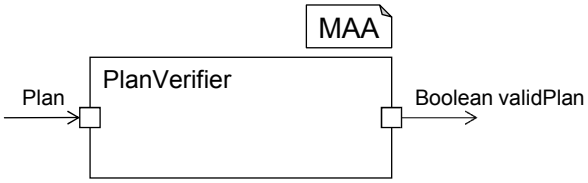


Figure 5.11: The `PlanVerifier` component checks if a passed plan is valid.

Figure 5.11 depicts the `PlanVerifier` component that obtains a plan via its incoming port and outputs a `Boolean` value indicating whether this plan is valid or not. This concept enables employment of further plan analysis. For example, it may check several constraints not reflected in the robot or world interfaces. For example, it can check whether a plan that results from a replanning process equals the former plan, which led to a failing action. A valid order of actions, as well as satisfaction of preconditions are already checked in the `Planner` component.

5.2.8 ActionExecuter Component

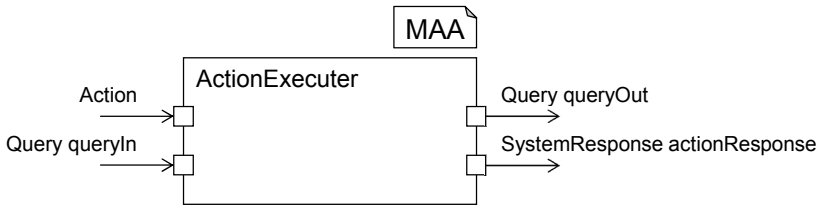


Figure 5.12: The `ActionExecuter` component controls the execution of actions.

Figure 5.12 presents the component `ActionExecuter` that is responsible for the execution of actions. After receiving an action via its incoming port `action`, the component requests information about the current world state via its outgoing port `queryOut`. On receipt of an answer on port `queryIn`, the component starts to execute the action by delegating the execution to the employed middleware. The implementation of the `ActionExecuter` is generated from the robot and world entities (see Section 5.4). The implementations of the entity interfaces that are also generated from entity models then

send the actions to be executed to the middleware of the robot (or the employed world middleware) and adapt the response from the middleware to `SystemResponse` messages that can be further evaluated in the architecture. The component's outgoing port `queryOut` indicates additional information about the action execution.

5.2.9 PropertyCalculator Component

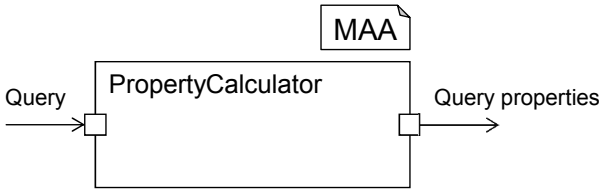


Figure 5.13: The `PropertyCalculator` component obtains queries for dynamic properties of robot and world. It fetches these properties from robot and world interfaces and returns them via the outgoing port `properties`.

Component `PropertyCalculator`, depicted in Figure 5.13, requests properties of the current robot and world states via the employed middlewares. It has an incoming port `query` of type `Query` and an outgoing port `properties`, which is of type `Query` as well. An object of type `Query` consists of a list of properties. A single property represents a map entry mapping a tuple of keys to a single value. For example, an entry may map a one-tuple consisting of a unique person identifier to a String representing the location of the person associated with the identifier. On receipt of a query, the component iterates through all properties the query consists of. While iterating, the component calculates and sets the values associated with the keys of the properties. Eventually, it emits the modified query. The implementation of the `PropertyCalculator` component is generated from robot and world entity models (Section 5.4).

5.3 Runtime Environment

This section describes the runtime environment of the reference architecture that is invariant to an actual application's tasks, domain, entities, and goals. It introduces a logging mechanism that is used system-wide for logging of errors, and as event bus, e.g., for asynchronous communication between the task processor and the user interfaces. This section also explains the interfaces that must be implemented for each robot and each world entity.

5.3.1 Model Realization Base Classes

`Actions`, `Goals`, `Plans`, and `Tasks` each correspond to runtime classes that are extended by the generated classes for each respective model. These runtime classes contain fields and methods that are required in all instances of the respective model type. The class

`Action` contains a variable of type `Identifier`. Two actions are equal to each other if their identifiers are equal to each other. The class `Goal` contains an `Identifier` variable as well. Besides that, it has three Boolean variables, indicating if a goal is locked, unlocked, or parametrized with the command `unlockAll`. Class `Plan` extends an `ArrayList` of `Actions`. To identify, which goal a plan belongs to, the constructor requires it as argument and stores it in a variable of type `Goal`.

Each `Task` object again contains an identifier and a `List` of `Goals` that have to be fulfilled to execute this task. Furthermore, each task can have a `Map<String, String>` that stores goal parameters as type-value pairs. Queries inside the system architecture extend the runtime class `Query`. They are identified by a consecutively numbered, globally unique Integer value and contain a `List` of type `Property`. Each property is identified by a contained variable of type `Identifier`.

5.3.2 Robot and World Interfaces

This subsection describes the entity interfaces generator for robot and world entities. They map properties and actions to middleware specific implementations, for example, to SmartSoft. In this case, property queries are mapped to `tcp-smartsoft` calls, which is described in Section 5.5.3. To keep the entity models of robot and world independent of the used middleware, only interfaces are generated for the entities. From each entity model we generate one interface. That means, for one robot and one world entity, we generate one interface for the world and one for the robot. The interfaces then can be implemented for different middlewares. This enables to exchange the middleware while assisting to retain the same functionality and increases reusability of the application independent entities. An application consists of one robot entity and one world entity. The implementations for each robot and world interface can be accessed via the generated class `EntityManager`. The `EntityManager` contains two static methods to access the current object of the implementation of the entities. To avoid inconsistencies, the entity manager is the only way to access the current world and robot entities. Each generated entity interface has an abstract method `void update()`, which updates the entity state by requesting relevant information from its environment, i.e., the middleware. The entity interfaces contain abstract methods for all actions and properties defined in the entity models.

For each action in the entity model we generate one abstract method without return type. The implementation then should execute the action by implementing the respective method from the interface, for instance with calls to the middleware. Action methods in the interface do not have a return value, since the result of the action execution is checked in the component `ActionExecuter`. This decouples the reference architecture's reaction on a positive/negative action execution from the call to the middleware. We decouple the call from the answer handling, because different middlewares can use different forms of communication with the reference architecture. The storage of the result of the action execution depends on the application, which contradicts to the application independence of entity interface. Thus it is not part of the entity interfaces. For each property defined in an entity model, we generate two abstract methods in the entity interface. Each of

these methods has the same parameters as the properties in the entity model. The return value of these methods is of the same type as the property defined in the entity model it is generated from. One method is used to query the property from the middleware. The other method that is generated from each entity property is used to determine the current state of the query that is stored in the entity implementation. This is required for the planner component. For example, if the robot entity defines that the robot can pick up and deliver *items*, the planner requires to have access to all currently loaded items. The world entity interface also contains methods to query all known objects of each domain type (e.g., items, rooms, and persons). The planner needs these methods to obtain a finite set of all instances of the domain types.

5.3.3 System-Wide Logging

The `RobotLog` class is both used as message logger and event bus. It is implemented as a singleton class that is observable by `java.util.Observers`. The messages that the `RobotLog` logs are of type `LogMessage`. Each message is composed of a time stamp, a `LogType`, and a `String` containing a detailed description of the logged event. A `LogType` again is comprised of a `SeverityLevel` and a `String` prefix and suffix. The severity level can be `ERROR`, `WARNING`, `INFO`, or `DEBUG` (in decreasing order of severity). Each `LogType` has a default severity level, but the severity level can also be adjusted individually for each logged event. To prevent unintended ambiguous naming of `LogTypes`, each domain should define a class that stores all `LogTypes` of its domain as constants. The prefix of a `LogType` indicates the domain the message originates from, e.g., `PLANNER`. It has to be globally unique. The suffix defines the exact message type and must be unique for a given prefix. Table 5.1 depicts a list of all `LogTypes` used in the reference architecture and their default severity level. The `LogType` prefix `LOG` can be used if no domain for a message is specified.

By default, class `SysOutLogger` observes the `RobotLog` and prints all log messages to the console. It can be enabled/disabled with respective methods. Figure 5.14 depicts all methods of `RobotLog` and `SysOutLogger`.

Errors may include an `Exception`. Besides of the logging of events log messages are used for testing purposes. Tests can observe the logger and check whether a certain event has occurred. Additionally, the `RobotLog` is used for the communication of task processor and tablet UI. Since both are executed on different machines, using a logger as event bus is a convenient way to establish simple communication, for instance, for confirming loading an item to the robot via the dedicated tablet UI.

5.4 Generators for iserveU Models

This section explains which parts of the task processor are generated from entity, task, and goal models, which are explained in Chapter 4. Section 5.3.2 explains the generation of entity interfaces and the class `EntityManager`. Implementations for the

Prefix	Suffix	Default Severity
ARCHITECTURE	CONTROLLER_STATE	DEBUG
	IDLE	INFO
	NEW_TASK	INFO
	EXECUTING_TASK	INFO
	ABORTING_TASK	INFO
	FINISHED_TASK	INFO
	TASK_SUCCESS	INFO
	TASK_FAILED	WARNING
	REPLANNING	WARNING
	CALL_REMOTE_OPERATOR	WARNING
	REMOTE_OPERATOR_SUCCESS	WARNING
	REMOTE_OPERATOR_FAILED	WARNING
LOG	DEBUG	DEBUG
	INFO	INFO
	WARNING	WARNING
	ERROR	ERROR
KB	UNKNOWN_PERSON	ERROR
	UNKNOWN_LOCATION	ERROR
	UNKNOWN_ITEM	ERROR
ROBOT	CHARGING	INFO
	EMERGENCY_STOP	WARNING
	ITEM_LOADED	INFO
	ITEM_UNLOADED	INFO
	SUCCESS	INFO
	ERROR_UNKNOWN_LOCATION	ERROR
	ERROR_PATH_BLOCKED	ERROR
	ERROR_PERSON_LOST	ERROR
	ERROR_UNKNOWN_PERSON	ERROR
	ERROR_PERSON_NOT_FOUND	ERROR
	UNKNOWN_RESPONSE	ERROR
SMARTSOFT	NOT_RESPONDING	ERROR
TCP	CONNECTED	INFO
	DISCONNECTED	INFO
	CONNECTION_LOST	WARNING
USERINTERACTION	WAIT_FOR_INTERACTION	INFO
	OK	INFO
	ABORT	INFO
PLANNER	SUCCESS	INFO
	FAILED	ERROR
	EMPTY	INFO

Table 5.1: All `LogTypes` of logging messages in iserveU with their default `SeverityLevel`.

components `ActionExecuter`, `InitialStateProvider`, and `PropertyCalculator` are generated. Further, a code generator generates factories that instantiate the respective generated component implementation. The implementations of these com-

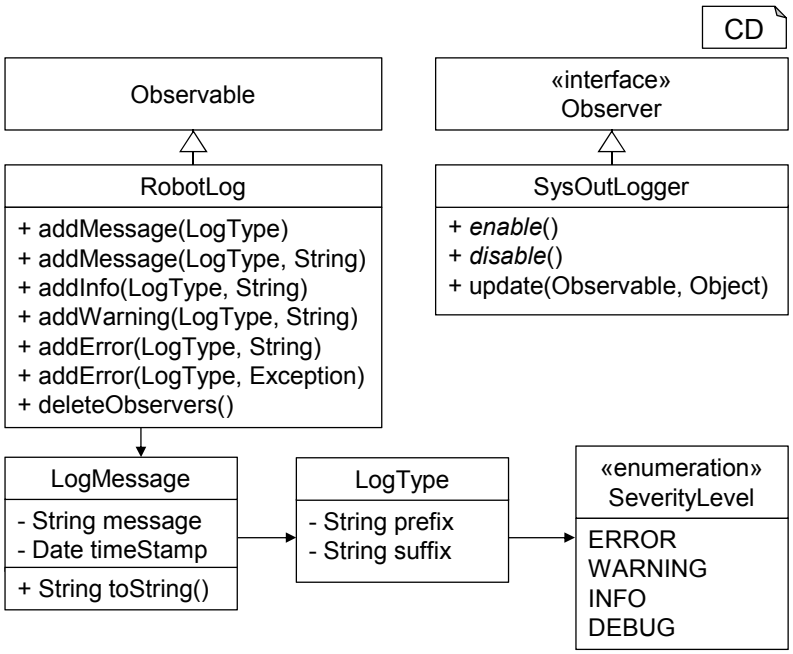


Figure 5.14: The class `RobotLog` is observable. By default, the `SysOutLogger` observes the `RobotLog`. The robot log logs `LogMessages` that consist of a certain `LogType`, a time stamp, and a `String` that describes the logged event. `LogType` consists of a `SeverityLevel`, a String prefix indicating the domain of the logged event, and a suffix String for a detailed description of the logged event type.

ponents are specific to the models of entities. The implementation of the component `IntialStateProvider` makes use of the generated `PropertyFactory` to create property objects for all properties defined in the robot and world interfaces. It calls the generated `stateOfX()` methods that are contained in the generated entity interfaces to set the current state in each property object. It adds all created properties to a list and returns it via its outgoing port, if the component receives $true$ on its incoming port. Similar to the `IntialStateProvider`, the implementation of the `PropertyCalculator` creates new instances of properties and assigns them their current value from robot and world interfaces. Whereas the `IntialStateProvider` queries all available properties of the domain, the `PropertyCalculator` returns properties part of a passed query.

The generated implementation of the `ActionExecuter` contains two methods for each action that is defined in the entities. One method to execute the action and one to send a query to the `PropertyCalculator`. This query is used to obtain the current state of the properties that are used in parameters of an action. The execute method first checks, if the responses for all queries are already available, and then checks the precondition of the respective action. If the precondition holds, the actions' method of the entity interface is called. The implementation of this method is responsible to delegate the action execution to the employed middleware. If the precondition is violated, the implementation of the action executer returns $false$ via the outgoing port `actionResponse`. A successful action execution is never sent from the generated implementation of the action executer. It

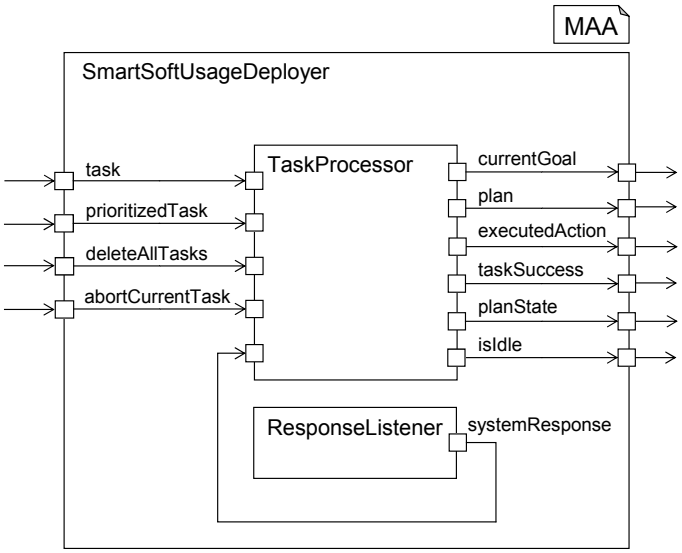


Figure 5.15: The `SmartSoftUsageDeployer` component is composed of the two components `TaskProcessor` and `ResponseListener`. It is the outermost component, which is deployed to the target system and started with class `SmartSoftUsageStarter`.

has to be injected into the reference architecture by the implementation of the `ResponseListener` component (see Section 5.5.2). Besides the component implementations, the generator generates implementations for all action and goal models for and all properties contained in the entity models. These implementations extend the runtime classes presented in Section 5.3.1. Additionally, a factory and an enumeration with identifiers for each model type are generated. The factories create instances of all models of a model type. For example, there is an `ActionFactory` that is able to create all types of actions that are modeled as part of the entire system. For task models, only a factory and an enumeration type with identifiers is generated.

5.5 Application-Specific Architecture Parts

This section describes the parts of the reference architecture that depend on a specific underlying middleware for the reference architecture. The middleware that is used for the implementation is SmartSoft. The section starts with a description of the MAA components that are used for this application scenario, the `SmartSoftUsageDeployer` and the `ResponseListener`. Section 5.5.3 explains the robot and world interface implementations for SmartSoft. The communication between the reference architecture and SmartSoft is explained in Section 5.5.4 and Section 5.5.5. Afterwards, the generated data types and the knowledge base (Section 5.5.6) implementations are introduced. The section ends describing of the architecture starter (Section 5.5.7).

5.5.1 SmartSoftUsageDeployer Component

Figure 5.15 depicts the `SmartSoftUsageDeployer` component. It is comprised of the components `TaskProcessor` and `ResponseListener`. The incoming ports `task` and `prioritizedTask` of the component `SmartSoftUsageDeployer` enable to add tasks either at the front or at the end of the task processor's task execution queue. `deleteAllTasks` resets this queue, and `abortCurrentTask` aborts the execution of the currently executed task. The values of the outgoing ports can be used for analyzing or monitoring the system state. The outgoing `systemResponse` port of the `ResponseListener` is connected to the `systemResponse` port of the `TaskProcessor`. All incoming ports of the `SmartSoftUsageDeployer` are connected to the `TaskProcessor` and all outgoing ports of the `TaskProcessor` are connected to the composed component `SmartSoftUsageDeployer`.

5.5.2 ResponseListener Component

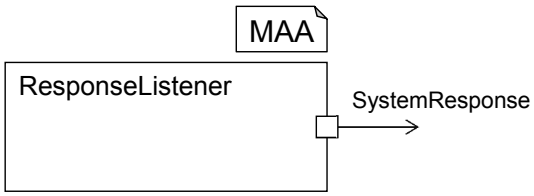


Figure 5.16: The `ResponseListener` component adapts message responses from the middleware to values of `SystemResponse` to be further processed inside of the `TaskProcessor` component.

The `ResponseListener` component, depicted in Figure 5.16, has one outgoing port `systemResponse` of the enumeration type `SystemResponse` (see Section 5.2.3). If the `SmartSoftCommunicationProtocol` has obtained a new `SystemResponse` as result of communication with SmartSoft, it is set as value of the outgoing port and thus integrated into the architecture. The value then is further processed in the `TaskProcessor` component.

5.5.3 Robot and World Implementation

Section 5.3.2 explains the robot and world interfaces that have to be implemented by each specific robot and world entity. This subsection explains the implementation of these interfaces. For example, the robot can move, load, and unload items and the world contains a graph with named locations as vertices and edges indicating adjacent locations. A location is of type `Room` and has a unique name which it can be identified with. An item is of type `Item` and, analog to the room, has a unique name. Besides delivering items, the robot can guide and follow persons. Each person has a name and a beacon id, and is identifiable by the beacon id. The domain types `Room`, `Item`, and `Person` are modeled as CD, from which Java classes are generated. The robot is controlled by the middleware

SmartSoft that stores a model of the world, which is used to build up the world for the architecture. Current values of properties reflected in SmartSoft's domain can be obtained by sending a SmartSoft query via a TCP connection. The value then is the result being sent back from SmartSoft. Similar to that, actions can be sent to SmartSoft via TCP. The result of the action execution is sent back as response to the action. All communication with SmartSoft is performed by the `Communicator` (see Section 5.5.4). The implementation of the entity interface of the world is the class `RoomsWorld`. It maintains a set of all `Rooms` in the world and stores an adjacency map, mapping each room to the set of rooms it is connected to. The class `SmartSoftCommunicationProtocol` wraps the communicator and offers methods to send all possible control commands to SmartSoft, while adapting the responses of these commands back to architecture-internal messages.

5.5.4 SmartSoft Communicator

The singleton `Communicator` class maintains the communication to SmartSoft via a TCP/IP client. The IP address and port it listens on can be set with setter methods to configure them during deployment of the system. The `Communicator` implements the interface `Runnable` and is executed in a separate thread. It can be started and stopped. If it is started, it continuously listens and waits for an incoming message from the `iserveUServer`. That means, a message is sent from SmartSoft to the server, which delegates it to the communicator. This is depicted in Figure 5.17. If a message is present, the communicator enqueues it to a `SynchronizedQueue`. From that queue, the message can be further processed by the reference architecture. This processing is done by the `SmartSoftCommunicationProtocol`.

5.5.5 Communication Protocol

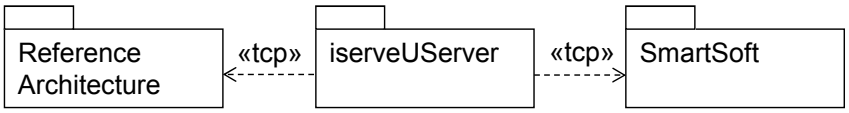


Figure 5.17: The reference architecture and SmartSoft contain a TCP client. These clients communicate with each other via the `iserveUServer` that delegates all messages from one client to the other.

This section gives a short overview of all messages that are exchanged between SmartSoft and the reference architecture. Figure 5.18 lists all responses from SmartSoft and their adaptations to `SystemResponses` used by the architecture. This protocol determines whether a failure of an action execution with a certain error type (e.g., the loss of a person) results in ACTION_FAILED or ACTION_ABORTED. ACTION_FAILED results in replanning of the planner, which in case of multiple consecutive failures eventually lead to a remote operator call, whereas ACTION_ABORTED discards the task that the failing action belongs to. The different success messages from SmartSoft are mapped to

the SystemResponse ACTION_SUCCEEDED. Similarly, the remote operator messages have respective SystemResponse messages.

SmartSoft Response	SystemResponse
(SUCCESS(OK))	ACTION_SUCCEEDED
(SUCCESS NIL)	ACTION_SUCCEEDED
(ERROR(UNKNOWN LOCATION))	ACTION_ABORTED
(ERROR(PATH BLOCKED))	ACTION_FAILED
(ERROR(PERSON LOST))	ACTION_ABORTED
(ERROR(UNKNOWN PERSON))	ACTION_ABORTED
(ERROR(PERSON NOT FOUND))	ACTION_ABORTED
(REMOTE-OPERATION-SUCCESS)	REMOTE_OPERATOR_SUCCEEDED
(REMOTE-OPERATION-FAILED)	REMOTE_OPERATOR_FAILED

Figure 5.18: A list of possible responses from SmartSoft and its adaptions to `SystemResponses` that also are processed in the reference architecture.

The architecture communicates with SmartSoft via a TCP connection (see Section 5.5.4) and follows a strict protocol. SmartSoft and the reference architecture provide communication interfaces via TCP clients. Both clients are connected to the iserveU server that delegates all messages from one client to the other one. Figure 5.19 depicts the commands that are sent to SmartSoft via the `CommunicationProtocol`, depending on the task type that is currently executed in the architecture. The right column represents all possible responses that SmartSoft replies with to the command given in the left column. The architecture reacts differently on different error types, as depicted in Figure 5.18. Some commands are parametrized with room names, symbolic location descriptions, or beacons that belong to a specific person. The values for these parameters are stored to and retrieved from knowledge bases. The information contained in knowledge bases are requested from SmartSoft with queries. All SmartSoft queries used by the architecture are listed in Figure 5.20.

5.5.6 Knowledge Base

The singleton `KnowledgeBase` class administrates the artifacts of the domain and information about them. It provides information about all items, their location and carrying status, and all persons, their location and whether they are currently guided or followed by the robot, which is not covered by SmartSoft. The knowledge base is a central storage that the reference architecture and the UIs can read from and write to. For instance, the UI displays all items from the `KnowledgeBase` that the robot can deliver. When the robot delivers the item it changes its location to the new room. The `KnowledgeBase` provides two methods to store and restore this data. To store an object, the data is serialized as a JSON file, which can be deserialized with the restore method. This enables the user to reuse an existing `KnowledgeBase` from a previous session (cf. Section 7.1).

Request	Possible Responses
(tcb-deliver-from-to ?from ?to)	(SUCCESS NIL) (ERROR(UNKNOWN LOCATION)) (ERROR(PATH BLOCKED))
(tcb-approach-location ?location)	(SUCCESS NIL) (ERROR(UNKNOWN LOCATION)) (ERROR(PATH BLOCKED))
(tcb-follow-person ?person)	(SUCCESS(OK)) (ERROR(PERSON LOST)) (ERROR(UNKNOWN PERSON)) (ERROR(PERSON NOT FOUND))
(tcb-guide-person ?person ?location)	(SUCCESS(OK)) (ERROR(PERSON LOST)) (ERROR(UNKNOWN PERSON)) (ERROR(UNKNOWN LOCATION)) (ERROR(PERSON NOT FOUND))
(tcb-joystick-navigation)	(SUCCESS NIL)
(tcb-stop-joystick-navigation)	(SUCCESS NIL)
(tcb-call-remote-operator)	(REMOTE-OPERATION-SUCCESS) (REMOTE-OPERATION-FAILED)

Figure 5.19: A table containing commands that can be sent to SmartSoft and all possible responses it replies with. Parameters are denoted with a leading '?'.

Query	Example Result
GET-CURRENT-LOCATION	KITCHEN
GET-BATTERY-VOLTAGE	24.1
GET-ALL-LOCATIONS	KITCHEN,ROOM1,ROOM2
GET-ADJACENT-LOCATIONS	(KITCHEN,ROOM1),(ROOM1, ROOM2)

Figure 5.20: A table with queries that can be sent to SmartSoft. The right column gives exemplary query result values.

5.5.7 Architecture Starter

The `Executer` maintains and controls the execution of the `SmartSoftUsageDeployer` component. It offers methods to start, stop, or run the reference architecture execution. Additionally, it has methods to inject values into the component's incoming ports. This is mainly the case if new tasks are added to the architecture or a task should be aborted. For the iserveU project, the executer offers the possibility to configure the reference architecture with different parameters. Among other things, it includes the configuration of an optional idle task that is executed if the robot is idle for a given time. If the idle task is enabled, the robot moves to its home station. Otherwise, it does nothing.

5.6 Exemplary Execution Excerpts

This section presents exemplary excerpts of the architecture information flow at runtime. Section 5.6.1 describes the setting in which the execution of an exemplary delivery task takes place. Afterwards, six scenarios explain the information flow in the scenario situations. The first scenario (Section 5.6.2) is the successful delivery of an item to a target location. The second scenario (Section 5.6.3) describes a situation in which the path is blocked with a temporary obstacle, which disappears after a short time. In the third scenario (Section 5.6.4) the obstacle does not disappear, but a remote operator manages to maneuver the robot to circumvent it. In the fourth scenario (Section 5.6.5), the remote operator is not able to circumvent the obstacle. The fifth (Section 5.6.6) and sixth (Section 5.6.7) scenarios depict situations where the task is aborted via the desktop UI and the tablet UI.

5.6.1 Setting

In all of the following exemplary excerpts, the architecture attempts to execute a `deliver` task. The task consists of one goal, and makes the robot deliver a configurable item to a configurable location. In the following examples, the robot should deliver the item *bandages* to room *r4043*. The robot is initially located in room *r4043*, while the item *bandages* is initially positioned in room *seminarraum*. To reach room *seminarraum* from room *r4043*, the robot has to traverse room *flur*. Vice versa, the robot has to traverse room *flur* to reach room *r4043* from room *seminarraum*.

5.6.2 Successful Deliver

The sequence diagrams depicted in Figure 5.21, Figure 5.22, Figure 5.23, and Figure 5.24 illustrate an exemplary excerpt of the information flow of the architecture on receipt of a new task. Further information about the task and the execution environment are given in Section 5.6.1. In this example, the task is executed successfully without any incidents.

The execution starts with component `controller` receiving the task (Figure 5.21) and ends after the task has been completely executed (Figure 5.24). Figure 5.21 illustrates the planning process and the execution of the first action. On receipt of the `deliver(r4043,bandages)` delivery task, component `controller` signalizes component `initialStateProvider` to output the current state. Further, it sends the first goal of the task to component `planner`. The `planner` component waits for the arrival of an initial state when receiving a new goal. Once component `planner` receives the initial state from component `initialStateProvider`, it calculates and outputs a plan that satisfies the goal under consideration of the initial state it received. In the next time slice, component `controller` receives the plan and forwards it to component `planVerifier`. After checking the plan, component `planVerifier` sends a message, indicating whether the plan satisfies its requirements to component `controller`. In this case, the plan is valid. Since the plan is valid, component `controller` extracts the actions that are necessary to fulfill the plan. The component sends the first action,

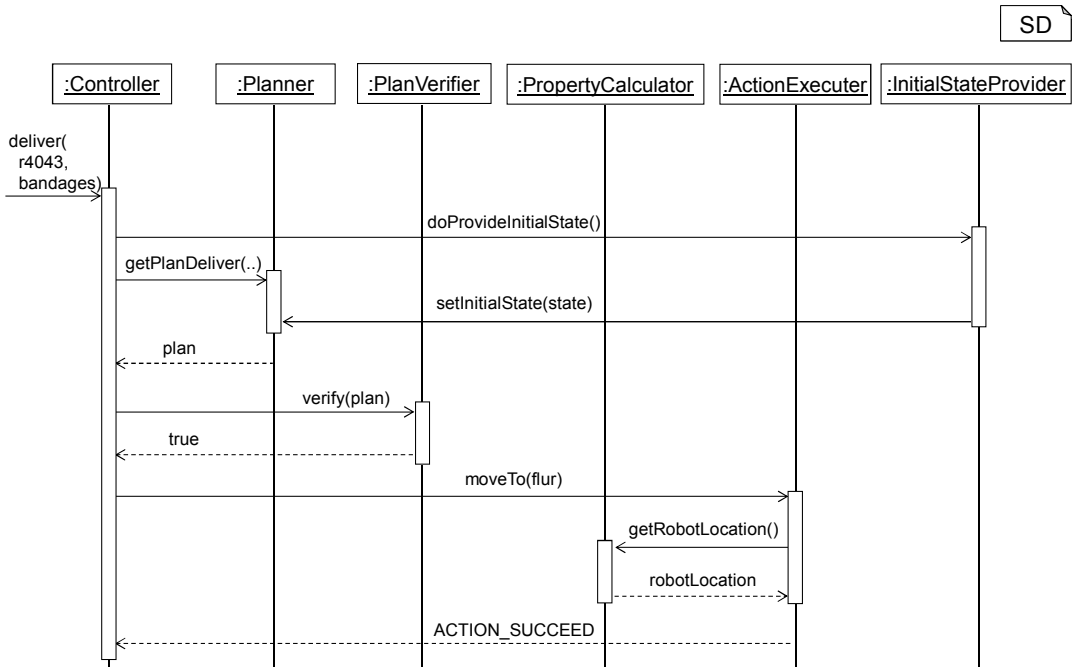


Figure 5.21: The planning process of a delivery task and the successful execution of a `moveTo(flur)` action.

which should make the robot move to room *flur*, to subcomponent `actionExecuter`. The `actionExecuter` has to check if the action's precondition is satisfied, hence it emits the corresponding query to component `propertyCalculator`, which answers with the appropriate properties. In this case, the action precondition is satisfied, the `actionExecuter` successfully executes it, and sends information about the execution to component `controller`.

Figure 5.22 depicts the execution of the next two actions. Component `controller` tells component `actionExecuter` to execute the action `moveTo(smrRaum)`. Component `actionExecuter` has to check if the action's precondition is satisfied, hence it emits the corresponding query to the `propertyCalculator` component. The component `propertyCalculator` answers with the appropriate properties. In this case, the action's precondition is satisfied, the `actionExecuter` successfully executes it, and sends information about the execution to component `controller`. Next, the `controller` component initiates the execution of the action `pickup(bandages)` by sending an appropriate message to component `actionExecuter`. The action execution is successful. Afterwards, component `actionExecuter` notifies component `controller` of the successful execution. In the end, the robot has to drop the item. The execution of these four actions is analogous to the previously described ones and is illustrated in Figure 5.23 and Figure 5.24.

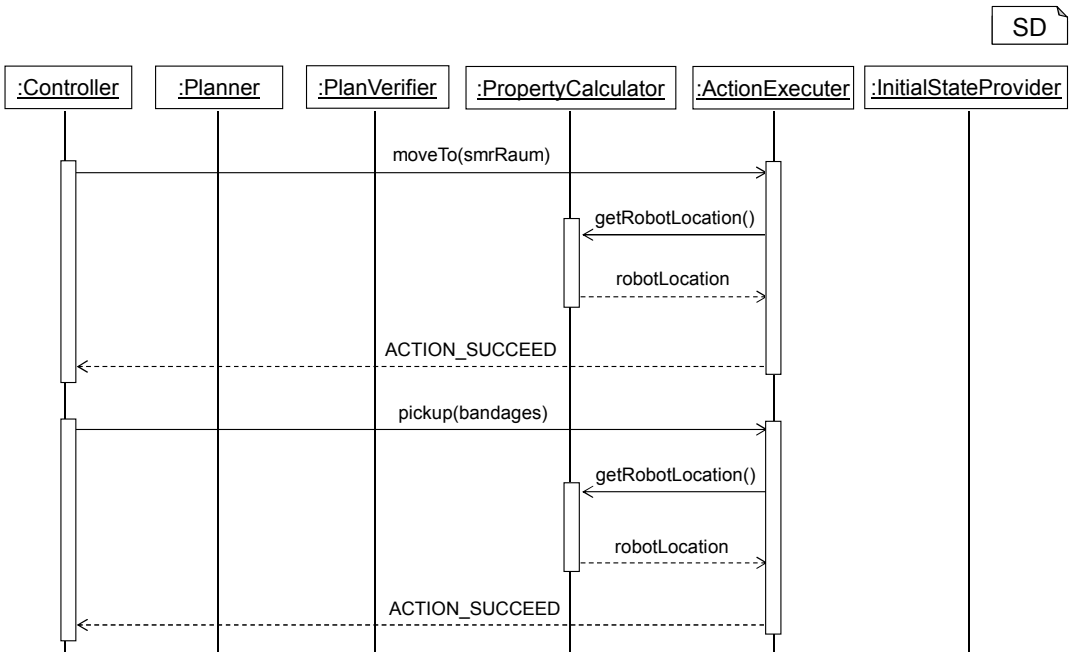


Figure 5.22: Successful execution of two tasks.

5.6.3 Path Temporarily Blocked

In this example, the path between room *r4043* and room *flur* is blocked when the robot tries to move from room *r4043* to room *flur*. After a while, the blockade between the two rooms is lifted. This situation is depicted in the sequence diagrams Figure 5.25.

In Figure 5.25 the architecture receives a new task and determines the actions to execute. The execution of the action `moveTo(flur)` fails because the path between room *r4043* and room *flur* is blocked. On receipt of the task, component `controller` signalizes component `initialStateProvider` to output the current world state. Further, it sends the goal of the task to component `planner`. The `planner` component waits for the arrival of an initial state when receiving a new goal. Once component `planner` receives the initial state from component `initialStateProvider`, it calculates and outputs a plan that satisfies the goal under consideration of the initial state it received. In the next time slice, component `controller` receives the plan and forwards it to component `planVerifier`. After checking the plan, component `planVerifier` sends a message, indicating whether the plan satisfies its requirements, to component `controller`. Since the plan is valid, component `controller` extracts the actions that are necessary to fulfill the plan. The component sends the first action, which should make the robot move to room *flur*, to subcomponent `actionExecuter`. The `actionExecuter` has to check if the action's precondition is satisfied, hence it emits the corresponding query to the `propertyCalculator` component, which answers with the appropriate properties. In this case, the action's precondition is satisfied, but the `actionExecuter` fails. The `actionExecuter` component sends a message to the `controller` component to indicate that the action execution has failed. The `controller` proceeds by executing the same task again. This is the same situation as depicted in Figure 5.21, except that the

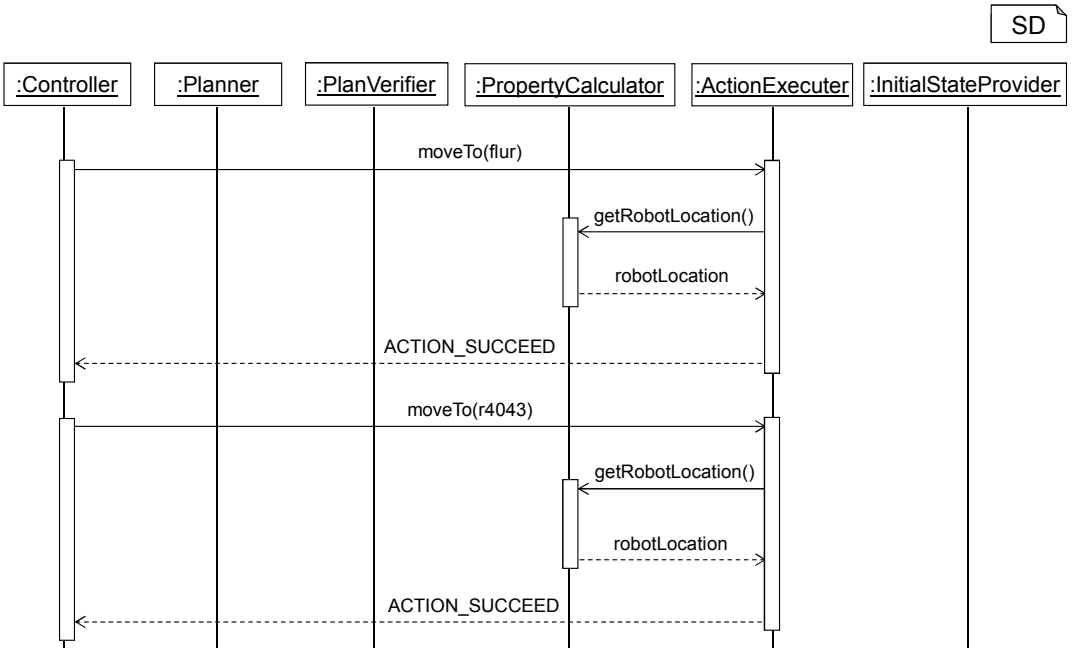


Figure 5.23: Successful execution of the tasks `moveTo(flur)` and `moveTo(4043)`.

`controller` does not receive the task from an external source, but from an internally maintained stack. This time, the execution of the action `moveTo(flur)` is successful since the blockade is lifted. The remaining architecture behavior to execute the task is illustrated in Figure 5.22, Figure 5.23, and Figure 5.24, analogous to the first scenario described in Section 5.6.2.

5.6.4 Path Permanently Blocked, Remote Operator Successful

In this example, the path between room *r4043* and room *flur* is blocked when the robot tries to move from room *r4043* to room *flur*. Further information about the task and the execution environment is given in Section 5.6.1. The blockade between the two rooms exists during the whole execution excerpt. Also, the robot is able to move the other way around from room *flur* to room *r4043* without any problems. Although the robot cannot move from room *r4043* to room *flur* by itself, the remote operator can navigate the robot through that bottleneck.

In Figure 5.25, the architecture receives the task. The action `moveTo(flur)` fails because the path from room *r4043* to room *flur* is blocked. The task is planned again, and because the path is still blocked, the situation described in Figure 5.25 occurs again. Since the execution of the action failed two times, component `controller` calls the remote operator (Figure 5.26). The remote operator is able to navigate the robot through a bottleneck. The remaining actions that are executed after a further replanning process are illustrated in Figure 5.21, Figure 5.22, Figure 5.23, and Figure 5.24.

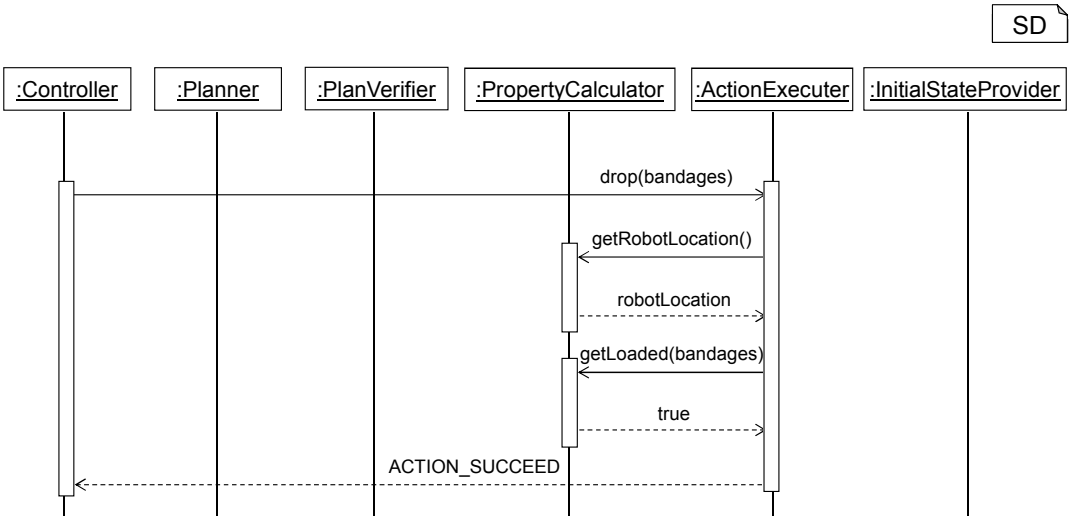


Figure 5.24: Successful execution of the task `drop(bandages)`.

5.6.5 Path Permanently Blocked, Remote Operator Not Successful

The sequence diagrams depicted in Figure 5.25 and Figure 5.27 illustrate an exemplary excerpt of the information flow of the architecture on receipt of a new task. Further information about the task and the execution environment is given in Section 5.6.1. In this example, the path between room *r4043* and room *flur* is blocked when the robot tries to move from room *r4043* to room *flur*. The blockade between the two rooms exists during the whole execution excerpt. Neither the robot can move from room *r4043* to room *flur* by itself, nor is the remote operator able to make the robot to bypass blockade.

In Figure 5.25, the architecture receives the task. The action `moveTo(flur)` fails because the path from room *r4043* to room *flur* is blocked. The task is planned again, and the action execution fails again because the path is still blocked (Figure 5.25). Since the execution of the action failed two times, component `controller` calls the remote operator (Figure 5.27). The remote operator is not able to make the robot cross over the blockade, either. Therefore, the `controller` discards the entire task since it cannot be executed. Afterwards, the architecture is ready to start executing of a new task.

5.6.6 Abort via Desktop UI

The sequence diagram in Figure 5.28 illustrates an exemplary excerpt of the information flow of the architecture on receipt of a new task. Further information about the task and the execution environment is given in Section 5.6.1. In this example, component `controller` is instructed to abort the task after it initiated the execution of an action.

On receipt of the task `deliver(r4043,bandages)`, component `controller` signalizes component `initialStateProvider` to output the current world state, and sends the appropriate goal to component `planner`. The `planner` component waits for the arrival of an initial state when receiving a new goal. Once component

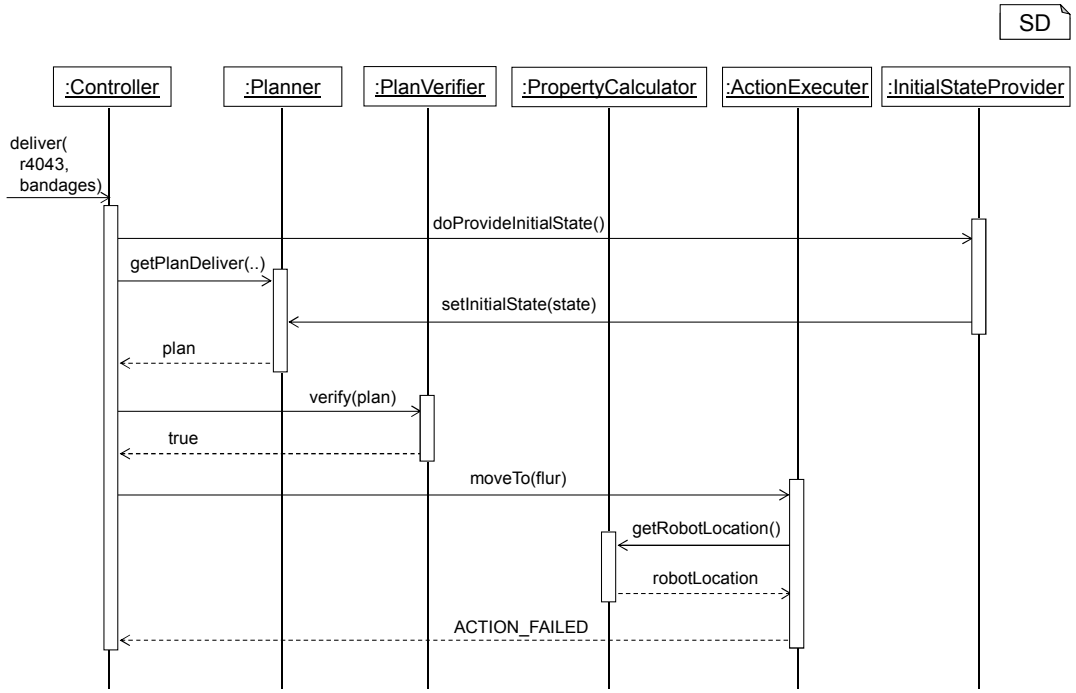


Figure 5.25: The planning process of a delivery task and the execution of a `moveTo(flur)` action, which fails due to a blocked path.

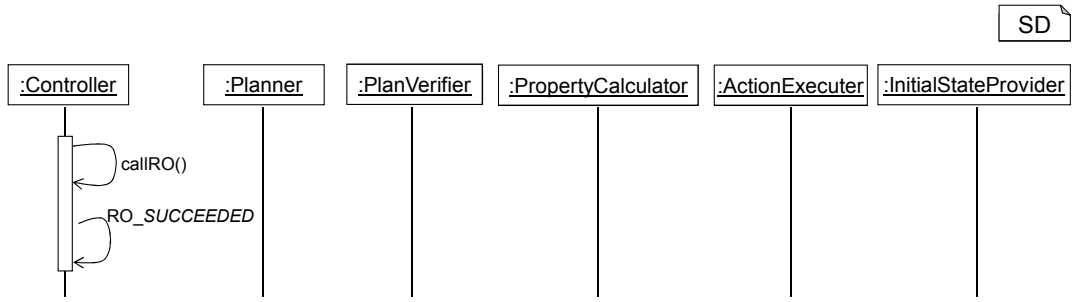


Figure 5.26: In this excerpt, component `controller` calls the remote operator. The remote operator succeeds.

`planner` receives the initial state from component `initialStateProvider`, it calculates and outputs a plan that satisfies the goal under consideration of the initial state it received. In the next time slice, component `controller` receives the plan and forwards it to component `planVerifier`. After checking the plan, component `planVerifier` sends a message, indicating whether the plan satisfies its requirements, to component `controller`. In this case, the plan is valid. Since the plan is valid, component `controller` extracts the actions that are necessary to fulfill the plan. The component sends the first action, which should make the robot move to room *flur*, to subcomponent `actionExecuter`. Now, two events happen simultaneously. While component `actionExecuter` checks if the actions precondition is satisfied by emitting an appropriate query to component `propertyCalculator`, component `controller` is instructed to abort the execution of the task that is currently executed. To do so, it

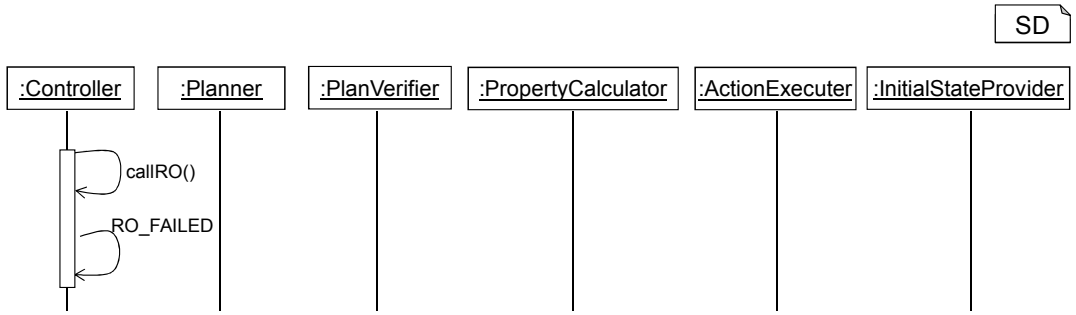


Figure 5.27: In this excerpt, component `controller` calls the remote operator. The remote operator fails.

interacts with an interface that is responsible for the physical realization of the actions performed by the robot. In this case, the task is aborted successfully. Nevertheless, component `actionExecuter` still tries to execute the action `moveTo(flur)`. To realize the execution, the component interacts with the same interface as component `controller` does to abort tasks. The interface informs component `actionExecuter` that the task has already been aborted (by component `controller`). Since the task has been aborted by component `controller`, component `actionExecuter` does not inform the `controller` about the result of the action execution. Now, the architecture is ready for starting a new task.

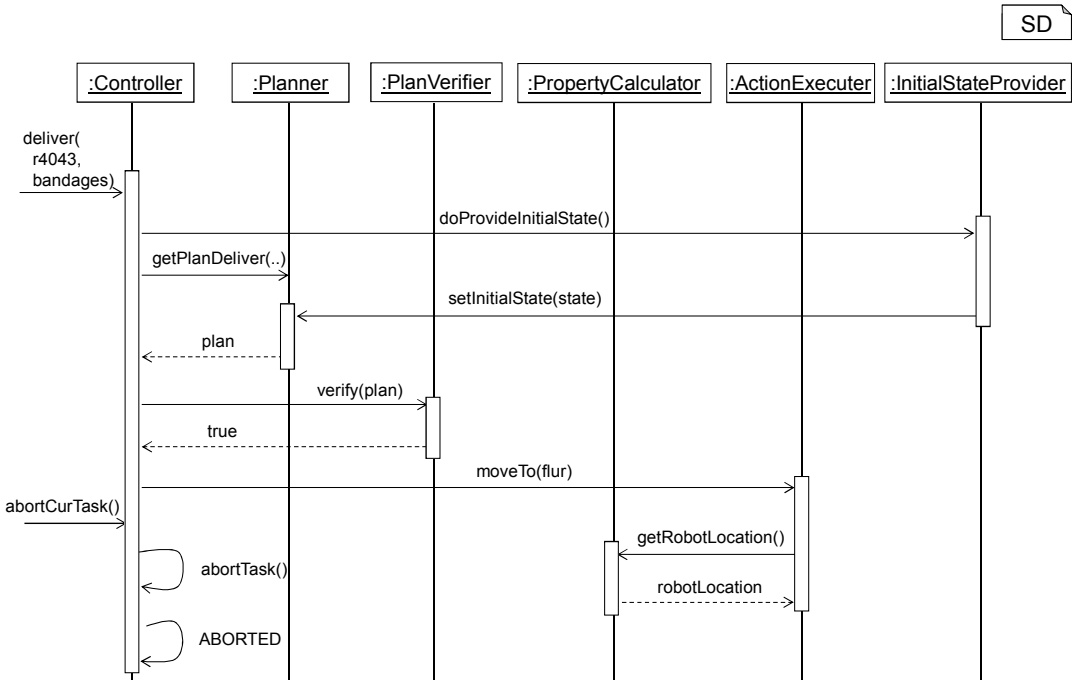


Figure 5.28: The execution of the task is aborted by the user during execution of the first action.

5.6.7 Abort via Tablet UI

The sequence diagrams depicted in Figure 5.21 and Figure 5.29 illustrate an exemplary excerpt of the information flow of the architecture on receipt of a new task. Further information about the task and the execution environment is given in Section 5.6.1. In this example, a user, who has to load the item *bandages* onto the robot, aborts the task execution. Figure 5.21 illustrates the necessary operations to make the robot move to room *flur*. In Figure 5.29, the user aborts execution of the task.

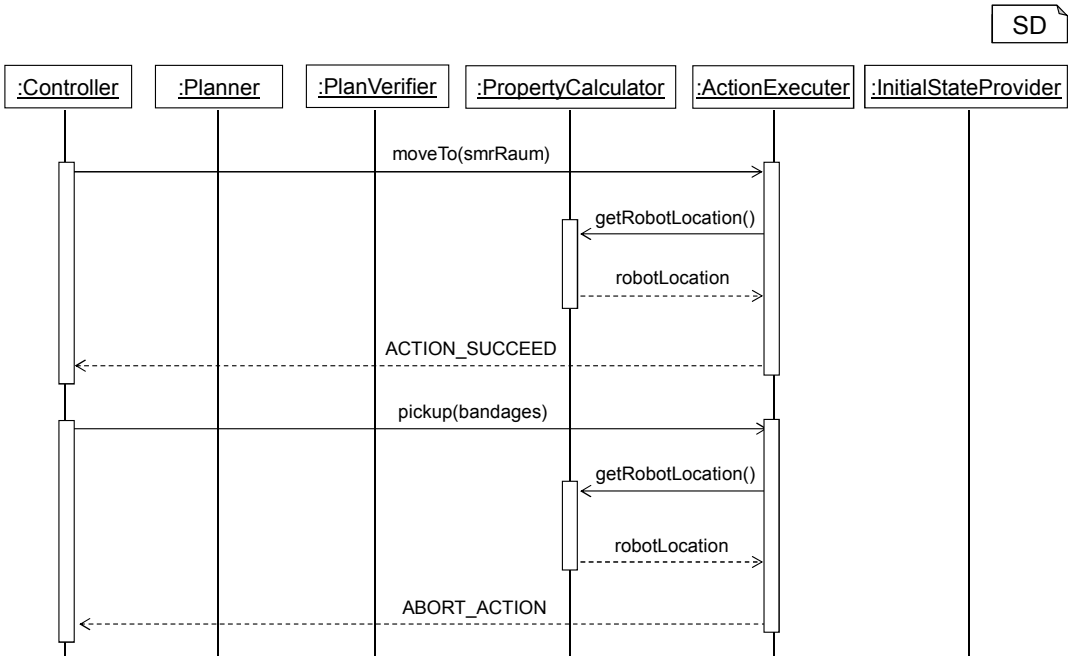


Figure 5.29: The execution of the `moveto(smrRaum)` action succeeds. Afterwards the execution of the `pickup(bandages)` action is initiated. As part of a pickup action, the user is prompted to load an item to the robot. If the user aborts this prompt, the action execution fails.

In Figure 5.29, component `controller` tells component `actionExecuter` to execute the action `moveTo(smrRaum)`. The `actionExecuter` has to check if the actions precondition is satisfied, hence it emits the corresponding query to component `propertyCalculator`, which answers with the appropriate properties. In this case, the actions precondition is satisfied, the `actionExecuter` successfully executes it, and sends information about the execution to component `controller`. Next, the `controller` component initiates the execution of the action `pickup(bandages)` by sending an appropriate message to component `actionExecuter`. Again, component `actionExecuter` has to check if the actions precondition is satisfied. In this case, the precondition is satisfied and the component starts the execution. As part of the action execution, a user is prompted to load an item on the robot. The user can either confirm or deny the loading by pressing one of two buttons on a UI. In this case, the user denies the loading, which is interpreted as the abortion of the action. Therefore, `actionExe-`

`cuter` aborts the action and indicates that by sending the message `ABORT_ACTION` to component `controller`.

5.7 Discussion

This section discusses, which parts of the architecture belong to robot or world entity and which parts depend on the used middleware. It explains the parts that have to be exchanged if a middleware different than SmartSoft is used. In the hospital application, the robot entity describes a robot that can move, load, and unload items. The task kinds are delivery, guide, and follow tasks. The world entity describes a graph with named locations as vertices and edges indicating adjacent locations. A major goal of the generality of the reference architecture is its independence of specific robot and world entities. If a different robot entity is used, only few reconfigurations have to be incorporated to the architecture. For instance, consider an entity that realizes refilling of storage shelves. Besides moving, the robot entity has to define actions to refill the storage spaces with a manipulator. In this scenario, the world entity would not consist of rooms and their interconnections, but of storage shelves that contain storage spaces of different sizes. To change the application scenario, the robot and world entity models have to be exchanged. The exemplary change of entities also requires to redefine new tasks, as for instance a task to find an empty storage shelf/space or a task to perform refilling of an empty space. The implementations of the new robot and world interfaces have to be handcrafted. Defining new tasks can require to define new goal models.

Section 5.5 explains which parts of the overall architecture are depending on the used middleware. If the middleware is exchanged with another middleware, as for instance, ROS, these parts have to be modified. The parts include, among other things, the communication with the middleware. A protocol for the communication with the middleware has to be implemented and the responses of the middleware have to be mapped to `SystemResponses` that the reference architecture can handle. A more detailed description of the middleware dependent architecture parts is given in Section 8.1. The modeling languages define platform and middleware independent models. Thus, the models can remain unchanged when exchanging the middleware.

Chapter 6

Translating Domain Expert Models to Robot Plans

Robotics application developers use the modeling languages presented in Chapter 4 to describe domain knowledge, tasks consisting of sequences of goals, and entities consisting of properties and actions. Solving tasks requires to derive actions to satisfy their accompanying goals step-by-step. Due to the fact that satisfying goals is a classic planning problem, we translate entities with actions and properties as well as domain knowledge into a planning language, use a planner to solve each goal, and transform the results back to models. While this incorporates two model transformations, it decouples planning functionality from the other system parts and hence allows independent planner development as well as an exchange on demand.

PDDL [MGH+98, FL03] is a widely used artificial intelligence planning language for which sophisticated planners exist. In our implementation of the reference architecture we employ the Metric-FF [HN01] planner to solve goals. Therefore, our planning infrastructure transforms entities, UML/P CDs, and goals to PDDL and supports solving PDDL problems with the Metric-FF [HN01] planner. The resulting artifacts are compatible to PDDL version 1.2. This project does not ship with any files of the Metric-FF planning software. To integrate Metric-FF, the corresponding software has be to installed and deployed as described in Chapter 8.

The planning infrastructure is well integrated into the system architecture described in Chapter 5. The `Planner` component type implementation (Section 5.2.5) uses this infrastructure to calculate plans for goals. Figure 6.1 overviews the main modules of the planning infrastructure and their dependencies to each other. At design time, application developers define entity models with properties and actions, UML/P CD models, which specify the domain knowledge utilized by the entities, and goals that represent desired environmental situations (Chapter 4). The infrastructure uses the parsers of the entity, UML/P CD, and goal languages for parsing corresponding models to obtain data structures encoding the abstract syntax of the models, the so called abstract syntax trees (ASTs). Afterwards, the infrastructure applies several intra-language model-to-model transformations to the entity ASTs to ensure some uniqueness of names properties for the PDDL artifacts generated later. Then, it uses the ASTs to create intermediate data structures which represent complete PDDL domain definitions and partial PDDL problems definitions. To complete the problem definitions, the infrastructure has to be provided

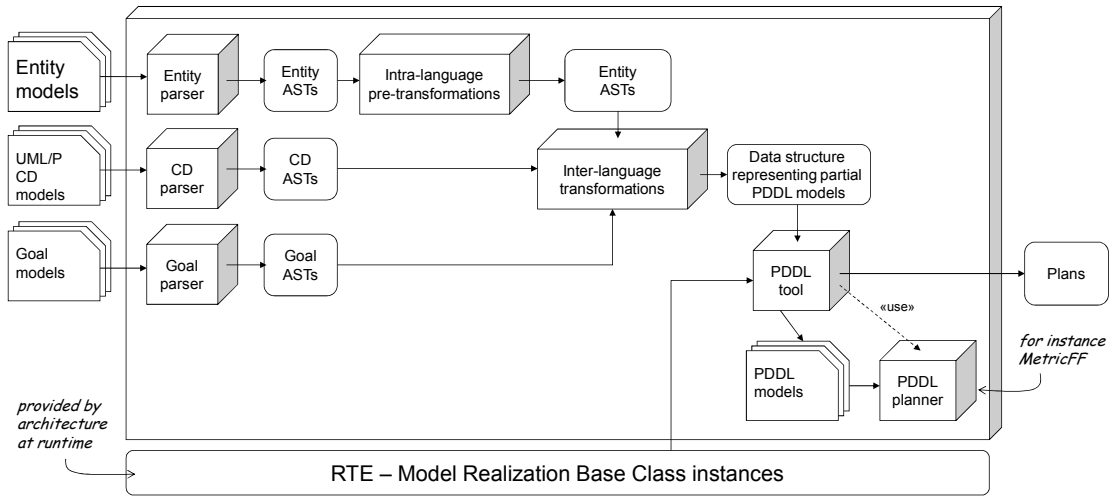


Figure 6.1: Overview of the transformation infrastructure.

Entity

```
1 package iserveu.worlds;
2 world transportWorld {
3   action openDoor() { /* ... */ }
4   property Boolean doorOpen(); //...
5 }
```

Listing 6.2: A world entity model excerpt consisting of a property `doorOpen` and an action `openDoor()`. The property and action transformations have not been applied to this entity model.

with more information, which is only available at runtime, such as an initial planning state or arguments for goals. Once these information are available in form of objects that are instances of RTE classes (Section 5.3.1) corresponding to goals and properties, PDDL models can be generated and the planning problem defined by these can be solved. The infrastructure outputs a plan in form of the planner's output and a mapping from the object names the planner used for planning to the corresponding instances of the RTE classes.

In the following, Section 6.1 describes intra-language pre-transformations. Based on their results, Section 6.2 presents the transformations from goal, entity, and domain models to PDDL. Finally, Section 6.3 discusses the transformations.

6.1 Intra-Language Model-to-Model Transformations

A single PDDL domain model is generated from two entity models, i.e., a robot entity and a world entity. To ensure the generated PDDL models are well-formed, an action transformation and a property transformation have to be applied to the entity ASTs between parsing and generating. Given a well-formed entity model, the action transformation

Entity

```
1 package iserveu.worlds;
2 world transportWorld {
3   action iserveu_worlds_transportWorld_openDoor() {/* ... */}
4   property Boolean iserveu_worlds_transportWorld_doorOpen();
5 }
```

Listing 6.3: This entity results from applying the property and action transformations to the world entity model depicted in Listing 6.2. The names of the property `doorOpen` and the action `openDoor` have been extended.

CD

```
1 classdiagram Items {
2   class Pencil { Location loc; }
3   class ColoredPencil extends Pencil { /* ... */ }
4   class Location { /* ... */ } // ...
5 }
```

Listing 6.4: A CD of three classes. Each of the classes is transformed to a PDDL type and used in the PDDL domain definition.

PDDL

```
1 (define (domain myDomain)
2   (:types
3     Pencil - object
4     ColoredPencil - Pencil
5     Location - object  ; ...
6   )
7   (:predicates
8     (Pencil_loc ?pencil - Pencil ?loc - Location) ; ...
9   ) ; ...
10 )
```

Listing 6.5: A domain containing the types and predicates resulting from transforming the classes of the CD depicted in Listing 6.4.

Entity

```
1 world myWorld {
2   property Room personLocation(Person person);
3   property Boolean itemLocation(Item item, Room room); //...
4 }
```

Listing 6.6: An entity that comprises two properties.

ensures that there is no action defined in the entity model having the same name as any other action defined in any other entity model. To this effect, the transformation prefixes each action name with the full-qualified name of its enclosing entity using underscores (_) as package delimiters. Similarly to the action transformation, the property transformation prefixes the full qualified name of the enclosing entity of each property to the name of the property using underscores (_) as package delimiters. This ensures there are no properties defined in distinct entity models having the same names. For instance, the entity model given in Listing 6.2 consists of a property `doorOpen` and an action `openDoor`. The transformation prefixes the names of the property and the action with the full-qualified names of their enclosing entity using underscores (_) as package delimiters. Listing 6.3 depicts the entity representing the transformed AST.

6.2 Transformations to PDDL

This section presents the transformation of goal, entity, and UML/P CD models to PDDL models. The transformations are described and exemplary illustrated with the aid of examples. For simplicity, the examples do not include the AST transformations (see Section 6.1) that are applied to the entity models beforehand. Additionally, we assume the reader is familiar with PDDL 1.2 [MGH+98].

6.2.1 Transformation of CD Models to PDDL Types and Predicates

Each class defined in a CD model is transformed to a PDDL type and each attribute of each class is transformed to a PDDL predicate. The generated PDDL types preserve the inheritance relation defined by the class diagrams and have the same names as the classes they originate from. For instance, the classes of the CD given in Listing 6.4 are transformed to the PDDL types depicted in Listing 6.5 ll. 2-6. Each attribute of each class of a CD model is transformed to a PDDL predicate. The name of the predicate is the name of the corresponding class concatenated to an underscore (_) and the name of the attribute. The predicate has two variables. The first variable has the type of the class, whereas the second variable has the type of the attribute. The class `Pencil` defined by the CD illustrated in Listing 6.4, for instance, has one attribute `loc`. Thus, the class defines together with its attribute `loc` the binary PDDL predicate `(Pencil_loc ?pencil - Pencil ?loc - Location)` (Listing 6.5, l. 8).

6.2.2 Transformation of Entity Properties to PDDL Predicates

Each property of each entity is transformed to a PDDL predicate. A property has parameters and a return value, whereas PDDL predicates correspond to parametrized Boolean predicates. Therefore, a property that has n arguments is transformed to a $(n + 1)$-ary PDDL predicate. The first n predicate parameters have the PDDL types corresponding

PDDL

```
(define (domain myDomain)
  (:predicates
    (personLocation ?person - Person ?result - Room)
    (itemLocation ?item - Item ?room - Room ?result - Boolean)
  ) ; ...
)
```

Listing 6.7: The predicates that result from transforming the properties of the entity depicted in Listing 6.6.

CD

```
classdiagram {
  class A { public B attrAB; }
  class B { public C attrBC; }
  class C { public Boolean boolOfC; }
}
```

Listing 6.8: The UML/P CD used as basis for the illustration of the transformations from entity to PDDL models.

Entity

```
action foo(A myA, B myB) {
  pre: myA.attrAB.attrBC == myB.attrBC; // ...
}
```

Listing 6.9: A precondition that consists of the comparison of two attributes.

PDDL

```
(:action foo
  :parameters (?myA - A ?myB - B )
  :precondition (and (exists (?X_0 - B ?X_1 - C )
                             (and (A_attrAB ?myA ?X_0)
                             (B_attrBC ?myB ?X_1)
                             (B_attrBC ?X_0 ?X_1)))) ; ...
)
```

Listing 6.10: The precondition results from transforming the precondition of the entity model depicted in Listing 6.9.

Entity

```
1 action foo(A a1, C c, A a2, B b) {
2   pre: m(a1.attrAB, c, a2.attrAB.attrBC) != b.attrBC.boolOfC;
3   /* ... */
4 }
```

Listing 6.11: The precondition checks whether the value assigned to an attribute is not equal to a value assigned to a property.

to the types of the parameters of the property. The last argument of the predicate has the PDDL type corresponding to the return value type of the property. Each PDDL predicates has the same name as its corresponding property. For instance, the properties depicted in Listing 6.6 are transformed to the predicates given in Listing 6.7.

6.2.3 Transformation of Entity Actions to PDDL Actions

Each action of each entity model is transformed to a PDDL action. The parameters and preconditions of entity actions correspond to parameters and preconditions of PDDL actions, whereas postcondition of entity actions correspond to PDDL effects. After transforming an entity's action, the resulting PDDL action has the same name as the entity's action. Further, the parameters of the resulting PDDL action have the same names and types as the corresponding parameters of the entity's action. The following exemplarily illustrates the transformations of preconditions and postconditions by examples. The types used in the following examples can be found in the CD model presented in Listing 6.8. It defines three types, each having an attribute of another type.

Transformation of Action Preconditions to PDDL Preconditions

A precondition of an entity's action is a Boolean expression that consists of logical conjunctions, logical disjunctions, logical negations, and expressions referencing properties as well as attributes of the enclosing action's parameters. During the precondition transformation, logical negations, conjunctions, and disjunctions are preserved. Expressions requesting property values and referencing attributes of parameters become suitable logical expressions in PDDL. In preconditions, values of attributes and properties may be queried and compared to parameters of the enclosing action, attributes of objects, or values requested from properties.

Listing 6.9 and Listing 6.10 illustrate the transformation of expressions comparing the values of two attributes to each other. The intended meaning of the resulting PDDL precondition (Listing 6.10) is the following: there must be x_0 and x_1 (l. 3) such that $x_0 = myA.attrAB$ (l. 4), $x_1 = myB.attrBC$ (l. 5), and $x_0.attrBC = x_1$ (l. 6).

Listing 6.11 and Listing 6.12 exemplarily illustrates the transformation of an expression checking whether the value of an attribute does not equal the value of a property. The resulting PDDL precondition (Listing 6.12) requires there must not be any x_0, x_2, and

PDDL

```
(:action entities_MyRobotEntity_foo
  :parameters (?a1 - A ?c - C ?a2 - A ?b - B )
  :precondition
    (not
      (exists (?X_0 - B ?X_1 - B ?X_2 - C
               ?X_3 - C ?X_4 - Boolean)
        (and (A_attrAB ?a1 ?X_0)
             (A_attrAB ?a2 ?X_1)
             (B_attrBC ?X_1 ?X_2)
             (B_attrBC ?b ?X_3)
             (C_boolOfC ?X_3 ?X_4)
             (m ?X_0 ?c ?X_2 ?X_4 )))) ; ...
)
```

Listing 6.12: The precondition results from transforming the precondition given in the entity model that is illustrated in Listing 6.11.

Entity

```
action foo(A myA1, A myA2) {
  pre:  // ...
  post: myA1.attrAB.attrBC == myA2.attrAB.attrBC;
}
```

Listing 6.13: The postcondition assigns the value of an attribute to another attribute.

x_4 (l. 5) such that $x_0 = a.attrAB$ (l. 6), $x_2 = sd.attrDE.attrEF$ (ll. 7-8), $x_4 = g.attrGH.boolOfH$ (ll. 9-10), and $m(x_0, c, x_2) = x_4$ (l. 11). Expressions in preconditions that compare the values of two properties are similarly transformed to PDDL.

Transformation of Action Postconditions to PDDL Effects

A postcondition of an entity action is a Boolean expression that consists of logical conjunctions and infix expressions over the equality (==) operator. Infix expressions in postconditions must reference properties, parameters, or their attributes. A postcondition can be interpreted as a sequence of assignments, where each either assigns a new value to a property or to an attribute. The transformation preserves logical conjunctions and transforms infix expressions to PDDL expressions for removing or adding facts. The following exemplarily illustrates the transformation of postconditions to effects.

The postcondition depicted in Listing 6.13, assigns the value of an attribute to another attribute. The PDDL effect resulting from the transformation is illustrated in Listing 6.14. The intended meaning of the PDDL postcondition is the following: for all values x_0, x_1, and x_3 (l. 5) such that $x_0 = myA1.attrAB$ (l. 6), $x_1 = myA1.attrAB.attrBC$ (l. 7), $x_3 = myA2.attrAb.attrBC$ (ll. 8-9), and $x_1 \neq x_3$ (l. 10), then delete the fact $x_1 = myA1.attrAB.attrBC$ (l. 12) and add the fact $x_3 = myA1.attrAB.attrBC$ (l. 13).

PDDL

```
1 (:action foo
2   :parameters (?myA1 - A ?myA2 - A )
3   :precondition ; ...
4   :effect
5     (and (forall (?X_0 - B ?X_1 - C ?X_2 - B ?X_3 - C )
6                  (when (and (A_attrAB ?myA1 ?X_0)
7                             (B_attrBC ?X_0 ?X_1)
8                             (A_attrAB ?myA2 ?X_2)
9                             (B_attrBC ?X_2 ?X_3)
10                            (not (= ?X_1  ?X_3))
11                       )
12                       (and (not (B_attrBC ?X_0 ?X_1))
13                            (B_attrBC ?X_0 ?X_3)))))) ; ...
14 )
```

Listing 6.14: Effect resulting from transforming the postcondition in Listing 6.13.

Goal

```
1 world  entities.RoomsWorld;
2 // ...
3 goal Deliver(Item item, Room room) {
4   predicate { RoomsWorld.itemLocation(item, room) }
5 }
```

Listing 6.15: The goal `Deliver` contains a predicate querying the value of a property.

If $myA1.attrAB.attrBC = myA2.attrAB.attrBC$ holds before action execution, the precondition does not have to be executed, as it would not change the planning state. Therefore, on the one hand, the requirement $x_1 \neq x_3$ (l. 10) ensures the effect does not change the planning state in case equality already holds before action execution. On the other hand, the requirement ensures that $myA1.attrAB.attrBC = myA2.attrAB.attrBC$ is not unintentionally deleted (l. 12) in case the values of the attributes are already equal to each other before executing the action. Expressions used in postconditions that assign the value of a property to another property or assign the value of a property to an attribute, and vice versa, are transformed similarly.

6.2.4 Transformation of Goals to PDDL Problems

Each goal is transformed to a PDDL problem. Unlike PDDL problems, goals can be parametrized. Therefore, a goal only partially defines a PDDL problem. Additionally, goals neither contain information about existing objects, nor about currently valid properties. In contrast, each PDDL problem defines a set of existing objects and an initial state consisting of facts relating objects (i.e., the corresponding domain's predicates).

PDDL

```
1 (define (problem Deliver)
2   (:objects
3     classtypesItem1412322831 - Item
4     classtypesItem1495608502 - Item
5     classtypesRoom412788346 - Room)
6   (:init
7     (entities_RoomsWorld_itemLocation
8       classtypesItem1412322831
9       classtypesRoom412788346 true)
10    (entities_RoomsWorld_itemLocation
11      classtypesItem1495608502
12      classtypesRoom412788346 true)
13    (entities_TransportRobot_robotLocation
14      classtypesRoom412788346 true))
15  (:goal
16    (exists
17      (?item - Item ?room - Room)
18      (and
19        (= ?item classtypesItem1495608502)
20        (= ?room classtypesRoom412788346)
21        (and
22          (entities_RoomsWorld_itemLocation ?item ?room true)
23         ))))
24 )
```

Listing 6.16: This PDDL problem was derived from the goal model (cf. Listing 6.15).

For these reasons, PDDL problem models are not generated directly after parsing, but during runtime when the current world and robot states are known, i.e., it is known which objects exist and which properties are valid at a certain point in time. Given a list of `Property` class (Section 5.3) instances, which encode facts holding at a certain point in time, and a `Goal` class (Section 5.3) instance, which assigns values to parameters of the corresponding goal, at runtime, a well-formed, complete PDDL problem can be generated.

The remainder of this section describes the transformation of goals to PDDL problems. The PDDL problem depicted in Listing 6.16, for instance, was derived at runtime from the goal model given in Listing 6.15, a list of `Property` class (Section 5.3) instances, and a `Goal` class (Section 5.3) instance. Given a list of `Property` class instance, a goal model, and a corresponding `Goal` instance, a PDDL problem is generated as follows:

1. Each attribute of each `Property` instance becomes an object in the PDDL problem (ll. 2-6). The identifier of each object is determined as follows: concatenate the parts of the full qualified class name of the object to the String “class” and suffix the objects hash-code to the result of the concatenation.

2. Each `Property` instance becomes an initial fact in the PDDL problem (ll. 7-19). The name of the fact is the full qualified name of the class of the `Property` instance where underscores (_) are used as package delimiters. Each attribute of the `Property` instance becomes a parameter of the fact. The identifier of the parameter is determined as in (1.). Since these facts are valid in the beginning, the last parameters of each fact is the object `true` of type `Boolean`, which is defined in the PDDL domain.

3. The goal formula of the PDDL problem (ll. 20-28) is generated from the predicate of the goal model, and the attributes of the `Goal` instance. The attributes of the `Goal` instance instantiate the parameters of the goal model. Therefore, the goal of the PDDL problem results from applying the following two transformation steps: First, transform the predicate given in the goal model the same way as preconditions of actions are transformed to PDDL preconditions (l. 26). Afterwards, enclose the result of the former transformation by a term that binds the goal's parameters by existential quantification (ll. 21-22) and require equality to the values defined by the attributes of the `Goal` instance (ll. 23-25).

6.3 Discussion

Section 6.1 described the intra-language pre-transformations that are applied to entity models to achieve well-formedness of generated PDDL artifacts. Based on their results, Section 6.2 presented the transformations from goal, entity, and domain models to PDDL. This section summarizes the functionality provided by the infrastructure, discusses the decoupling from application specific parts of the architecture (Section 5.5), points out its limitations, and suggests possible enhancements.

Functionality Overview

Altogether, the infrastructure is capable of calculating plans for solving problems defined by a static environmental description, a description of the currently available dynamic environmental conditions, and a desired environmental situation. Technically, the resulting plans are tuples of the output from external planning software for PDDL problems and a mapping from identifiers used by the planner to instances of the corresponding objects used by the architecture. The static environmental description is represented by a robot and a world entity model with actions and properties (Section 4.3), a UML/P CD modeling the domain utilized by the entities (Section 4.2), and a set of goal models defined for the entities (Section 4.5). The dynamic environmental conditions are represented by a list of property instances (cf. Section 5.3.1) that conform to the properties defined by the entities. The desired environmental situation is a list of goal instances (cf. Section 5.3.1) that conform to the goal models and are to be satisfied.

Decoupling from Application Specific Classes

One important design goal was to completely decouple the infrastructure from the generators for the languages presented in Chapter 4, i.e., the implementation of the transformation infrastructure is completely independent from the artifacts generated from entities, UML/P CD, and goals. On the one hand, this independence obviously ensures that changes in the generators do not affect the transformation infrastructure implementation. Further, the decoupling enables to reuse the transformation infrastructure in other contexts, e.g., in implementations produced by other generators. On the other hand, the strict separation between this infrastructure and the language generators has a disadvantage: The infrastructure has no knowledge about any generated code. Thus, it does not provide functionality for transforming plans calculated for PDDL problems to the appropriate data structure used by the reference architecture (Chapter 5), i.e., plans which are lists of action instances (cf. Section 5.3.1). Therefore, these transformations have to be realized by the architecture, i.e., the `Planner` component type implementation (Section 5.2.5).

Limitations and Possible Enhancements

The planning infrastructure transforms entity, UML/P CD, and goal models to PDDL models compatible to version 1.2. PDDL 3.1 [HDKN12] introduces object-fluents to the language, i.e., functions ranging over any object type. A possible enhancement is to integrate these into the transformation infrastructure. On the one hand, the integration possibly eases the necessary transformations to PDDL since property definitions could be transformed to object fluents, which have a similar signature, and thus would not necessarily have to be transformed to predicates anymore. On the other hand, the majority of existing planners do not support PDDL version 3.1 or are not as efficient as well established planners supporting previous versions of PDDL[1].

[1] International Planning Competition 2014 website: `https://helios.hud.ac.uk/scommv/IPC-14/planners_actual.html`

Chapter 7

Human-Robot Interaction

The reference architecture (Section 5.2) employs two UIs to control the robot. One is stationary and resides on a desktop PC, while the other uses a tablet mounted to the robot (cf. Figure 7.1). Section 7.1 presents the desktop UI, which provides an administrative interface (e.g., to manipulate the system's knowledge base). Afterwards, Section 7.2 describes the tablet UI, which enables interaction between humans and the robot (e.g., during task execution).

Figure 7.1: Illustration of a human-robot interaction within the iserveU project. The medical staff assigned tasks to a nearby robot using the tablet UI.

7.1 Desktop User Interface

The desktop UI (Figure 7.2) is responsible for administrating the knowledge base (Section 5.5.6), task initialization, and observing the logs (Section 5.3.3). Therefore, it has access to the overall architecture (e.g., the `KnowledgeBase`, `RobotLog`, and the robot and world implementations). For instance, it bears the possibility to define, store, or load knowledge bases during runtime. Knowledge bases reflect the current state of the

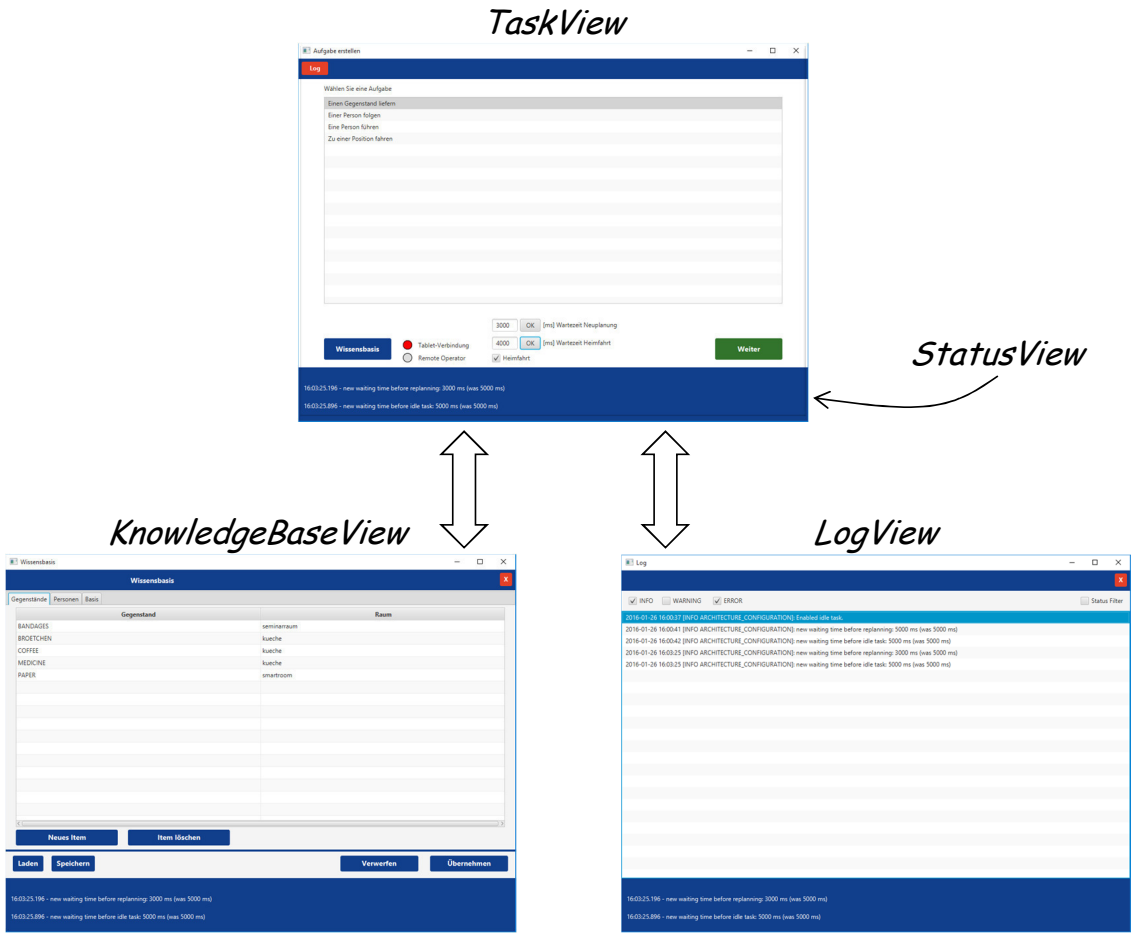


Figure 7.2: The desktop UI enables administrative users to instantiate tasks, change the knowledge base, or view the logged messages. This functionality is comprised in four views: `InstanceTaskView`, `KnowledgebaseFillView`, `LogView`, and `StatusView`.

environment with information about its available items, locations, and persons. The UI is capable of tracking changes in the knowledge base and the state of the robot. Thereby, users can observe changes at runtime and may adapt the knowledge base or some specific execution parameters such as the time for replanning (Section 5.2.4) or whether the robot should return home when idling. Nevertheless, its fundamental functionality is to initialize and forward tasks to the robot. For instance, a user initializes the deliver task by choosing an item and a destination from the knowledge base. This feature is available in both UIs and synchronized between them. Concurrent instantiation of a task is prohibited.

The desktop UI is based on JavaFX[1] and consists of different views (e.g., for the task management, robot and task state, knowledge base, and event logging). Each of these views is managed by the `InstanceTaskGUI`, which handles the UI, the corresponding controllers, and the data used by the desktop UI. Additionally, the component has an observer that tracks messages of the `RobotLog` (Section 5.3.3). Some messages trigger changes in

[1] JavaFX: `http://docs.oracle.com/javafx/`

the desktop UIs settings and others are informational. The `InstanceTaskGUILogger` stores and provides relevant log messages for the corresponding views.

The Component `InstanceTaskGUI`

The component `InstanceTaskGUI` initializes and manages all relevant views of the desktop UI. Therefore, it provides methods to switch between the different views, knows available tasks and their possible parameters based on the knowledge base. By observing the `RobotLog` it reacts to specific events and might change 1. the visibility of specific buttons and 2. the states of various connection and activity indicators.

The Component `InstanceTaskView`

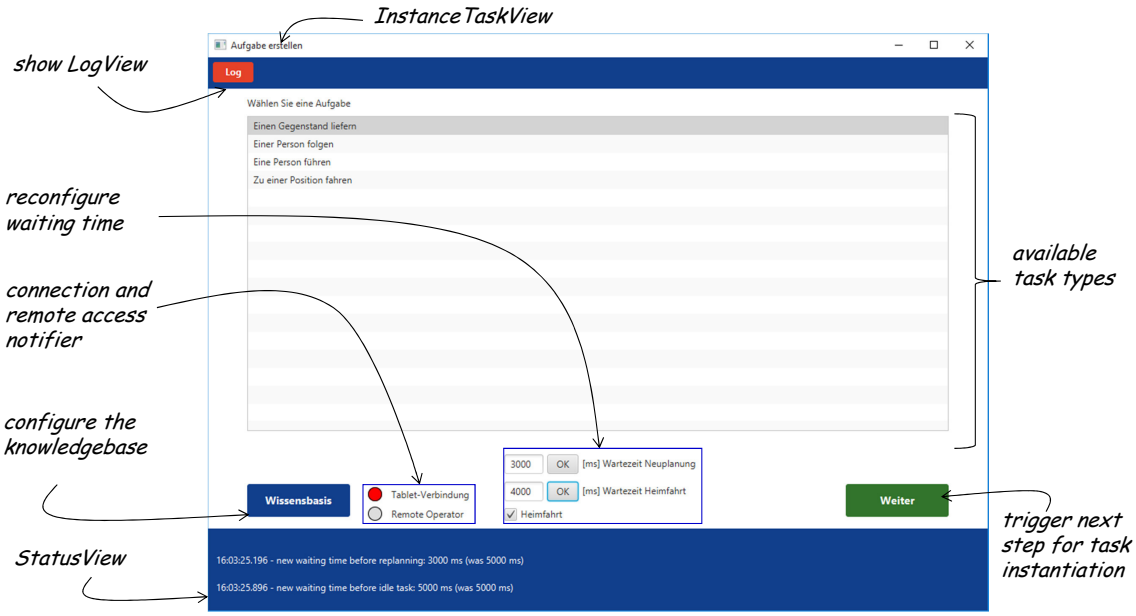


Figure 7.3: The `InstanceTaskView` is the main view for task instantiation and provides the user with various settings.

Figure 7.3 presents the view for submitting tasks to the robot. Initially, it displays the list of available tasks. Depending on the user's selection it asks her to provide the required task parameters by displaying corresponding input forms (e.g., to select the item and room for the delivery task). If all parameters are set, the view controller calls the task factory to create a task. Thereon, the view controller calls the `Executer` (Section 5.5.7) component to execute it.

The view does not allow to initialize a task if the tablet UI is currently in use. Furthermore, it configures the waiting time until replanning proceeds, the waiting time until the robot is considered idle, and whether the robot should return to its home station, if idle. Besides the task initialization, the view enables to trigger the `LogView` with the "Log" button in the top-left and the `KnowledgeBaseView` with the "Wissensbasis" button

in the bottom-left border. Furthermore, the actual connection state to the tablet UI, the last two messages of the robot log, and whether the the robot is controlled by the remote operator are shown.

The Component `KnowledgebaseFillView`

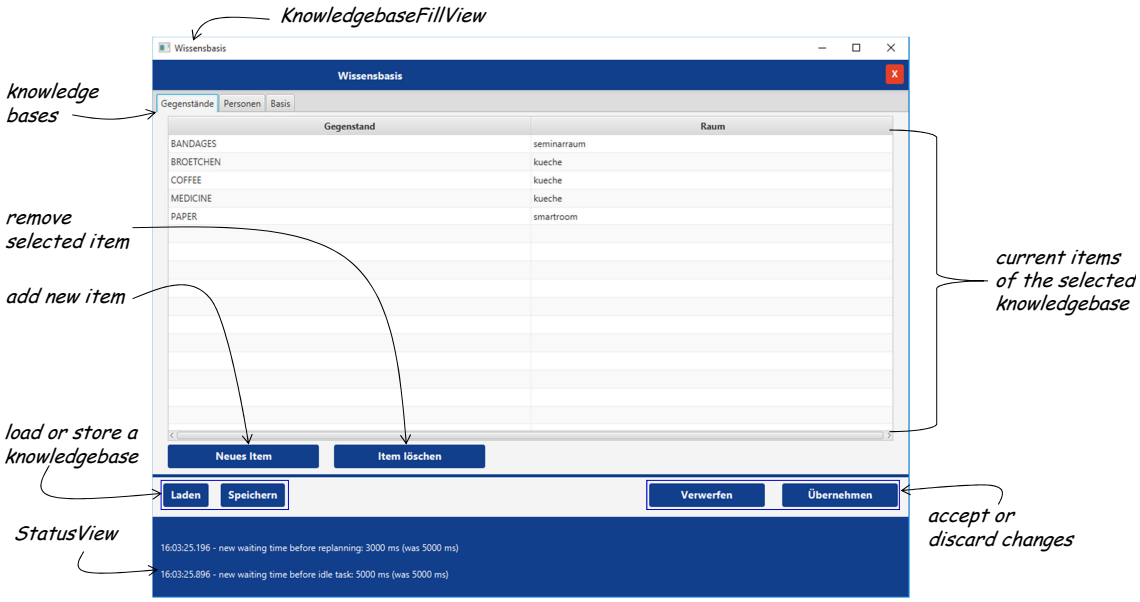


Figure 7.4: The `KnowledgebaseFillView` view provides various options for knowledge base manipulation.

Figure 7.4 shows the `KnowledgebaseFillView` that enables the manipulation of the knowledge base. Each knowledge base is depicted in the top-left corner in form of a tab. Two explicit buttons “Neues Item” and “Item löschen” enable to add a new item to the selected knowledge base or to remove a selected item from the knowledge base. Changes do not take effect immediately, but users may discard them, explicitly apply them, or export them to a file. Buttons to load and store knowledge bases are provided in the bottom-left border while the buttons for the abortion and confirmation of changes are displayed at the bottom-right border. Changes are always tracked to warn about unsaved changes when trying to exit the view.

The Components `InstanceTaskGUILogger`, `LogView`, and `StatusView`

The desktop UI observes the `RobotLog` to track relevant messages. The `StatusView` displays the recent two log messages. The view is displayed simultaneously to the other views, e.g., `InstanceTaskView`. Additionally, a `LogView` (Figure 7.5) displays administration-relevant log messages stored by component `InstanceTaskGUILogger`. This view enables users to filter them by severities (e.g., error, warning, and info). Memory usage is thereby reduced by storing only the last 100 messages.

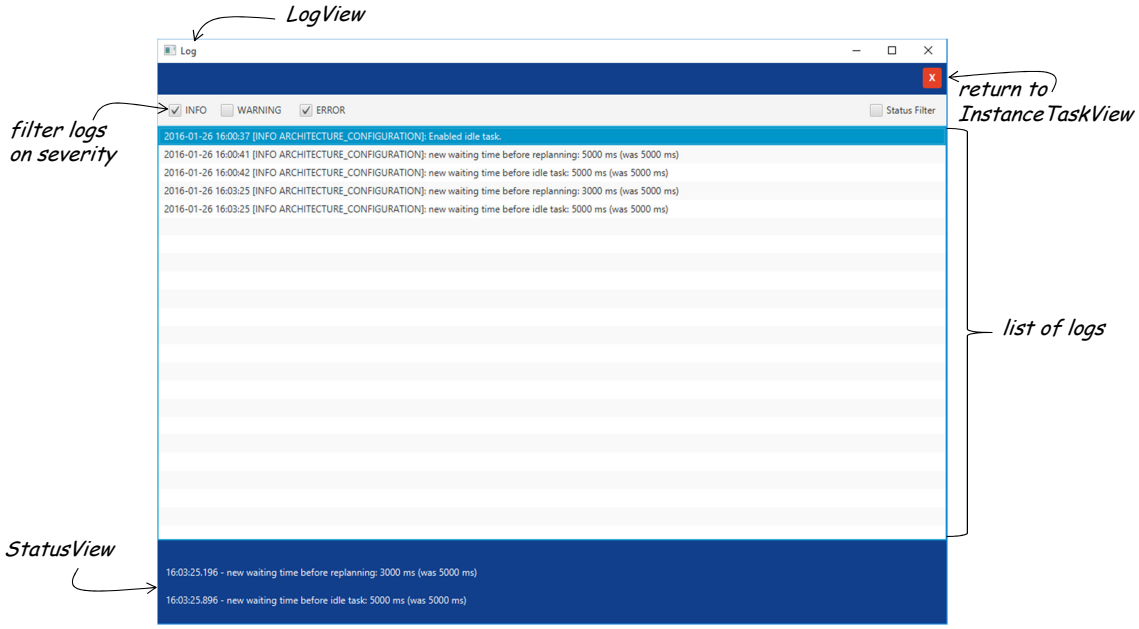


Figure 7.5: The `LogView` displays the last 100 log messages and enables to filter them by different severities.

7.2 Tablet User Interface

The tablet UI (see Figure 7.6) is an Android[2] application deployed on a tablet mounted to the robot. It is responsible for direct interactions with users and enables to assign tasks to the robot while it is nearby. For instance, during the deliver task the robot requests the user to load an item to the robot. The user is able to confirm or cancel this request locally. Therefore, the tablet UI is optimized for a simple and fast access. The design is separated into top, center, and bottom areas (e.g., the `MainActivity` in Figure 7.6). At the top of each activity is an actionbar that is divided into three parts: a button on the left forwards to the `LogActivity` in each activity, a notification bar in the center, and a button for further settings. A list for available tasks is presented at the center of the activity. An item of this list has to be chosen until a task can be further processed. To get to the next activity or back to the previous one, the activities provide corresponding buttons at the bottom.

7.2.1 Activities in Different Scenarios

For an improved understanding of how the various Android activities are designed and applied, it is beneficial to examine the activities based on common application scenarios. Three different scenarios have to be mentioned: the start of the application, the creation of a task, and the notification about the current state of a task.

[2] Android: `https://www.android.com/`

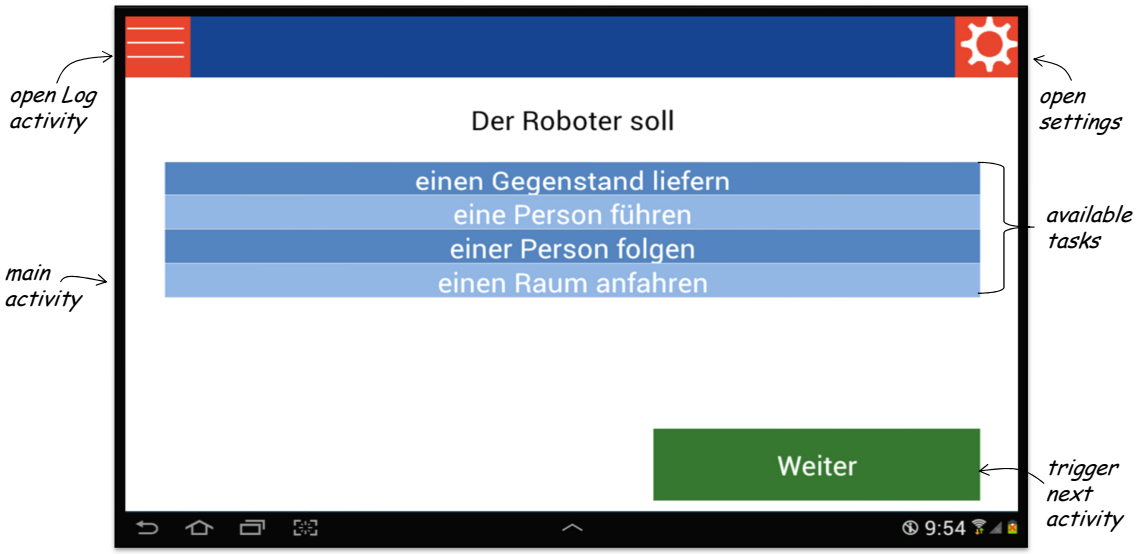


Figure 7.6: The `MainActivity` is the core of the tablet UI and allows user to select a task, to open the `LogActivity`, or to switch to the `SettingsActivity`.

Starting the Application

Starting the tablet UI triggers an `InitialActivity`. It serves as a waiting screen that displays the application name on screen while it protects the user from an inaccurate utilization before the `Text-To-Speech` service, which is used for acoustic task notifications, and the `ConnectionHandlerService` service have been initialized. Afterwards, the activity forwards to the `MainActivity` (Figure 7.6).

Task Creation

Figure 7.7 depicts an excerpt of a possible activity call sequence to create a task. The `MainActivity` serves as initial starting point for interacting with the robot. In the `MainActivity`, the user is able to select between four possible tasks: deliver an item, guide a person, follow a person, and move to a room. Pressing the “Weiter” button invokes the `LoadingActivity`.

The `LoadingActivity` serves as a mediator between the activities for human interaction. Therefore, it contains a reference to the previous activity that triggered the call and requests the specific data list with respect to the previous activity. A request between the `LoadingActivity` and the `connection` service is handled via inter-process communication. During this request operation the user is notified that the UI is requesting item from the server. If the `LoadingActivity` receives an item list, it triggers the next activity call.

The `SupplyActivity` (Figure 7.8) is triggered by the `LoadingActivity` if the user selected a delivery task in the main activity. In this activity, the list comprises the available items that might be delivered to a location. Therefore, the user is able to select a specific item in this interaction activity. Additionally, the bottom section of the design is enhanced by a “Zurück” button that forwards to the previous activity. If the user presses the

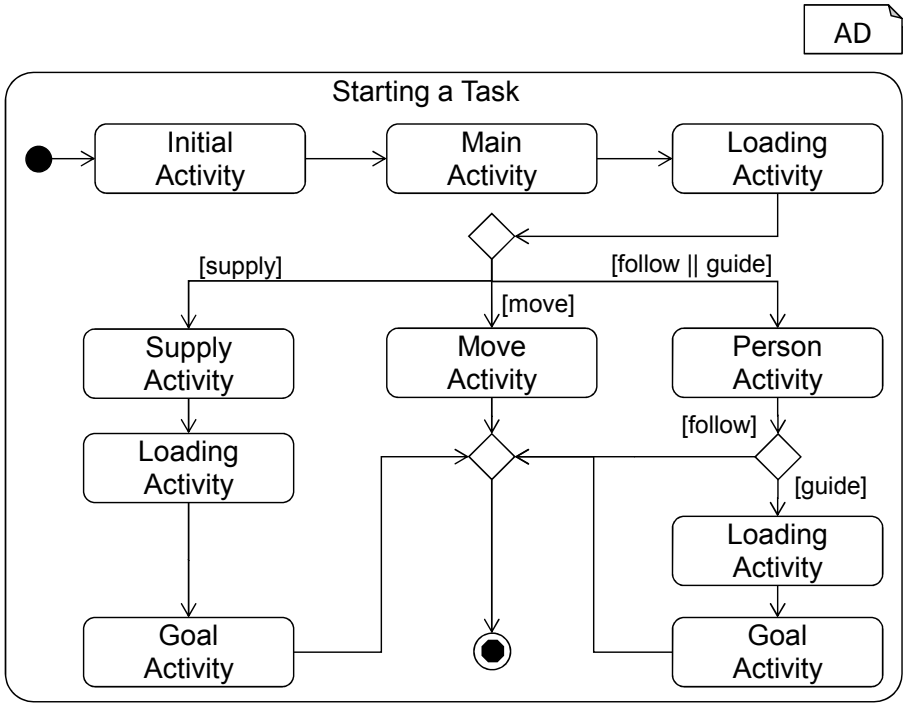


Figure 7.7: This activity diagram displays the sequences of activity calls, which are necessary to start a task on the tablet. Possible tasks in this scenario are move, supply, follow, and guide tasks.

"Weiter" button, the activity triggers the `LoadingActivity` with a request for rooms. Finally, the `LoadingActivity` triggers the `GoalActivity`.

The `GoalActivity` might be either triggered for a deliver or a guide task. Previously selected information for those respective tasks are enhanced by the destination location. Since the activity receives the accumulated information for its current task, it is able to decide whether the information are for a deliver or for a guide task. As a result, the information are forwarded to the `connection` service to create the task. Section 7.2.2 presents how the communication of the connection service is handled.

If a user selects a movement task in the `MainActivity`, the `LoadingActivity` requests a list of all reachable rooms from the knowledge base and triggers the `MoveActivity`. This activity differs from the previously mentioned ones by its forwarding button in the right corner at the bottom. This button ensures that a room is chosen and the necessary information for a movement task is transmitted.

The `PersonActivity` is used for the follow and guide tasks. Both require the user to select one person from a list. Further processing and forwarding in this activity depends on the selected task in the `MainActivity`. When the required information for a follow task are complete, the "Weiter" button forwards the task information directly to the `connection` service. Otherwise, the gathered information is forwarded to the `LoadingActivity` that triggers a further `GoalActivity` call.

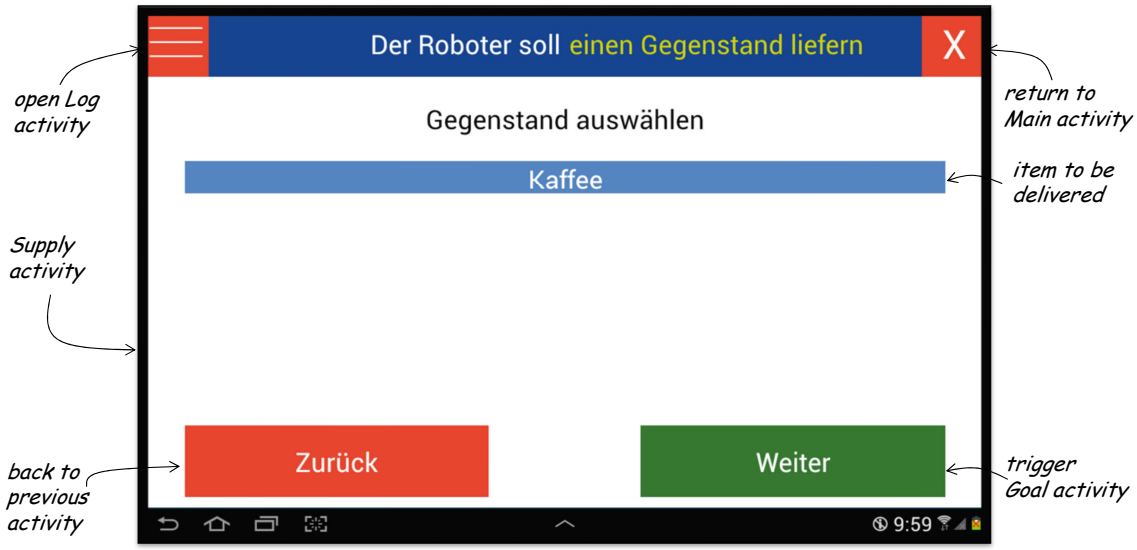


Figure 7.8: The `SupplyActivity` is used to select an item that has to be delivered to a specific destination, which has to be chosen in a further activity. Therefore, the activity provides the possibilities to trigger the `LogActivity` activity, return to `MainActivity` or previous activity, and to trigger the `GoalActivity` for further supply task processing.

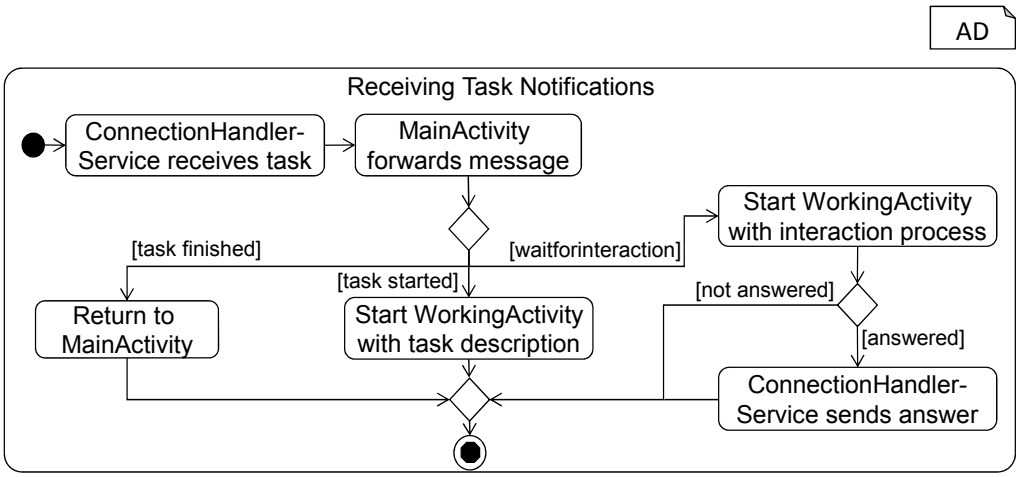


Figure 7.9: The activity diagram represents the activities for receiving task notifications when a task is executed. The `ConnectionService` service receives messages with information about the current state. These information are forwarded to the `MainActivity` via background communication (Section 7.2.2). The `MainActivity` immediately triggers the `WorkingActivity` with respect to the content.

Task Notification

The last scenario is the notification for the state of a running task. This provides the users information about the current task and signalizes that an interaction with the robot is required. Since the robotics platform does not contain a dedicated display, the interaction with the user is handled by the tablet UI. The entire inter-process communication between

activities is processed in the background and does not affect most of the activities. This is guaranteed by a lock and release agreement between the desktop and tablet UIs. Figure 7.9 represents a possible activity diagram for this scenario.

The `ConnectionHandlerService` receives messages from the desktop UI and forwards them to a message handler of the `MainActivity`. Based on the received content, the handler decides which activity has to be started. The set of possible messages that have to be handled consists of `task started`, `task finished`, and `wait for interaction` messages. A `task finished` message always triggers a `MainActivity` call while `task creation` and `wait for interaction` trigger a `WorkingActivity` call. These calls are performed regardless of the current state of the table UI.

Figure 7.10: The `WorkingActivity` is responsible for human-robot interaction during task execution. The depicted state of the activity provides information about a deliver task, which are the robot's destination and the item it delivers. The user is able to abort task execution.

The `WorkingActivity` (Figure 7.10) bears two functionalities: it provides information about the currently executed task and handles interactions between user and robot during task execution. For the first functionality, the activity displays the current task on screen as long as no human interaction is required. The `WorkingActivity` instances do not change their state themselves. Instead, another `WorkingActivity` is stacked onto the previous activity whereas a `finished task` message empties the stack. Thus, there is a stack of `WorkingActivity` calls for a deliver task. Starting with a `task started` message, the activity displays the information of the tasks until it receives a `wait for interaction` message. These messages trigger a new `WorkingActivity` call, which notifies the user via `Text-To-Speech` and a text message on the screen to interact with the tablet. An interaction is handled with a confirmation whether the user was able to fulfill the task. Note, if the users do not interact with the tablet UI in a given time, the current task is automatically aborted. Thereon, the confirmation is forwarded to the desktop UI by the `connection` service. The tablet UI switches back to the previous

	Command	Parameter	Description
1	req;item		Requests all items
2	req;room		Requests all rooms
3	req;person		Requests all persons
4	req;log		Requests last 50 logs
5	inst;deliver	?item;?room	Instantiates task to deliver the item to the room
6	inst;follow	?person	Instantiates a task to follow a person
7	inst;lead	?person;?room	Instantiates a task to leads a person to a room
8	inst;move	?room	Instantiates a task to go to the specified room
9	ui	?response	Delegates ui response (true or false)
10	taskabort		Stops the current task
11	tabletInUse		Blocks actions on desktop UI
12	tabletIsIdle		Release actions on desktop UI

Table 7.1: Communication protocol between the `ConnectionHandlerService` on the tablet and the `ConnectionHandler` class on the server side.

`WorkingActivity` call. This might be done several times until a finish call empties the stack and forwards to the `MainActivity`.

7.2.2 Tablet Communication

Due to the fact that the tablet is used in a dynamic environment, the tablet UI has to adapt to environmental changes at runtime. These changes might be invoked by new available network hotspots as well as changes in the respective knowledge bases. Thus, the communication between tablet and its environment has to be dynamic and reliable. A suitable model for this kind of network is the client-server model, which is established via a reliable TCP network protocol. Therefore, non-blocking socket channels are used for the tablet-desktop communication. While the desktop UI and its responsible `ConnectionHandler` are assigned the server role, the `ConnectionHandlerService` remains as thin client that receives the live content of the environment from the server. Therefore, the `ConnectionHandler` observes the `RobotLog`, filters relevant data, and stores them in a queue. The `ConnectionHandler` also processes the data accordingly to its appearance. Nevertheless, each data entry might trigger various communication operations. The following communication protocols ensure a proper way of communication between server and client.

Protocol for Client to Server Communication

Table 7.1 depicts the applied `ConnectionHandlerService` to `ConnectionHandler` protocol. These commands are concatenations of fixed command Strings and their corresponding parameters, which are denoted by a "?" sign. For instance, rows 1 to 4 consist of four kinds of data requests that might be sent during an activity swap. In this case, the used commands are fixed and require no further input. Task instantiation is possible with previously requested data of the environment. Besides the fixed commands,

the respective commands in row 5 to 8 expect further input necessary to create a task on the server side. If the `ConnectionHandler` receives these commands, it splits the task information in their essential parts, creates a task with the specific task factory, and forwards the task to the task `executer`. Additionally, the tablet is used for specific tasks that require human-robot interaction such as the delivery task. The interaction during task execution is restricted to `Boolean` responses, the used command in row 9 is enhanced by a Boolean value represented by a `String`. The `ConnectionHandler` processes this value to a specific log message and adds it to the `RobotLog`. Moreover, the tablet user is allowed to abort the current task during execution with the `taskabort` command, which triggers the `Executer` (Section 5.5.7) on the server side. To ensure that concurrent actions of desktop and tablet user do not interfere with each other, the tablet is able to lock and unlock the desktop UI with the commands in lines 11 and 12. For instance, the desktop UI is blocked if a user interacts with the tablet UI. This ensures that no task is instantiated and executed from the desktop UI until the interaction has been finished.

Protocol for Server to Client Communication

The server side of this communication model is responsible for providing live data as well as processing the received data from the tablet. Therefore, the `ConnectionHandler` uses specific commands for communication.

	Command	Parameter	Description
1	items	?entry1;?entry2;...	Response with a list of items
2	persons	?entry1;?entry2;...	Response with a list of persons
3	rooms	?entry1;?entry2;...	Response with a list of rooms
4	logs	?entry1;?entry2;...	Response with a list of the last 50 logs
5	uiinteraction	?textMessage	Sends an UI request to the tablet with an specific message
6	task	?taskDescription	Sends the active task description to the tablet
7	ro;call		Notifies that the remote operator starts maintenance
8	ro;success		Informs that remote operation succeeded
9	ro;fail		Informs that remote operation failed
10	ok		Used to check for connection issues

Table 7.2: Protocol for the communication between the `ConnectionHandler` class and the `ConnectionHandlerService` service. This table depicts the protocol for direction server to client.

These commands are represented in Table 7.2. The first four rows depict the commands and parameters that correspond to the first four request commands of Table 7.1. These commands are concatenated with all respective data entries, each delimited by a semicolon. The next two rows represent commands that are used for displaying task execution on the tablet as well as requesting user input. Both commands are extended with a respective message for the user interaction or task description and passed immediately to the tablet. This ensures a proper and live notification on the tablet for new tasks and user interactions. Moreover, row 7 to 9 are used for maintenance information of the remote operator. These

commands trigger an immediate change to the maintenance activity of the tablet UI. Further, the `ok` command is used for detection of connection issues. The command is sent in constant, short time intervals to ensure a timed connection issue detection.

Chapter 8

Deployment

The previous chapters introduced modeling languages (Chapter 4), described the reference architecture (Chapter 5), and its UIs (Chapter 7). Based on these, Section 8.1 presents a use case scenario and demonstrates the integration of domain-specific handcrafted code. The remaining sections show how to deploy and launch the reference architecture using Robotino robots. The robots are controlled via the SmartSoft[1] middleware. Therefore, Section 8.2 explains launching a SmartSoft deployment. Afterwards, Section 8.3 shows how to deploy the reference architecture and how to start the UIs.

Figure 8.1: Illustration of a Robotino3 platform that is moving and interacting within a dynamic hospital environment during the iserveU evaluation.

8.1 Deployment Scenario

This section presents a use case scenario employing the developed DSLs supplemented with handcrafted Java code. The scenario includes an autonomously driving robot that is capable of guiding and following persons carrying a beacon as well as transporting items (e.g., bandages and medicine) between rooms in a hospital environment (cf. Figure 8.1). The corresponding robot entity, domain, task, and goal models abstract from application domain-specific (here the hospital environment) world entity information. Hence, they are reusable in combination with world entities requiring guide and follow as well as transportation capabilities.

Figure 8.2 overviews the necessary steps and tool infrastructure for using the system. Domain experts develop task, goal, domain, application, and entity models. Then, the

[1] SmartSoft Website: `http://www.servicerobotik-ulm.de/drupal/?q=node/19`

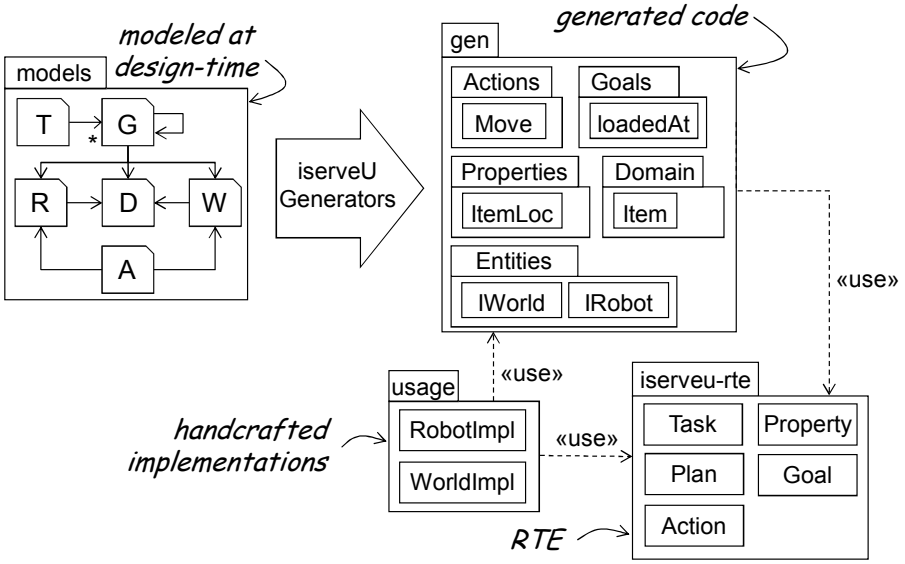


Figure 8.2: Overview of the development process: Task, goal, entity, application models, and the dependencies between each other are modeled at design time. The generators then produce parts of an executable system from these models as further described in Section 5.4. The generated classes are assisted in providing their functionality by the RTE infrastructure (Section 5.3.1). Since the generated code is not complete, it has to be complemented with handcrafted robot and world implementations.

corresponding generators produce parts of an executable system as stated in Section 5.4. They generate, inter alia, classes representing actions of entities, goals, properties, and domains. Additionally, as further explained in Section 5.3.2, for each entity model, the generator produces an interface (Figure 8.2, `IWorld` and `IRobot`) consisting of a method for each action and for each property of the entity. The interface generated for a world entity model additionally contains a method for each class of the domain imported by the entity. Intuitively, invoking such a method generated for an action should result in physically executing the action. The invocation of such a method generated for a property should yield information about the current world and robot states regarding the property and the currently existing objects (which are instances of the classes described in the domain model). The methods generated for the classes modeled by the domain should return subsets of objects that are relevant for planning. Technically, these objects are of the type `Query`, which is described in details in Section 5.3.1.

Since the generated code is missing robot and world specific domain knowledge, the generated code is not complete. It has to be complemented with handcrafted world and robot implementations (Figure 8.2, `RobotImpl` and `WorldImpl`) that must implement the interfaces generated for the entity models (`IWorld` and `IRobot`). The following sections present specific domain, entity, task, application, and goal models and illustrate the integration of handcrafted code.

8.1.1 Logistics Domain

Figure 8.3 depicts a CD modeling the domain of the scenario presented in this chapter. For the representation of items, rooms, and persons, the CD consists of the three classes `Item`, `Room`, and `Person`. Each has an attribute `name` of type `String`, while the class `Person` additionally has an attribute `beaconID` of type `String`. Items, rooms and persons can be identified by their names. In the scenario presented in this chapter, persons must carry electronic beacons to be localized by the robot. Each beacon has a unique identifier. The `beaconID` attribute of a `Person` encodes which beacon, if any, the person carries.

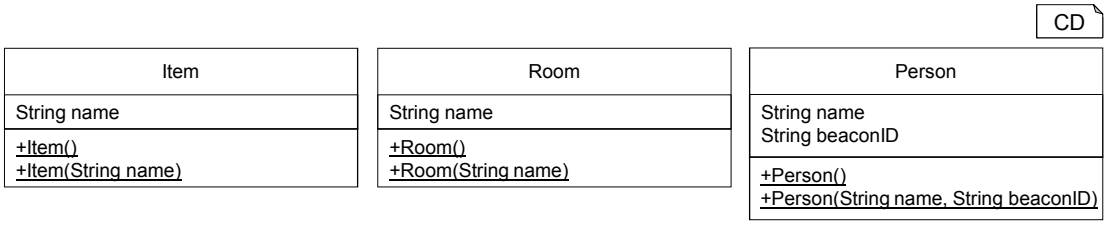


Figure 8.3: CD modeling the domain of the scenario presented in this chapter.

8.1.2 Rooms World Entity

Listing 8.4 depicts the `RoomsWorld` entity model. The world entity states the domain `LogisticsDomain` (l. 3) and therefore imports the data types defined by the domain model. Further, the entity consists of five properties and does not contain any action. The property `itemLocation(Item item, Room room)` (l. 6) indicates whether the current position of `item` is `room`. Querying the property `personLocation(Person person)` (l. 7) yields the current position of `person`. To keep track of the persons, which have already been followed by the robot, the entity contains the properties `wasFollowed(Person person)` (l. 9) and `wasGuided(Person person, Room room)` (l. 8). The former indicates whether `person` was followed at least once, whereas the latter shows whether `person` was guided to `room` at least once. The property `adjacent(Room r1, Room r2)` (l. 10) indicates whether the rooms `r1` and `r2` are adjacent to each other.

8.1.3 Transport Robot Entity

Listing 8.5 depicts the `TransportRobot` entity model. The robot entity states the domain `LogisticsDomain` (l. 2) and the world entity `RoomsWorld` (l. 3). It consists of actions for managing concerns regarding robot movement (ll. 9-13), item handling (ll. 15-26), and guide and follow initiation (ll. 28-36). Querying the property `robotLocation()` (l. 6) yields the robot's current position. The robot moves between adjacent rooms by executing a `move(Room from, Room to)` action (ll. 9-13). The action can only be executed if the robots' current position is room `from` and the rooms `from` and

Entity

```
1 package smartsoftusage.entities;
2
3 domain smartsoftusage.types.LogisticsDomain;
4
5 world RoomsWorld {
6   property Boolean itemLocation(Item item, Room room);
7   property Room personLocation(Person person);
8   property Boolean wasGuided(Person person, Room room);
9   property Boolean wasFollowed(Person person);
10  property Boolean adjacent(Room r1, Room r2);
11 }
```

Listing 8.4: The world entity model `RoomsWorld`. The entity imports the domain `LogisticsDomain` (l. 3) and contains five properties (ll. 6-10) for querying item (l. 6) and person locations (l. 7), querying if a person was guided (l. 8) or followed (l. 9) by the robot, and for querying if two rooms are adjacent to each other (l. 10).

`to` are adjacent to each other (l. 11). This supports the planner in calculating a proper path for the robot's movement at runtime. Executing the action changes the position of the robot to room `to` (l. 12). The property `hasLoaded(Item item)` (l. 7) indicates whether the robot has loaded a specific item. The action `pickUp(Item item, Room room)` (ll. 15-20) can be executed when the robot's current position as well as the position of `item` is `room` (ll. 16-17). On action execution, the `item` is loaded onto the robot and therefore removed from `room` (l. 19). Analogously, when the robot has loaded a specific `item` and its current position is `room`, the robot is able to execute the action `drop(Item item, Room room)` (ll. 22-26), which results in the robot unloading the `item` in `room` (l. 25). The precondition of action `follow(Person person)` (ll. 28-31) is always true. Hence, it can always be executed spontaneously without any restrictions[2]. Executing the action results in setting the `wasFollowed` property of the world entity to reflect the robot has followed `person` (l. 30). Similarly, the action `guide(Person person, Room room)` can be executed spontaneously and executing it results in setting the `wasGuided` property of the world entity to reflect the robot has guided `person` to `room` (l. 35).

8.1.4 Transport Tasks

This section presents two parametrized tasks. They operate in the context of the `LogisticsDomain` described in Section 8.1.1. The execution of a `DeliverItem` task (Listing 8.6) results in the placement of the given `item` into the specified `room`. To this effect, the task requires the satisfaction of a `Deliver` goal (l. 5). It has one parameter `item` of type `Item` and one parameter `room` of type `Room` (l. 4). The task `Follow-`

[2]Note that the planner optimizes solutions, so these actions only occur when required.

Entity

```
1 package smartsoftusage.entities;
2 domain smartsoftusage.types.LogisticsDomain;
3 world  smartsoftusage.entities.RoomsWorld;
4
5 robot TransportRobot {
6   property Room robotLocation();
7   property Boolean hasLoaded(Item item);
8
9   action move(Room from, Room to) {
10    pre:  robotLocation() == from &&
11          RoomsWorld.adjacent(from, to);
12    post: robotLocation() == to;
13  }
14
15  action pickUp(Item item, Room room) {
16    pre:  RoomsWorld.itemLocation(item, room) &&
17          robotLocation() == room;
18    post: hasLoaded(item) && !RoomsWorld.itemLocation(item,
          room);
19  }
20
21  action drop(Item item, Room room) {
22    pre:  hasLoaded(item) && robotLocation() == room;
23    post: !hasLoaded(item) &&
24          RoomsWorld.itemLocation(item, room);
25  }
26
27  action follow(Person person) {
28    pre:  true;
29    post: RoomsWorld.wasFollowed(person);
30  }
31
32  action guide(Person person, Room room) {
33    pre:  true;
34    post: RoomsWorld.wasGuided(person, room);
35  }
36 }
```

Listing 8.5: A simplified version of the `TransportRobot` entity model.

Task

```
1 import smartsoftusage.goals.*;
2 domain smartsoftusage.types.LogisticsDomain;
3
4 task DeliverItem(Item item, Room room) {
5   Deliver(item, room);
6 }
```

Listing 8.6: The tasks `DeliverItem`, which imports the `LogisticsDomain`, defines two parameters and uses one `Deliver` goal.

Task

```
1 package smartsoftusage.tasks;
2
3 import smartsoftusage.goals.*;
4 domain smartsoftusage.types.LogisticsDomain;
5
6 task FollowPerson(Person person) {
7   Followed(person);
8 }
```

Listing 8.7: The tasks `FollowPerson`, which imports the `LogisticsDomain`, defines one parameter and contains one `Followed` goal.

`Person(Person person)` (Listing 8.7) leads the robot to follow `person`. To meet this requirement, it consists of one parameter `person` of type `Person` (l. 4) and requires the satisfaction of a `Followed` goal (l. 5).

8.1.5 Transport Goals

This section presents the goals `Deliver` (Listing 8.8) and `Followed` (Listing 8.9) utilized by the tasks `DeliverItem` (Listing 8.6) and `FollowPerson` (Listing 8.7), which are described in Section 8.1.4. Both goals refer to the `LogisticsDomain` (Figure 8.3) as well as the entities `TransportRobot` (Listing 8.5) and `RoomsWorld` (Listing 8.4). Each of them consists of a predicate that references a property defined in the `RoomsWorld` entity. The `Deliver` goal (Listing 8.8) has one parameter `item` of type `Item` and one parameter `room` of type `Room` (l. 7). It requires the satisfaction of the predicate `RoomsWorld.itemLocation(item, room)` (ll. 8-10), which is satisfied if the current location of the `item` is `room`. The `Followed` goal (Listing 8.7) has one parameter `person` of type `Person` (l. 7) and contains the predicate `RoomsWorld.wasFollowed(person)` (ll. 8-10), which is satisfied if the robot started and finished following the `person`.

Goal

```
1 package smartsoftusage.goals;
2
3 domain smartsoftusage.types.LogisticsDomain;
4 robot  smartsoftusage.entities.TransportRobot;
5 world  smartsoftusage.entities.RoomsWorld;
6
7 goal Deliver(Item item, Room room) {
8   predicate {
9     RoomsWorld.itemLocation(item, room)
10  }
11 }
```

Listing 8.8: The goal `Deliver` depends on the `LogisticsDomain`, the robot entity `TransportRobot`, and the world entity `RoomsWorld`. It defines two parameters and contains a predicate over the world state.

Goal

```
1 package smartsoftusage.goals;
2
3 domain smartsoftusage.types.LogisticsDomain;
4 robot  smartsoftusage.entities.TransportRobot;
5 world  smartsoftusage.entities.RoomsWorld;
6
7 goal Followed(Person person) {
8   predicate {
9     RoomsWorld.wasFollowed(person)
10  }
11 }
```

Listing 8.9: The goal `Followed` depends on the `LogisticsDomain`, the robot entity `TransportRobot`, and the world entity `RoomsWorld`. It defines one parameter and contains a predicate over the world state.

8.1.6 Integrating Handcrafted Code

As illustrated in Figure 8.2, the code produced by the generators has to be complemented with handcrafted code. To complete it, users have to create robot and world implementations for the generated interfaces `IWorld` and `IRobot`. The names of the handcrafted classes must be equal to names of their corresponding entity models. Further, entity models and their corresponding classes must be placed in the same package. For instance, considering the entities presented in Listing 8.5, the name of the corresponding handcrafted class has to be `TransportRobot` (cf. `RobotImpl` (Figure 8.2)) and the class has to be placed in the package `smartsoftusage.entities`. The interfaces generated from the entities `TransportRobot` (Listing 8.5) and `RoomsWorld` (Listing 8.4) as well as the handcrafted classes `TransportRobot` and `RobotWorld` are illustrated in Figure 8.10.

The interface `IRobot` as given in Figure 8.10 is generated from the `TransportRobot` entity (Listing 8.5). It contains one method for each action and two methods for each property of the corresponding entity. Each method generated for an action has the return type `void`, the same name as the action, and one parameter for each parameter of the action. The parameters have the same names and types as the parameters of the corresponding action. For instance, the method `void drop(Item item, Room room)` (Figure 8.10) is generated for the action `drop(Item item, Room room)` (Listing 8.5, ll. 20-23). Similar to the methods generated for actions, the first method generated for a property has the same name and signature as the property, as for instance `Boolean hasLoaded(Item item)` (Figure 8.10, interface `IRobot` and Listing 8.5, l. 7). The second method generated for a property has no parameters and has the name of the property starting with an uppercase letter, prefixed with the String `"stateOf"`. Its return type is a set of tuples representing all information about the relation defined by the corresponding property. The tuples arity is equal to one plus the number of the parameters of the query. The type of the first parameter of the tuple is bound to the type of the query. The other type parameters of the tuple are bound to the types of the parameters of the query. The method `Set<Pair<Boolean, Item>> stateOfHasLoaded()` (Figure 8.10, interface `IRobot`), for instance, is generated from `Boolean hasLoaded(Item tem)` (Listing 8.5, l. 7).

The interface `IWorld` as given in Figure 8.10 is generated from the entity `RoomsWorld` (Listing 8.4). Analog to the generated interface `IRobot`, the `IWorld` interface contains one method for each action and two methods for each property of its corresponding entity. Additionally, it contains one method for each class of its corresponding domain. The methods generated for the domain are expected to return all instances of the corresponding class that are relevant to the planner. They do not define any parameters and their names are equal to the name of the corresponding classes starting with an uppercase letter, prefixed with the String `"getAll"`. The return type of such a method is `Set<X>` where `X` is the name of the class the method is generated from. The method `Set<Item> getAllItem()` (Figure 8.10, interface `IWorld`), for example, is generated for the class `Item` (Figure 8.3).

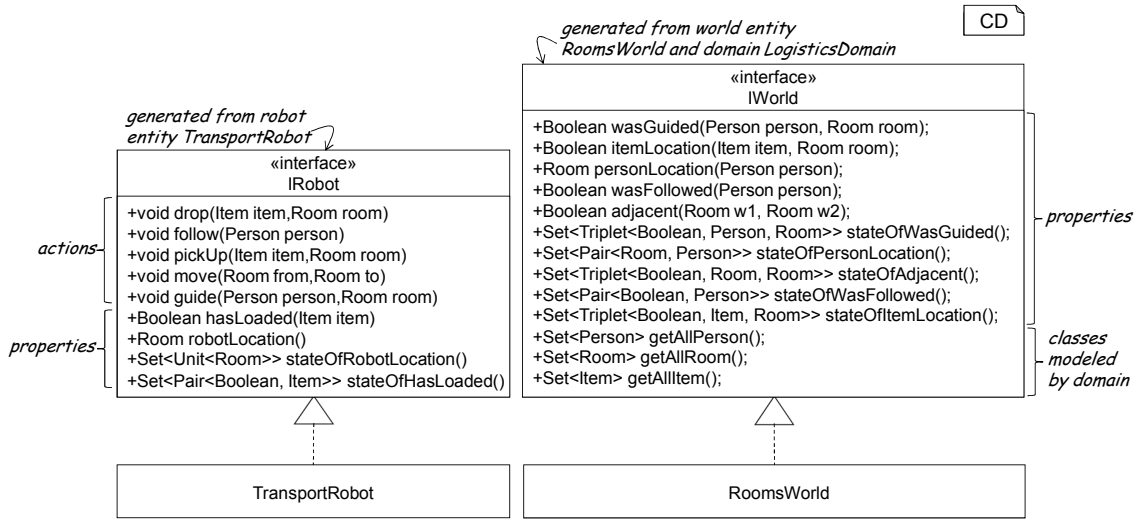


Figure 8.10: The interfaces `IRobot` and `IWorld` generated from the entities `TransportRobot` and `RoomsWorld` are implemented by the handcrafted classes `TransportRobot` and `RoomsWorld`.

Bash

```
1 $ sudo sh start-deployment.sh menu
```

Listing 8.11: Bash command to start the SmartSoft deployment.

8.2 Starting SmartSoft

This section describes how to start the SmartSoft[3] deployment. First, one has to connect via VNC to the robot. To start the deployment the bash command depicted in Listing 8.11 has to be executed in folder `/root/tmp/Deployment` from the robot. Executing the command opens a separate window for each SmartSoft component.

In the scenario control window (Figure 8.12) `Start Scenario` has to be selected. To choose the desired scenario, the `SmartLispServer` window requires to be opened (Figure 8.13). The `SmartLispServer` processes the commands that are sent to SmartSoft. For instance, it maps the symbolic positions (e.g., a room's name) to coordinates. The coordinates are used to navigate the robot to the desired symbolic destination.

8.3 Deployment of Architecture and User Interfaces

SmartSoft provides a high level interface that serves as communication endpoint. The reference architecture connects to it via the iserveU server (cf. Section 5.5.4). This section describes how to deploy, and start the iserveU server (Section 8.3.1), the architecture, the desktop UI (Section 8.3.2), and the tablet UI (Section 8.3.3). To deploy or start these components, one has to download the latest build as zip file from our website[4]. Its folder

[3] SmartSoft Website: `http://www.servicerobotik-ulm.de/drupal/?q=node/20`
[4] Reference architecture desktop UI: `http://www.se-rwth.de/materials/iserveu/`

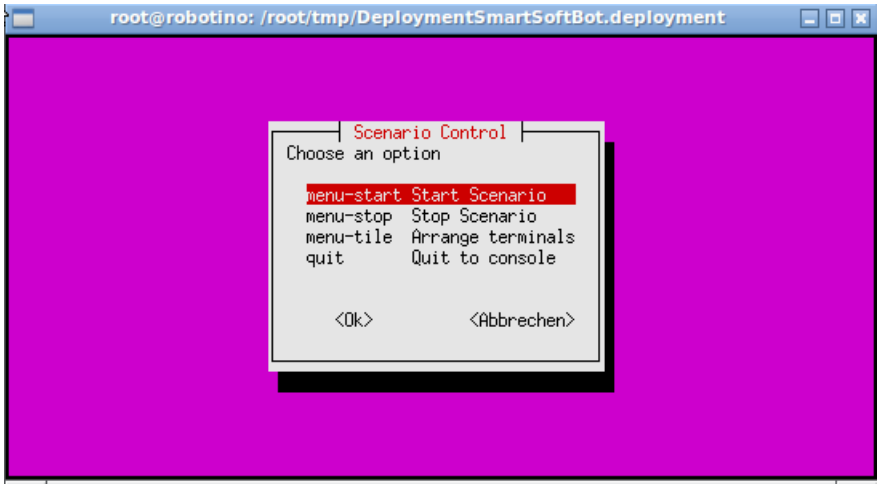


Figure 8.12: The SmartSoft Scenario Control window with options to start, stop or quit the deployment.

structure is depicted in Figure 8.14. Afterwards, Metric-FF [Met] in version 1.0 has to be installed to the folder `smartsoft-usage/src/main/resources/Metric-FF/Metric-FF-PLATFORM-BUILD/` (Figure 8.14) of the previously downloaded zip file. Finally, the iserveU server has to be downloaded from our companion website[5].

8.3.1 Starting the iserveU Server

To start the iserveU server, its jar file has to be copied to a target device (e.g., the robotics platform). The server is the first deployment component to start and runs on the Robotino. It is deployed automatically with the SmartSoft deployment and by default the server is expecting messages via port 20021. This port needs to be accessible and open on the device. It is possible to change it by launching the server with command line argument as depicted in Listing 8.15.

8.3.2 Deploying Reference Architecture and Desktop UI

The reference architecture is always started together with the desktop UI. For starting the desktop UI and the reference architecture, one has to enter the folder of the previously downloaded zip file. The folder contains two starter jars, two knowledge base json files, a source folder with all required models and Metric-FF files. The desktop UI and the reference architecture can be started as depicted in Listing 8.16. Afterwards, the knowledge base selection window is displayed (Figure 8.17). Section 7.1 describes how to operate the desktop UI.

[5]iserveU server: `http://www.se-rwth.de/materials/iserveu/`

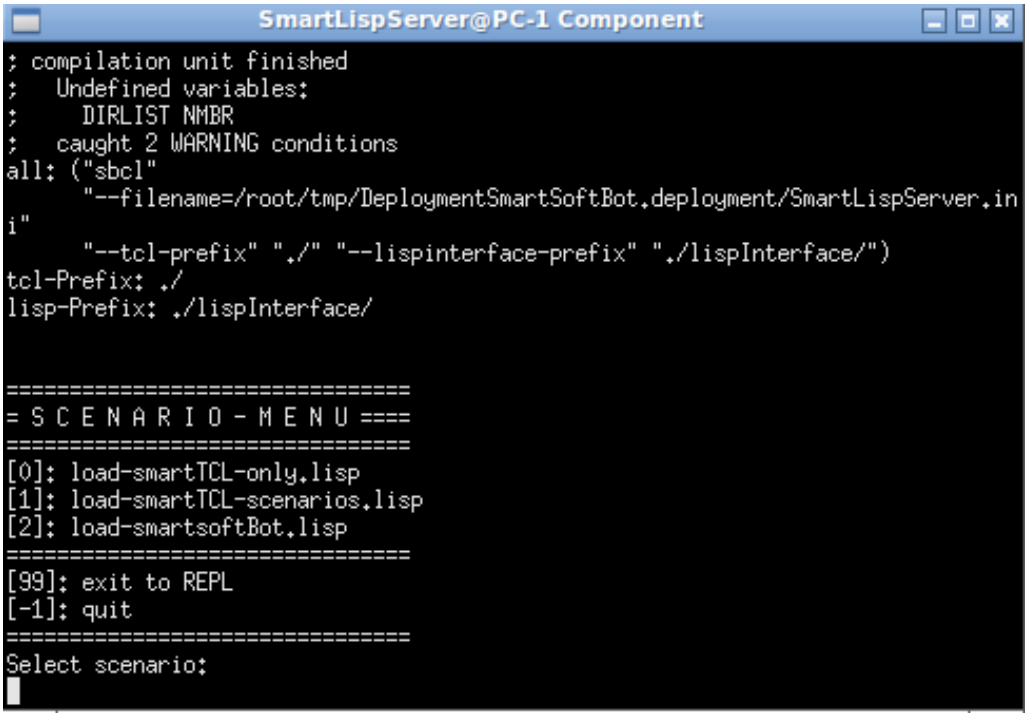


Figure 8.13: LispServer component window with three scenarios.

8.3.3 Starting the Tablet User Interface

The Android app supports user interaction at runtime. Executing the following steps is necessary before starting the app. First of all, the "app-release.apk"[6] has to be deployed to an Android device. The app requires a minimum Android version of 4.0.4 (Figure 8.18). Starting the app requires the server as well as the reference architecture and the desktop UI to run and to be in the same local network. After the launch, the app tries to connect to the desktop UI. The default port is configured as 20024 and can be configured together with the IP (Section 7.2).

8.4 Generalizing the Infrastructure

A collection of specific models has been created that describe tasks to deliver items, guide persons, and follow persons for a movable robot in a hospital domain. As the implementation abstracts from technology and domains, it can be applied to different domains and platforms with little effort. Section 8.4.1 describes how to generalize models to other domains. Section 8.4.2 explains the required modifications in an application with a different robotics middleware.

[6]Android tablet UI: `http://www.se-rwth.de/materials/iserveu/`

```
smartsoft-usage/
├── HLISimulation.json
├── Krankenhaus.json
├── starter.jar
├── starterWithHLI.jar
└── src/
    └── main/
        ├── models/
        │   └── ...
        └── resources/
            └── Metric-FF/
                ├── Metric-FF-LINUX-BUILD/
                └── Metric-FF-WINDOWS-BUILD/
                    └── *.exe
```

Figure 8.14: Folder structure of the downloaded zip. It contains two knowledge base files, two executable jars, a models folder and a folder Metric-FF where Metric-FF has to be installed too.

Bash

```
1 $ java -jar iserveu-server.jar -port <port>
```

Listing 8.15: Bash command to start the iserveU server.

8.4.1 Adapting Models

The iserveU models (cf. Section 8.1) define tasks, goals, entities, and domain types. To change the world domain the robot is used in, or to use the tasks defined for a robot entity on another robot platform, the entity models can be adapted or exchanged. Also, for a specific world and robot, the domain of tasks can be altered. This section describes the changes that an adaption of domain, entity, task, and goal models entail. Figure 8.19 depicts the corresponding model modifications.

Models serve as abstraction from specific domains, and a change in the models may require to adapt handcrafted implementations. Domain types are modeled as UML/P CDs [Rum11]. If the domain changes, the knowledge base, which stores and maintains the domain types, has to be adapted to support new types. The domain types may be used in entity, task, and goal models. To support the domain type's usage in the models, these

Bash

```
1  $ java -jar starter.jar -robip <server host>
2                          -robport <server port>
3                          -appport <port of appservices>
```

Listing 8.16: Bash command to start the reference architecture.

Figure 8.17: The knowledge base selection window (Section 7.1) is displayed after execution of the starter jar.

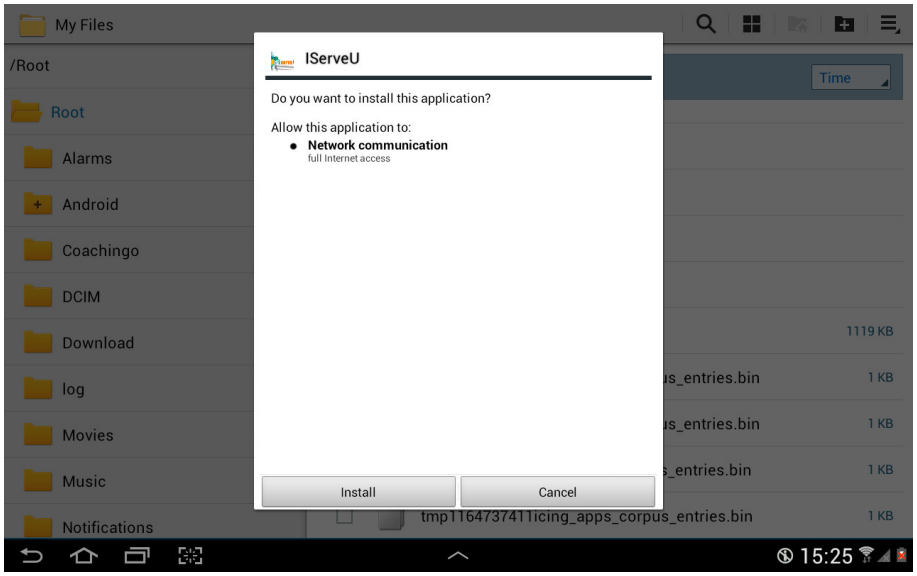


Figure 8.18: Install the iserveU app from the Android file explorer.

models have to be adapted. Besides that, domain types can affect the communication. If this is the case, the implementation of robot and/or world interfaces has to be adapted. The desktop UI enables to exchange and to load and store different knowledge bases. The domain types in the tasks are adapted from task models automatically. The following describes the changes that have to be made if an existing model is modified or a new model is added to the application. This is also depicted in Figure 8.19.

For the addition of a new task, a new task model has to be created. If a task model is modified, the application has to be modified at different locations. Whereas the desktop UI queries the available task types from the models automatically, new task types have to be added to the tablet UI manually. This results in adapting the task view and modifying the instantiation of new task objects. Besides that, the interrupt handling in the implementation of the interface `InterruptHandler` (Section 5.2.4) is responsible to instantiate task objects in case the robot should execute an idle task.

A modification of an existing goal model affects the task models that reference this goal. Besides that, test cases have to be adapted. The implementation of new test cases that

cover the added and modified functionality of the goal model is strongly recommended. To add a new goal to the application, the goal model has to be implemented and referenced from a task model.

If the robot or world entity model is altered, the robot, respectively world, interface implementations have to be modified accordingly to support new or modified properties and actions. The remaining parts of an application do not rely on implementations of the interfaces rather than the generated interfaces themselves. Considering the complexity and interweaving of the parts of an application, these changes are rather small.

The effort a modification of existing models entails is very little. In case of task, goal, action, and domain models, only models that refer to the modified model have to be adapted. Entity models require adapting the implementation of the generated entity interfaces.

8.4.2 Transfer to Other Platforms

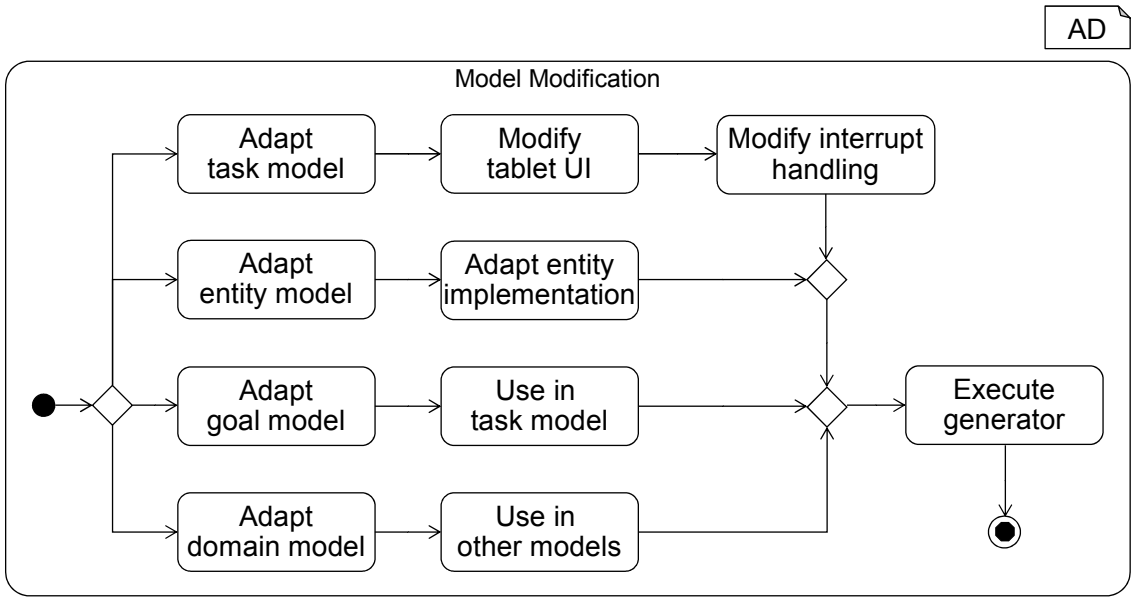


Figure 8.19: Changes entailed by modifications to different model kinds.

The implementation described in this chapter uses SmartSoft as middleware for controlling the robot. SmartSoft is a robotics middleware that enables to execute high-level commands on the robot. The architecture abstracts from the middleware and acts on symbolic actions. Only the implementation of the entities' interfaces interacts with the middleware. These interfaces define methods that execute certain actions or query certain properties that are defined in the entity models. They are responsible for invoking the employed middleware. The implementation of the middleware interactions could be defined in different ways. For SmartSoft as middleware, iserveU uses a fixed protocol and communicates with the middleware via TCP/IP (see Section 5.5.5 and Section 5.5.4). TCP/IP communication enables to execute the architecture on a different platform than the robot platform. The robot platform thus only has to run SmartSoft. Different middlewares can have different ways to notify whether a command has been executed successfully or not.

These responses received from the middleware have to be adapted to `SystemResponse` (see Section 5.2.3). The resulting system responses have to be provided to the reference architecture via the implementation of component `ResponseListener` (explained in Section 5.2 and Section 5.5.2).

Chapter 9

Related Work

This section highlights work related to the three aspects of our contribution. First, Section 9.1 discusses related modeling languages. Afterwards, Section 9.2 examines related approaches to integrated MDE with planning. Finally, Section 9.3 investigates related software architectures.

9.1 Modeling Languages

A textual DSML for modeling robot abilities is presented in [RHP+10]. The language enables to define sequences of actions, which are independent from specific robot platforms. The DSML provides the concepts *resource component*, *action*, and *variable*. A resource component contains a set of actions. Actions describe abstract abilities of a robot and are attached to resource components. Variables are globally available and can store action results. With this infrastructure, tasks are programmed imperatively and there is no decoupling between the tasks of a domain and robot abilities. In contrast to the approach presented in this report, actions are bound to specific robot platforms via specific code generators. This requires that robot experts develop proper code generators, which they should not be burdened with.

In [DPMH12], an ontology to describe a robot behavior is presented. It focuses on service robotics in health care, where the most important concepts are *objects*, *robots* and *persons*. These concepts are ordered hierarchically with objects being the most general kind of concept. They are also subject to inheritance. Relations between concepts are either actions that are executed on the robot, other objects, or senses. Robots can use the latter to recognize an object's properties. The approach also supports control structures that decide whether an action or a sense is used, and ultimately describe the behavior of the robot. Actions, senses, and control structures are described in textual tables, which can be utilized to implement a robot platform at a subsequent date. In our approach, robots also provide actions. However, the properties of domain objects are available on the model level and, thus, there are no *sense* models necessary. Instead, the implementation of sensing is subject to the individual platform and transparent to the domain expert. Also, the modeling languages (cf. Chapter 4) omit control structures and the presented reference architecture (cf. Chapter 5) delegates task solving to a sophisticated planner instead.

9.2 Integrating Modeling Languages with Planning

The presented reference architecture (cf. Chapter 5) uses PDDL [MGH+98], which could be used to model tasks, goals, actions, and properties without the developed DSMLs (cf. Chapter 4). PDDL provides predicates with typed parameters and safety conditions. Domains can extend other domains and define types, constants, predicates, and actions. Actions have preconditions and effects. A problem implements a domain and describes an initial state as well as a goal state. With PDDL, modelers can describe planning domains and their constituents of arbitrary complexity - something the presented DSMLs (cf. Chapter 4) protect developers from. Furthermore, translation from sequences of actions to effects in the real world (e.g., by mapping actions to robot API calls) is not foreseen with PDDL. The same issues hold for the KPLANNER language [HL09], which can generate plans with loops, and Readylog [FL08], which is a behavior modeling language with underspecification that extends GOLOG [LRL+97].

9.3 Robotics Reference Architectures

Due to the heterogeneity of robotics, there are only few reference architectures [LOC00, GRH+09], but various architectural styles [QGC+09, SSL11, BKH+13]. The three layer reference architecture presented in [LOC00] aims to facilitate extensibility. The first layer is the deliberate layer, which is connected to a HRI from which it receives commands (which resemble tasks). Based on the commands, an integrated planner component calculates a state change, which is sent to the middle layer. This task execution layer maps the received state to a behavior which can be processed by the underlying reactive layer. The reactive layer calculates control signals for the robot hardware based on the robot's position and the requested behavior, for instance, to avoid collisions. The architecture can be easily extended by adding new behaviors. The architecture presented in this report (cf. Chapter 5) also has a UI to receive tasks. These tasks are composed of goals which are processed by a planner. The planner calculates a solution for the task at hand that consists of a sequence of actions. These actions are sent to the middleware that derives the actual control signals. Applications developed with this infrastructure can be easily expanded by adding more tasks, goals or extending the entity models. Additionally, the architecture itself is not tied to a specific hardware, which facilitates to reuse it with further platforms.

The Care-O-Bot [GRH+09] is a service robot that supports caregivers. It is controlled via an integrated HRI that enables the user to select commands for the robot. The command is passed to a symbolic planner, which calculates a list of actions to fulfill the given command. The current world state is stored in a SQL database and the planner receives necessary information about the environment from it. The Care-O-Bot architecture is tied to a specific platform and does not decompose tasks into goals.

Chapter 10

Conclusion

We presented a modeling infrastructure to facilitate engineering of complex service robotics applications while retaining a separation of concerns between domain experts, integration experts, and robotics experts. To this effect, we introduced abstract models describing tasks, goals, domain types, and entities. These models support domain experts in describing requirements to the system (for instance in form of expected robot entity properties) and their causalities liberated from the notational noise and accidental complexity of a GPL. Integration of domain-specific models benefits from the same abstraction by modeling large parts of the execution infrastructure using the MAA C&C ADL. The MAA reference architecture also facilitates execution infrastructure extension via well-defined interfaces for additional components. Sophisticated code generation automatically translates tasks, goals, and entities of the domain, as well as architecture models into integrated executable artifacts. These include entity interfaces produced for robot and world experts to implement and hence ground the abstract actions and causalities of the models into reality. The resulting reference architecture implementation can easily be extended by handcrafted user interfaces and employs state of the art planning from tasks to sequences of actions. The latter are interpreted in the context of robot and world entities that decouple logical actions from their platform specific implementations. Ultimately, this integration supports easy reuse of tasks, goals, and entities in different contexts and hence facilitates service robotics application engineering.

Bibliography

[ABH+16] Kai Adam, Arvid Butting, Robert Heim, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Model-driven separation of concerns for service robotics. In *16th International Workshop on Domain-Specific Modeling (DSM 2016)*, pages 22–27. ACM, 2016.

[BDD+93] Manfred Broy, Frank Dederich, Claus Dendorfer, Max Fuchs, Thomas Gritzner, and Rainer Weber. The Design of Distributed Systems - An Introduction to FOCUS. Technical report, TUM-I9202, SFB-Bericht Nr. 342/2-2/92 A, 1993.

[BKH+13] Herman Bruyninckx, Markus Klotzbücher, Nico Hochgeschwender, Gerhard Kraetzschmar, Luca Gherardi, and Davide Brugali. The BRICS Component Model: A Model-Based Development Paradigm For Complex Robotics Software Systems. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1758–1764, New York, NY, USA, 2013. ACM.

[BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement*. Springer Verlag Heidelberg, 2001.

[BW98] Alan W. Brown and Kurt C. Wallnau. The Current State of CBSE. *IEEE Software*, 15(5), 1998.

[DPMH12] James P. Diprose, Beryl Plimmer, Bruce A. MacDonald, and John G. Hosking. How People Naturally Describe Robot Behaviour. In *Proceedings of Australasian Conference on Robotics and Automation*, pages 3–5, 2012.

[FL03] Maria Fox and Derek Long. PDDL2. 1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.

[FL08] Alexander Ferrein and Gerhard Lakemeyer. Logic-based Robot Control in Highly Dynamic Domains. *Robotics and Autonomous Systems*, 56(11):980–991, 2008.

[FN72] Richard E. Fikes and Nils J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial intelligence*, 2(3):189–208, 1972.

[Fow10] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, 2010.

[FR07] Robert France and Bernhard Rumpe. Model-Driven Development of Complex Software: A Research Roadmap. In *Future of Software Engineering 2007 at ICSE.*, pages 37–54, 2007.

[GHK+15] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiß, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. Integration of Handwritten and Generated Object-Oriented Code. In *Model-Driven Engineering and Software Development*, Communications in Computer and Information Science 580, pages 112–132. Springer, 2015.

[GKR+08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 925–926, 2008.

[GRH+09] Birgit Graf, Ulrich Reiser, Martin Hägele, Kathrin Mauz, and Peter Klein. Robotic Home Assistant Care-O-bot® 3 - Product Vision and Innovation Platform. In *Advanced Robotics and its Social Impacts (ARSO), 2009 IEEE Workshop on*, pages 139–144, Nov 2009.

[HDKN12] Andreas Hertle, Christian Dornhege, Thomas Keller, and Bernhard Nebel. Planning with Semantic Attachments: An Object-Oriented View. In *European Conference on Artificial Intelligence (ECAI)*, 2012.

[HKR+11] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-oriented Architectural Variability Using MontiCore. In *Software Architecture Conference (ECSA'11)*, pages 6:1–6:10. ACM, 2011.

[HL09] Yuxiao Hu and Hector Levesque. Planning with loops: Some new results. In *ICAPS Workshop on Generalized Planning*, page 37, 2009.

[HMSNR+15] Robert Heim, Pedram Mir Seyed Nazari, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Modeling Robot and World Interfaces for Reusable Tasks. In *Intelligent Robots and Systems Conference (IROS'15)*, pages 1793–1798. IEEE, 2015.

[HN01] Jörg Hoffmann and Bernhard Nebel. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, pages 253–302, 2001.

[HRW16] Robert Heim, Bernhard Rumpe, and Andreas Wortmann. Extending Architecture Description Languages With Exchangeable Component Behavior Languages. In *Conference on Software Engineering & Knowledge Engineering (SEKE'16)*, pages 1–6. KSI Research Inc., Fredericton, Canada, 2016.

[KRV10] Holger Krahn, Bernhard Rumpe, and Stefen Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010.

[LOC00] Mattias Lindström, Anders Orebäck, and Henrik I. Christensen. Berra: A Research Architecture for Service Robots. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, pages 3278–3283. IEEE, 2000.

[LRL+97] Hector J. Levesque, Raymond Reiter, Yves Lesperance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A Logic Programming Language for Dynamic Domains. *Journal of Logic Programming*, 31(1-3):59–83, 1997.

[Met] Metric-FF website https://fai.cs.uni-saarland.de/hoffmann/metric-ff.html.

[MGH+98] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL-The Planning Domain Definition Language. Tech report cvc tr-98-003/dcs tr-1165, Yale Center for Computational Vision and Control, 1998.

[MLM+13] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione, and Antony Tang. What Industry Needs from Architectural Languages: A Survey. *IEEE Transactions on Software Engineering*, 39(6):869–891, 2013.

[MT00] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 2000.

[NHWW16] Arne Nordmann, Nico Hochgeschwender, Dennis Wigand, and Sebastian Wrede. A Survey on Domain-Specific Modeling and Languages in Robotics. *Journal of Software Engineering for Robotics (JOSER)*, 2016.

[OMG10] Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure Version 2.3 (10-05-05), May 2010. `http://www.omg.org/spec/UML/2.3/Superstructure/PDF/` [Online; accessed 2015-12-17].

[QGC+09] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009.

[RHP+10] Michael Reckhaus, Nico Hochgeschwender, Paul G. Ploeger, Gerhard K. Kraetzschmar, and Sankt Augustin. A Platform-independent Programming Environment for Robot Control. In *Proceedings of the 1st International Workshop on Domain-Specific Languages and models for ROBotic systems (DSLrob'10)*, 2010.

[RRRW14] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems . In *1st International Workshop on Model-Driven Robot Software Engineering (MORSE 2014)*, CEUR Workshop Proceedings 1319, pages 66 – 77, York, Great Britain, July 2014.

[RRRW15] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics (JOSER)*, 6(1):33–57, 2015.

[RRW14] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Aachener Informatik-Berichte, Software Engineering, Band 20. Shaker Verlag, December 2014.

[RRW15a] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Composing Code Generators for C&C ADLs with Application-Specific Behavior Languages (Tool Demonstration). In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2015, pages 113–116, New York, NY, USA, 2015. ACM.

[RRW15b] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Tailoring the MontiArcAutomaton Component & Connector ADL for Generative Development. In Uwe Aßmann, Colin Atkinson, Erik Burger, Thomas Goldschmidt, and Ralf Reussner, editors, *Proceedings of the 2015 Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*, pages 41–47, L'Aquila, Italy, July 2015. ACM New York, NY, USA.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, München, Deutschland, 1996.

[Rum11] Bernhard Rumpe. *Modellierung mit UML, 2te Auflage*. Springer Berlin, September 2011.

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring, 2te Auflage*. Springer Berlin, Juni 2012.

[Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

[SSL11] Christian Schlegel, Andreas Steck, and Alex Lotz. Model-Driven Software Development in Robotics : Communication Patterns as Key for a Robotics Component Model. In Daisuke Chugo and Sho Yokota, editors, *Introduction to Modern Robotics*. iConcept Press, 2011.

[VSB+13] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, Simon Helsen, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley Software Patterns Series. Wiley, 2013.

[WHR14] Jon Whittle, John Hutchinson, and Mark Rouncefield. The State of Practice in Model-Driven Engineering. *Software, IEEE*, 31(3):79–85, 2014.

[Wil01] David S. Wile. Supporting the DSL Spectrum. *Computing and Information Technology*, 4:263–287, 2001.

List of Figures

List of Listings

Related Interesting Work from the SE Group, RWTH Aachen

Agile Model Based Software Engineering

Agility and modeling in the same project? This question was raised in [Rum04]: "Using an executable, yet abstract and multi-view modeling language for modeling, designing and programming still allows to use an agile development process." Modeling will be used in development projects much more, if the benefits become evident early, e.g with executable UML [Rum02] and tests [Rum03]. In [GKRS06], for example, we concentrate on the integration of models and ordinary programming code. In [Rum12] and [Rum16], the UML/P, a variant of the UML especially designed for programming, refactoring and evolution, is defined. The language workbench MontiCore [GKR+06, GKR+08] is used to realize the UML/P [Sch12]. Links to further research, e.g., include a general discussion of how to manage and evolve models [LRSS10], a precise definition for model composition as well as model languages [HKR+09] and refactoring in various modeling and programming languages [PR03]. In [FHR08] we describe a set of general requirements for model quality. Finally [KRV06] discusses the additional roles and activities necessary in a DSL-based software development project. In [CEG+14] we discuss how to improve reliability of adaptivity through models at runtime, which will allow developers to delay design decisions to runtime adaptation.

Generative Software Engineering

The UML/P language family [Rum12, Rum11, Rum16] is a simplified and semantically sound derivative of the UML designed for product and test code generation. [Sch12] describes a flexible generator for the UML/P based on the MontiCore language workbench [KRV10, GKR+06, GKR+08]. In [KRV06], we discuss additional roles necessary in a model-based software development project. In [GKRS06] we discuss mechanisms to keep generated and handwritten code separated. In [Wei12] demonstrate how to systematically derive a transformation language in concrete syntax. To understand the implications of executability for UML, we discuss needs and advantages of executable modeling with UML in agile projects in [Rum04], how to apply UML for testing in [Rum03] and the advantages and perils of using modeling languages for programming in [Rum02].

Unified Modeling Language (UML)

Starting with an early identification of challenges for the standardization of the UML in [KER99] many of our contributions build on the UML/P variant, which is described in the two books [Rum16] and [Rum12] implemented in [Sch12]. Semantic variation points of the UML are discussed in [GR11]. We discuss formal semantics for UML [BHP+98] and describe UML semantics using the "System Model" [BCGR09a], [BCGR09b], [BCR07b] and [BCR07a]. Semantic variation points have, e.g., been applied to define class diagram semantics [CGR08]. A precisely defined semantics for variations is applied, when checking variants of class diagrams [MRR11c] and objects diagrams [MRR11d] or the consistency of both kinds of diagrams [MRR11e]. We also apply these concepts to activity diagrams [MRR11b] which allows us to check for semantic differences of

activity diagrams [MRR11a]. The basic semantics for ADs and their semantic variation points is given in [GRR10]. We also discuss how to ensure and identify model quality [FHR08], how models, views and the system under development correlate to each other [BGH+98] and how to use modeling in agile development projects [Rum04], [Rum02]. The question how to adapt and extend the UML is discussed in [PFR02] describing product line annotations for UML and more general discussions and insights on how to use meta-modeling for defining and adapting the UML are included in [EFLR99], [FELR98] and [SRVK10].

Domain Specific Languages (DSLs)

Computer science is about languages. Domain Specific Languages (DSLs) are better to use, but need appropriate tooling. The MontiCore language workbench [GKR+06, KRV10, Kra10, GKR+08] allows the specification of an integrated abstract and concrete syntax format [KRV07b] for easy development. New languages and tools can be defined in modular forms [KRV08, GKR+07, Völ11] and can, thus, easily be reused. [Wei12] presents a tool that allows to create transformation rules tailored to an underlying DSL. Variability in DSL definitions has been examined in [GR11]. A successful application has been carried out in the Air Traffic Management domain [ZPK+11]. Based on the concepts described above, meta modeling, model analyses and model evolution have been discussed in [LRSS10] and [SRVK10]. DSL quality [FHR08], instructions for defining views [GHK+07], guidelines to define DSLs [KKP+09] and Eclipse-based tooling for DSLs [KRV07a] complete the collection.

Software Language Engineering

For a systematic definition of languages using composition of reusable and adaptable language components, we adopt an engineering viewpoint on these techniques. General ideas on how to engineer a language can be found in the GeMoC initiative [CBCR15, CCF+15]. As said, the MontiCore language workbench provides techniques for an integrated definition of languages [KRV07b, Kra10, KRV10]. In [SRVK10] we discuss the possibilities and the challenges using metamodels for language definition. Modular composition, however, is a core concept to reuse language components like in MontiCore for the frontend [Völ11, KRV08] and the backend [RRRW15]]. Language derivation is to our believe a promising technique to develop new languages for a specific purpose that rely on existing basic languages. How to automatically derive such a transformation language using concrete syntax of the base language is described in [HRW15, Wei12] and successfully applied to various DSLs. We also applied the language derivation technique to tagging languages that decorate a base language [GLRR15] and delta languages [HHK+15a, HHK+13], where a delta language is derived from a base language to be able to constructively describe differences between model variants usable to build feature sets.

Modeling Software Architecture & the MontiArc Tool

Distributed interactive systems communicate via messages on a bus, discrete event signals, streams of telephone or video data, method invocation, or data structures passed between software services. We use streams, statemachines and components [BR07] as well as expressive forms of composition and refinement [PR99] for semantics. Furthermore, we built a concrete tooling infrastructure called

MontiArc [HRR12] for architecture design and extensions for states [RRW13b]. MontiArc was extended to describe variability [HRR+11] using deltas [HRRS11, HKR+11] and evolution on deltas [HRRS12]. [GHK+07] and [GHK+08] close the gap between the requirements and the logical architecture and [GKPR08] extends it to model variants. [MRR14] provides a precise technique to verify consistency of architectural views [Rin14, MRR13] against a complete architecture in order to increase reusability. Co-evolution of architecture is discussed in [MMR10] and a modeling technique to describe dynamic architectures is shown in [HRR98].

Compositionality & Modularity of Models

[HKR+09] motivates the basic mechanisms for modularity and compositionality for modeling. The mechanisms for distributed systems are shown in [BR07] and algebraically underpinned in [HKR+07]. Semantic and methodical aspects of model composition [KRV08] led to the language workbench MontiCore [KRV10] that can even be used to develop modeling tools in a compositional form. A set of DSL design guidelines incorporates reuse through this form of composition [KKP+09]. [Völ11] examines the composition of context conditions respectively the underlying infrastructure of the symbol table. Modular editor generation is discussed in [KRV07a]. [RRRW15] applies compositionality to Robotics control. [CBCR15] (published in [CCF+15]) summarizes our approach to composition and remaining challenges in form of a conceptual model of the "globalized" use of DSLs. As a new form of decomposition of model information we have developed the concept of tagging languages in [GLRR15]. It allows to describe additional information for model elements in separated documents, facilitates reuse, and allows to type tags.

Semantics of Modeling Languages

The meaning of semantics and its principles like underspecification, language precision and detailedness is discussed in [HR04]. We defined a semantic domain called "System Model" by using mathematical theory in [RKB95, BHP+98] and [GKR96, KRB96]. An extended version especially suited for the UML is given in [BCGR09b] and in [BCGR09a] its rationale is discussed. [BCR07a, BCR07b] contain detailed versions that are applied to class diagrams in [CGR08]. To better understand the effect of an evolved design, detection of semantic differencing as opposed to pure syntactical differences is needed [MRR10]. [MRR11a, MRR11b] encode a part of the semantics to handle semantic differences of activity diagrams and [MRR11e] compares class and object diagrams with regard to their semantics. In [BR07], a simplified mathematical model for distributed systems based on black-box behaviors of components is defined. Meta-modeling semantics is discussed in [EFLR99]. [BGH+97] discusses potential modeling languages for the description of an exemplary object interaction, today called sequence diagram. [BGH+98] discusses the relationships between a system, a view and a complete model in the context of the UML. [GR11] and [CGR09] discuss general requirements for a framework to describe semantic and syntactic variations of a modeling language. We apply these on class and object diagrams in [MRR11e] as well as activity diagrams in [GRR10]. [Rum12] defines the semantics in a variety of code and test case generation, refactoring and evolution techniques. [LRSS10] discusses evolution and related issues in greater detail.

Evolution & Transformation of Models

Models are the central artifact in model driven development, but as code they are not initially correct and need to be changed, evolved and maintained over time. Model transformation is therefore essential to effectively deal with models. Many concrete model transformation problems are discussed: evolution [LRSS10, MMR10, Rum04], refinement [PR99, KPR97, PR94], refactoring [Rum12, PR03], translating models from one language into another [MRR11c, Rum12] and systematic model transformation language development [Wei12]. [Rum04] describes how comprehensible sets of such transformations support software development and maintenance [LRSS10], technologies for evolving models within a language and across languages, and mapping architecture descriptions to their implementation [MMR10]. Automaton refinement is discussed in [PR94, KPR97], refining pipe-and-filter architectures is explained in [PR99]. Refactorings of models are important for model driven engineering as discussed in [PR01, PR03, Rum12]. Translation between languages, e.g., from class diagrams into Alloy [MRR11c] allows for comparing class diagrams on a semantic level.

Variability & Software Product Lines (SPL)

Products often exist in various variants, for example cars or mobile phones, where one manufacturer develops several products with many similarities but also many variations. Variants are managed in a Software Product Line (SPL) that captures product commonalities as well as differences. Feature diagrams describe variability in a top down fashion, e.g., in the automotive domain [GHK+08] using 150% models. Reducing overhead and associated costs is discussed in [GRJA12]. Delta modeling is a bottom up technique starting with a small, but complete base variant. Features are additive, but also can modify the core. A set of commonly applicable deltas configures a system variant. We discuss the application of this technique to Delta-MontiArc [HRR+11, HRR+11] and to Delta-Simulink [HKM+13]. Deltas can not only describe spacial variability but also temporal variability which allows for using them for software product line evolution [HRRS12]. [HHK+13] and [HRW15] describe an approach to systematically derive delta languages. We also apply variability to modeling languages in order to describe syntactic and semantic variation points, e.g., in UML for frameworks [PFR02]. Furthermore, we specified a systematic way to define variants of modeling languages [CGR09] and applied this as a semantic language refinement on Statecharts in [GR11].

Cyber-Physical Systems (CPS)

Cyber-Physical Systems (CPS) [KRS12] are complex, distributed systems which control physical entities. Contributions for individual aspects range from requirements [GRJA12], complete product lines [HRRW12], the improvement of engineering for distributed automotive systems [HRR12] and autonomous driving [BR12a] to processes and tools to improve the development as well as the product itself [BBR07]. In the aviation domain, a modeling language for uncertainty and safety events was developed, which is of interest for the European airspace [ZPK+11]. A component and connector architecture description language suitable for the specific challenges in robotics is discussed in [RRW13b, RRW14]. Monitoring for smart and energy efficient buildings is developed as Energy Navigator toolset [KPR12, FPPR12, KLPR12].

State Based Modeling (Automata)

Today, many computer science theories are based on statemachines in various forms including Petri nets or temporal logics. Software engineering is particularly interested in using statemachines for modeling systems. Our contributions to state based modeling can currently be split into three parts: (1) understanding how to model object-oriented and distributed software using statemachines resp. Statecharts [GKR96, BCR07b, BCGR09b, BCGR09a], (2) understanding the refinement [PR94, RK96, Rum96] and composition [GR95] of statemachines, and (3) applying statemachines for modeling systems. In [Rum96] constructive transformation rules for refining automata behavior are given and proven correct. This theory is applied to features in [KPR97]. Statemachines are embedded in the composition and behavioral specification concepts of Focus [BR07]. We apply these techniques, e.g., in MontiArcAutomaton [RRW13a, RRW14] as well as in building management systems [FLP+11].

Robotics

Robotics can be considered a special field within Cyber-Physical Systems which is defined by an inherent heterogeneity of involved domains, relevant platforms, and challenges. The engineering of robotics applications requires composition and interaction of diverse distributed software modules. This usually leads to complex monolithic software solutions hardly reusable, maintainable, and comprehensible, which hampers broad propagation of robotics applications. The MontiArcAutomaton language [RRW13a] extends ADL MontiArc and integrates various implemented behavior modeling languages using MontiCore [RRW13b, RRW14, RRRW15] that perfectly fit Robotic architectural modeling. The LightRocks [THR+13] framework allows robotics experts and laymen to model robotic assembly tasks.

Automotive, Autonomic Driving & Intelligent Driver Assistance

Introducing and connecting sophisticated driver assistance, infotainment and communication systems as well as advanced active and passive safety-systems result in complex embedded systems. As these feature-driven subsystems may be arbitrarily combined by the customer, a huge amount of distinct variants needs to be managed, developed and tested. A consistent requirements management that connects requirements with features in all phases of the development for the automotive domain is described in [GRJA12]. The conceptual gap between requirements and the logical architecture of a car is closed in [GHK+07, GHK+08]. [HKM+13] describes a tool for delta modeling for Simulink [HKM+13]. [HRRW12] discusses means to extract a well-defined Software Product Line from a set of copy and paste variants. [RSW+15] describes an approach to use model checking techniques to identify behavioral differences of Simulink models. Quality assurance, especially of safety-related functions, is a highly important task. In the Carolo project [BR12a, BR12b], we developed a rigorous test infrastructure for intelligent, sensor-based functions through fully-automatic simulation [BBR07]. This technique allows a dramatic speedup in development and evolution of autonomous car functionality, and thus enables us to develop software in an agile way [BR12a]. [MMR10] gives an overview of the current state-of-the-art in development and evolution on a more general level by considering any kind of critical system that relies on architectural descriptions. As tooling infrastructure, the SSELab storage, versioning and management services [HKR12] are essential for many projects.

Energy Management

In the past years, it became more and more evident that saving energy and reducing CO2 emissions is an important challenge. Thus, energy management in buildings as well as in neighborhoods becomes equally important to efficiently use the generated energy. Within several research projects, we developed methodologies and solutions for integrating heterogeneous systems at different scales. During the design phase, the Energy Navigators Active Functional Specification (AFS) [FPPR12, KPR12] is used for technical specification of building services already. We adapted the well-known concept of statemachines to be able to describe different states of a facility and to validate it against the monitored values [FLP+11]. We show how our data model, the constraint rules and the evaluation approach to compare sensor data can be applied [KLPR12].

Cloud Computing & Enterprise Information Systems

The paradigm of Cloud Computing is arising out of a convergence of existing technologies for web-based application and service architectures with high complexity, criticality and new application domains. It promises to enable new business models, to lower the barrier for web-based innovations and to increase the efficiency and cost-effectiveness of web development [KRR14]. Application classes like Cyber-Physical Systems and their privacy [HHK+14, HHK+15b], Big Data, App and Service Ecosystems bring attention to aspects like responsiveness, privacy and open platforms. Regardless of the application domain, developers of such systems are in need for robust methods and efficient, easy-to-use languages and tools [KRS12]. We tackle these challenges by perusing a model-based, generative approach [NPR13]. The core of this approach are different modeling languages that describe different aspects of a cloud-based system in a concise and technology-agnostic way. Software architecture and infrastructure models describe the system and its physical distribution on a large scale. We apply cloud technology for the services we develop, e.g., the SSELab [HKR12] and the Energy Navigator [FPPR12, KPR12] but also for our tool demonstrators and our own development platforms. New services, e.g., collecting data from temperature, cars etc. can now easily be developed.

[BBR07] Christian Basarke, Christian Berger, and Bernhard Rumpe. Software & Systems Engineering Process and Tools for the Development of Autonomous Driving Intelligence. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 4(12):1158–1174, 2007.

[BCGR09a] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Considerations and Rationale for a UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 43–61. John Wiley & Sons, November 2009.

[BCGR09b] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Definition of the UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 63–93. John Wiley & Sons, November 2009.

[BCR07a] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 2: The Control Model. Technical Report TUM-I0710, TU Munich, Germany, February 2007.

[BCR07b] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 3: The State Machine Model. Technical Report TUM-I0711, TU Munich, Germany, February 2007.

[BGH+97] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and Complete Object Interaction Descriptions. In *Object-oriented Behavioral Semantics Workshop (OOPSLA'97)*, Technical Report TUM-I9737, Germany, 1997. TU Munich.

[BGH+98] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Systems, Views and Models of UML. In *Proceedings of the Unified Modeling Language, Technical Aspects and Applications*, pages 93–109. Physica Verlag, Heidelberg, Germany, 1998.

[BHP+98] Manfred Broy, Franz Huber, Barbara Paech, Bernhard Rumpe, and Katharina Spies. Software and System Modeling Based on a Unified Formal Semantics. In *Workshop on Requirements Targeting Software and Systems Engineering (RTSE'97)*, LNCS 1526, pages 43–68. Springer, 1998.

[BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, Februar 2007.

[BR12a] Christian Berger and Bernhard Rumpe. Autonomous Driving - 5 Years after the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System. In *Automotive Software Engineering Workshop (ASE'12)*, pages 789–798, 2012.

[BR12b] Christian Berger and Bernhard Rumpe. Engineering Autonomous Driving Software. In C. Rouff and M. Hinchey, editors, *Experience from the DARPA Urban Challenge*, pages 243–271. Springer, Germany, 2012.

[CBCR15] Tony Clark, Mark van den Brand, Benoit Combemale, and Bernhard Rumpe. Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages*, LNCS 9400, pages 7–20. Springer, 2015.

[CCF+15] Betty H. C. Cheng, Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe, editors. *Globalizing Domain-Specific Languages*, LNCS 9400. Springer, 2015.

[CEG+14] Betty Cheng, Kerstin Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi Müller, Patrizio Pelliccione, Anna Perini, Nauman Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann, and Norha Villegas. Using Models at Runtime to Address Assurance for Self-Adaptive Systems. In *Models@run.time*, LNCS 8378, pages 101–136. Springer, Germany, 2014.

[CGR08] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, TU Braunschweig, Germany, 2008.

[CGR09] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within Modeling Language Definitions. In *Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, LNCS 5795, pages 670–684. Springer, 2009.

[EFLR99] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Meta-Modelling Semantics of UML. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 45–60. Kluver Academic Publisher, 1999.

[FELR98] Robert France, Andy Evans, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. *Computer Standards & Interfaces*, 19(7):325–334, November 1998.

[FHR08] Florian Fieber, Michaela Huhn, and Bernhard Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik-Spektrum*, 31(5):408–424, Oktober 2008.

[FLP+11] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. State-based Modeling of Buildings and Facilities. In *Enhanced Building Operations Conference (ICEBO'11)*, 2011.

[FPPR12] M. Norbert Fisch, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. The Energy Navigator - A Web-Platform for Performance Design and Management. In *Energy Efficiency in Commercial Buildings Conference(IEECB'12)*, 2012.

[GHK+07] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, and Bernhard Rumpe. View-based Modeling of Function Nets. In *Object-oriented Modelling of Embedded Real-Time Systems Workshop (OMER4'07)*, 2007.

[GHK+08] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In *Proceedings of 4th European Congress ERTS - Embedded Real Time Software*, 2008.

[GKPR08] Hans Grönniger, Holger Krahn, Claas Pinkernell, and Bernhard Rumpe. Modeling Variants of Automotive Systems using Views. In *Modellbasierte Entwicklung von*

eingebetteten Fahrzeugfunktionen, Informatik Bericht 2008-01, pages 76–89. TU Braunschweig, 2008.

[GKR96] Radu Grosu, Cornel Klein, and Bernhard Rumpe. Enhancing the SysLab System Model with State. Technical Report TUM-I9631, TU Munich, Germany, July 1996.

[GKR+06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänspezifischer Sprachen. Informatik-Bericht 2006-04, CFG-Fakultät, TU Braunschweig, August 2006.

[GKR+07] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Textbased Modeling. In *4th International Workshop on Software Language Engineering, Nashville*, Informatik-Bericht 4/2007. Johannes-Gutenberg-Universität Mainz, 2007.

[GKR+08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 925–926, 2008.

[GKRS06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, and Martin Schindler. Integration von Modellen in einen codebasierten Softwareentwicklungsprozess. In *Modellierung 2006 Conference*, LNI 82, Seiten 67–81, 2006.

[GLRR15] Timo Greifenberg, Markus Look, Sebastian Roidl, and Bernhard Rumpe. Engineering Tagging Languages for DSLs. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 34–43. ACM/IEEE, 2015.

[GR95] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical Report TUM-I9533, TU Munich, Germany, October 1995.

[GR11] Hans Grönniger and Bernhard Rumpe. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems*, LNCS 6662, pages 17–32. Springer, 2011.

[GRJA12] Tim Gülke, Bernhard Rumpe, Martin Jansen, and Joachim Axmann. High-Level Requirements Management and Complexity Costs in Automotive Development Projects: A Problem Statement. In *Requirements Engineering: Foundation for Software Quality (REFSQ'12)*, 2012.

[GRR10] Hans Grönniger, Dirk Reiß, and Bernhard Rumpe. Towards a Semantics of Activity Diagrams with Semantic Variation Points. In *Conference on Model Driven Engineering Languages and Systems (MODELS'10)*, LNCS 6394, pages 331–345. Springer, 2010.

[HHK+13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, and Ina Schaefer. Engineering Delta Modeling Languages. In *Software Product Line Conference (SPLC'13)*, pages 22–31. ACM, 2013.

[HHK+14] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. User-driven Privacy Enforcement for Cloud-based Services in the Internet of Things. In *Conference on Future Internet of Things and Cloud (FiCloud'14)*. IEEE, 2014.

[HHK+15a] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, Ina Schaefer, and Christoph Schulze. Systematic Synthesis of Delta Modeling Languages. *Journal on Software Tools for Technology Transfer (STTT)*, 17(5):601–626, October 2015.

[HHK+15b] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. A comprehensive approach to privacy in the cloud-based Internet of Things. *Future Generation Computer Systems*, 56:701–718, 2015.

[HKM+13] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. First-Class Variability Modeling in Matlab/Simulink. In *Variability Modelling of Software-intensive Systems Workshop (VaMoS'13)*, pages 11–18. ACM, 2013.

[HKR+07] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An Algebraic View on the Semantics of Model Composition. In *Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'07)*, LNCS 4530, pages 99–113. Springer, Germany, 2007.

[HKR+09] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Scaling-Up Model-Based-Development for Large Heterogeneous Systems with Compositional Modeling. In *Conference on Software Engineeering in Research and Practice (SERP'09)*, pages 172–176, July 2009.

[HKR+11] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-oriented Architectural Variability Using MontiCore. In *Software Architecture Conference (ECSA'11)*, pages 6:1–6:10. ACM, 2011.

[HKR12] Christoph Herrmann, Thomas Kurpick, and Bernhard Rumpe. SSELab: A Plug-In-Based Framework for Web-Based Project Portals. In *Developing Tools as Plug-Ins Workshop (TOPI'12)*, pages 61–66. IEEE, 2012.

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, October 2004.

[HRR98] Franz Huber, Andreas Rausch, and Bernhard Rumpe. Modeling Dynamic Component Interfaces. In *Technology of Object-Oriented Languages and Systems (TOOLS 26)*, pages 58–70. IEEE, 1998.

[HRR+11] Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer, and Frank van der Linden. Hierarchical Variability Modeling for Software Architectures. In *Software Product Lines Conference (SPLC'11)*, pages 150–159. IEEE, 2011.

[HRR12] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, February 2012.

[HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta Modeling for Software Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VII*, pages 1 – 10. fortiss GmbH, 2011.

[HRRS12] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Evolving Delta-oriented Software Product Line Architectures. In *Large-Scale Complex IT Systems. Development, Operation and Management, 17th Monterey Workshop 2012*, LNCS 7539, pages 183–208. Springer, 2012.

[HRRW12] Christian Hopp, Holger Rendel, Bernhard Rumpe, and Fabian Wolf. Einführung eines Produktlinienansatzes in die automotive Softwareentwicklung am Beispiel von Steuergerätesoftware. In *Software Engineering Conference (SE'12)*, LNI 198, Seiten 181–192, 2012.

[HRW15] Katrin Hölldobler, Bernhard Rumpe, and Ingo Weisemöller. Systematically Deriving Domain-Specific Transformation Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 136–145. ACM/IEEE, 2015.

[KER99] Stuart Kent, Andy Evans, and Bernhard Rumpe. UML Semantics FAQ. In A. Moreira and S. Demeyer, editors, *Object-Oriented Technology, ECOOP'99 Workshop Reader*, LNCS 1743, Berlin, 1999. Springer Verlag.

[KKP+09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Domain-Specific Modeling Workshop (DSM'09)*, Techreport B-108, pages 7–13. Helsinki School of Economics, October 2009.

[KLPR12] Thomas Kurpick, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Modeling Cyber-Physical Systems: Model-Driven Specification of Energy Efficient Buildings. In *Modelling of the Physical World Workshop (MOTPW'12)*, pages 2:1–2:6. ACM, October 2012.

[KPR97] Cornel Klein, Christian Prehofer, and Bernhard Rumpe. Feature Specification and Refinement with State Transition Diagrams. In *Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems*, pages 284–297. IOS-Press, 1997.

[KPR12] Thomas Kurpick, Claas Pinkernell, and Bernhard Rumpe. Der Energie Navigator. In H. Lichter and B. Rumpe, Editoren, *Entwicklung und Evolution von Forschungssoftware. Tagungsband, Rolduc, 10.-11.11.2011*, Aachener Informatik-Berichte, Software Engineering, Band 14. Shaker Verlag, Aachen, Deutschland, 2012.

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering, Band 1. Shaker Verlag, März 2010.

[KRB96] Cornel Klein, Bernhard Rumpe, and Manfred Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model. In

Workshop on Formal Methods for Open Object-based Distributed Systems, IFIP Advances in Information and Communication Technology, pages 323–338. Chapmann & Hall, 1996.

[KRR14] Helmut Krcmar, Ralf Reussner, and Bernhard Rumpe. *Trusted Cloud Computing*. Springer, Schweiz, December 2014.

[KRS12] Stefan Kowalewski, Bernhard Rumpe, and Andre Stollenwerk. Cyber-Physical Systems - eine Herausforderung für die Automatisierungstechnik? In *Proceedings of Automation 2012, VDI Berichte 2012*, Seiten 113–116. VDI Verlag, 2012.

[KRV06] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In *Domain-Specific Modeling Workshop (DSM'06)*, Technical Report TR-37, pages 150–158. Jyväskylä University, Finland, 2006.

[KRV07a] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Domain-Specific Modeling Workshop (DSM'07)*, Technical Reports TR-38. Jyväskylä University, Finland, 2007.

[KRV07b] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'07)*, LNCS 4735, pages 286–300. Springer, 2007.

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In *Conference on Objects, Models, Components, Patterns (TOOLS-Europe'08)*, LNBIP 11, pages 297–315. Springer, 2008.

[KRV10] Holger Krahn, Bernhard Rumpe, and Stefen Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010.

[LRSS10] Tihamer Levendovszky, Bernhard Rumpe, Bernhard Schätz, and Jonathan Sprinkle. Model Evolution and Management. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 241–270. Springer, 2010.

[MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer*, 43(5):42–48, May 2010.

[MRR10] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. A Manifesto for Semantic Model Differencing. In *Proceedings Int. Workshop on Models and Evolution (ME'10)*, LNCS 6627, pages 194–203. Springer, 2010.

[MRR11a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *Conference on Foundations of Software Engineering (ESEC/FSE '11)*, pages 179–189. ACM, 2011.

[MRR11b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. An Operational Semantics for Activity Diagrams using SMV. Technical Report AIB-2011-07, RWTH Aachen University, Aachen, Germany, July 2011.

[MRR11c] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 592–607. Springer, 2011.

[MRR11d] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Modal Object Diagrams. In *Object-Oriented Programming Conference (ECOOP'11)*, LNCS 6813, pages 281–305. Springer, 2011.

[MRR11e] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Semantically Configurable Consistency Analysis for Class and Object Diagrams. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 153–167. Springer, 2011.

[MRR13] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views. In Meyer, B. and Baresi, L. and Mezini, M., editor, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, pages 444–454. ACM New York, 2013.

[MRR14] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Verifying Component and Connector Models against Crosscutting Structural Views. In *Software Engineering Conference (ICSE'14)*, pages 95–105. ACM, 2014.

[NPR13] Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In *Model-Driven Engineering for High Performance and Cloud Computing Workshop*, CEUR Workshop Proceedings 1118, pages 15–24, 2013.

[PFR02] Wolfgang Pree, Marcus Fontoura, and Bernhard Rumpe. Product Line Annotations with UML-F. In *Software Product Lines Conference (SPLC'02)*, LNCS 2379, pages 188–197. Springer, 2002.

[PR94] Barbara Paech and Bernhard Rumpe. A new Concept of Refinement used for Behaviour Modelling with Automata. In *Proceedings of the Industrial Benefit of Formal Methods (FME'94)*, LNCS 873, pages 154–174. Springer, 1994.

[PR99] Jan Philipps and Bernhard Rumpe. Refinement of Pipe-and-Filter Architectures. In *Congress on Formal Methods in the Development of Computing System (FM'99)*, LNCS 1708, pages 96–115. Springer, 1999.

[PR01] Jan Philipps and Bernhard Rumpe. Roots of Refactoring. In Kilov, H. and Baclavski, K., editor, *Tenth OOPSLA Workshop on Behavioral Semantics. Tampa Bay, Florida, USA, October 15*. Northeastern University, 2001.

[PR03] Jan Philipps and Bernhard Rumpe. Refactoring of Programs and Specifications. In Kilov, H. and Baclavski, K., editor, *Practical Foundations of Business and System Specifications*, pages 281–297. Kluwer Academic Publishers, 2003.

[Rin14] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems*. Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, 2014.

[RK96] Bernhard Rumpe and Cornel Klein. Automata Describing Object Behavior. In B. Harvey and H. Kilov, editors, *Object-Oriented Behavioral Specifications*, pages 265–286. Kluwer Academic Publishers, 1996.

[RKB95] Bernhard Rumpe, Cornel Klein, and Manfred Broy. Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme - Syslab-Systemmodell. Technischer Bericht TUM-I9510, TU München, Deutschland, März 1995.

[RRRW15] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics (JOSER)*, 6(1):33–57, 2015.

[RRW13a] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. In *Software Engineering Workshopband (SE'13)*, LNI 215, pages 155–170, 2013.

[RRW13b] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Conference on Robotics and Automation (ICRA'13)*, pages 10–12. IEEE, 2013.

[RRW14] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Aachener Informatik-Berichte, Software Engineering, Band 20. Shaker Verlag, December 2014.

[RSW+15] Bernhard Rumpe, Christoph Schulze, Michael von Wenckstern, Jan Oliver Ringert, and Peter Manhart. Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. In *Software Product Line Conference (SPLC'15)*, pages 141–150. ACM, 2015.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, München, Deutschland, 1996.

[Rum02] Bernhard Rumpe. Executable Modeling with UML - A Vision or a Nightmare? In T. Clark and J. Warmer, editors, *Issues & Trends of Information Technology Management in Contemporary Associations, Seattle*, pages 697–701. Idea Group Publishing, London, 2002.

[Rum03] Bernhard Rumpe. Model-Based Testing of Object-Oriented Systems. In *Symposium on Formal Methods for Components and Objects (FMCO'02)*, LNCS 2852, pages 380–402. Springer, November 2003.

[Rum04] Bernhard Rumpe. Agile Modeling with the UML. In *Workshop on Radical Innovations of Software and Systems Engineering in the Future (RISSEF'02)*, LNCS 2941, pages 297–309. Springer, October 2004.

[Rum11] Bernhard Rumpe. *Modellierung mit UML, 2te Auflage*. Springer Berlin, September 2011.

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring, 2te Auflage*. Springer Berlin, Juni 2012.

[Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

[SRVK10] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. Metamodelling: State of the Art and Research Challenges. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 57–76. Springer, 2010.

[THR+13] Ulrike Thomas, Gerd Hirzinger, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. A New Skill Based Robot Programming Language Using UML/P Statecharts. In *Conference on Robotics and Automation (ICRA'13)*, pages 461–466. IEEE, 2013.

[Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011.

[Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering, Band 12. Shaker Verlag, 2012.

[ZPK+11] Massimiliano Zanin, David Perez, Dimitrios S Kolovos, Richard F Paige, Kumardev Chatterjee, Andreas Horst, and Bernhard Rumpe. On Demand Data Analysis and Filtering for Inaccurate Flight Trajectories. In *Proceedings of the SESAR Innovation Days*. EUROCONTROL, 2011.