Model Transformations in eHome Systems

Priit Salumaa

Master Thesis
Aachen, 11. November 2005

Submitted to the Department of Computer Science III
RWTH Aachen University

Corrector: Prof. Dr.-Ing. Manfred Nagl
External Corrector: Prof. Dr. rer. nat. Otto Spaniol
Supervisor: Dipl.-Inform. Ulrich Norbisrath

Mu isa ütles kord: “Elus viivad edasi
vaid julged otsused.”

My father once said: “Only brave decisions
will take you forward in life.”

Acknowledgements

This work has been one of the challenges of my life. I hereby thank sincerely the people who have supported and guided me from my heart:

See töö ei oleks saanud teoks ilma inimesteta mu kodus, Eestis. See on olnud üks mu elu raskemaid ülesandeid ning kestnud viimased kaks aastat. Ma poleks nii kaugele jõudnud oma perekonna, sugulaste ja sõprade toeta. Siinkohal tahangi ma teid kõiki kogu südamest tänada. Tänu teile, mu isa ja ema, teie kasvatusele, olen ma see, kes olen ning kirjutan neid sõnu siin. Teie olete õpetanud mind elama ja lükanud sahana lahti tee, mis praeguseks on toonud mind siia, kodust nii kaugele. Mu õde-venda Liisi ja Paul, te olete olnud mulle alati seltsiks ja näidanud, mis asi on südametunnistus ning mida tähendab kellegi eest hoolitseda. Ma tänan teid, mu kallid vanaisa ja vanaema, kes te olete mind alati hoidnud ja õpetanud haridust väärtustama. Ma tahan ka tänada oma sõpru Sveni, Jaanust, Vahurit ja Askot. Te olete mind pannud vaimustuma arvutiteadusest, aidanud rasketel hetkedel ning ühtlasi näidanud, et peale töö ja kooli on elus ka muud.

Kõiki inimesi, kellele ma tänu võlgnen, või kes mulle olulised on, on üsna võimatu üles lugeda. Selleks kuluks veel sama palju paberit ja sõnu, kui mu magistritöö võtab. Seega, ma tänan teid kõiki, oma tuttavaid ja sõpru Eestis või eestlasi Saksamaal, kes te olete olnud mulle toeks ja pere eest.

I want to thank all the people who have supported and guided me through these two years in Germany. It has been a wonderful experience here to work with you or share the kitchen. My special gratitude belongs to my academic advisor Ulrich Norbisrath. Ulrich, you are more than a mentor and a colleague, you have become a dear friend to me during these years.

I want to thank Prof. Dr. Ing. Manfred Nagl who introduced and accepted me to the Department of Computer Science 3 at the RWTH Aachen University. He and Prof. Dr. rer. nat. Otto Spaniol have also been my advisors in the Graduate School "Software for Mobile Communication". Participating in the graduate school has given a great additional academic value to my studies. I want to thank all the people from our eHome group for their support and cooperation. Special thanks go to people who have supported me in writing this thesis: from Kassel University – Prof. Dr. Albert Zündorf, Christian Schneider, and Leif Geiger; from our department – Adam Malik, Liviana Manolescu, Michael Kirchof, Erhard Schultchen, Christof Mosler, Daniel Retkowitz, Ibrahim Armac, Tim Schwerdtner, and Daniel Rose; from Estonia – Merily Plado and Redi Koobak. I would also like to thank the DAAD and Siemens for providing me with financial support for my Master's studies.

I affirm thereby that I composed this work independently and used no other sources and tools but the specified ones and that I marked all quotes as such.

Hiermit versichere ich, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, 11. November 2005 Priit Salumaa

Priit Salumaa

Model Transformations in eHome Systems

Contents

Chapter 1

Introduction

The modern trends in software engineering are looking towards more generic approaches for software development. The latest developments in the field of software engineering bring us to advanced techniques like Model Driven Architecture (MDA) [KWB03], visual languages, or graph-rewriting approaches. This is a logical development in the software engineering considering the history and evolution of programming languages. After the machine code was invented, the programming languages have evolved towards higher abstraction levels which has given us dynamic addressing, variable names, reusable function declarations, object-orientation, design patterns, or visual modelling. It is obvious that generations of programming languages have been developed to facilitate software development and this trend will probably carry on through the whole evolution of software development. We refer to the software development techniques, currently at the highest level of abstraction as *model-driven development*. Model-driven development relies completely on modelling of software models and replaces programming with code generation.

The idea of model-driven development has not yet materialized for the software development on a large scale, and is probably not applicable in every area of software development. However, there are areas and application fields which use these generic techniques successfully. For example, model-driven development techniques are applied in the field of *eHomes* also known as *smart homes*. The present thesis about eHomes focuses on the following topics: development of an appropriate model for eHomes and handling of its transformations during the specification, configuration, and deployment process of eHome systems; development of tool support for the process; and the research on advanced model-driven development techniques for creation of the eHome model and the tools supporting the above mentioned process for eHome systems.

This thesis was developed within the framework of research activities of the *Integrated eBusiness Systems in Home Automation* group (also known as the *eHome group*) [RWT]. The eHome group is a part of the Department of Computer Science 3 at the RWTH Aachen University. Our department has a remarkable history in the field of research on graph transformation theory and graph-rewriting systems. In fact, the graph-rewriting system called PROGRES [Sch91] was developed in the late eighties at our department. The work on this system contributes a great deal to research on model-driven development. As the eHome group itself is active in the home automation field, concentrating on eHomes, eHome services, their development and deployment, we use the model-driven development techniques extensively in our work. The research made by the eHome group has also contributed to the work with numerous enabling technologies, such as component based service frameworks (OSGi [CG01]), different communication protocols (X10 [X1004], UPnP [JW03]), semantic webs and knowledge bases (OWL [BvHH^{+}03]), to name but a few.

The term *eHome*, also known as smart home, denotes a living space, which is equipped with different appliances. These appliances are computer controlled devices, with features that can be combined in order to offer value added services to the inhabitant of the eHome. These value added services, in other words *eHome services* are related to the fields of comfort, security, or infotainment. eHome services offer the inhabitant a greater value than just a set of electronic appliances present in the living space.

The main emphasis of the work of the eHome research group lies in developing a specification, configuration and deployment process (SCD-process) for eHome systems. This process is developed to introduce eHomes to the mass markets by breaking an important price-barrier to enter the market. The SCD-process dissolves the current expensive software *development* process for eHome systems, by replacing it with a low-cost software *configuration* process for the systems. The eHome systems and SCD-process are discussed in more detail in Chapter 3.

The basis for the SCD-process is the eHome model. This is a general meta-model describing common aspects for all eHomes. The eHome model instance is derived from the eHome model and describes the aspects of a particular eHome. The eHome model instance provides the necessary details to specify and configure an eHome system for the particular eHome. The eHome model instance is created and refined during the SCD-process and contains all the information necessary to deploy the desired eHome system into an ordinary home thus, transforming it into an eHome. The first topic of this thesis deals with the eHome model structure and transformations of the eHome model instance during the SCD-process (see sections 3.3 and 3.6).

The second topic of the thesis concentrates on tool support for the SCD-process. The eHome group uses a graph based Fujaba tool suite [Zün99] in the software development. Fujaba is a graph-rewriting system similar to PROGRES and can be regarded as a successor to PROGRES, because the Fujaba project was initiated in the late nineties by the people who developed PROGRES. But compared to PROGRES, Fujaba relies on the latest UML [RJB99] modelling techniques and produces by default a fully executable Java code generated from the UML models. Fujaba is used in the scope of this thesis to specify fully operational eHome model (see Chapter 3) for the eHomeConfigurator tool (see Chapter 4). The discussion why it is Fujaba and not PROGRES that is used is given in sections 2.3 and 6.1.

eHomeConfigurator is a an application largely developed in the framework of this master thesis, with an important role in all the phases of the SCD-process. The phases of the SCD-process can be derived from its name: specification, configuration, and deployment. According to the phases, the eHomeConfigurator tool is used to specify the eHome with its constructional ground plan, appliances, and the already existing services. During the configuration phase, the tool configures the eHome model instance according to the selected services, sub-services, corresponding software components, and appliances. After the configuration phase resulting in a complete configuration, the eHome model instance is ready for deployment. The complete configuration provides the services descriptions required by the eHomeConfigurator tool for deploying the corresponding software components onto the service gateway in the eHome.

The implementation of the eHomeConfigurator tool comprises the visualisation of eHome model instance's structure. The model works with activity buttons and menus for the models activities. There are also editors for different views on the model instance and its contexts, wizards to guide the user stepwise through more complex operations on the model. The development of the eHomeConfigurator tool introduces the so called translator implementation problem handled in Chapter 2. The translator problem lies in the implementation and maintenance overhead for translators used to transform and synchronize the structures of the eHome model instance with the structures needed for visualisation or on other technological platforms.

The third topic of this thesis focuses on solving the translator problem. The research on this problem includes the application of triple graph grammars [Sch94] in software development to specify bidirectional translators to synchronise different object models during runtime. This thesis develops a Fujaba-related generic translation and synchronisation framework called *Transformation Rules for Incremental Model Synchronization (TRIMoS)* (see chapters 6 and 7). The framework enables to replace the hand coded translators with visually specified rule sets.

The TRIMoS framework has currently shown a lot of potential for synchronisation of different object models. For example, the translator development problem[1] tackled in this thesis can be solved in a more generic way using TRIMoS framework. The future work on TRIMoS framework should have two main goals: firstly, the development and refinement of TRIMoS into a well defined and documented Java technology API; and secondly, implementation of the automatic activity invocation mechanisms inside the framework to propagate only the method calls[2] between the synchronized models.

In summary, it can be said that this thesis focuses firstly on the development of an appropriate model for eHomes and handling of its transformations during the SCD-process. Secondly, the development of tool support for this process is handled. And last but not least, the TRIMoS framework is developed to improve the eHomeConfigurator development, which might be applicable in the object-oriented software engineering in general.

1.1 Structure of the Work

The next chapter contains the motivation of this thesis, which describes two development scenarios. The first one illustrates the current development and maintenance process for the eHomeConfigurator tool and the second one describes the desired outcome of this thesis. Chapter 3 gives an overview of the first topic of this thesis.

[1]The problem is illustrated by two development scenarios in Chapter 2. The first scenario represents the currently used development steps for the tool in the framework of this master thesis and the second one presents the desired outcome of this master thesis, solved with the TRIMoS approach (see Chapter 6).

[2]The method calls are not related to the structures of the related models.

It introduces eHomes, eHome systems, the SCD-process, describes the technical solution and the structure of the developed eHome model. The life cycle of the eHome model instance is viewed from the perspective of the SCD-process.

Chapter 4 introduces the tool support for the SCD-process and the eHomeConfigurator project in general. The development of the eHomeConfigurator tool is discussed in more detail in the sections where the modules implemented in the scope of this thesis are handled. The discussion raises the development problem of translators in the eHomeConfigurator tool. The solution approach for this problem will be handled in the following chapters.

Firstly, the theoretical background for the solution approach is introduced in Chapter 5. The theoretical background covers the graphs, graph transformations, graph grammars and triple graph grammars (TGG). The TRIMoS approach solving the translator problem is based on triple graph grammars and is introduced in Chapter 6. TRIMoS is a synchronisation framework, which helps to replace the hand coded translators in eHomeConfigurator with translators specified by means of visual rules. Chapter 6 deals with the TRIMoS transformation rules and their differences from the classical TGG approach, but also the semantics of the rules. Chapter 7 focuses on the implementation of the TRIMoS framework and synchronisation mechanisms. This chapter discusses in detail the design of the framework, the object structures created during the runtime of the framework and the implementation of the interpretation of different TRIMoSrule object stereotypes. The adaptations of the JGraph API to be used with the TRIMoS framework are tackled upon in the same chapter.

The next chapters give an overview of the future work in Chapter 8 and related work in Chapter 9. Chapter 10 summarises the main findings of the research and implementation of the trimos approach. The thesis is finalized with appendix A containing an example TRIMoS rule set for environment editor of the eHomeConfigurator tool, a list of figures, bibliography and an index.

Chapter 2

Motivation

The motivation for this thesis is primarily to solve the current development problem of frequent eHome model changes. Every model change involves quite a severe and error-prone coding overhead to adapt the eHomeConfigurator for those changes. This thesis describes the eHome model and runtime transformations of the eHome model instance (see Section 3.3) in the eHomeConfigurator tool during the SCD-process. The execution of the SCD-process created the interest in developing the eHomeConfigurator. Another interesting aspect of the thesis for us is to explore new Fujaba-related possibilities, because this tool is an outstanding example supporting the model-driven development practices and is most extensively used when developing the eHomeConfigurator.

The Fujaba tool suite is used because the eHomeConfigurator project originates from a lab course [Nor03], where Fujaba as a tool was studied and the initial form of the eHome model and eHomeConfigurator was developed. It was a logical choice to continue using the Fujaba tool since the author of this thesis was one of the students participating in the lab course and it would be an enormous overhead to develop the project from scratch using a different selection of development tools.

During the eHomeConfigurator project, the underlying eHome model has been redesigned about eight times. Additionally, numerous minor adjustments have been made to the model structure in the course of the project. Every major design decision involves nearly all structural changes, which can be made in the UML model in general: adding, removing classes; adding, changing, removing relations; adding, changing, removing class attributes, and methods – see Figure 2.1. All these changes require manual programming to adapt the eHomeConfigurator tool for the changes made. To emphasize the primary goal of this thesis, which is to research more generic ways for the eHomeConfigurator development and to illustrate the problems related to the goal, there are two *development scenarios* presented in the next section.

2.1 Example Scenarios

The objective of the eHomeConfigurator development is to enable working with the eHome model and to create the surroundings which encapsulate the whole life cycle of the eHome model instance (see Chapter 4). The eHome model instance is trans-

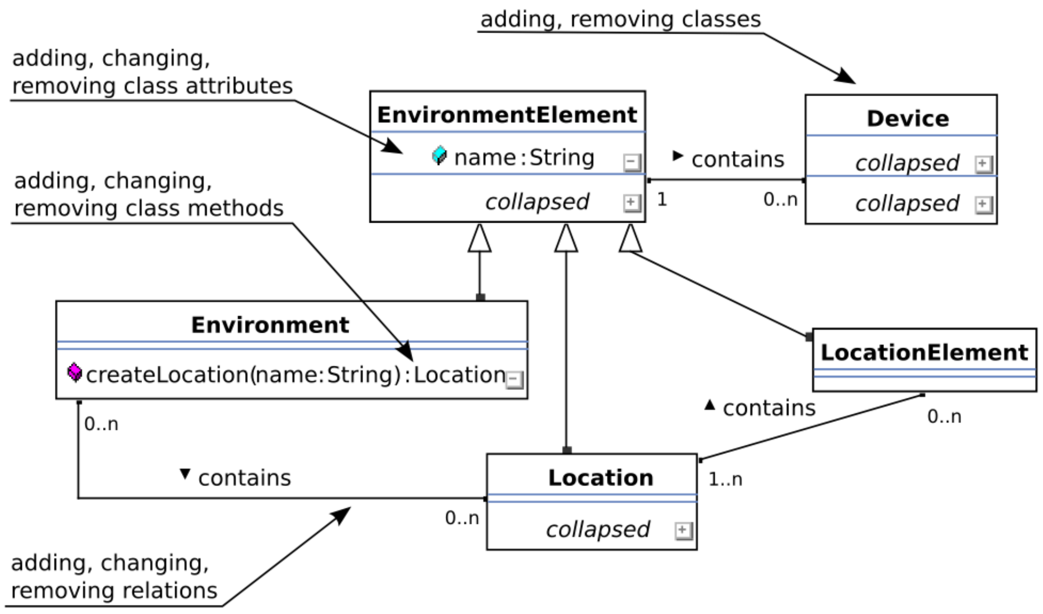


Figure 2.1: The possible changes in the eHome model

formed and refined during its life-cycle according to the specific eHome environment and requirements of the inhabitants of the specific eHome. All changes to the model instance structure during the SCD-process are performed using the eHomeConfigurator tool. The next two sub-sections present two alternative eHome configurator development scenarios.

Both scenarios have one common step, which consists of the creation of the eHome model using Fujaba (see Chapter 3). Fujaba allows us to describe the eHome model's static and dynamic structure by means of UML diagrams. After the model is designed, the corresponding Java code is generated with Fujaba. The eHome-Configurator is developed to encapsulate the code generated for the eHome model, which is the inner data structure for the eHomeConfigurator tool. This tool enables working with the eHome model instance serving as the user interface of the eHome model instance.

2.1.1 First development scenario for eHomeConfigurator

In the first case, the graphical user interface (GUI) of the eHomeConfigurator (see Figure 2.2) has to be developed manually. The tool development starts with programming the main frame with different menus and general tool bar buttons, also relating the menus and buttons with the activities of the eHome model. Every context (see Section 3.5) of the eHome model requires an editor panel programmed to work with the context concerned, and every editor requires activity buttons for the activities of the eHome model[1] corresponding to the context view reflected in

[1]Activity is a synonym for a method in object oriented model, i.e. activities are the methods defined in classes of the eHome model.

the editor. These buttons have to be placed and set to call the activities in the appropriate model context. For the input parameters of every activity, the input forms have to be programmed. In addition, the visualisation of the editor has to be implemented. In the case of visual editors, the translators visualizing the graph structure of the eHome model instance have to be programmed. These translators create the corresponding visualizing JGraph [Com] structures on the editor panel surface.

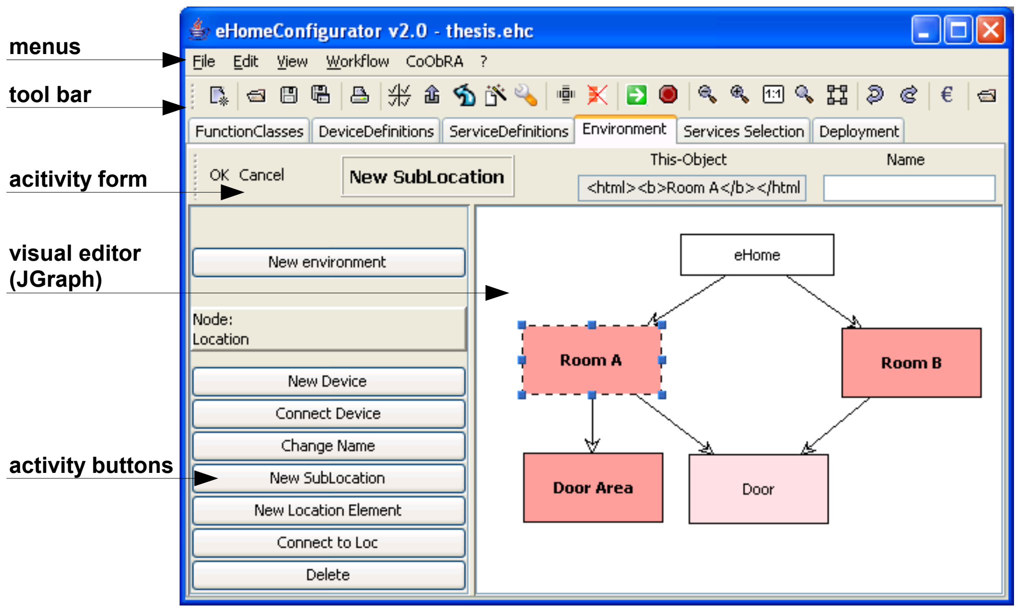


Figure 2.2: The environment editor of the eHomeConfigurator tool.

We consider as a simple illustrative example, a model change which states that locations can have sub-locations. A sub-location models a part of a larger location. For instance, a living-room can have a window area as its sub-location for advanced illumination control in the eHome. This change is quite trivial and is done in the environment context of the eHome model (see Figure 2.3). In the eHome model's UML class diagram, the `Location` class has to be provided with a self-relation on this class like in Figure 2.3. This is done with Fujaba. The self-relation has to be enabled through an activity in the model dynamics. Activities are specified by Fujaba *story diagrams*. There is a story diagram specified for the `Location` class (see Figure 2.4), which defines the method `createLocation(String name)` generated later into the Java code. The Java code is generated with Fujaba, as well. These development steps complement the model with sub-location concept and dynamics necessary to create the sub-locations.

The structure of the model has changed in the environment context, which corresponds to the building structure of the eHome. The environment editor of the eHomeConfigurator tool (see Figure 2.2) has to be adapted correspondingly. To visualize this structural change, the translator that translates the environment context

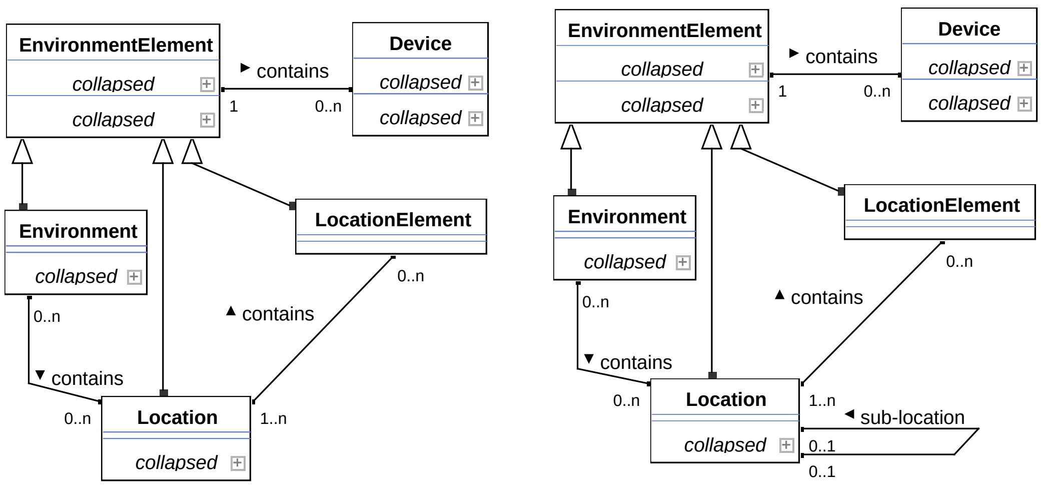


Figure 2.3: The environment context of the eHome model without (on the left) and with the sub-location concept (on the right).

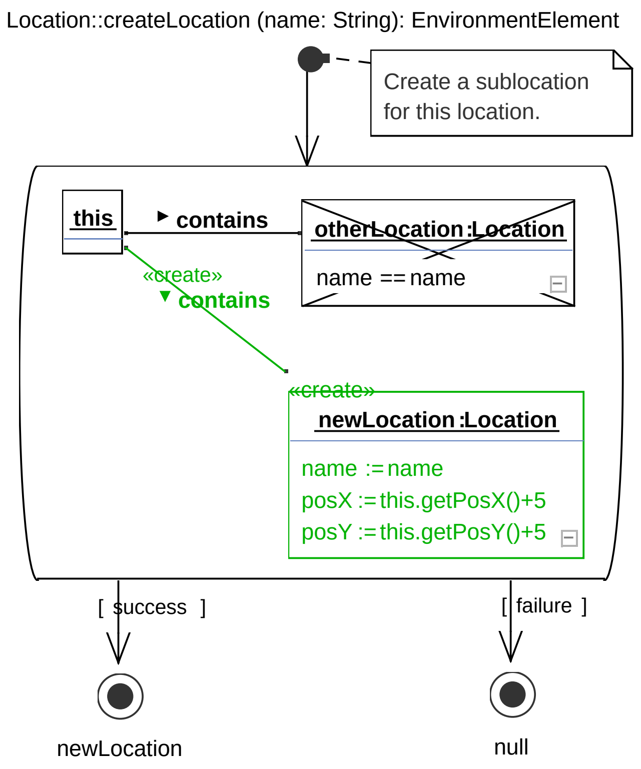


Figure 2.4: The activity for creating sub-location

of the model instance into the JGraph structures, has to be adapted to consider the new self-relation on the `Location` class.

In a situation, where Room A has the Door Area as its sub-location and the Door Area is important for the security service as an area under surveillance, the translator of the environment editor has to be re-programmed to visualize this situation on the JGraph panel of the editor. The adaptation of the translator is done manually. Manual programming is performed by providing the editor with a button and an input form for the new activity `createLocation(String name)` described in Figure 2.2. These steps finalize the change request for adding a sub-location concept into the eHome model.

2.1.2 Improved development scenario for eHomeConfigurator

Similarly, in the second case: after the eHome model has been developed, also the eHomeConfigurator has to be developed. However, in this case, the sub-location concept is needed to be introduced into the model with no manual programming. The whole user-interface with its different API technologies is initialised by a set of visual initialization rules resembling UML object diagrams. The runtime behaviour such as invocation of activities on the model instance and the visualization of model instance's graph-like structure, is also described by the set of visual rules. The visual rules are comprehensive and express the same object structure as described in Fujaba for the eHome model. Currently, nearly all this can be done using the *generic activity invocation mechanism* [NSSK05] (see Section 4.3.2), which instead of visual rules involves XML configuration files, and secondly, with the *TRIMoS framework*, which uses visual rules (see chapters 6 and 7). Both tools are extended or developed largely in the framework of this thesis.

The modifications on the eHomeConfigurator to adapt it to the illustrative new sub-location concept have only two simple steps. First, a TRIMoS rule set for the environment editor has to be complemented with one new rule describing the self-relation on the `Location` class (see Figure 2.5). The second step enables the button for the method creating a new sub-location. This is done by changing the XML configuration of the generic activity invocation mechanism. The example is given in Listing 2.1.

```
<ACTIVITY name="createLocation" label="New SubLocation">
    <TOOLTIP>
        Creates a new sub-location connected to the location.
    </TOOLTIP>
    <CONTEXTS>
        <ENVIRONMENT/>
    </CONTEXTS>
    <PARAM label="Name">
        <TOOLTIP>The name of the new sub-location.</TOOLTIP>
    </PARAM>
</ACTIVITY>
```

Listing 2.1: The XML configuration for a new method `createLocation(String name)`.

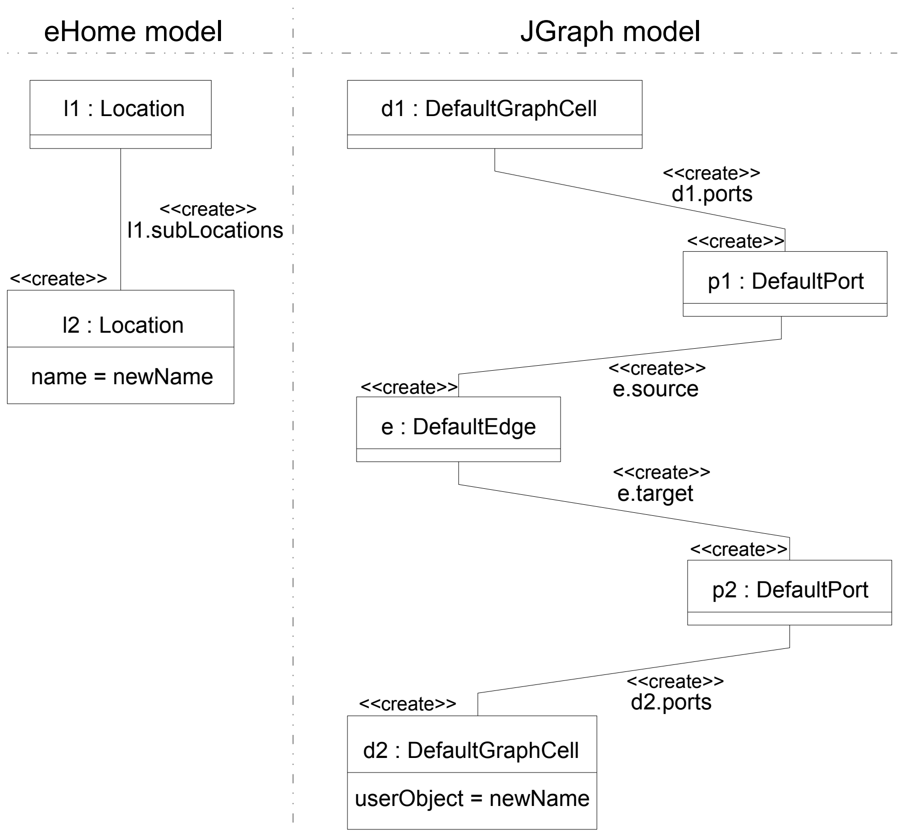


Figure 2.5: The TRIMoS rule to create a sub-location and corresponding JGraph structures. The rules follow the UML object diagram notion. The postcondition rule elements are denoted with stereotype `create`.

2.2 Motivation Conclusion

Considering these two development scenarios, it is obvious that in the first case, every change in the eHome model results in a considerable coding overhead. Procedures described in the second development scenario, on the other hand, are more desirable and easier to perform, especially from the perspective of software maintenance. This can be concluded from the following facts:

1. In the first case, the development and adaptations in the eHomeConfigurator have to be made by reading, refactoring and reprogramming the tools code. This means changing the code of the visual editor translators, the code for activity buttons, their input forms, and eHomeConfigurator menus. This process is error-prone, requires extensive debugging and testing.

2. In the second case, there are XML configuration and visual rule sets to be specified and maintained. Although the XML configuration is also changed by hand, it requires considerably smaller effort. The visual rules are comprehensible – easy to understand and to write down.

3. As to maintenance, it is easier to find and change visual rules than analyse the translator code and search for the code segments where the changes have to be made.

4. The hand-coded translators are not bidirectional. Since the TRIMoS framework is based on triple graph grammars (see Chapter 5), it works as bidirectional transformation framework.

5. The visual rules can also be changed during runtime of the tool in testing, because the TRIMoS framework has interpretative nature.

So far, the development of the TRIMoS system has shown a lot of potential for the future work. This thesis tackles the problem of frequent structural changes of the eHome model in the case of eHomeConfigurator maintenance as described in the development scenarios. The future work on the TRIMoS implementation may also concern the automatic activity invocation. The TRIMoS system could be changed in the way that the visual rules would have the expression power to relate method calls on different models to carry out related computations, which have no effect on the graph-like structure of the models[2]. Considering this development, the given generic activity invocation mechanism will be redundant. TRIMoS can also be extended and refined by further development to a production-level Java API having simpler interface and the defined procedures to operate it with. These further developments are beyond the scope of this thesis.

2.3 Solution Sketch

As indicated, there is a most urgent problem in the maintenance of the translator code which transform the eHome model structures to the JGraph structures. The solution for the translator problem is quite elegant. The structures of the eHome model and JGraph model have to be linked to follow the structural changes of one another reactively and automatically. This is done using the triple graph grammar based TRIMoS approach (see Chapter 6). A graph grammar itself defines a set of production rules on graphs, i.e. graph rewriting rules. The set of the graph rewriting rules defines a graph language – a finite or infinite set of graphs, which can be produced/ constructed according to the grammar of this language (see Chapter 5).

The idea of triple graph grammars is to bind two different graph grammars with the third graph grammar. We call these two graph grammars left- and right-hand side grammar. The connection of the left- and right-hand side grammars is done per graph rewriting rule on both sides via a third graph rewriting rule in between. In case a rewriting rule is executed on either of the left- or right-hand side graph, the third graph in between triggers the related graph rewriting rule on the other side.

The content of this thesis is to extend and develop the TRIMoS system, which was initially developed at the University of Kassel, Department of Computer Science and Electrical Engineering by the Research Group Software Engineering as a proof of concept for simple form of an interpreter for triple graph grammar rules. Before the beginning of the work on this thesis, the TRIMoS system was able to produce simple

[2]For example, statistical computations

tree-like structures related to each other. It was not able to delete the elements from the structures produced or keep them in sync in the perspective of deletion or more complex graph structures. The reasoning behind the usage and development of TRIMoS and not using the PROGRES system developed in our chair is the following: the eHomeConfigurator uses the Java programming language specific visualisation technologies not supported and enabled by PROGRES. We have situations where the visual actions on user interface imply the eHome model instance's structural changes beneath the user interface. For example, if device D on the eHomeConfigurator display is moved from room A to room B, the link between the corresponding `Device` object D and `Location` object A has to be destroyed in the structure of the eHome model instance and created between the `Device` object D and `Location` object B.

The usage of the TRIMoS system implies that instead of the hand-coded translators, there will be one set of TRIMoS rules for every visual editor. These rules are edited with a simple TRIMoS editor which visualises one graph rewriting rule from the left-and right-hand side graph grammar (see Figure 2.5) and enables the developer to create, examine, or manipulate these two rule sides. The third graph grammar rule in between is hidden and handled automatically by the TRIMoS system during runtime.

If the change occurs in the eHome model, only the set of rules belonging to the visual editor addressing the context of the eHome model where the change occurred has to be changed. In the second illustrative development scenario, where a sublocation concept was introduced, there has to be a new rule (the rule is shown on Figure 2.5) added to the rule set defining the environment editor of the eHomeConfigurator tool (see Figure 2.2). This kind of an approach has smaller development overhead, is more comprehensive, and less error-prone.

2.4 Summary

This chapter gave an overview of the motivations behind this thesis. We introduced an eHomeConfigurator development related problem, as well as two development scenarios, where the second scenario is the desired outcome of this thesis and presents development techniques with minimal coding overhead. In the next chapters, we will give an overview of the SCD-process, the eHome model and the eHomeConfigurator development. We will also introduce the theory behind the TRIMoS solution and the approach itself.

Chapter 3

The eHome Model and the SCD-process

The eHome Group located at RWTH Aachen University, Department of Computer Science 3 is active in the research field dealing with home automation, smart homes also known as *eHomes*. The eHomes are living spaces equipped with computer controlled electronic appliances and services combining the functionalities of those appliances. These services offer the inhabitant an additional value which is greater than just a plain set of devices and the sum of their functionalities. The term *eHome system* denotes a computer system the presence of which at home transforms it into an eHome. The eHome system consists of all the hardware and software required to provide eHome services in the home environment.

3.1 The eHome System

Figure 3.1 illustrates the structure of the eHome system. The eHome system comprises all the hardware, software and supporting systems for providing the smart home environment with its services for the inhabitants (multiple users). For example, eHome services may cover the fields of security, comfort or infotainment. According to Figure 3.1, an eHome system has three levels:

1. the hardware level contains appliances such as cameras, sensors for movement or temperature, lamps, heater systems, media devices, the local and remote communication devices to interface the inhabitant with the eHome system (computers, PDAs, mobile phones, etc.), but also the *residential gateway* housing the service gateway and eHome service software.

2. the software level includes the eHome service software, the service gateway as the service middleware for eHome services running on the residential gateway, the client software running on the communication and interface devices.

3. the supporting systems deal with services provided by the service providers to support the eHome with additional features, information, for example weather or traffic information, news, digital media, etc.

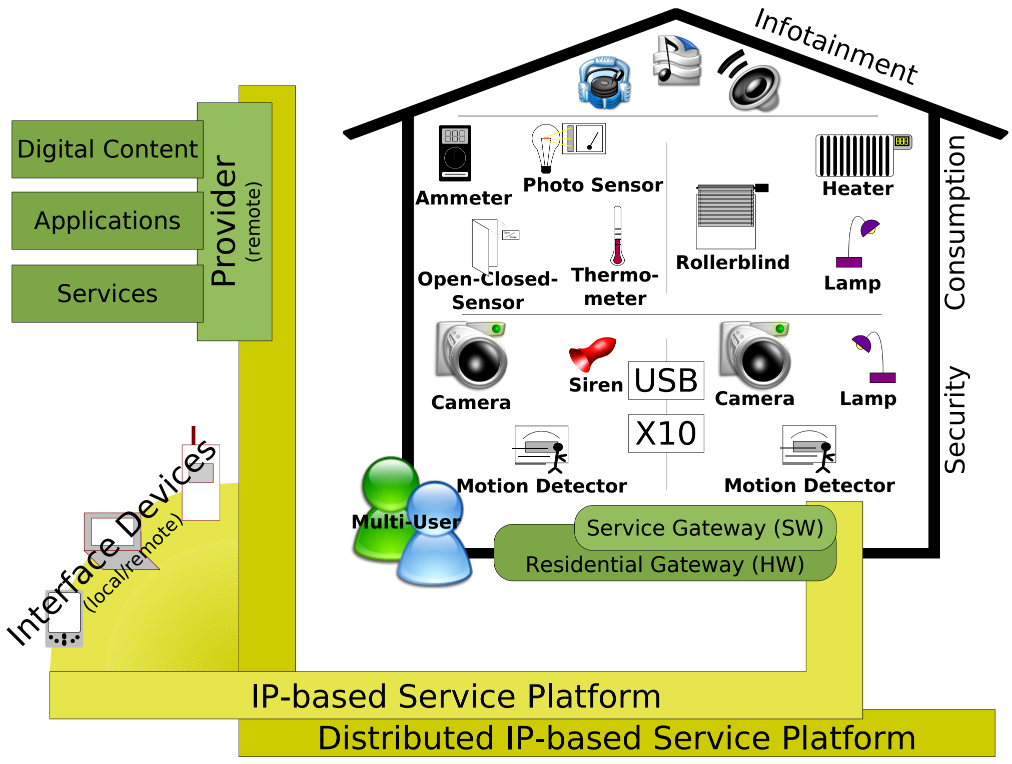


Figure 3.1: The structure of the eHome system

The eHome research group deals with the eHome system related problems, especially the development and architecture of these systems but also with the related business processes. One of the key problems up to now is how to introduce eHomes to the masses. The research on this problem is embodied into the work on the Specification, Configuration and Deployment process (SCD-process) for eHome systems. This thesis contributes to the research on the SCD-process, dealing more specifically with the tool support for this process and with the design and SCD-process related transformations of the underlying eHome model.

The next sections of this chapter and the next chapters cover the SCD-process and its enabling factors like the eHome model (see Section 3.3) and tool support (see Chapter 4). In this chapter, we will first introduce the SCD-process. Secondly, we will focus on the eHome model, its technological solution, structure, and finally we will deal with the transformations of the model instance during its life cycle in the SCD-process. The tool support for the SCD-process will be addressed in Chapter 4.

3.2 The SCD-process for eHome Systems

The reason for developing the *Specification, Configuration and Deployment process (SCD-process) for eHome Systems* is to establish a low cost process to introduce eHomes to the mass-market. The main obstacle preventing eHomes from becoming common in the welfare society is the relatively high price of the software driving the eHome since it needs to be developed or adapted for every particular eHome. Despite the fact that the appliances used in smart homes are getting constantly cheaper, very little research has been done on the subject of eHomes, except for a few enthusiastic development projects [inH05, T-C05]. The main reason for this is the large amount of coding work needed to complete this kind of project.

The idea of the SCD-process is to establish an iterative chain of procedural techniques to automate the creation of the eHome as much as possible. This means that we are looking for ways to automate and support the process of specifying, configuring and deploying an eHome system into the normal home - transforming the regular home into an eHome. This is achieved by means of tool support, reuse and mere configuration of software components for providing eHome services. It is essential that the shift from the normal home to an eHome did not involve developing the software but merely configuring and deploying it. Furthermore, the configuration and deployment of the given components must also be done automatically. As the result, the specification of the home environment, selection of the desired eHome services and installation of necessary appliances are the only activities performed manually.

As the name of the SCD-process indicates, the process consists of the following *phases*:

1. Specification of the eHome environment and necessary services. During this phase, the architectural information about the eHome is captured – how the rooms in the home are located and connected with the different location elements such as doors and windows. The given appliances and their location in the home environment is described – in which rooms or on which location elements the devices are positioned. The already existing eHome services are also identified – when modifying the configuration of the eHome, the already existing eHome services have to be specified. Whenever new eHome services are needed, they are selected and added to the specification. Only top-level services are selected.

   Along with the eHome environment, the services used later in the eHome environment, plus the required devices and functionalities need to be defined and specified beforehand, as well. For more information about the specification phase see sections 3.6.2.

2. Automatic configuration of the selected services. The services selected in the specification phase are automatically configured. This means that the necessary devices still missing from home are added to the configuration. Likewise, the required sub-services that are missing are selected to meet the functional requirements of the selected services. For example, if the lighting service needs at least one lamp per room and one switch to control the lamp, these devices are added to the configuration. Furthermore, the corresponding driver component services for the lamp and switch controllers are added to the configuration. For more information about the configuration phase, see Section 3.6.3.

3. Deployment of the service configuration onto the service gateway in the eHome. The software components specified and configured during the first two phases

are deployed automatically onto the service gateway residing in the eHome. The software components are also initialized properly and launched automatically. For more information about the deployment phase, see Section 3.6.2.

3.3 The eHome Model

The whole SCD-process is strongly related to the *eHome model.* The eHome model is a meta model describing real eHomes, depicting aspects relevant for the SCD-process, eHome systems and the runtime of these systems. In other words, the eHome model is a general model defining a language to express the specific eHomes. This model contains the information required to model the specific home environment, appliances, services, etc. The model describing one particular eHome is called the *eHome model instance.*

We chose this notion to call the general model the *eHome model* and the specific model derived from it the *eHome model instance*, since the eHome model is an object-oriented model consisting of classes, their relations and the dynamics of the classes. The eHome model instance, on the other hand, is an object model constructed and maintained for one particular eHome during the runtime of tools supporting the SCD-process and runtime of the eHome system (for more information about tool support, see Section 4.1). We use the object-oriented terminology by calling the object model derived from the eHome model class structures the eHome model instance since this is literally a structure of the objects, i.e. the instances of the classes. The instance of the eHome model represents one particular eHome with all its specific details and configuration and is the basis for the SCD-process.

The model instance is actively involved during the runtime of the eHome system as it provides information about the home environment for eHome services. This enables the development of context aware eHome services which need the real-time information about the inhabitant locality, and the states of other services, or appliances. In other words, the eHome model instance is also a communication medium for eHome services.

The author of this thesis has been the main contributor to the eHome model development and its embedding into the SCD-process in the past two years, as well as a senior developer of the tools supporting the work with the eHome model instance and the SCD-process. The next sections give an overview of the development of the eHome model, its contents and its function in the SCD-process. The tool support for the process is handled in Chapter 4.

3.4 Technical Solution for the eHome Model

The eHome model is developed using the Fujaba tool suite [Zün99]. The Fujaba tool suite enables the development of an object-oriented model and the generation of a fully executable Java code for the model. The model has its statics and dynamics. The static structure of the model is designed using a UML class diagram describing the classes of the model and their relations to each other. During the runtime, the objects of the model classes are created according to the structure of the class diagram.

There have been several other versions of the eHome model before the current one. The first model was designed using the ontology web language (OWL) and the tools supporting the SCD-process were therefore developed using knowledge base specific technologies [KNS04]. The reason why there was a change in technology to use object-oriented models and the Fujaba tool instead, was to enable a bigger variety of choices for development of the SCD-process supporting tools. The object-oriented model is easier to operate with during the tool support software development and has proven to be the right technological choice.

The dynamics or runtime behaviour of the eHome model is described by means of Fujaba *story diagrams* [FNTZ98]. The Fujaba story diagrams are a combination of the UML activity diagrams and UML collaboration diagrams. The story diagrams can be considered as activity diagrams having their own activity type called *story-activity* which enables describing the interaction of objects during the operational sequence of the program or the time-flow of the program execution. In terms of methods of the classes it means that every method in the class is described by a story diagram. The story diagrams enable the full expression power of the Java programming language.

By describing the eHome model using UML diagrams, no manual coding is needed. After designing the model in Fujaba, the model's Java code is generated with the same tool. The code is compilable and error free from the human error perspective. The only problems appearing in code are due to incorrect modelling of the classes or their methods.

The structure of the eHome model is quite complex. It includes six different contexts starting with the building structure of the eHome and ending with the environment information for services during runtime. The next section will introduce each of the six contexts together with their static structure. The complete model will not be visualized as one unit due to its complexity.

3.5 Contents of the eHome Model

The eHome model has six different contexts. These contexts provide all the information needed to support the SCD-process and the runtime of the eHome system. The contexts are the following:

1. The *functionality context.* This context covers the functionalities of the eHome services, as well as the appliances. See Section 3.5.1.

2. The *device definition context.* This context describes the devices and their properties. See Section 3.5.2

3. The *environment context.* This context includes information about the building structure of the eHome, the given appliances and eHome services. See Section 3.5.3.

4. The *service context.* This context represents the services, their functional requirements and presence in the eHome. See Section 3.5.4

5. The *service instance context.* The context expresses the runtime configuration of services. See Section 3.5.5

6. The *inhabitant context.* This is a slice of the model which contains information about the inhabitant in the eHome. See Section 3.5.6.

3.5.1 Functionality Context

The smallest but one of the most important contexts of the eHome model is the part describing functionalities of the services, as well as the appliances (see Figure 3.2). The functionalities are described by means of `Function` class which has a self-relation describing that one functionality can be refined by another one. For example, a detection functionality can be refined by the functionalities: movement detection, smoke detection, glass breakage detection, etc. (see example in Section 3.6.2).

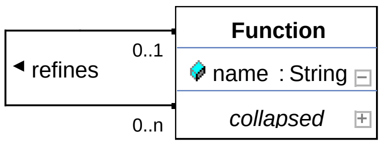


Figure 3.2: The functionalities context of the eHome model

The self-relation of the `Function` class results in the tree structure of objects from this class. Functions are defined by their names and the refinement relation of the functionalities should be used in such a way that the most general functionality is the root of the tree and the leaves are the most specific functionalities (see Figure 3.10 in Section 3.6.2).

3.5.2 Device Definition Context

The devices used in the environment specification phase of the SCD-process are predefined. In the case of a device, or several devices of the same type specified in the eHome environment (see Section 3.5.3), there must be a device definition describing the device, including the manufacturer information, the name, and attributes. This device is specified in the environment as an instance of the device definition with a specific hardware address (IP address, house code for X10, USB address, etc.) and recognizable name. In figure 3.3, the device definition is represented by `DeviceDe-finition` class. The device as the instance of its definition is represented by the `Device` class and the attributes by the `Attribute` class.

In earlier versions of the eHome model the devices were regarded as the aggregate of functionalities. This was needed for meeting the requirements for eHome services. Although the essential part of the devices is still described by means of functionalities, it is done by using the software component (see Section 3.5.4). This is because the hardware itself and the physical connections have no significant role in the software configuration step during the SCD-process since in the case of the higher level software and user interfaces, the hardware is visible only through hardware driver components, their attributes and access methods. The communication between appliances is also performed over lower level drivers and communication pro-

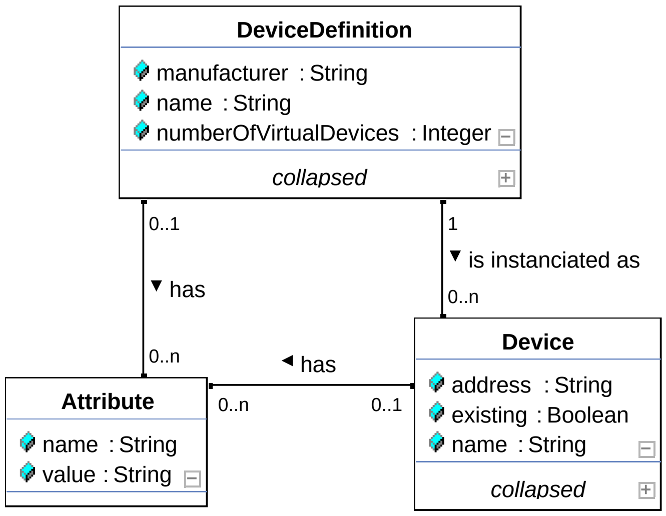


Figure 3.3: The device definition context of the eHome model

tocols, so physical connections are not relevant for the SCD-process (see sections 3.2 and 3.6.3).

3.5.3 Environment Context

The environment context of the eHome models the location information according to the ground plan of the home, the given appliances and the given eHome services. The environment context can provide a set of different environments which are connected via locations or location elements for the eHome. For example, the house can be connected with an external garage, by a hall, or just a door. Since there is also a sub-location concept implemented, it means that, for example, a floor can have its rooms as sub-locations or the room can have different service-related areas near the windows, doors or a TV set. This kind of modelling freedom and generality gives us a mechanism powerful enough to express any kind of architectural designs. We mostly concentrate on floor plans, but we can also express 3D designs using the connective and descriptive location elements.

The environment context has the `EnvironmentElement` class as a super-class for any other class describing location information of the home (see Figure 3.4). `EnvironmentElement` aggregates the common features of the classes describing the home environment. Since the classes `Environment`, `Location`, and `LocationElement` inherit from the `EnvironmentElement` class, they have a relation to the `Device` class. This means that every element in the environment context can have appliances related to them. For example, in the living room, there can be a lamp, a media set with speakers and a LCD screen, controlling switches, etc.; the door in the room can have a movement detector attached to it; and the window can have a class-brake detector attached to it.

The (see Figure 3.4) `Environment` class depicts the environment which is a logical independent entity of the living space with its locations, appliances and eHome services, a typical eHome. The `Location` class corresponds to one logical location entity in the eHome environment, for example, a floor, a room or a part of the room. Locations are important for services because the location is a smallest unit covered by a service according to our service selection strategies (see sections 3.5.4 and 3.6.2).

The `Location` class has a self-relation `sub-location` for the sub-location concept. This relation defines a hierarchical location graph expressing logical substructures in the locations themselves. This graph must be a directed acyclic graph (DAG). It makes no sense to have a transitive cycle, where one location is simultaneously a parent and a sub-location of another location. For example, the first floor contains a living room as a sub-location and the living room has a window area as its sub-location. Whereas, if the window area has the first floor as its sub-location, the first floor would also be the sub-location of the living room which in turn is a sub-location of the first floor. This would be a contradiction in the sub-location concept we have.

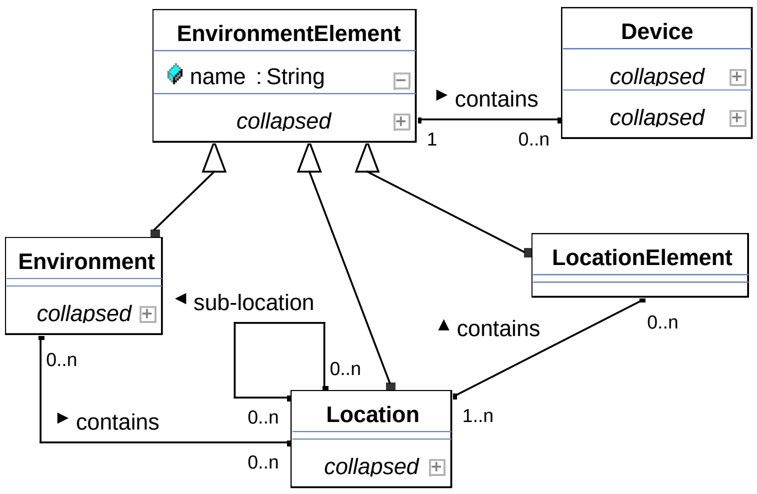


Figure 3.4: The environment context of the eHome model

The `LocationElement` class describes one logical component of a location - windows, doors, or why not an elevator shaft. The `LocationElement` objects like doors can be shared between locations so that logical connections appear. Thus, environments, locations, and connecting location elements can form a complex graph that represents the logical connections[1] in an architectural design. An example of the environment is depicted in Figure 3.9.

3.5.4 Service Context

The service context of the eHome model represents eHome service descriptions, or more accurately, the definitions of eHome services. It is crucial for the SCD-process that the eHome service software components configured during the automatic configuration phase are modelled beforehand since the automatic configuration relies

[1]The connections describe the relations between the entities in the architectural design thus, modelling also the three dimensional relations in the building.

on the abstract description of the eHome service software. The Service context does not include the runtime configuration of the selected services, which has to be deployed onto the service gateway in the eHome during the deployment phase of the SCD-process.

Figure 3.5 outlines the service context of the model. The service itself is modelled by the `Service` class which contains information such as the id, name, type, and description of the service, but also the information on the resource URI of the corresponding software component installed during deployment phase of the SCD-process. The `Service` class is an abstract description of the corresponding software component executed during the runtime of the eHome system.

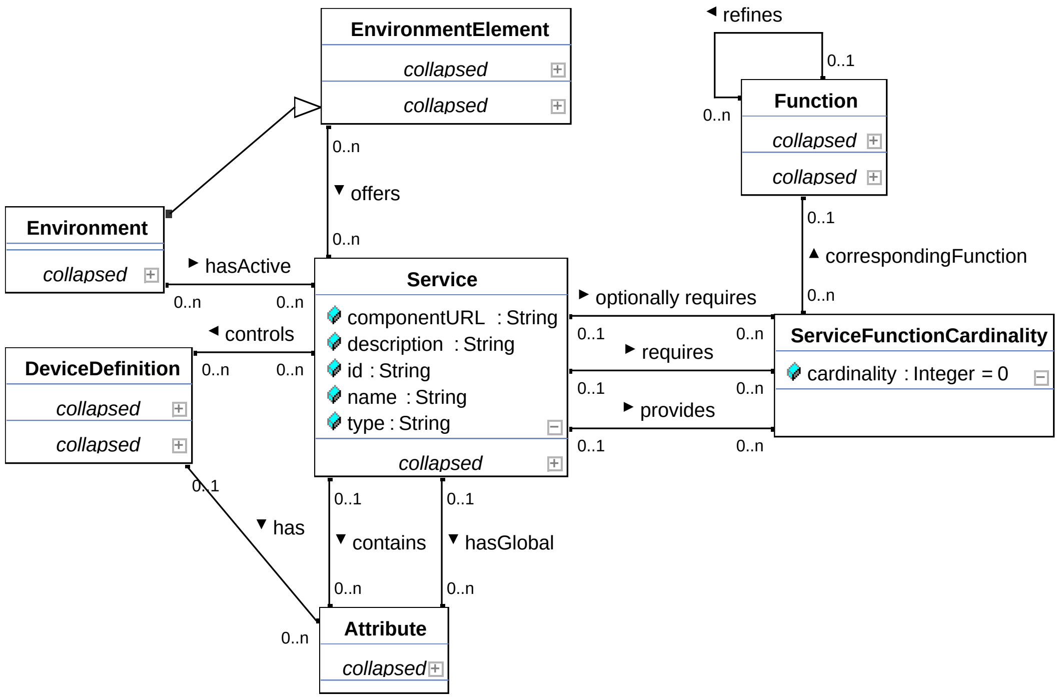


Figure 3.5: The service context of the eHome model

The essential part of the service description is specified by functionalities. The `Service` class has three indirect relations to the `Function` class over the `Service-FunctionCardinality` class. The service is described by the functionalities it provides, requires, and optionally requires. As mentioned in Section 3.5.2, the functionalities of the devices are considered to be a part of driver services. `Service-FunctionCardinality` class is used since the configuration step of the SCD-process requires cardinalities on the functional requirements of the service (for more information and examples see sections 3.2 and 3.6.3).

The functionalities give the SCD-process a dynamic composition and dependency resolution during the automatic configuration step forming a certain abstraction layer for service composition. The services may require functionalities in order to combine them and offer this combination or to be able to provide additional functionalities.

The `Service` class has two relations to the `Attribute` class. These two relations model the global attributes of the class – general information for the service, not dependent on a specific eHome; and specific attributes – attributes which are set for the runtime instance of the service (see Section 3.5.5).

Services are related to the environment information. The relation between `EnvironmentElement` and `Service` class implies that the corresponding environment element offers this service for the eHome inhabitants. This relation is typically used for locations. It means that the most straight-forward strategy[2] is to define in which rooms the specific services are provided for inhabitants. The links between locations and the service are created when the service is selected during the specification phase of the SCD-process. An example of this is provided in Figure 3.15, where the `Music Follows Person` service is available in the location `Living-room`.

Services also have a relation to the `Environment` class which indicates whether the selected service for the home is configured during the automatic configuration step or not. It is also stated whether the environment `hasActive` services and not active services. Not active services can be configured and started later according to the home-owner's needs.

The `Service` and `DeviceDefinition` class are related, as well, as the devices are controlled by the software. This is the case for services which in fact are driver components of the devices. The device driver component provides the other services or the end-user with functionalities attributive to the controlled device. This reasoning concludes with the fact that devices as such have no great significance for the automatic configuration phase of the SCD-process. An example of a device driver service is presented in Figure 3.12 or 3.18.

3.5.5 Service Instance Context

In the service context, the service is modelled by its functional dependencies. The model instance context models the runtime configuration of eHome services in the eHome system. During the configuration step of the SCD-process, the structures of the eHome model instance corresponding to this context are filled automatically (see Section 3.6.3). The service instance context models the service configuration during runtime of the eHome system. To give a better overview, the context is visualized on two figures 3.6 and 3.7.

According to Figure 3.6 the `ServiceObject` class models the service instance and has a relation to the `Service` class indicating which service is instantiated as the service object. The idea of the service instantiation is to assign a `ServiceObject` with its specific configuration to every `EnvironmentElement` which provides the selected `Service`. Thus, Figure 3.7 shows a relation between the `ServiceObject` and the `EnvironmentElement class`. This relation `is in` corresponds to the relation `offers` in Figure 3.5.

Similarly, the `controls` relation between the `ServiceObject` and `Device` classes (see Figure 3.7) corresponds to the `controls` relation between the `Service` and `DeviceDefinition` classes (see Figure 3.5). The `controls` relation described in the service instance context models the situation where a service instance can control

[2]The strategy used by our eHome research group.

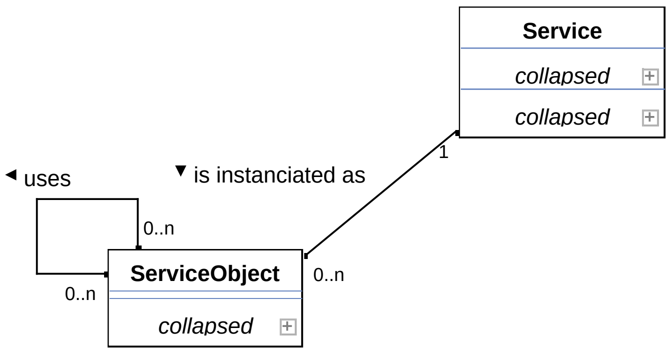


Figure 3.6: The relation between service and service instance of the eHome model

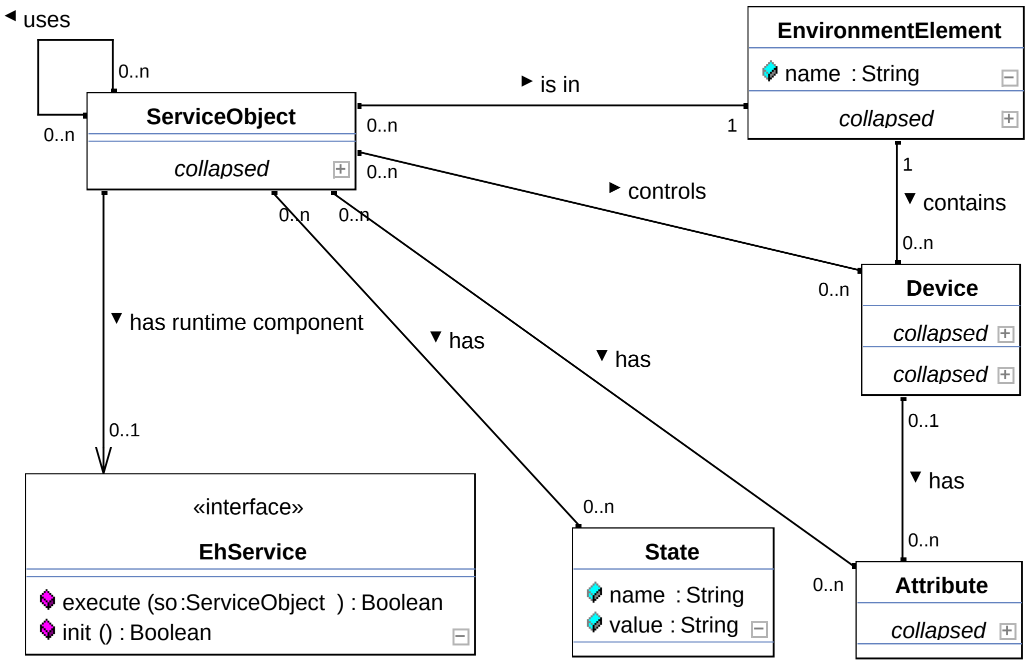


Figure 3.7: The service instance context of the eHome model

specific devices, using its specific configuration. This relation does not reflect the general relation between the device driver service component and the device type it controls.

The service instance can have states described by the `State` class. The states describe runtime attributes of the service, which can be changed during runtime. For example, in the case of the device driver service component, the state can also describe a state of the controlled device – on/off.

The number and types of the attributes of the service instance itself are described by the service it instantiates. The values are set separately for each service object. For example, by the e-mail notification service the attribute e-mail address is described with arbitrary default value. In the case of the instance of this service, the e-mail address has to be provided during the configuration step of the SCD-process.

The service instance object in the model has also a reference to its implementing software component. This is realized via the relation `has runtime component` with the `EhService` interface. It means that every software component implementing an eHome service modelled in the eHome model instance also has to implement this interface. This relation enables the model to be aware of the implementing software components and also to call methods upon them according to the `EhService` interface (more information about this in Section 3.6.5).

But the most important relation in the service instance context for the configuration step in the SCD-process is the self-relation `uses` of the `ServiceObject` class. This relation expresses the usage relation between the services during runtime. In the service context, there is no explicit relation between the services. The reason for that is to use the functional abstraction layer[3] around the services for the service composition to enable straightforward automatic configuration of the services during the SCD-process.

As mentioned before, the service instance context is filled with objects and data during the configuration step of the SCD-process. The tools supporting the automatic configuration (see Chapter 4) take into account the selected services, their functional requirements, and then compose a suitable set of services to enable the selected services in eHome. The suitable set is composed as the structure of the service instances where the usage relations between the services are expressed explicitly. The composition uses the indirect `provides`, `requires`, and `optionally requires` relations between the `Service` class and the `Function` class (see Figure 3.5). These relations are used to construct a usage and dependency graph of the service instances, expressing the runtime structure and configuration of the eHome services in the eHome system. For more information about the configuration step and eHome model see Section 3.6.3.

3.5.6 Inhabitant Context

The inhabitant context models the person-related information in eHomes. The person related information is irrelevant for the SCD-process, because the process involves the specification, configuration and deployment of eHome systems in general. The emphasis of the process lies on the resource and dependency resolution. Nevertheless, information about the person plays an essential role during the runtime of the eHome system, for example, when communicating information about the inhabitant's location to the services. Person related information is also important for other eHome processes like business processes [Kir05] and possible transportation of services between different eHome environments[4]. For those reasons the person related information is modelled in the eHome model.

Figure 3.8 presents the super class `Person` of the classes `Inhabitant` and `Customer`, modelling the most important properties of the person under consideration.

[3]The services are described as entities having functional imports and exports, so far these functions have been specified as semantic labels on the conceptual abstraction level.

[4]The transportation of services is studied under the latest research topic on virtual eHomes in the eHome group.

The `Customer` class is important for eHome related business processes which is a topic beyond the scope of this thesis, but is thoroughly handled in [Kir05]. The `Inhabitant` class models the inhabitant of the eHome. The relation `is In` between the classes `Inhabitant` and `Location` represents the location information of the inhabitant describing where the person is currently located (for example, if mother is in the kitchen). This information is needed for the services which require information about the location of the people in the house.

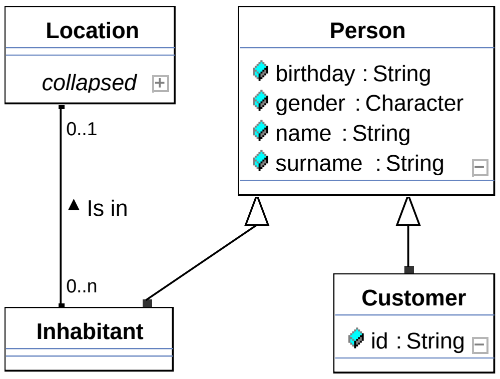


Figure 3.8: The inhabitant context of the eHome model

3.6 The Life Cycle of the eHome Model Instance

This section gives an overview of the life cycle of the eHome model instance in four stages throughout the SCD-process and runtime of the eHome system. We describe how the eHome model instance is formed and transformed during the SCD-process and which model contexts are involved in specification, configuration, and deployment phases of the SCD-process; but also how the model instance is changed and used during the runtime of the eHome system.

As mentioned, the eHome model is designed using Fujaba tool, but this model is actually a meta model for specific homes. This particular eHome is modelled by using the eHome model instance. This instance is constructed and modified during the SCD-process and also during the runtime of the eHome system. The life cycle of the eHome model is iterative due to the iterative nature of the SCD-process but also because nearly every software engineering process is iterative. Therefore, the life cycle of the eHome model instance can return to its earlier stages according to the phases of the SCD-process and runtime of the eHome system.

To give a better idea of the changes throughout of the life cycle of the eHome model instance during the SCD-process and runtime of the eHome system, we introduce an illustrative example. This example consists of a simple scenario of the application of the SCD-process for a small apartment. This home is converted into an eHome by deploying one eHome service into its living space. The next subsections give the illustrative scenario and stages of the life cycle of the eHome model instance.

3.6.1 Illustrative Example of the SCD-process Application Case

As an example of the SCD-process, we have a small apartment, consisting of a hall, a bathroom, a living-room, and a bedroom. The living-room has a small kitchen corner for cooking (see Figure 3.9). This is a typical bachelor apartment for one person. The owner of this home is a bachelor who loves music and wants to hear the music everywhere in his apartment, while taking a bath, for instance. Nearly all of his music collection is on the PC. He can buy an expensive sound system connected to the computer and supporting at least three pairs of speakers. However, he would expect his sound system to be a little smarter than that. In this case, he would like to have the following feature: if a particular person listens to the music, the music played at the time follows the person from room to room.

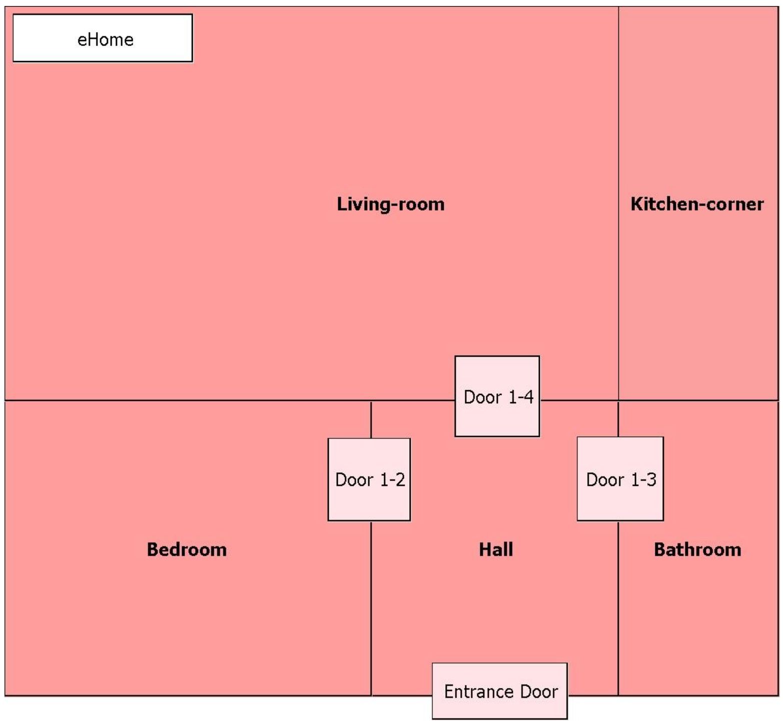


Figure 3.9: The floor plan of the example apartment.

In order to meet this additional requirement and solve the music coverage problem, it is possible to install a few additional devices and one corresponding eHome service for the inhabitant, transforming his living space into an eHome. The required service is called *Music Follows Person service.* This service routes the music from one room to another, particularly to the speaker systems in these rooms. The Music Follows Person service is activated by the person in one of the rooms by switching the corresponding switch. The preselected music stream starts to play in this room. When the person leaves the room, the music stops playing in the room and is played in the room the person enters. The movements of the person are tracked by a person detection system.

Using the SCD-process and the corresponding tool support for the process, the described eHome service is selected and deployed into the eHome. The example of the music loving bachelor and Music Follows Person service is used to illustrate the following sections and to give a better idea of the changes in the eHome model instance during the SCD-process.

At this point, the example only consists of the wish of the customer and the floor plan of his apartment. In the next sections, this example is refined and extended relating it to the different model contexts and the SCD-process. The next subsections correspond to the four stages, which form the life cycle of the eHome model instance.

3.6.2 Changes in the eHome Model Instance Structure During the Specification Phase

The specification phase of the SCD-process addresses most of the model contexts, in particular the functionalities, device definitions, devices, the environment, service, and inhabitant contexts. As the name of the phase states, all the necessary data for the next process phases is gathered during specification as user or third party input.

The specification process involves several parties. It is even possible that a home owner does not participate in the process, except by submitting an order for the desired services for the eHome system. In this case, the third parties like the corresponding service providers perform the whole SCD-process.

The specification phase can be viewed on two levels:

1. the specification of the common bearings of eHomes and eHome systems like device definitions and services used in numerous eHomes;

2. the specification of the characteristics of the particular eHome like its floor plan, existing devices, and selected services.

We will first address the general part of the specification and then move on by describing the specification of a particular eHome.

Specification of Common Aspects for eHomes

Considering the example where the Music Follows Person serviceis needed, it is necessary to specify the functionalities, devices, and services before it is possible to offer the services for the customer. Thus, the specification starts with addressing the functionalities.

The required set of functionalities have to be defined beforehand. The corresponding model context (see Section 3.5.1) implies that the functionalities have to be organized into a tree structure. The tree expresses the refinement hierarchy of the functionalities, in which the children of a functionality refine the parent. For example, detection can be refined by movement detection and smoke detection. Figure 3.10 outlines a tree of functions which are needed to describe the Music Follows Person service and the required sub services described later in this section. The functionality tree in the figure consists of three main branches: `detection`, `music follows person`, and `drive`. The branches are refined where necessary, for example the detection function is refined by detecting `person`, `movement`, and `switching`.

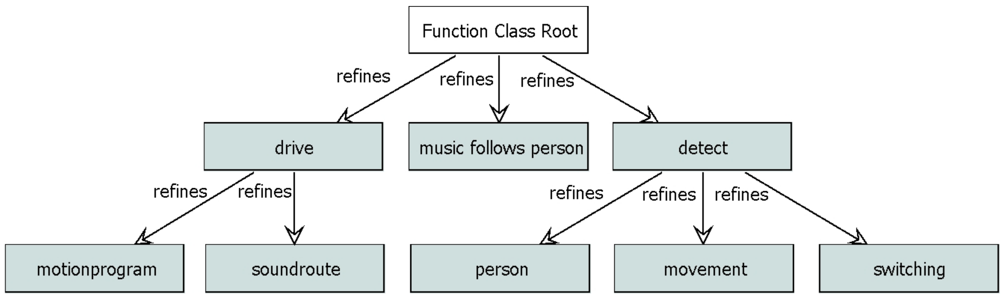


Figure 3.10: The necessary functionalities for the Music Follows Person service.

Likewise, the devices used by the services have to be specified. For example, Figure 3.11 describes a simple web-cam definition in the device definition context of the eHome model. This web-cam can be used for getting a video stream or pictures from a room used for movement detection or person recognition. The description of the device is rather simple. It only describes a device as an entity with necessary attributes, for example, an USB camera number, IP address, or some other addressing attribute.

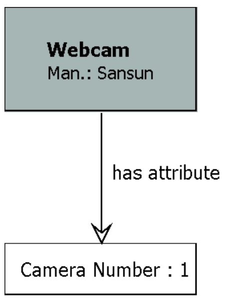


Figure 3.11: The definition of a web-cam used by the person tracking system.

The device definition is as simplified as it is since the information on the functionalities of the devices is described elsewhere, namely in the service context. For example, the service encapsulating the device driver of the named web-cam is presented in Figure 3.12. This service aggregates the web-cam functionalities for providing a picture stream and information about the movement in the room, providing an aggregated functionality `motionprogram`. This functionality can be used by other services or software. The number in the brackets behind the `provides` relation represents the cardinality of the functionality. In this case “-1” indicates that this functionality as a resource can be used by an indefinite number of other services. The cardinalities are discussed more thoroughly in Section 3.6.3.

Apparently the services can be developed and defined so that they wrap the functionalities of the devices. But services can also be developed to use other services and offer an additional value for other services or for eHome inhabitants directly. Figure 3.13 depicts a `Person Detector` service for tracking a person from room to room. This service definition requires the described `motionprogram` functionality

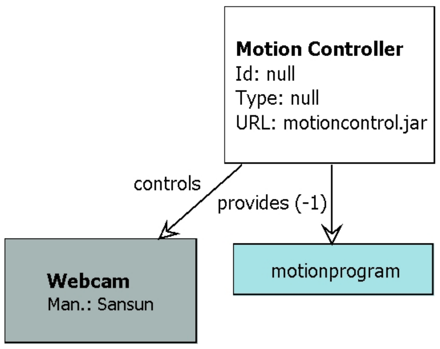


Figure 3.12: The specification of the Motion Controller service, which uses a webcam to detect movement.

and `switching` functionality[5]. These functionalities are combined by this service to offer `person` detection functionality. The `person` detection functionality can be used directly by the Music Follows Person service (see Figure 3.14).

The two required functionalities are used by the `Person Detector` service in the following way: when the switch is pressed, the person tracking is activated and the tracking itself is done via the `motionprogram` functionality offered by the Motion Controller service (see Figure 3.12). The Motion Controller service supplies the Person Detector service with image data, which can be processed to track the person under consideration.

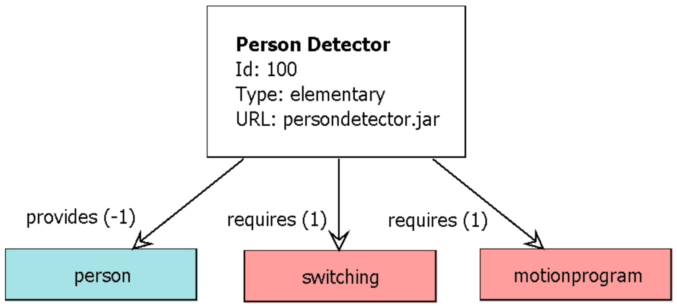


Figure 3.13: The specification of the Person Detector service, which tracks person movements in the eHome.

Figure 3.14 describes the example of the Music Follows Person service needed by the bachelor living in the five room apartment. This service offers the `music follows person` functionality the bachelor needed, i.e. this service offers its functionalities directly for the inhabitant. To offer this functionality, the Music Follows Person service requires not only the `person` detection functionality but also a sound routing functionality `soundroute`. It means that if the person is detected in a room, the sound stream assigned to this person is routed into this room. The `person` detection functionality is provided by the Person Detector service. It is obvious that there is also a service offering the `soundroute` functionality. However, neither

[5]`switching` functionality refines the `detection` functionality.

this nor the on/off Switching service[6] offering the `switching` detection functionality required by the Person Detector servicewill be described in this section.

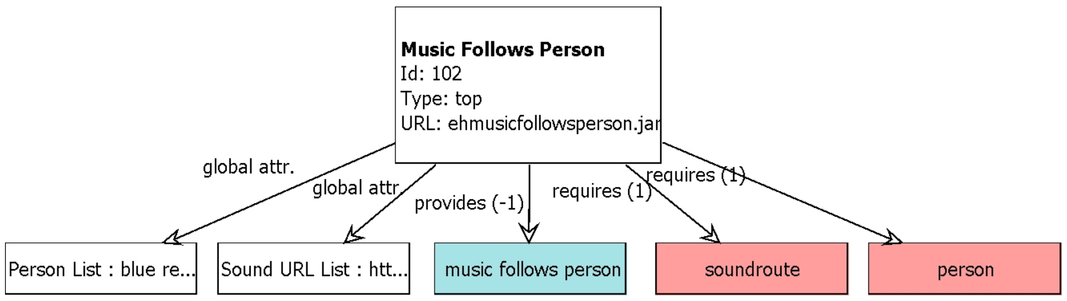


Figure 3.14: The specification of the Music Follows Person service.

As we can see, the services are designed so that they can be composed to enable the reuse of the service software. The service composition is done via the functionalities. The functionalities represent the abstraction layer which provides the necessary flexibility needed for the service composition done during the configuration phase of the SCD-process (see Section 3.6.3).

The three presented services describe one part of the composition tree. The lowest level Motion Controller service controlling the web-cam device is used by the Person Detector service, and the latter one is used by the Music Follows Person service. These dependencies are explained in sections 3.6.3 and 3.6.5.

Up to this point, the activities in the specification phase have been of general nature. It means that the specification of functionalities, device definitions and services can in general be applied for every particular home. The next activities described in this section are specific for every single eHome considering the home environment and the customer requirements.

Specification of the Particular eHome

The specification phase of the SCD-process covers the specification of the eHome environment addressing the environment context of the eHome model. The specification of the particular eHome by the home owner himself or some service provider makes sense after the functionalities, device definitions and services have been specified by the device manufacturer, service provider or other parties. Hence, executing the SCD-process to create an eHome has clearly the goal to have the eHome services running in the living space.

The specification of the home environment has the objective of modelling and defining the surroundings for the running eHome services. Modelling of the environment begins with specifying the floor plan of the eHome, the rooms, doors, windows etc. (see Figure 3.9) – the locations, sub-locations, and location elements (see the environment context in Section 3.5.4).

Figure 3.9 shows a floor plan of our explanatory example (see Section 3.6.1). The figure depicts the `eHome` environment from the environment context, with six

[6]The Switching service is depicted in Figure 3.18.

locations: `Hall`, `Bedroom`, `Bathroom`, `Living-room`, and `Kitchen-corner` as a sub-location of the `Living-room`[7]. The `Hall` has a location element `Entrance Door`, but also location elements (doors in this case) connecting it to the other three locations.

This example corresponds to the environment context of the eHome model (see Section 3.5.4), where an `Environment` class has the `contains` relation to the `Location` class and `Location` class has the `contains` relation to the `LocationElement` class. This example includes the sub-location concept, but no devices modelled in the environment. The device modelling will be addressed in the next section in the scope of our example, the bachelor's eHome environment. Although the example does not include the already existing devices in the bachelor's apartment, they can also be modelled beforehand in the specification phase of the SCD-process. The configuration phase then includes the devices in the configuration of the eHome.

The specification phase consists of the service selection activity. The home-owner has to select the services he/she likes to have in his/her eHome. In our example, the bachelor would like to have the Music Follows Person service in his apartment. The result of this selection must be captured in the model. Figure 3.15 presents the same sample environment information as Figure 3.9, but with another layout for the objects under consideration. Additionally, it presents the service selection information – the `Music Follows Person` is selected for the `Living-room`.

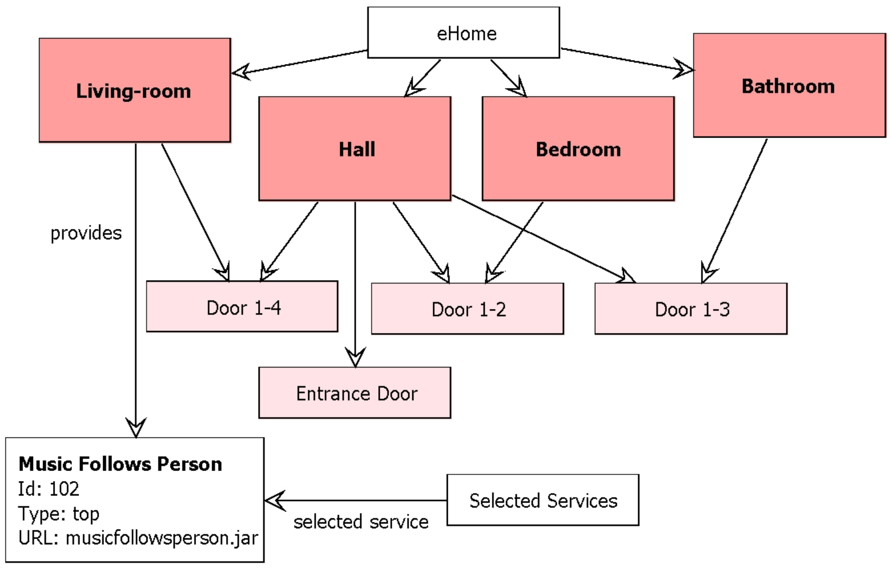


Figure 3.15: The Music Follows Person service is selected for the living-room.

In our example, the bachelor wants to have the Music Follows Person service. This service is selected and it is also determined in which room the service will be running in. For the sake of simplicity, Figure 3.15 shows only the case where the service will only be running in the living-room which is not the case in the complete example described in Section 3.6.1. The same figure shows the connection between

[7]The `Kitchen-corner` sub-location of the `Living-room` will be considered as a part of the example for the specification phase but not in the following process phases since it provides no additional explanatory value.

the `Living-room` and the `Music Follows Person` objects. This corresponds to the `offers` relation between the classes `EnvironmentElement` and `Service` in the service context (see Figure 3.5 in Section 3.5.4). The fact that the Music Follows Person service is selected is presented by the connection between the `Selected Services` and `Music Follows Person` objects corresponding to the `hasActive` relation between the `Environment` and the `Service` classes in the service context (see Figure 3.5 in Section 3.5.4).

Although Figure 3.15 describes the situation where the Music Follows Person service is selected only for the living room, we assume that the service will also be run in the bedroom and in the bathroom. Thus, similar connections between the `Music Follows Person` object and objects `Bedroom` and `Bathroom` should be present, but are omitted to simplify the figure. This note is important since in the next section about the configuration phase of the SCD-process we assume that the service is running also in the two other rooms.

3.6.3 Changes in the eHome Model Instance Structure During the Configuration Phase

The aim of the configuration phase is to create a configuration of the eHome system. The configuration phase is performed automatically. The resulting configuration is deployed into the eHome, initialized and executed. The configuration phase is performed in respect to the selected services in the specification phase. The automatic configuration is handled in detail in [Sch05b].

During the configuration phase of the SCD-process, the eHome model instance is complemented with the following parts:

1. necessary devices are added into the environment context (see Section 3.5.3) needed to put the eHome system configuration into the practice according to the services present in the configuration.

2. the lower level service instances are added into the service instance context (see Section 3.5.5) being driver components for the devices in the eHome. These services provide the functionalities offered in reality by the devices and required by other higher level services.

3. necessary service instances are created according to the selected services. The service instance is the instance level entity for the corresponding service. Service instances are necessary since they differ from eHome to eHome. The instance structure is composed according to the functionality requirements. This is done recursively by solving the overall dependency problem for the selected but also for the lower level services. The result is a service instance graph (see Figure 3.16), in other words, eHome system configuration. This graph has the instances of the selected services as roots. The leaves are typically the service instances controlling the devices directly or providing system resources.

As mentioned, the configuration of the eHome system is done automatically. This fact is the key-point of the whole SCD-process since the central idea behind this process is to minimize the overhead to set up an eHome. This is achieved through the reuse of the eHome service software, by mere composition and configuration of the software components, and automation of the configuration and deployment of the eHome system. The SCD-process should brake the excessive price barrier concerning the eHome software on the home automation market, in contrast to the current situation where every eHome establishment has been a standalone software development project like described in [inH05, T-C05].

We discussed in Section 3.6.2 that the standard policy for the service selection in the eHome is to assign service to a specific location or multiple ones. Thus, the service context is automatically complemented with devices and the corresponding device attributes necessary for the services running in the locations. Figure 3.16 illustrates the result of the automatic configuration activity in the example case where the Music Follows Person service is desired in the living-room, bedroom, and bathroom.

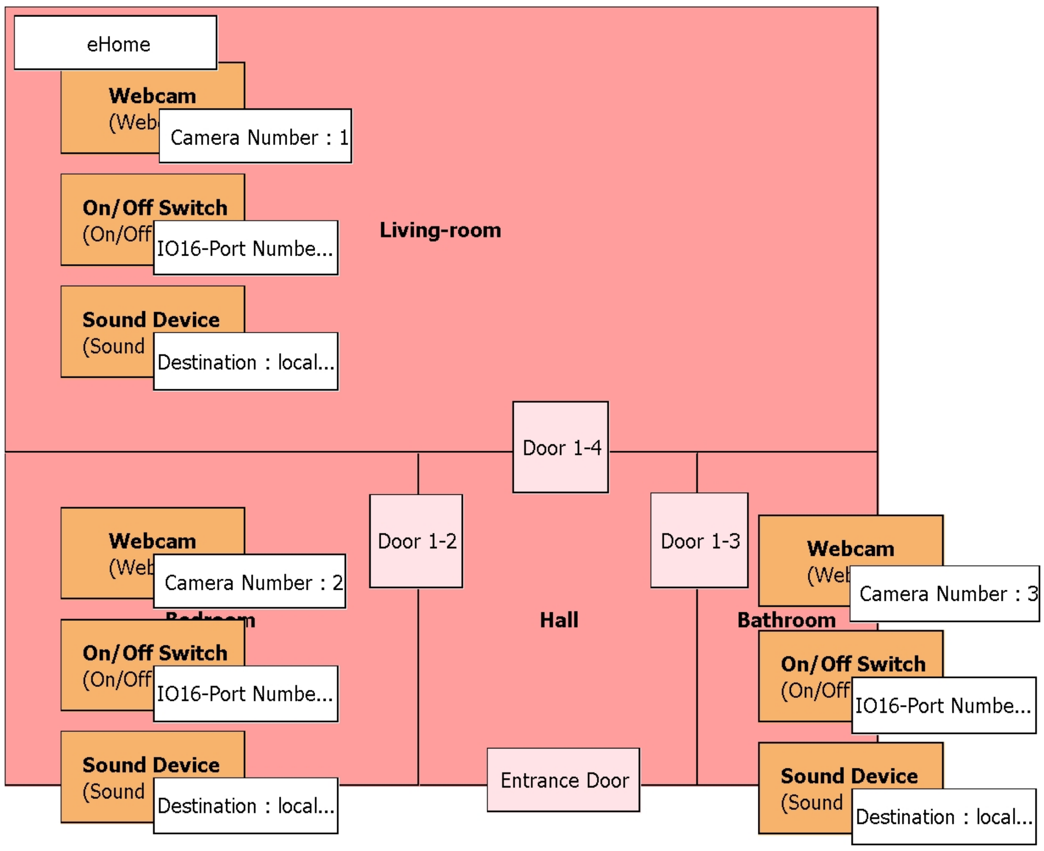


Figure 3.16: The necessary devices in the eHome environment supporting the Music Follows Person service.

There are three new devices added for every room. The web-cam for person tracking, the on/off switch to start and stop the service and the sound device for sound playback. All devices have also attributes. For example, in the case of the web-cam, it has the camera number to address it over the USB connection. The attribute assignments have to be done by hand during the process. But this is inevitable in any configuration process. It might become redundant if the latest research results in the fields of service and resource discovery are integrated into the SCD-process.

If some required devices are already present in the locations, they are reused in the configuration. It is also possible to choose the alternatives during the configuration phase. This is done, if the alternatives are present and the person triggering the automatic configuration wants to be aware of the alternatives. Last but not least, it is also possible to configure the services only according to the devices present in the eHome. Devices are in this case as the preconditions for the service configuration.

It is constructive from Section 3.6.2 and figures 3.14, 3.13, and 3.12 that the devices added to the configuration are not directly usable by the Music Follows Person service, but there are other services present in the configuration offering necessary functionalities for the Music Follows Person service. This issue is addressed in the service instance context, which expresses the runtime configuration of the eHome services.

To be accurate, the service instance context (also noted as service object context) is complemented with five service instances:

1. the Motion Controller service instance controlling the web-cam,
2. the Switching service instance controlling the on/off switch,
3. the Person Detector service instance using the later two,
4. the Sound Router service instance controlling the sound device,
5. the Music Follows Person service instance using the Person Detector service instance and Sound Router service instance.

The service instance configuration to offer the Music Follows Person service only for the living-room in our exemplary case, is depicted in Figure 3.17. For simplicity the figure presents only a fragment of the complete service instance configuration graph. This fragment covers the service instances required to offer the Music Follows Person service only in the living-room of apartment in our example. The similar parts of the graph are instantiated actually for every location where the Music Follows Person service is offered. In our case: living-room, bedroom, and bathroom. The service instances are noted with the `- object` extension in their names. The uses and controls relations are respectively denoted with the `uses` and `controls` edges in the configuration graph. The attributes of the service instances are indicated with `attr` edges. The edges relating attributes to the devices but also the device affiliation with locations are not labelled.

The configuration like the one in Figure 3.17 is generated automatically. The goal by the automatic service composition is to find the service instance dependency graph to fulfil the functional requirements of the top-level services selected by the customer. During the composition the functional requirements of the services and functionalities provided by the services are considered. The dependency graph is constructed using the `uses` self-relation on the `ServiceObject` class – see the service instance context in Section 3.5.5.

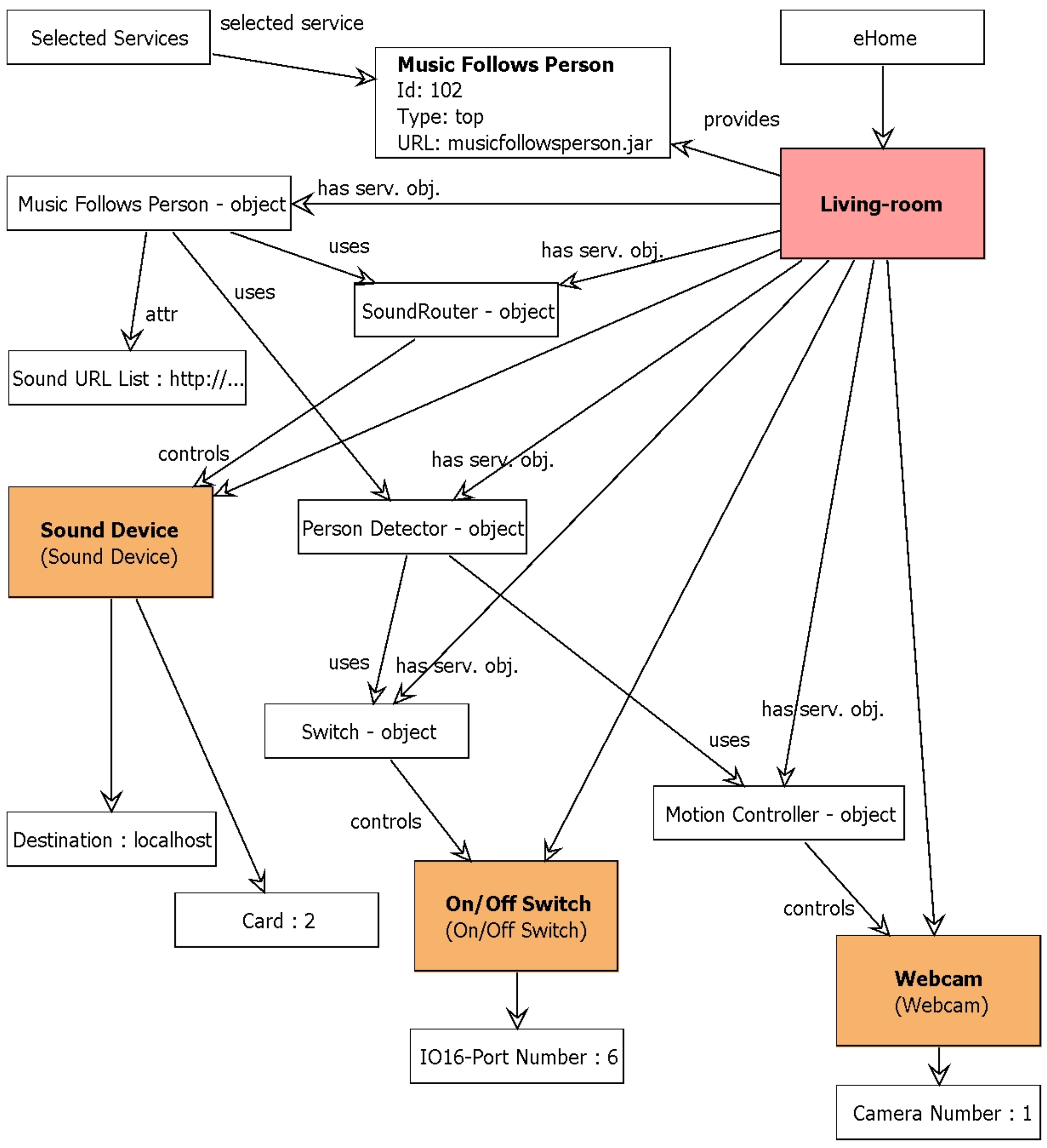


Figure 3.17: The fragment of the service instance configuration graph.

For example, let us consider the path (see Figure 3.17) in the service instance configuration graph: `Music Follows Person` object using `Person Detector` object using `Motion Controller` object which controls the `Webcam` device. This path can be derived from the functional dependencies in the service context by considering the following dependencies: the `Music Follows Person` service (see Figure 3.14) requires `person` detection functionality, the `Person Detector` (see Figure 3.13) service provides `person` detection functionality and requires `motionprogram` functionality, the `Motion Controller` service (see Figure 3.12) provides the `motionprogram` functionality and controls the `Webcam` device.

This example illustrates how the service instance dependency graph is constructed in the service instance context over the `uses` relation on the `ServiceObject` class. It also gives an idea of the role service functionalities play during the automatic configuration. The services can have three types of functionalities related to them: required functionalities, optionally required functionalities and provided functionalities. The required and optionally required functionalities express the resources needed for the service to enable the execution of the service. The provided functionalities are the resources the service provides during its runtime. This kind of functionality division enables the service composition on the functional abstraction level. It enables loose coupling between the services and also selection between the alternatives, since the same functionality required by a service can be provided by several other services. The optional requirements create additional diversity in the service composition.

The *cardinalities* on the service functionalities are the means for applying constraints on the resources offered or required by the functionalities. The cardinalities are defined for the service functionalities during the specification of the service by the service provider who designs and/or implements the specified service. For example, the Motion Controller service provides `motionprogram` functionality with cardinality "-1" (the number in the brackets behind the `provides` relation in Figure 3.12). For the service composition, it means that this resource can be used by infinite number of services at a time.

We can also consider the example, where the Switching service controls a toggle switch with one button. It is reasonable to assume that this switch can be used by only one service at a time to avoid conflicts appearing in the case of the shared resource. Therefore, the controller service for this switch must offer the switching functionality with cardinality "1". This type of a service is visualized in Figure 3.18. This example implies that if there is more than one service installed in the same location requiring the switching functionality, there are also a corresponding number of service instances added into the service instance graph, and also a corresponding number switching devices added into the environment. We can consider a illumination control service in the eHome requiring the switching functionality besides the Music Follows Person service. In this case, there must be also two switches present in the corresponding rooms of the eHome and two Switching service instances in the configuration.

The cardinalities are also used by the service requirements. The cardinality at the required service indicates the quantity of the functionality required. Relying again on the `Switch` service example (see Figure 3.18), we can reckon with a service controlling the two level heating system of the room. This service would require switching functionality with cardinality "2", since the first switch is used to turn on and off the radiators on the walls and the second switch is to turn on and off the floor heating. If this kind of service is installed in the location, there are two Switching service instances added into the configuration and respectively two toggle switch devices into the corresponding location. If some other Switching service would offer the `switching` with the cardinality "2" or greater than two, the heating control service would need only one instance of this kind Switching service. If the Switching service is offering the `switching` functionality with the cardinality greater than two, the same instance can be used even by other services requiring the `switching` functionality.

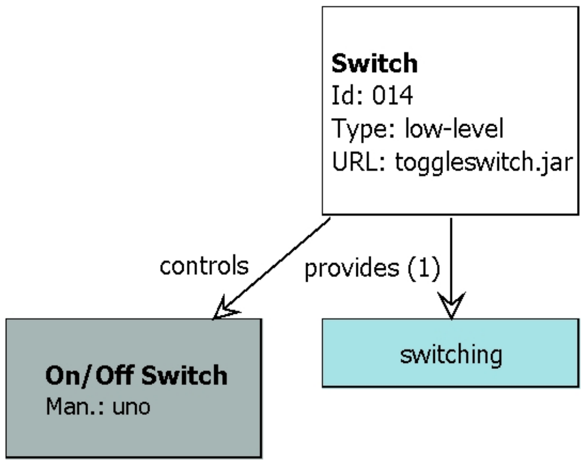


Figure 3.18: The specification of the Switching service to control a toggle switch.

The idea of automatic configuration is simple because of the functional abstraction layer it uses. But it still lacks the rigorous mechanism for composition verification on the software component level. The automatic configuration does not check formally, if the combined components actually can work together and will give a reasonable result during runtime. This could be done using parametric contracts [Reu01] – the additional white-box description of the black-box software components describing the resource requirements, used communication protocols and logical behaviour. This kind of addition would help by the composition verification and the integration of the components produced by completely different software manufacturers without any additional software development.

3.6.4 Changes in the eHome Model Instance Structure During the Deployment Phase

The deployment phase of the eHome system does not affect the eHome model instance greatly. This step focuses on installation of necessary software components on the service gateway to enable the selected and configured eHome services in the eHome. Thus, there are only two aspects changed in the eHome model:

1. Installation is executed according to the service instance context of the eHome model. All service instances present in the configuration are installed onto the gateway. After the installation the corresponding service software components are registered in the eHome model instance at the corresponding service instances objects (see the class `ServiceObject` in Figure 3.7). This is the first change in the eHome model instance. Since all the installed software components must implement the `EhService` interface the registration is realized via the `has runtime component` relation between the `ServiceObject` class and `EhService` interface (see Figure 3.7).

2. After installation the services are initialized and executed. The services are initialized recursively according to the usage graph of service instances. During

the initialization the eHome model instance is complemented with the state information for the service objects. This is the second change in the eHome model instance. The necessary `State` class instances are added for the class `ServiceObject` instances via the `has` relation (see Figure 3.7).

The service instance graph presented in Figure 3.19 complements the service instance graph depicted in Figure 3.17 with state objects. The `Switch` service object has the state `switch` with value 1 or 0 indicating, if the toggle switch device is in the state on or off. The `Person Detector` service object has the state `person` with the value to signalling which predefined person is in the room[8].

3.6.5 Changes in the eHome Model Instance Structure During the Runtime of the eHome System

Likewise, the deployment phase of the SCD-process and also the runtime of the eHome system does not change the structure of the eHome model instance drastically. During the runtime, only the state information of the eHome service instances is changing mostly. But also the inhabitant information is changed during runtime of the eHome system. This results from the fact that the eHome services are *context-aware services*. The eHome services use the eHome model instance

1. to get the location information they are operating on,
2. to get the state information of other services,
3. to communicate the state information to other services,
4. to get the dynamically changing inhabitant information in the eHome.
5. to send messages/ control other services over the service instance and their state objects.

Since the eHome model instance is available for the services during the runtime of the eHome system, the services have at their disposal the whole information encapsulated into the eHome model instance. For the services the most important of the eHome model contexts are the ones containing environment, service, service instance, and inhabitant information. The environment context provides the eHome environment information. The service context provides the general information on services. The service instance context provides the runtime information of the services. The inhabitant information provides useful information for the services concerning the inhabitant locality and preferences.

We will concentrate on the service instance context since this context enables the communication and control flows for the services during the runtime of the eHome system. The communication and control is performed via the states of the services. For example, one service can be a listener to a state of another service, thus being aware of the changes of this state.

[8]There are additional states omitted in Figure 3.19 since the figure would be uncomprehensive otherwise. There should be states also by other service instance objects, and in some cases even more than one.

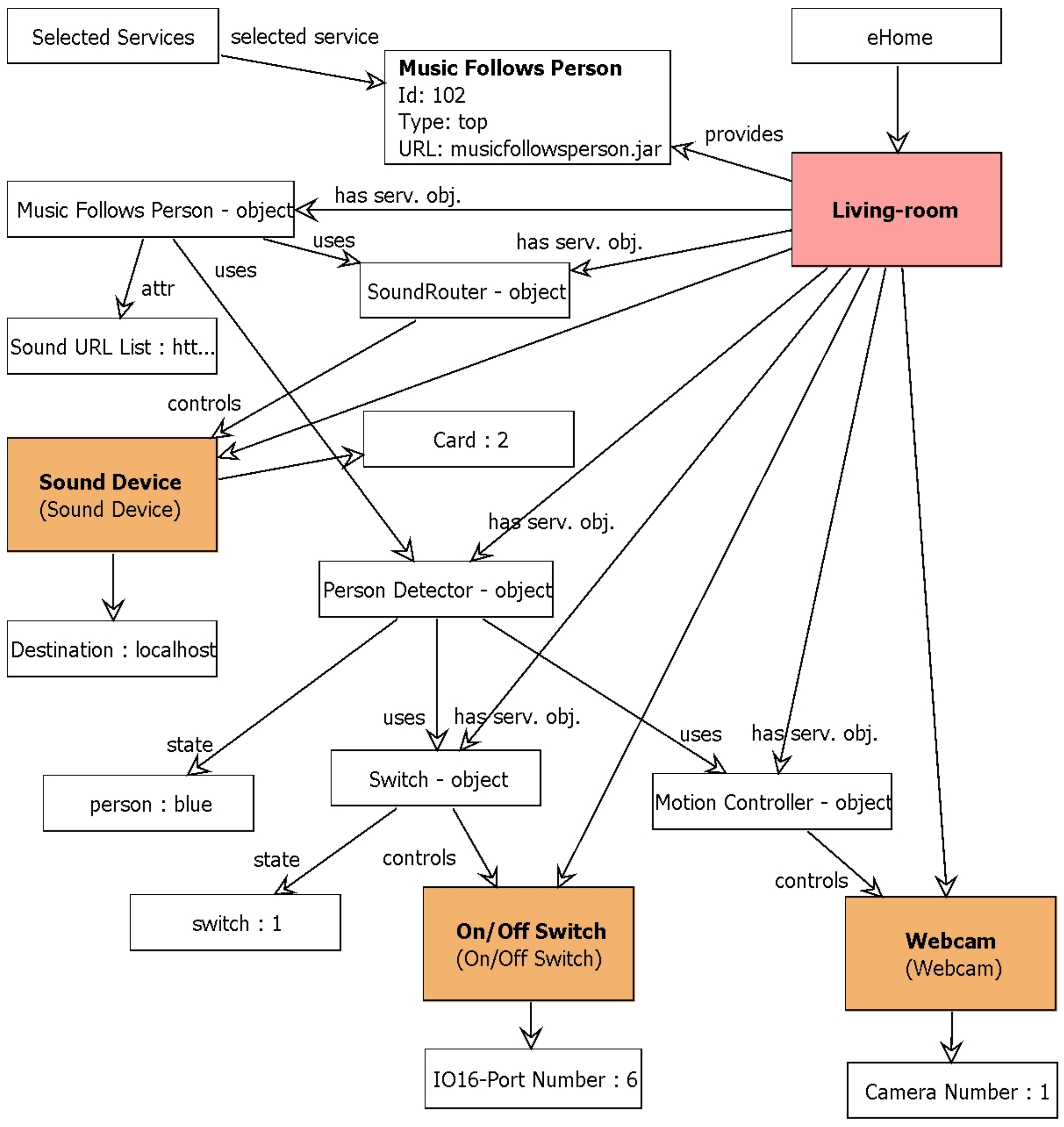


Figure 3.19: The structure of the eHome model's service instance context during runtime of the eHome services.

Figure 3.19 illustrates our example where the bachelor wants to have a Music Follows Person service at his home. According to the example the service `Person Detector` listens to the state `switch` of the service `Switch`. If the service `Switch` changes its state `switch`, the `Person Detector` service can act correspondingly by starting the person detection procedures and setting its `person` state, since the change in the state `switch` signals that the person tracking should be started. The `Music Follows Person` service listening to the changes of the `person` state object can then route the music according to this information into the corresponding room.

3.7 Summary

This chapter gave an overview of the eHome systems and SCD-process for eHome systems. We discussed thoroughly the eHome model and its structure – how it reflects the aspects of the eHomes in reality. We also introduced the concept of the eHome model instance, how it is involved in SCD-process and which are the changes, i.e. transformations performed on this model instance during the process.

The next chapter we will handle the eHome model instance transformations during the SCD-process in the context of the eHomeConfigurator tool. This tool is the general instrument to support the SCD-process. Since the author of this thesis has been one of the main developers of the eHome model and the eHomeConfigurator tool, we will handle two main aspects of the tool. Firstly, we will discuss the architecture and development of this tool. Secondly, how this tool supports the SCD-process, particularly from the eHome model instance's point of view.

Chapter 4

The Tool Support for the SCD-process

The SCD-process relies heavily on the tool support. There are tools supporting all the three phases of the SCD-process and the necessary eHome model instance transformations during these phases. The development of the tools supporting the process is run in the scope of the open-source eHomeConfigurator project [NSM04]. The project has been active since the beginning of the year 2004. The roots of the project lie in the lab-course [Nor03] conducted at our Department of Computer Science 3. The author of this thesis was one of the students participated in the lab-course and one of the three main contributors to the development of the early prototype for the eHomeConfigurator tool. Over ten different developers have contributed to the project during the entire course of it, but the core of the development team has been so far the author of this thesis, his scientific advisor Ulrich Norbisrath, and another graduate student Adam Malik.

The open-source characteristic of the project is chosen to attain better possibilities for distribution of the developed software, to find partners and contributors in the private and commercial fields. The project is licensed with GNU Lesser General Public Licence (LGPL) [GNU99]. This licence supports the free distribution of the source code and the software itself, but allows also the commercial entities to link their proprietary products with our software. The aim of our licensing policy is to reach:

1. the developers in the open-source community,
2. the end-users who use and test this open-source software,
3. companies active in the fields of the home automation and interested in cooperation with the eHomeConfigurator project and eHome research group.

The tools and software developed in the project are united into one general eHomeConfigurator tool. The unification has a purpose to have one tool supporting the entire SCD-process throughout its phases. The eHome model covered in the previous chapter is also considered to be a part of this tool. The following sections give an overview of the eHomeConfigurator tool. We will discuss the structure of the eHomeConfigurator and implementation of its separate modules in more detail. We will also consider the tool in action by discussing how this tool supports the example introduced in Section 3.6.1. This example will be considered from the perspective of the eHomeConfigurator tool supporting the SCD-process. The tool is used to carry out the specification of the bachelor's home environment, configuration of the Music Follows Person service and deployment of this service into the bachelors home.

4.1 The eHomeConfigurator Tool

The research work of the eHome group on the SCD-process has been going on more than three years. During this time, there have been several different prototypes and solution ideas for the tool support. Initially there was a tool chain designed (see also Figure 4.1), which consisted of:

1. Protégé [NFM00], the open-source tool used for the specification of the eHome ontology, developed outside of the eHome group;
2. ComponentPreselector [Kre04], the tool supporting interactive component selection for the eHome system;
3. DeploymentProducer [Skr04], the tool importing the initial configuration document and producing the XML complete configuration document, i.e. the installation instructions as deployment configuration;
4. RuntimeInstancer [Kli04], the tool installing software referenced in the XML deployment configuration document onto the service gateway.

This was a restricted and specific solution limited by the chosen technologies and integration approach. The model used by ComponentPreselector was a knowledge base with a very specific structure lacking generality. The integration between the tools in the tool chain was achieved by the XML document production and transformation approach. The DTDs of these XML documents were complex and highly specific for the OSGi platform. The initial configuration document had to be written and edited by hand.

The tool chain depicted in Figure 4.1 worked as follows. The knowledge base (eHome ontology) was designed in Protégé and had to be changed for every specific home. The initial configuration consisting of the device and functionality information was written by hand in XML. The initial configuration document was refined during the SCD-process by the cooperation of the DeploymentProducer and ComponentPreselector, where the latter tool was using the eHome ontology. The resulting complete configuration[1] was then installed on the service gateway by the RuntimeInstancer.

The current approach does not use knowledge base related technologies but the classical object-oriented technology and the eHome model to support the SCD-process. The reasoning behind the selection of the technology has been given in Section 3.4. The integration between tools supporting the SCD-process is done using the eHome model itself. In this case, the eHome model as the communication

[1]I.e. the deployment configuration.

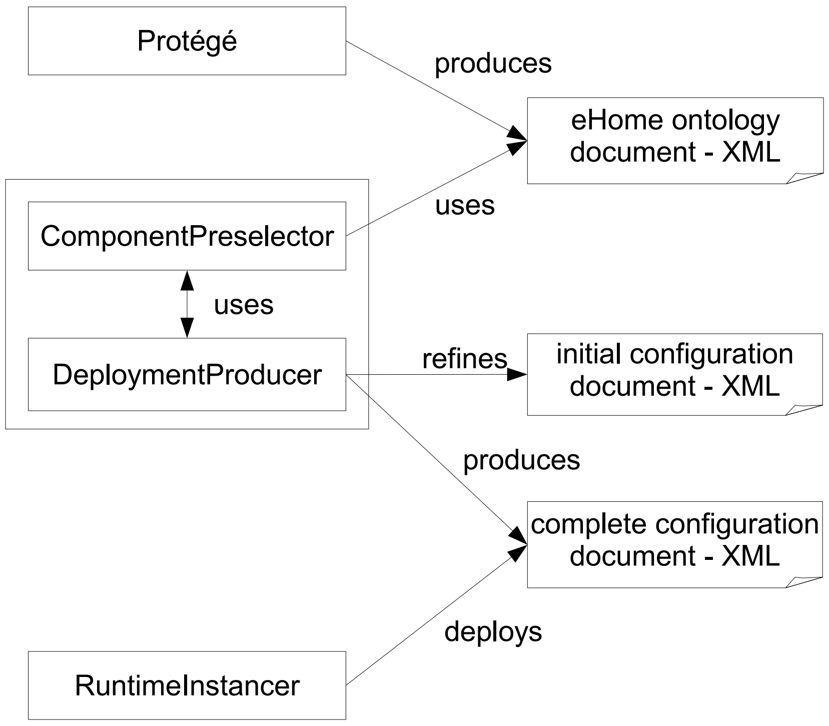

Protégé
produces
eHome ontology document - XML
uses
ComponentPreselector
uses
DeploymentProducer
refines
initial configuration document - XML
produces
complete configuration document - XML
deploys
RuntimeInstancer

Figure 4.1: The tool chain supporting SCD-process before the eHomeConfigurator project.

medium does not only support the SCD-process, but also the runtime of the eHome system (see Section 3.6.5). There is no need for the XML parsers and un-parsers and complex XML-reflect API structures to be dealt with during the tool development. The eHome model is the unified entity for the data exchange and refinement having a fixed and simple communication interface. The object-oriented eHome model gives larger possibilities to extend the tools supporting the SCD-process beyond their current scope if necessary. The complete model structure is available for tools using it and for the future tools requiring the access to the eHome model instance (see Section 4.2).

Figure 4.2 illustrates the structure of the eHomeConfigurator tool. The eHomeConfigurator tool has four main modules. The three: Specificator, Auto-Configurator, and Deployer support each directly the respective SCD-process phase. The fourth module: DataHolder is responsible for the encapsulation of the eHome model instance and is used by all of the other three modules.

In general, the aim of the eHomeConfigurator is to support the SCD-process, i.e. to provide to end-users an interface to the eHome model instance. The eHomeConfigurator tool is used to carry out the changes on the eHome model instance throughout the entire SCD-process. The tool is also responsible for the persistency of the model instance.

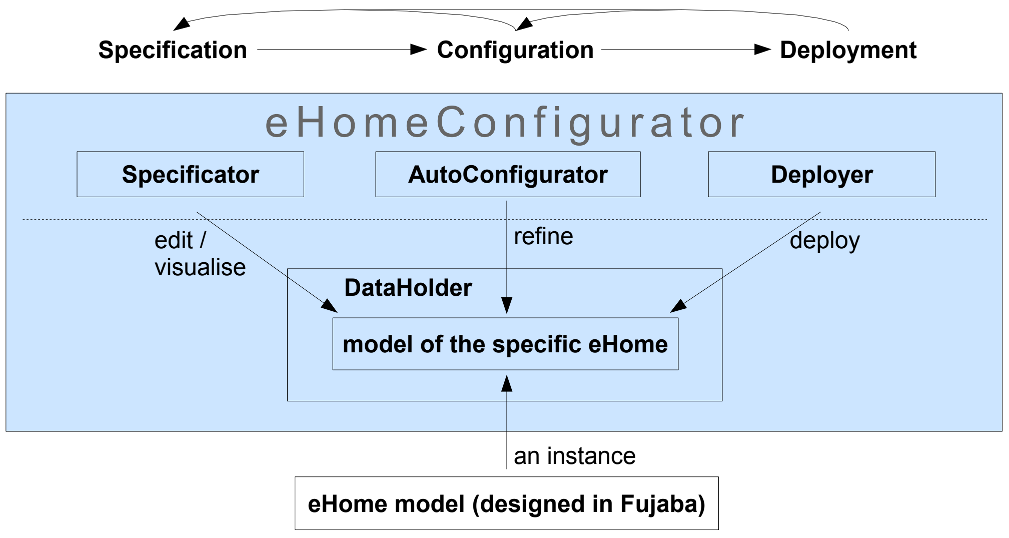


Figure 4.2: The general architecture of the eHomeConfigurator tool according to the SCD-process.

The eHomeConfigurator tool has been developed in the way that it can be run as a standalone application. This enables the work on the eHome model instance separately from the eHome system. The eHomeConfigurator can also be started as a component bundle in the OSGi framework. This allows changing and monitoring the eHome model instance during the runtime of the eHome system making the eHomeConfigurator an integral part of this system.

4.2 The Implementation of the DataHolder Module

This section focuses on the DataHolder module, its functions and structure. DataHolder operates as the container for the eHome model instance. It has a firm interface to access the model instance. The main functions of the DataHolder are: saving the model into a file with a given name, loading the model from the file-system, offering redo and undo functionalities for changes on the model instance. These functions are implemented using the respective features provided by the CoObRA framework [Sch03]. We refer to the storing and versioning mechanism of CoObRa object repository as CoObRa persistency. CoObRa framework provides also the model instance monitoring features by following the JavaBeans specification [Sun97]. For example, this allows the eHome services to “tap into” the model as property change listeners on the objects in the model instance, so that services can respond to the eHome model instance changes immediately.

Besides the reasoning given in Section 3.4 for our technological and instrumental choices, there is one more reason behind choosing the Fujaba tool to design the eHome model. This is because the CoObRA framework is linked with Fujaba in the way that Fujaba is able to generate the code for the CoObRA persistent classes. This is important because in despite of the powerful features CoObRA framework offers, the usage of the framework is not as straightforward as a developer would wish. In other words, the development of the CoObRA persistent model and its classes is complex. The implementation of the designed classes has to correspond to the CoObRA specification considering the property change mechanisms. Fortunately, using Fujaba hides this complexity from the developer. It is possible to make the designed classes in the model CoObRA persistent with one click in Fujaba. The Fujaba-generated Java code for the classes contains thereupon the code corresponding to the required property change mechanisms. This makes the application of CoObRA framework straightforward.

The DataHolder is designed in the way that it can be used by several different tools at a time. If the eHomeConfigurator runs as an OSGi bundle:

1. the Specificator module (see Section 4.3) and Deployer module (see Section 4.6) can use the eHome model instance simultaneously.

2. the eHome services can use the eHome model instance too, because also the services are OSGi bundles.

3. the same DataHolder module can be used by several Specificator modules.

Since CoObRA is an object repository functioning like concurrent versioning system (CVS) for objects or object-oriented database CoObRA, it enables the work of different DataHolders on the same model instance. This functionality is not used by the eHomeConfigurator project yet, but can be taken under consideration to have an alternative mechanism allowing distributed working on the same eHome model instance.

4.3 The Implementation of the Specificator Module

The Specificator module is the user interface for the entire eHomeConfigurator tool and it supports the specification phase of the SCD-process. We will begin describing the user interface of the Specificator and proceed describing the important structural details of the Specificator module. The concrete relation between the user interface and the SCD-process will be given in Section 4.7.

The graphical user interface (GUI) of the Specificator module is developed using the Java Swing *application programming interface* (API). The GUI consists of the main application frame, a menu bar, a tool bar, and a tabbed panel (see Figure 4.3). The menu consists of general selections for different actions like redo, undo, save, load, or zoom on the view. The tool bar represents mostly the same activities but there are also some additions. For example, the buttons for the wizards related to the automatic configuration (see Section 4.5), the button for launching the Deployer module (see Section 4.6), and the activity to start the Dynamic Object Browsing System (DOBS) [GZ02].

DOBS is a system visualizing the Java object heap. This allows the developer to browse the object structures handled during the runtime of the Java virtual machine. For example, the selection of the Music Follows Person service is depicted in Figure 4.3. After pressing the DOBS button one can see the service object in the

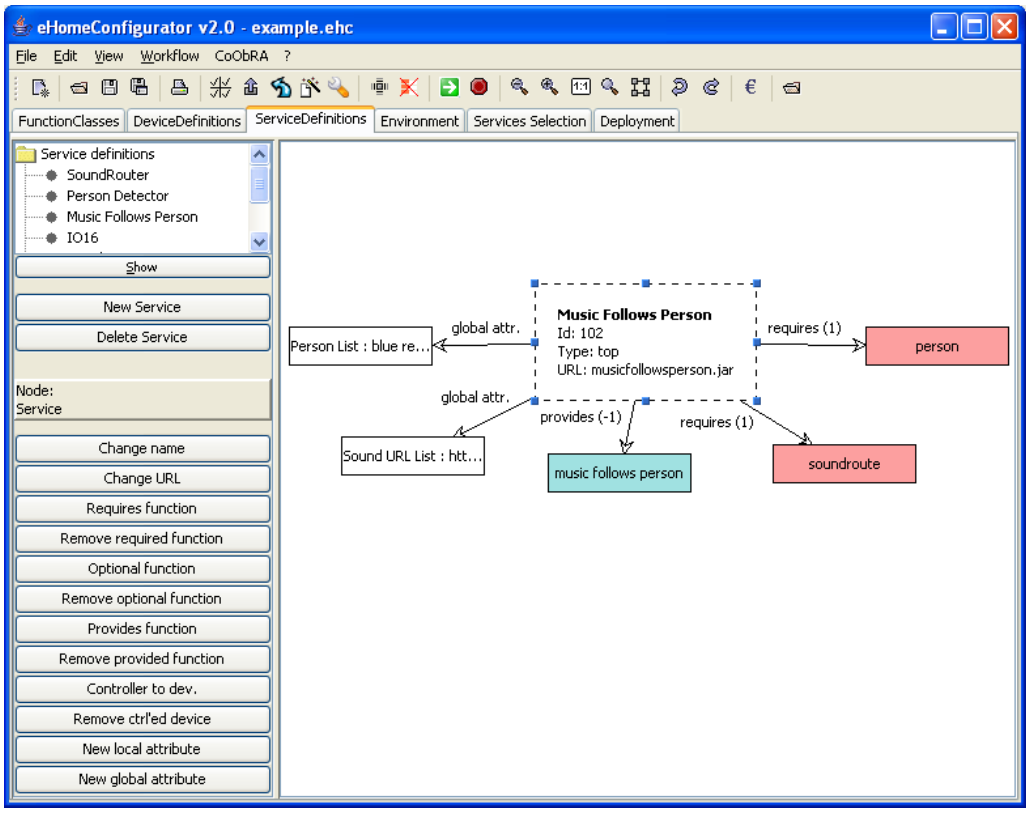


Figure 4.3: The screen shot of the eHomeConfigurator tool – the Specificator module.

DOBS editor with all its field values and also methods. It is possible to browse the other objects related to this service object in a similar way. The DOBS tool provides advanced visual features for debugging and information acquisition on the program and model structure.

The most important components of the GUI are the tabs on the tabbed panel. There are necessary editors and information panels located on the tabs for the eHome model instance manipulation. There are two types of tabs: the *editor tabs* (handled in the scope of this thesis, see an example on Figure 4.3) and the *information panels* (see for more information in [Mal05] and an example on Figure 4.6). For every eHome model context (see Section 3.5) except the inhabitant context, there is one graphical editor tab to work on the respective context. This sums up to five different editors. In Figure 4.3 the Music Follows Person service is visualized in the service editor's graph panel.

The JGraph API [Com] is used to visualise the eHome model contexts on the editor tabs. One could propose that the visualisation could be done using DOBS itself, but DOBS lacks the visualisation features needed by the eHomeConfigurator tool, which is used by the end-users. JGraph offers prepared layout algorithms and extended control on the layouts of the graphs and is specially designed for visualisation of graphs.

The development of an editor consists of two main steps. The first step is to create the translator traversing the eHome model instance according to the model context. The translator visualizes the model context in the graph panel in the editor. Secondly, the activities necessary for the transformations on the model instance are made available to the user. This is done by creating a button into the editor for every required activity and creating alternative drop-down menu for activities, which appears if the right mouse button is clicked on the model object in the graph panel. By pressing the button in the editor or using alternatively the drop-down menu, the input form corresponding to the activity's parameters has to appear together with buttons for execution and cancellation of this activity.

The layout of the editors is mostly the same[2]. Thus, the development of an editor means only sub-classing the `EditorPanel` class. For example, for environment editor, the `EnvironmentPanel` class extends the `EditorPanel` class. In general case, the translators for the JGraph panel and optionally the JTree panel are derived from the `Translator` class. The translators are completed with graph traversing code. Then, they are attached to the class extending the `EditorPanel` class. In the case of environment editor there is a JGraph translator implemented for the `EnvironmentPanel` and the visual part of the editor is ready. The activity buttons and corresponding input forms are created dynamically by the generic activity invocation mechanism. The creation of the translators and using of the generic activity invocation mechanism will be handled in the next sections.

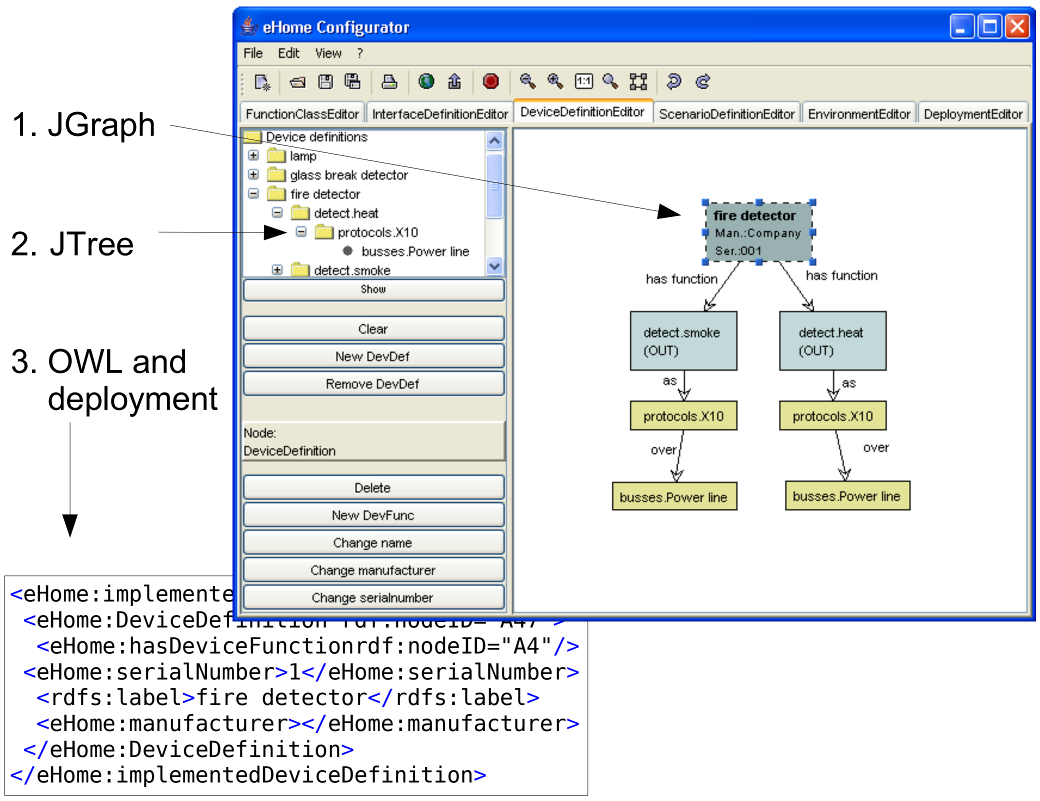
The development of an editor can be seen as the application of the framework, which is designed following the *framework design* principles [GHJV95]. The common behaviour for the editors is generalized into abstract classes which form a mechanism performing as an editor in the Specificator tool by cooperation. Thus, the development of the particular editor requires sub-classing of predetermined abstract classes in the tool and coding the specific behaviour into the corresponding sub-classes to attain the specific properties of the desired editor.

4.3.1 The Translators in the eHomeConfigurator Tool

The translators are the bridges between the eHome model and other Java technologies. The translators are used to create the JGraph, JTree, or OWL structures corresponding to the eHome model instance's structures (see Figure 4.4). The translators are mostly used for the visualisation of the eHome model, i.e. translating the model instance's structures into JGraph and JTree structures. At the moment there are up to two translators involved for every editor: the JGraph translator and the JTree translator. The methods in these translators traversing the specific context in the eHome model are hand-coded. The idea to automate the translator development, to avoid the manual coding of the translators is one of the key topics of this thesis.

The translators were also used to transform the eHome model's and the model instance's structures into the OWL structures using the Jena version 2 Semantic Web API [McB04]. The OWL translators were used to integrate the eHomeConfigurator

[2]The tree component and the button bar on the left, the graph panel on the right, and input form appearing on top – see Figure 4.3.



```
<eHome:implemente
 <eHome:DeviceDefinition rdf:nodeID="A4">
  <eHome:hasDeviceFunctionrdf:nodeID="A4"/>
 <eHome:serialNumber>1</eHome:serialNumber>
  <rdfs:label>fire detector</rdfs:label>
  <eHome:manufacturer></eHome:manufacturer>
 </eHome:DeviceDefinition>
</eHome:implementedDeviceDefinition>
```

Figure 4.4: The translators in the eHomeConfigurator tool.

tool in early stages of the project with the ComponentPreselector and Deployment-Producer tools developed in the frames of [Kre04] and [Skr04]. This integration approach failed because the implementation overhead to modify the ComponentP-reselector and DeploymentProducer was too great. This failure is also a motivation for the automated translator development.

There was a translator created to generate the XML deployment configuration document from the eHome model's structures [Akh05]. This translator integrated successfully one of the older versions of the eHome model instance and RuntimeIn-stancer tool. Thus, contributing a vital feedback for the development of the Deployer module (see Section 4.6) of the eHomeConfigurator tool.

The entire translator development follows the *template design pattern* [GHJV95]. This means that the common properties and behaviour of the translators have been gathered in the `Translator` super class excluding the specific behaviour of the particular translators. In this way every new translator has to inherit from the `Translator` class and implement a `protected abstract void construct()` method to traverse the necessary parts of the eHome model. For example, the environment editor has an `EnvironmentTranslator` class which implements the specific traversal routines in the `construct()` method to process the environment context of the eHome model.

The `Translator` class also provides a number of useful methods for the development of the specific translators and encapsulates the complexity of the JGraph API. In this way, the development of the translators used in the tool is made as straightforward as possible and contains as less repeated coding as possible.

4.3.2 The Generic Activity Invocation Mechanism

The generic activity invocation mechanism was inspired by the Upgrade framework [Jäg00] implementing a similar method invocation mechanism. This mechanism was developed in lab-course by Erhard Schultchen along with the first version of the Specificator [Nor03] and was extended by the author of this thesis to cope with void methods and inheritance of Java classes [NSSK05].

The generic activity invocation mechanism in the Specificator module has the goal to make the activities of the eHome model, i.e. the methods of the model's classes, available on the eHomeConfigurator GUI. This means that the methods of the classes in the model are made available by creating the necessary buttons and input fields of the GUI dynamically. The idea behind the activity invocation is quite simple. The necessary methods are described within the XML configuration file. The configuration file is read at the tool start-up. The buttons are generated dynamically for the editors according to the configuration during the runtime of the tool. The method invocation is performed using the Java Reflection API [FF04]. First, we will describe the XML configuration, then the button and input form generation, and in the end of this section, the method invocation itself.

The example for the XML configuration of the generic activity invocation mechanism can be seen in Listing 4.1. The configuration XML consists of class descriptions according to its DTD. The class description contains the descriptions of the methods of this class, which are desired to appear in the GUI. Listing 4.1 gives an example of the XML configuration for a method `public Environment newEnvironment(String name, EnvironmentRoot root)` of the class `Environment`. This method creates a new environment object and links it with the environment root object.

A developer of the Specificator module wants to see the button for this method with the label `Button Name` in the editor `ENVIRONMENT_EDITOR`. If the user presses this button, the input form appears in the top side of the environment editor (similar to Figure 4.5) to get the values for the arguments `name` and `root`. The developer wants to see on the input form some descriptive information for the first argument and wants the second argument to be fixed as one specific object/element in the eHome model instance. The XML configuration for the `Environment` is described in Listing 4.1.

Listing 4.1 gives an overview on the tags used in the XML configuration. The tag `CLASS` defines which class is under consideration. The tag `ACTIVITY` describes the method. The `CONTEXTS` tag contains the predefined list of the editor contexts where the considered method is available as an activity. The tag `PARAM` describes the argument of the method. The order of the arguments of the described method in the configuration is determined by the method signature. The `TOOLTIP` tag captures the tool-tip or description of the element specified by the tag containing this `TOOLTIP` tag. Hence, the XML configuration mostly relates just the labelling and tool-tip information with the described method and helps to generate the input fields form for the activity. We do not handle the type information of the arguments in the XML configuration. The verification of the input fields together with the type checking is done using the Java Reflection API, while the execution of the activity by the activity invocation mechanism.

```
<CLASS name="Environment">
    <ACTIVITY name="newEnvironment" label="New Environment">
        <TOOLTIP>Create new environment</TOOLTIP>
        <CONTEXTS>
            <ENVIRONMENT_EDITOR/>
        </CONTEXTS>
        <PARAM label="Name">
            <TOOLTIP>The of the environment.</TOOLTIP>
        </PARAM>
        <PARAM label="Environment root">
            <VALUE type="fixed">
                <REF key="ENVIRONMENT_ROOT"/>
            </VALUE>
        </PARAM>
    </ACTIVITY>
    ...
</CLASS>
```

Listing 4.1: The example for the XML configuration for a method handled by the generic activity invocation mechanism in Specificator module.

Nevertheless, it is also possible with the `VALUE` tag to fix objects from the eHome model instance as fixed parameters in the scope of the `PARAM` tags. In this case, the corresponding input field is not enabled for editing like the "This-object" field in Figure 4.5. Additionally, it is possible to describe the selection lists of predefined parameters in the `VALUE` tags. The input field is either a drop-down menu or editable drop-down menu for a selection list.

We consider the return values not to be relevant in the activity configuration for several reasons. This mechanism is designed to satisfy the GUI needs and the end user would not get a lot information out of some object returned. Most of the activities involved in the model have the direct impact on the structure of the model instances, thus the result of the activity invocation is directly visible in the graph panel of the corresponding editor. Nevertheless, if exceptions appear, the exception information is displayed with dialogue windows to notify the user about malfunctions.

At the start-up of the eHomeConfigurator tool the Specificator module reads the configuration XML file and creates the buttons for static methods onto the left side panel of the corresponding editors. In Figure 4.5 the static method of the class `DeviceDefinitionRoot` creating the device definitions is represented by the button `New DevDef`.

The non-static activities are also represented by buttons, but separately from the buttons for the static activities. The buttons are displayed according to the selection

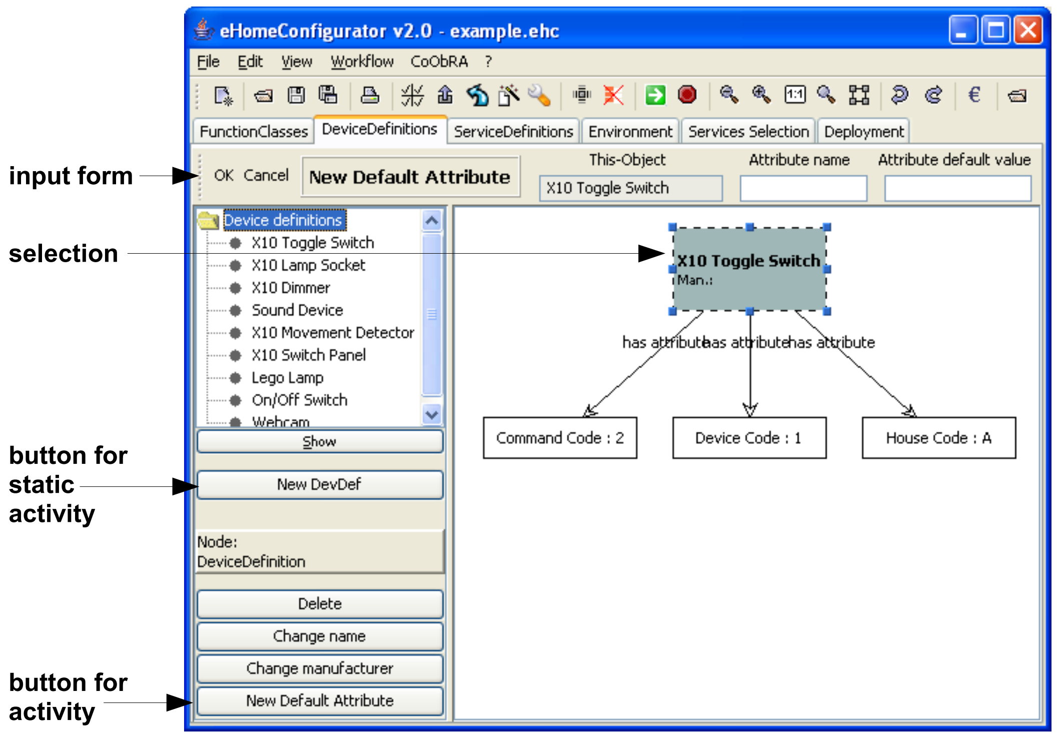


Figure 4.5: GUI elements for the `public Attribute createAttribute(String name, String byDefault)` method in the `DeviceDefinition` class.

performed in the graph panel. If the selection changes, the buttons are generated accordingly on the left side of the editor. In Figure 4.5, there is a `DeviceDefinition` object selected in the graph panel. In the case of the selection event, the buttons appear on the left side panel corresponding to the XML configuration and these buttons represent available activities for selected object in this editor. In Figure 4.5, there are four buttons for the selected `DeviceDefinition` object although, the `DeviceDefinition` class has nearly thirty public methods, which could be made available in this editor.

Figure 4.5 shows the `New Default Attribute` button. This button corresponds to the method `public Attribute createAttribute(String name, String byDefault)` from the class `DeviceDefinition`. By pressing this button the input form appears with: corresponding input fields, description of the selected object (indicated as `This-Object`), and buttons for execution and cancellation of the activity on the upper side of the editor. This behaviour is the same for static and non-static activities, i.e. static and non-static methods. The button labels, number of input fields and tool tips for this method are determined with quite a similar XML configuration as described by Listing 4.1.

After the input form is filled with data, and the `OK` button is pressed, the method corresponding to the activity is called on the object which is selected in the graph panel. The number and the types of the input values are checked by the activity controller using the Java Reflection API. The activity controller also checks if the method exists for the corresponding object and the attribute values match the method signature. If this validation returns with a positive result, the method is executed on the object. If there is no exception thrown, the input form is removed and the graph panel is refreshed in the editor. Otherwise, the exception information is displayed on the pop-up dialogue window.

4.4 The Main Problem at the Specificator module Development

The editor development has been optimised by the underlying software design to be as straightforward as possible having minimal overhead on coding. Still, there is a problem remaining. The development of the editors is still not generic enough. The frequent and severe eHome model changes require a great deal of maintenance work. We do not consider the eHome model instance changes during the SCD-process and runtime of the eHome system here, but the structural changes of the eHome model itself.

During the eHomeConfigurator project, there have been at least eight bigger structural changes in the eHome model. The model has been partly redesigned during these changes. Every change requires manual recoding of translators. This is error-prone, requires quite a high level of programming expertise and development time. Luckily there already is some degree of genericity introduce by the design of the editors and generic activity invocation mechanism. This problem is illustrated in Chapter 2 of this thesis. Chapter 2 discusses an example about implementation of the sub-location concept into the model, the steps for the translator reprogramming, and adoption of the activity invocation configuration XML.

The translator-related development problem is tackled in the theoretic part of this thesis (chapters 5, 6, and 7). We introduce the triple graph grammar (TGG) based approach to develop generic bidirectional translators. In this case the translators are not hard-coded, but specified with special TGG rules. These rules are interpreted during the runtime of the tool and provide the synchronisation mechanism between the eHome model instance and JGraph structures. The developed mechanism is bidirectional: the specified translators synchronise transformations of related models in both directions. The translators in the eHomeConfigurator are unidirectional by just "exporting" the eHome model structures to other technologies. Our synchronisation approach also enables to propagate changes in JGraph structures directly to the eHome model instance. (see Chapter 6).

4.5 The Auto-Configurator Module

The work on the Auto-Configurator module of the eHomeConfigurator is handled in [Mal05] and [Sch05b], thus being out of the scope of this thesis. The Auto-Configurator has the task to construct the service instance context (see Section 3.5.5) according to the eHome services, which were selected during the specification phase. The Auto-Configurator creates an eHome service configuration which can be deployed into the eHome. There are corresponding wizards available to guide the user through the automatic configuration process, and corresponding launch buttons reside on the tool bar of the Specificator module.

The Auto-Configurator module represents the tool support for the configuration phase of the SCD-process. This module embodies all necessary activities required by the configuration phase. It automates nearly completely the whole configuration process by narrowing the manual activities down to selecting the alternatives and describing the attributes. The manual activities are supported by the wizard available through the Specificator module.

The result of the configuration phase is the structure in the service instance context of the eHome model instance. The generated structures express the runtime configuration of the eHome services – which services are required, and how they work together. These structures are the input for the deployment phase of the SCD-process.

4.6 The Implementation of the Deployer Module

The Deployer module is designed and implemented to install the eHome system configuration onto the service gateway at the eHome. This configuration is constructed by the Auto-Configurator Module. The start-up of the Deployer module is the sequential activity to the automatic configuration. This marks the transition to the deployment phase of the SCD-process.

The execution of the Deployer module is also possible over the Specificator module's GUI. There exists a button on the tool bar of the Specificator, which launches the Deployer. The Deployer analyses the service instance context (see Section 3.5.5) structure of the eHome model instance and performs for the installation the following steps:

1. Determining the *list of services* which have to be installed. This requires the traversal of the service configuration graph in eHome model instance's service instance context. The traversal is performed using the `uses` relation on `ServiceObject` class (see Figure 3.7). The traversal is performed recursively by following this relation between the service objects. For example, considering Figure 3.17, the composed service list would contain Switching service, Motion Controller service, Person Detector service, Sound Router service, and Music Follows Person service.

2. *Installing every necessary service* from the list of services onto the service gateway. The Deployer uses the features offered by the bundle context of the OSGi gateway to install the service software onto the gateway. During this step the installed bundle components are also registered in the Deployer module for future handling. This means for example, that Deployer has references to implementations of Switching service and Music Follows Person service

3. After installation of the services, *every installed service is started* in the framework. Deployer tries to start all installed services sequentially. If some errors occur, the cycle for starting the services is finished and started over by giving the framework more time to deal with probably unresolved dependencies

between the services. This approach lacks of elegant handling of the resource resolution, while starting the services assuming that the service configuration constructed by the Auto-Configurator module is correct.

The installation and starting of the services could be also done using the bundle listener (`BundleListener` interface) features provided by the OSGi framework API, but this approach introduces particular class-loader problems. If the bundle listener concept is used, the resource dependencies between the services are probably still not solved correctly. It is because the installation events might arrive in random and do not carry the resource resolution information.

4. The started *services are registered in the eHome model instance.* The Deployer implements the service listener (`ServiceListener`) interface of the OSGi API. This enables the Deployer to get the service events informing it about the life cycle changes of the services. After a service has been started, the framework sends the service registration event. Upon receiving this event the service is registered at its corresponding `ServiceObject`'s in the eHome model instance as the implementation for the service described by the particular `ServiceObject`. For example, the `Music Follows Person` object in Figure 3.17 has after this step a reference to the implementation of the Music Follows Person service installed on the service gateway.

5. All the *eHome services are initialized with `init()` method* of the `EhService` interface (see Figure 3.7). In this additional initialization routine lies a flaw in service design, which has to be addressed in future work. The initialization requirement by the `init()` method for the service development should be omitted and initialization should be implemented in the `start` method of the eHome service `Activator` class.

6. Finally all the *eHome services are executed in the eHome context* with `execute(ServiceObject so)` method known from the `EhService` interface. This provides the service with the information about eHome context. The `ServiceObject` provides the information about other required services, the environment, the states and attributes, but also the controlled devices, and the abstract description of the service itself (see figures 3.6 and 3.7). During this method-call the service can add the state information into the eHome model instance by creating and relating the `State` class objects with the corresponding `ServiceObject` instances (see the service instance context structure in Section 3.5.5). For example, the Switching service object `Switch` on Figure 3.19 creates a state `switch` with value `1`.

   The services upon the `execute` method is called, call also often themselves the `execute` method upon the services they are using. But this depends on the implementation and is used by developers in the case the services placed higher in the usage hierarchy need the state information of the used services.

The Deployer module is tested and works with two different OSGi implementations: a proprietary Prosyst mBedded server 5.x [Pro] and an open source implementation of the OSGi framework Knopflerfish [Kno04].

4.7 The eHomeConfigurator in the Action

We will consider the eHomeConfigurator tool now in the scope of the example introduced in Chapter 3, where the bachelor living in the apartment desires the Music Follows Person service (see Section 3.6.1). As mentioned, the eHomeConfigurator tool embodies the tool support for the SCD-process and the Specificator module is the GUI for this tool. We will describe how the tool is used to support our example. According to the SCD-process there is an editor available for nearly every eHome model context to specify the necessary input information for the following phases of the process. There are also corresponding trigger buttons and wizards for the following configuration and deployment phase of the SCD-process. A more detailed description regarding the tool support for this process follows in next subsections.

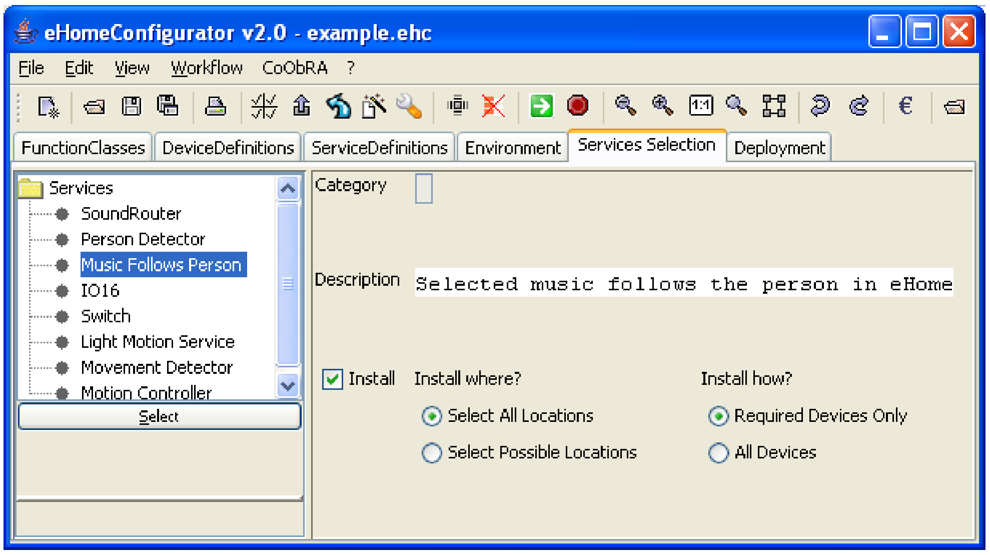
4.7.1 Specification with the eHomeConfigurator

In the considered example (see Section 3.6.1) the bachelor desires the Music Follows Person service. An external service provider has to specify the functionalities required and provided by this services in the eHome system configuration. There is a function class editor in the Specificator module to create and edit functionality refinement trees like depicted in Figure 3.10. In fact, the descriptive figures in sections 3.6.1 – 3.6.4 are all exported from the different editors of the Specificator module[3].

After the functionalities are specified the devices are next in turn. Devices are specified using the device definition editor (see Figure 4.5) and can be specified by device manufacturers or again by service providers. Services, their requirements and features on the functional abstraction level have to be specified by service providers in the service editor (the example is seen in Figure 4.3). The following specification step is to describe the floor plan of the eHome in the environment editor. An example of the floor plan for the bachelors apartment described in the environment editor can be seen in figures 4.7 and 3.9. The floor plan includes the description of rooms and their interconnections via doors, windows, etc. Additionally to the floor plan the devices are specified, which are already included in the home and are planned to be the part of the eHome system. The function, device definition, service and environment editor address respectively the function, device definition, service, and environment contexts of the eHome model (see sections 3.5.1, 3.5.2, 3.5.4, and 3.5.3).

The next specification step includes the service selection. This is done using the service selection information panel in the Specificator module (see Figure 4.6). The service selection requires the information where the service is activated and if it is installed considering just the requirements of the service or additionally the optional requirements [Mal05, Sch05b]. The selection information panel addresses the service context of the eHome model (see Section 3.5.4).

[3]Specificator is able to export the graphics of the editors in scalable vector graphics (SVG) format, joint photographic experts group (JPEG) format, and graphics interchange format (GIF).


eHomeConfigurator v2.0 - example.ehc
File Edit View Workflow CoObRA ?
FunctionClasses DeviceDefinitions ServiceDefinitions Environment Services Selection Deployment
Services
SoundRouter
Person Detector
Music Follows Person
IO16
Switch
Light Motion Service
Movement Detector
Motion Controller
Select

Category

Description Selected music follows the person in eHome

[✓] Install

Install where? [✓ Select All Locations] Select Possible Locations

Install how? [✓ Required Devices Only] All Devices

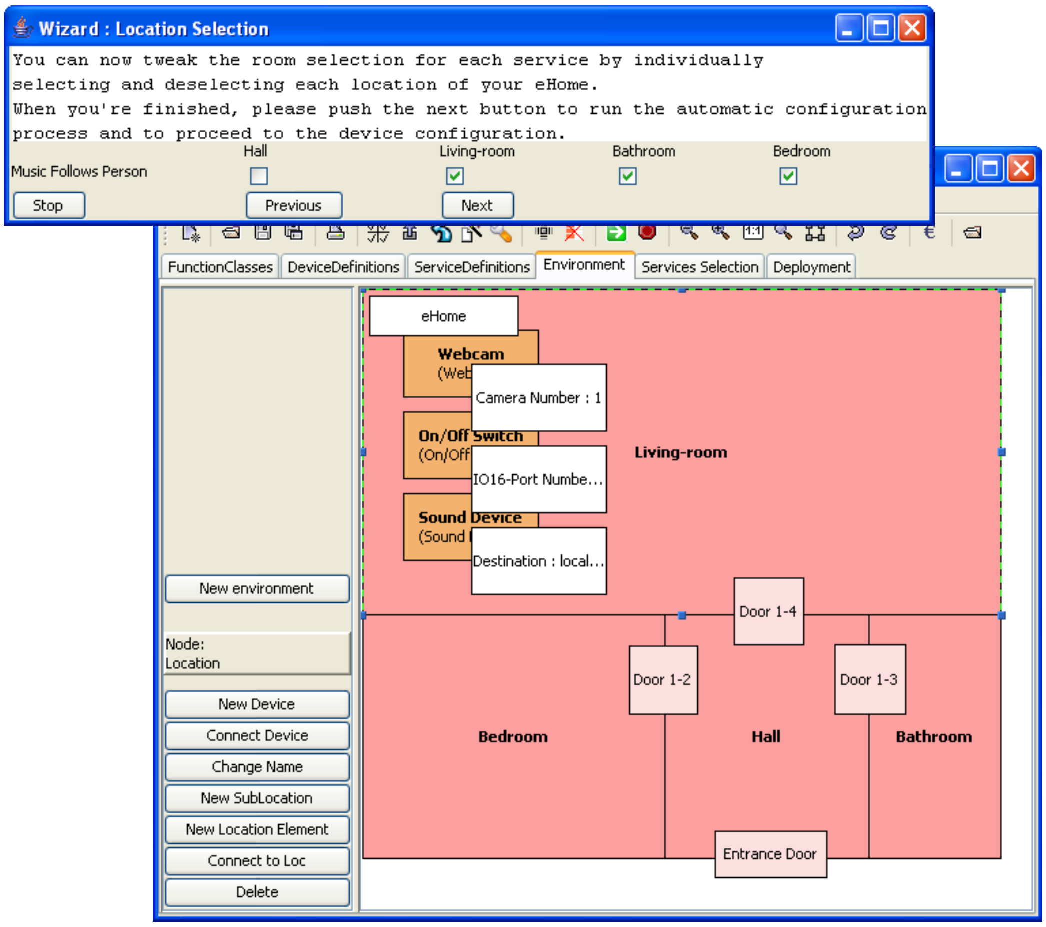
Figure 4.6: The service selection panel in the Specificator module.

4.7.2 Automatic Configuration of eHome Service Software with the eHomeConfigurator

The specification phase of the SCD-process finishes with the environment specification and service selection activities. The process proceeds with configuration phase which addresses the service instance context of the eHome model (see Section 3.5.5). The automation is implemented in the Auto-Configurator module of the eHomeConfigurator and supported by the corresponding wizard available through the Specificator GUI [Sch05b]. Since the service selection is also added to the wizards, the first two windows of the wizard offer similar functionalities as the selection information panel discussed above. In Figure 4.7, there is the second window of the wizard, which gives additional selection options – where exactly the selected service has to be available. In the bachelors example, he has selected the Music Follows Person service to be installed into all of the rooms in this apartment, but the hallway.

The wizard continues with the selection of the alternatives for possible devices and service components. It can happen that some requirement of the service can be satisfied by several services or devices. For example, one could use for person detection the cheaper USB web-cams or more expensive IP cameras. IP cameras should be used, if a security service is installed in addition and it requires more elaborate monitoring devices. The alternatives can be chosen per-room in the eHome, but also simultaneously for the entire environment. The device and service attributes are specified in the next wizard step[4]. This finishes the automatic configuration requiring minimal amount of user interaction to obtain the information which can not be obtained automatically.

[4]For example, the USB camera numbers to address web-cams in the case of our Music Follows Person service or some e-mail address for the e-mail notification service used by the security service.


Wizard : Location Selection

```
You can now tweak the room selection for each service by individually
selecting and deselecting each location of your eHome.
When you're finished, please push the next button to run the automatic configuration
process and to proceed to the device configuration.
```

	Hall	Living-room	Bathroom	Bedroom
Music Follows Person	☐	✓	✓	✓

Figure 4.7: The automatic configuration wizard in the eHomeConfigurator with the environment editor of the Specificator module in the background.

The deployment editor of the Specificator module can be used to browse the finished configuration. One layouted small piece of the configuration graph exported from the deployment editor can be seen in Figure 3.17 and the deployment editor itself in Figure 4.8[5]. The deployment editor has its use also during the runtime of the eHome system.

4.7.3 Deployment of the eHome Service Configuration and Monitoring of the eHome System Runtime with the eHomeConfigurator

The eHomeConfigurator tool is used to install the services given in the configuration. For this, the tool must be started on the OSGi service gateway. In this case, the eHomeConfigurator will be an integral part of the deployed eHome system. The Deployer module is started through the button available on the tool bar of the

[5]Figure 4.8 shows an un-comprehensive deployment graph. This illustrates the fact that deployment graphs are very complex in the case of more sophisticated eHome systems.

Specificator GUI. The deployment information is displayed on the console of the service gateway. After the successful deployment the corresponding dialogue window displays the appropriate result. The deployment editor in the eHomeConfigurator is refreshed to display additional runtime information of the eHome system reflected on the eHome model's service instance context (see Section 3.5.5).

During the deployment and the `execute(ServiceObject so)` method calls on the top level services, the service instance context is completed with state information (see Section 4.6). This is presented in Figure 3.19, which is the graphics export from the deployment editor of the Specificator module. The editor is depicted in Figure 4.8 showing the fraction of the more complex configuration graph. The selected state object in the editor indicates that the Person Detector service has a `State` with name `Person` and value `blue`, i.e has detected the person with code `blue`.

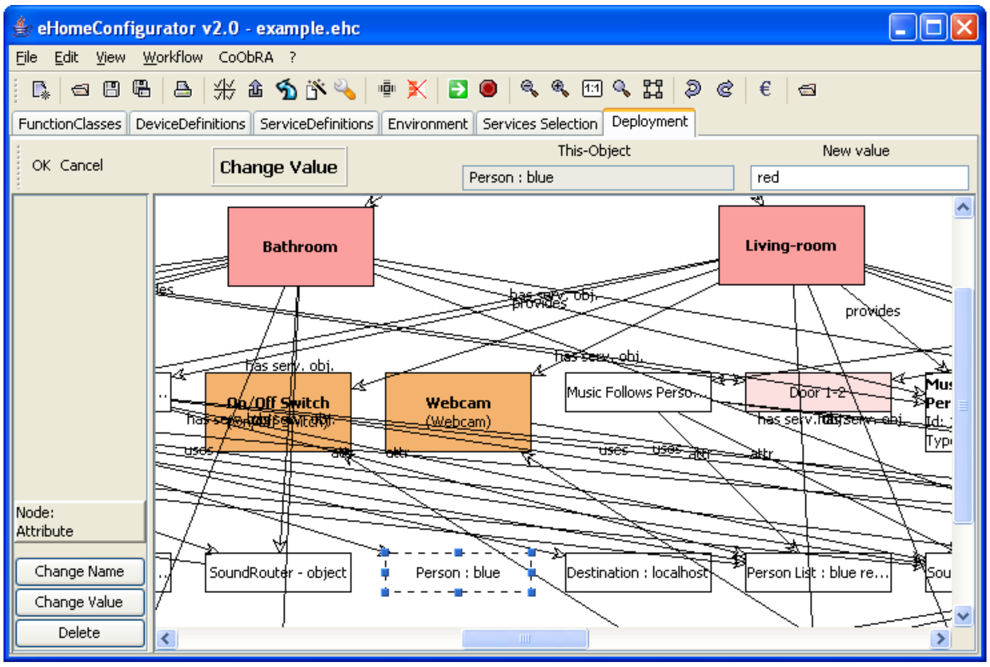


Figure 4.8: The deployment editor in the Specificator module.

The deployment editor visualises the contents of the service instance context of the eHome model instance. It enables during the runtime of the eHome system to browse and access the model instance's deployment and runtime relevant information. It offers also the manipulation of the model instance. For example, the user can change the value of the `Person` state to `red`,to alter directly the behaviour of the person detection service. This again has the result on the behaviour of the Music Follows Person service. Also the other services and devices can be controlled in similar way. This shows that the eHomeConfigurator tool acts like the preliminary user interface for the eHome system. DOBS can also be started from the Specificator module which gives even more accessing and browsing options. The deployment editor and DOBS offer together useful testing and debugging features for the eHome system developer.

4.8 Summary

In this chapter the tool support for the SCD-process was discussed. The tool supporting the process is called eHomeConfigurator. It consists of different modules and GUI to support the entire process. We also described the modules of the eHomeConfigurator: DataHolder, Specificator, Auto-Configurator and Deployer. The implementation of the DataHolder, Specificator, and Deployer where considered in more detail, since this thesis contributes largely to the development of these modules. In the end of this chapter, it was described how the eHomeConfigurator supports the SCD-process in practice by describing the usage of the eHomeConfigurator to deploy the Music Follows Person service introduced in Section 3.6.1.

The main problem during the development and maintenance of the Specificator module is related to the translator development (see Section 4.4). Translators have the task to transform eHome model instance structures into the structures of the JGraph, Java Swing, and Jena Semantic Web API's. It is possible that new translators have to be developed for other technological APIs' in the future, what is a time-consuming task. Additionally Every eHome model change requires the manual reprogramming of the translators, what is laborious and error-prone. This problem is solved by developing and using the TRIMoS framework, a triple graph grammar based reactive synchronisation mechanism between different models. This topic will be handled in next chapters by discussion on the proper theoretical background and the TRIMoS framework.

Chapter 5

Theoretical Background

In the previous chapters we introduced the eHome model (see Section 3.3) and its participation in SCD-process (see Section 3.6). We also handled the tool support for the SCD-process (see Chapter 4) and saw the development problem rising related to the translators used in the eHomeConfigurator tool (see Section 4.4). We will describe how to solve this problem with the help of the TRIMoS framework handled in Chapter 7. TRIMoS is a triple graph grammar based synchronisation framework for graph-like models. Before discussing the solution to the translator problem in Chapter 6, we have to introduce the underlying graph theory.

5.1 Graph Grammars

Nearly all current software development techniques involve *graphs*. The different software models on architecture level or during runtime as well diagrams or models specified with visual notations[1] can all be considered as graphs. Computations on these kind of graph-like models can be described with graph transformations. Our eHome model is an object-oriented model described with UML class diagrams and activity diagrams. Therefore, the theoretical approaches dealing with graphs and graph transformations are suitable to handle the eHome model related technical aspects and problems.

This section gives an overview on graph transformations and related concepts and approaches. The most important source for the whole section is the fundamental handbook on graph transformation theory [Roz97]. A short comprehensive overview on graph grammars can also be found in [Hec05].

Graph grammars as an approach for rewriting have been invented in the early seventies to compensate the deficiencies in term rewriting and (string) Chomsky grammars [Pra71, EPS73]. The idea is to rewrite graphs which are non-linear structures. The graph grammars are the means for precise modelling of local transformations on graphs. Before introducing the graph grammars we will give a definition for a graph.

[1]For example, UML family languages.

Definition 5.1 *Graph*

> *A quadruple* $\mathbf{G} := (V, E, s, t)$ *is a* graph*, where* V *is the set of vertices,* E *is a set of edges, and* $s, t : E \rightarrow V$ *are two functions which assign source and target vertices to edges.*

This definition for a graph is equivalent but somewhat different from traditional notation where the graph is given just as duple with sets of nodes and edges, whereby edges are duples of nodes. This notation is chosen to have simpler formulations for the following definitions in this chapter. Definition 5.1 omits the labelling functions for the vertices and edges, since they do not have a significance in the context of this chapter. More precise definitions can be found in [Roz97].

A *graph grammar* consists of the finite set of graph productions (also known as transformation rules) and a starting graph, thus defining a language whose words are graphs. Productions rewrite a graph and are in the form $p : L \rightarrow R$. Graphs L and R are called the left- and the right-hand side of production. Figure 5.1 shows a production defining a partial correspondence between the elements on its left- and right-hand side. The production determines which nodes and edges have to be preserved, deleted, or created during the application of the production on the given graph, as described in the following Definition 5.2.

Definition 5.2 *Graph Production*

> *Any duple of graphs* $p := (L, R)$ *with* p *defining a partial correspondence between the* L *(left-hand side graph) and* R *(right-hand side graph) is a* production*. If* p *is applied to a given (host) graph* G*, it produces another graph* G'*. This is denoted by:* $G - p \rightarrow G'$*, with respect to (left- and right-hand side)* matching morphism $g : L \rightarrow G$ *and* $g' : R \rightarrow G'$*, where* $g'|_L = g$*, i.e.* g *and* g' *are identical mappings with respect to the left-hand side graph* L*. The application with respect to the morphism* g *is denoted by* $G - p(g) \rightarrow G'$*.*

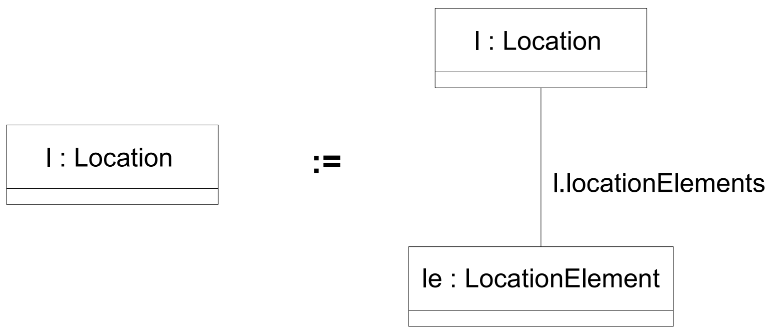


Figure 5.1: A graph production. The left-hand side of the production is on the left of the ":=" sign. The right-hand side of the production introduces new elements into the graph

Figure 5.1 shows a graph transformation, where a new edge *l.locationElements* and a node *le* : *LocationElement* are added to the graph. This transformation is performed on the graph, if there is a node matching with *l* : *Location* node inn the left-hand side of the graph production. In this case the *l.locationElements* is connected to the node which was matched with the *l* : *Location*. The created node *le* : *LocationElement* is connected to the other end of the *l.LocationElements* edge.

The application of the production on the graph G with correspondence to the match found for the left-hand side of the production follows quite a simple mechanism. Every object in G which matches an element in L that has no corresponding element in R will be deleted. Symmetrically each element from R which has no corresponding element in L will be added to G. All remaining elements in G are preserved. Thus, the derived graph G' is constructed as $G' = G \setminus (L \setminus R) \cup (R \setminus L)$. The operations for the graph are handled in Definition 5.4.

There are two different approaches to graph grammars: the *gluing* and the *connecting* approach, which are also respectively known as algebraic and algorithmic approach. The theory behind the first one relies on category theory and special pushout constructions. The second one relies on set theory. The differences in these approaches narrow down to three aspects: the definition of the graph, how the left-hand side of the production is matched with the graph, and how the left-hand side is replaced with the right-hand side of the production in the graph. The matching and replacement mechanisms are sometimes also denoted as embedding mechanism.

The connecting approach deals mostly with context-free graph grammars, where the left side of the production consists only of one node. Thus, nodes are replaced by graphs. There is more interest in the algebraic approach – it is used by most of the graph transformation systems, because of involvement of context sensitive grammars and the grater expression power. This approach is divided into the double-pushout (DPO) and the single-pushout (SPO) approach. These approaches rely on different pushouts, i.e. gluing conditions. Gluing conditions determine how the embedding mechanism behaves in the case the graph elements are deleted.

The first problematic aspect is that matches can contain conflicts by matching host graph elements and left-hand side of the production in the way that the production specifies both deletion and preservation of the element. The second problem is related to the dangling edges which might appear if the vertices are removed from the graph. The DPO approach does not allow conflicting matches and production applications producing dangling edges. The SPO approach allows both: conflicting matches and creation of dangling edges by preferring deletion over preservation in the conflicting match case and by deleting dangling edges.

It is also important to consider a concept of type graphs, because the system implementation level deals mostly with typed constructs. The most common example in the object-oriented software development is the relation between the classes and the objects, i.e. the relation between the class and the instance of the class. The class in is the type definition for the object. Thus, the graph containing classes and relationships between them is the type graph[2] for the corresponding object graph, which is constructed during the runtime of the implemented system. A similar rela-

[2]Type graph can also be considered as a graph schema for its instance graphs.

tion is between the class diagram and object diagram in the UML, which visualize respectively the class and object structures.

Object graphs contain additionally pre-defined attributes. Thus, the attributes must also have a type declaration. The notation $o : C$ represents the vertex (object) o of type (a class) C and $a : T$ represents the type level declaration for attribute with name a and type T. Thus, the *type graph* represents the type level and is instantiated as individual snapshots, the *instance graphs*. For example, the eHome model is a type graph for the eHome model instance.

When fixing a type graph TG, a graph production $p : L \rightarrow R$ has to be considered as production with name p having a pair of instance graphs over TG as its left- and right-hand side. The graphs L and R are compatible, i.e. the vertices with the same identity in L and R have the same type and attributes, and edges have the same type, source, and target. The graph transformation depicted in Figure 5.1 has two instance graphs corresponding to the left- and right-hand side. Their type graph is in fact the environment context of the eHome model, described in Figure 3.4.

There are two known problems related to application of the transformation rules. The first one is the *non-determinism of the rule application*. The left-hand side i.e. preconditions of the transformation rule can be matched with different sets of elements on the host graph[3]. The application of the transformation can yield in different results when applied on different part of the host graph, i.e. let m_1 and m_2 be different matching morphisms, then $G - p(m_1) \rightarrow G'$, and $G - p(m_2) \rightarrow G''$, and $G' \neq G''$.

The second problem is the replacement of substructures in an unknown context. It reflects the need for *universally quantified operations*. For example, to address the maximal set of all vertices of some type reachable over specified connections from fixed vertices in the rule – all locations the location element is connected to. In software engineering the problem is solved by implementing a so called *multi objects* concept known from UML object diagrams. This concept is implemented in the PROGRES [Sch91] framework but also in the Fujaba [KNNZ99] tool used in the eHomeConfigurator project (see Chapter 4) to design the eHome model.

5.2 Triple Graph Grammars (TGG)

Most of the text in this chapter relies on [Sch94, Pra71]. Schürr describes in the fundamental paper [Sch94] the concept of triple graph grammars. TGGs are invented to support the specification of interdependencies between graph-like data structures, to support development on generic implementation frameworks as well as to be the basis for development of incremental or batch-oriented data integration tools.

The motivation behind the research of triple graph grammars lies in the following draw-back of graph rewriting systems: graph rewriting systems are usually restricted to the specification of processes which perform in-place graph modifications and transform one instance of a class of graphs into another instance of the same class. This drawback hinders development of tools which check the consistency of simultaneously existing related data structures. Also, integration tools which take a complex source graph as an input and translate it into the new related target graph.

[3]The graph under transformation.

The example for related graph structures could be the requirements and design documents of the software system. Another example would be graphical specification of a function and the syntax tree of the corresponding function in the source code.

The triple graph grammars have evolved out of pair graph grammars [Pra71]. Pair graph grammars are used for tree-to-tree translations and are too limited because of restrictions to context-free productions and one-to-one correspondences between objects in related data structures. Triple graph grammars include context-sensitive productions with complex left- and right-hand sides.

The concept of triple graph grammars incorporates fine-grained m-to-n inter-graph relationships with the following characteristics:

1. Elements of one graph are related to distinct elements of another graph.

2. Correspondences between vertices and edges of the source and the target graph are at least 1-to-n. This means that arbitrary number m of nodes in source graph can correspond to an arbitrary number n of nodes in target graph.

3. Related graphs contain references as well as private elements. For example, some edges or vertices of one graph might have no references to the elements of the other graph.

In general, inter-graph relationships themselves carry information about on-going translation or analysis process. For example, there can be dependencies between inter-graph relations according to the order of the relations (one part of the graph was translated earlier as the other part) or validity constraints on relations (the relationship A is only valid if the relationship B exists). To express these kind of complex inter-graph relationships, they will be modelled as a separate correspondence graph with references to related source and target graphs by a morphisms between source and target graphs. Triple graph data integrators will be specified by the means of graph productions, which rewrite three graphs (the source, the correspondence and the target graph) in parallel, in sequel.

To define triple graph grammars formally, we have to start by giving the elementary definitions for graph, graph morphisms etc. The graph corresponds to the graph given in Definition 5.1 with one restriction. The graph has finite sets of nodes and vertices. The finiteness of the graph is required to guarantee termination of the possible translations.

Definition 5.3 *Graph Morphism*

> *Let* $\mathbf{G} := (V, E, s, t)$ *and* $\mathbf{G}' := (V', E', s', t')$ *be two graphs. A pair of functions* $h := (h_V, h_E)$ *with* $h_V : V \to V'$ *and* $h_E : E \to E'$ *is a* graph morphism *from* G *to* G', *i.e.* $h : G \to G'$, *iff* $\forall e \in E : h_V(s(e)) = s(h_E(e)) \wedge h_V(t(e)) = t(h_E(e))$.

Definition 5.4 *Graph Operators*

> *The* operators *"*$\subset$*" for "proper sub-graph", "*$\subseteq$*" for "sub-graph", "*$\cup$*" for "union of graphs with gluing of identified nodes and edges (nodes and*

edges with same identifiers)", "$\cap$", and "$\setminus$" etc. are defined as usual for graphs, and with $h : G \to G'$ being a morphism the $h(G) \subseteq G'$ denotes the sub-graph *in G', which is the image of h.*

Definition 5.5 *Monotonic Production*

Any duple of graphs $p := (L, R)$ with $L \subseteq R$ is a monotonic production *and p applied to a given graph G produces another graph $G' \supseteq G$, denoted by: $G \sim p \rightsquigarrow G'$, with respect to redex selecting morphisms $g : L \to G$ and $g' : R \to G'$, iff:*

1. *$g'|_L = g$, i.e. g and g' are identical mappings with respect to the left-hand side graph L.*
2. *g' maps new vertices and edges of $R \setminus L$ onto unique new vertices and edges of $G' \setminus G$.*

The monotonic productions define productions which only create graph structures and do not delete them. This is not a restriction to the translator development or TRIMoS framework as seen in Section 6.2.1. The TRIMoS approach applies also monotonic productions. After defining the fundamental terminology it is possible to define also the triple productions and their application on the graph triples.

Definition 5.6 *Graph Triple*

Let LG, RG, and CG be three graphs, and

$$lr : CG \to LG,\ rr : CG \to RG$$

are those morphisms which represent m-to-n relationships between the left-hand side graph LG and the right-hand side graph RG via the correspondence graph CG in the following way:

$$x \in LG \text{ is related to } y \in RG \iff \exists z \in CG : x = lr(z) \wedge rr(z) = y.$$

The resulting graph triple *is denoted as follows:*

$$GT := (LG \leftarrow lr - CG - rr \rightarrow RG).$$

Definition 5.7 *Triple Production*

Let $lp := (LL, LR)$, $rp := (RL, RR)$, and $cp := (CL, CR)$ be monotonic productions. Furthermore, $lh : CR \to LR$ and $rh : CR \to RR$ are graph morphisms such that their restrictions $lh|_{CL} : CL \to LL$ and $rh|_{CL} : CL \to RL$ are morphisms too, which relate the left- and right-hand sides of productions lp and rp via cp to each other. The resulting triple production *is denoted as follows:*

$$p := (lp \leftarrow lh - cp - rh \rightarrow rp).$$

The application of such a triple production p to a graph triple

$$GT := (LG \leftarrow lr - CG - rr \rightarrow RG)$$

produces another graph triple

$$GT' := (LG' \leftarrow lr' - CG' - rr' \rightarrow RG')$$

i.e.

$$GT \sim p \rightsquigarrow GT'$$

which is up to isomorphism uniquely defined. In sequel, there are applications of triple productions, where the redex or the result for their left- or right-hand side production application is already known in the form of morphism g. These restrictions are denoted for rewriting GT into GT′ by

$$GT \sim p(g) \rightsquigarrow GT'.$$

It is obvious that one triple production involves and replaces three rather similar conventional productions in the graphs under transformation. The triple productions in TRIMoS framework are called TRIMoS rules (see Section 6.2). Figure 5.2 shows a triple production where correspondence graph production has nodes drawn with dashed line. The left- and right-hand side graph productions are depicted respectively on the left and right of the correspondence graph production. The three productions are drawn in a way that the left-hand side graph is above the right-hand side graph. The triple production in the figure defines a relation between the eHome model and the JGraph structures. If the location element is added for the location, the corresponding JGraph structure is completed with structures for an edge and a graph cell. The definitions introduced so far enable to define triple graph grammars.

Definition 5.8 *Triple Graph Grammar (TGG)*

Triple graph grammar (TGG) *is a pair*

$$\mathcal{TGG} := \langle (p : (lp \leftarrow lh - cp - rh \rightarrow rp))_{p \in P}, GT_0 \rangle$$

where the first component is a family of triple productions indexed by production names in P and

$$GT_0 = (LG_0 \leftarrow lr_0 - CG_0 - rr_0 \rightarrow RG_0)$$

is the start graph triple*.*

The *TGG* specifies a language of graph triples and is composed of triple productions, where each production component is responsible for generating or extending the corresponding graph in the graph triple. The set of terminal labels is omitted to identify the graphs belonging to the generated language since the graph languages are not the focus of this thesis. The TGG can be considered as a translator, for example translating the eHome model instance's environment context into JGraph structures[4].

[4]Section 6.1 presents a TGG as a translator specified by TRIMoS rule set.

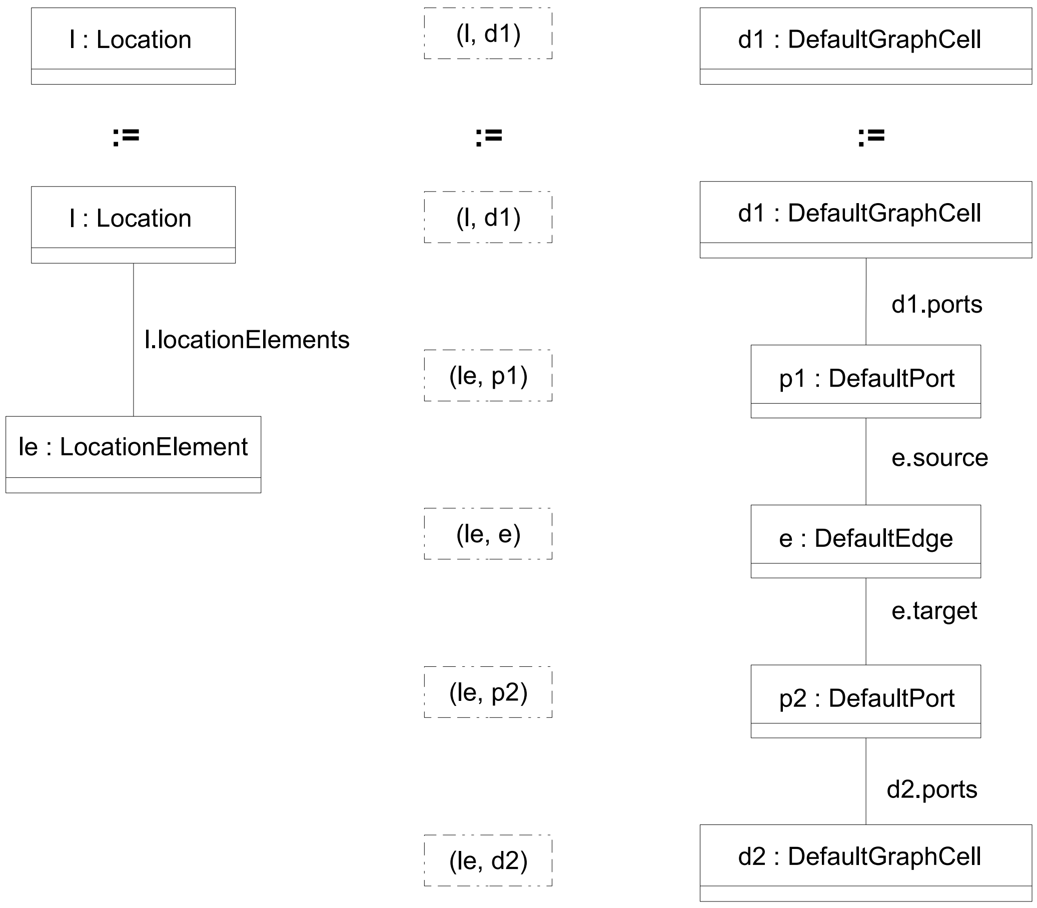


Figure 5.2: A triple production. Left- and right-hand side graph productions are separated and related with the correspondence graph production in the middle.

Specification of triple graph grammar productions can be used to develop different tools: LR-translators, which take according to the triple graph productions left-hand side graph structure as an input and return a corresponding right-hand side graph structure; RL-translators, which do the vice versa; correspondence analysers, which monitor the relationships between a given graph structures; and synchronisation tools which keep left-hand side and right-hand side graphs in sync.

Developing a LR-translator tool is a rather difficult task. In general it requires a graph parser for context sensitive productions, which is able to recover a sequence of production applications yielding a given source graph. Development of corresponding RL-translator is a nearly identical task, because of the symmetrical nature of relations. Therefore we can speak about bidirectional transformation, analysis and synchronisation process.

In Schürr's paper [Sch94], there are only monotonic productions covered – any production's left side is part of the production's right side. This should simplify the development of LR- or RL-translators considerably. It is also claimed that triple graph grammars are not intended to model editing processes on related graphs, but are a generative description of graph languages and their relationships. The paper provides also a proof for two essential results. The first result states that every triple production can be separated into a left-local triple production which rewrites the left-hand side graph only, and a left-to-right translating triple production which keeps the left-hand side graph unmodified and adjusts the correspondence and the right-hand side graph. The second result states that the application of a sequence of triple productions is equivalent to the application of the corresponding sequence of left-local productions followed by the sequence of left-to-right transformations. We present these two results here.

Proposition 5.1 *LR-Splitting of Production Triples*

A given production triple

$$p := ((LL, LR) \leftarrow lh - (CL, CR) - rh \rightarrow (RL, RR))$$

may be split into the following pair of equivalent production triples:

1. $p_L := ((LL, LR) \leftarrow \varepsilon - (\emptyset, \emptyset) - \varepsilon \rightarrow (\emptyset, \emptyset))$ *is the* left-local production *for* p*, where* $\emptyset$ *is the empty graph and* ε *is an inclusion of the empty graph into any graph.*
2. $p_{LR} := ((LR, LR) \leftarrow lh - (CL, CR) - rh \rightarrow (RL, RR))$ *is the* left-to-right translating production *for* p*.*

For these production triples and any graph triples

$$GT := (LG \leftarrow lr - CG - rr \rightarrow RG),$$

$$GT' := (LG' \leftarrow lr' - CG' - rr' \rightarrow RG'),$$

and a morphism $lg' : LR \rightarrow LG'$ *the following proposition holds:*

$$GT \sim p(lg') \rightsquigarrow GT' \iff$$

$$\iff \exists HT : GT \sim p_L(lg') \rightsquigarrow HT \wedge HT \sim p_{LR}(lg') \rightsquigarrow GT'.$$

Proposition 5.2 *Permutation of Left-Local and Left-to-Right Productions*

Given n *production triples* $p^1 \dots p^n$ *and morphisms* $lg^1 \dots lg^n$ *which determine the application results of the left-hand side production components* $p^1 \dots p^n$*. We can prove that:*

$$p^1(lg^1) \circ \dots \circ p^n(lg^n) =$$

$$= (p^1_L(lg^1) \circ \dots \circ p^n_L(lg^n)) \circ (p^1_{LR}(lg^1) \circ \dots \circ p^n_{LR}(lg^n)).$$

These results are particularly important because they prove that we can construct a LR-translator which determines a sequence of applicable left-local productions, which are basically simple graph productions to produce given source graph. After that, the translator also generates a target graph using the corresponding sequence of left-to-right transformations. The resulting right side graph obtained in this kind of process is equivalent to the right side graph obtained during the application of triple productions in incremental synchronisation process between the source and target graph.

In general, the TGG triple productions, i.e. TGG transformation rules and relations between the left- and right hand side has to be defined explicitly by the person specifying the rules. In other words, the morphism relating the right- and left-hand side productions of the triple production (see Definition 5.7) have to be defined explicitly by the developer. In the next chapter we will introduce another TGG approach introduced in the scope of this thesis. The TRIMoS approach has simpler TGG transformation rules without explicit correspondence graph and no need for the specification of the relating morphisms. The TRIMoS framework is developed to solve the problems indicated in Section 4.4. Its theoretic concepts are given in the Chapter 6 and the implementation of the TRIMoS framework will be discussed in Chapter 7.

Chapter 6

TRIMoS TGG Approach

We introduced the translator development problem in Sections 2.2 and 4.4. The problem states that the development and maintenance of translators to transform the eHome model into the structures of other technologies is laborious and error-prone. This statement is supported by the personal experience of the author of this thesis, who has been developing and maintaining the translators for eHomeConfigurator project during the past two years. In this chapter, we will handle an approach, which introduces an advanced solution for the translator development.

The approach is called *Transformation Rules for Incremental Model Synchronization* (TRIMoS). This is a triple graph grammar based translator and synchronisation framework. This approach introduces a new type of triple graph transformation rules and covers also the implementation of the supporting framework. The TRIMoS development was started at the University of Kassel. In the beginning it was a small interpretor for simple triple graph grammar rules. TRIMoS was able to produce and synchronize simple tree-like structures related to each other. It did not provide the deletion of elements from the produced structures. It was also not able to synchronise the related structures in respect of deletion and could not handle graphs in general.

This thesis takes the TRIMoS prototype as a basis for further research and development. The TRIMoS rules are improved by the new rule element type called *optional create* (see Section 6.2). Also the bugs the implementation's code are fixed. It is completed to support the new rule type and to support complex graph structures. For example, TRIMoS will be able to support the synchronisation of the eHome model instance's environment context with JGraph structures. Thereby, the environment context can hold arbitrary graphs consisting of locations, sub-locations and environment elements. The implementation is also completed with the mechanism responsible for the deletion of elements in synchronized models and an adapter API for the JGraph API (see Chapter 7).

In contrast to many other batch-oriented or incremental approaches the TRIMoS approach is a reactive TGG approach. The specified translators are not only used to produce the corresponding graphs for the input graph like in batch-oriented approaches, but are also used to keep the related graphs in sync during runtime. Every change in a graph on one side of the relation is answered immediately with corresponding changes on the other side. This approach is thus particularly useful for application development of GUIs.

The entire research and development of the TRIMoS system has been motivated and performed according to the requirements from application development, including the requirements originating from the eHomeConfigurator project. We will give the details of the motivation in the next section. In the following sections we will introduce the TRIMoS transformation rules, the semantics and the runtime execution of the rules.

6.1 Motivation

The main motivation for TRIMoS development is to be able to specify and not hand-code the translators (see the development problem in Section 4.4) transforming one model to another or keeping the two models in sync. A *TRIMoS translator* is specified with a set of TRIMoS-specific triple graph grammar rules also known as TRIMoS rules. The TRIMoS interprets the rule set during the runtime of the system monitoring the models, the correspondence of which is defined by the rules. In the case of any changes in the synchronized models, TRIMoS propagates the changes according to the rules to the other side. Chapter 7 describes how this propagation is done.

TRIMoS rules are a restricted form of the TGG rules. The TRIMoS rules omit the correspondence graph from the TGG rules. The correspondence graph is modelled implicitly instead. This form of a TGG rule is motivated by the need for simpler rules than the traditional TGG transformation rules. The simpler form is also justified by the fact that different APIs used in application development produce similar structures. The TRIMoS rules are discussed in Section 6.2.

The eHomeConfigurator was developed using Java technologies. An alternative for TRIMoS development to solve the translator problem, was to use PROGRES [Sch91] in combination with the Upgrade framework [Jäg00]. Unfortunately, this combination was not compatible with chosen Java technologies. The combination of PROGRES and Upgrade was also not sufficient for the visualisation and layout needs on eHomeConfigurator's GUI. The Upgrade framework is not able to set an aggregation of the nodes of one graph to correspondence with the nodes of the other graph[1]. Additionally, if using Upgrade, the developer has to either configure given filters or to program the special un-parsers if the filters do not manage to visualize the graphs[2] specified in PROGRES [Sch91].

6.2 TRIMoS TGG Transformation Rules

The TRIMoS framework uses TGG transformation rules to specify the translation and synchronisation mechanism, keeping the left- and the right-hand side graphs (see Definition 5.6) in sync. We will consider the terms graph and model equivalent for our approach. The graphs we are dealing with are object models constructed during the runtime. Moreover, the type graphs are modelled with UML class diagrams and

[1]This can be solved by implementing a special aggregation node into the model.

[2]For example, in the case of special traversals.

instance graphs correspondingly with UML object diagrams. The object diagram notation is used for TRIMoS transformation rules (shortly, TRIMoS rules). There are additionally more principal differences between TRIMoS rules and classical triple productions with the following distinctions:

1. The graph productions in the TRIMoS rules are not modelled like the classical graph productions (see Figure 5.1), but the left- and the right-hand side of the production are presented as one graph with stereotypes for the nodes and edges (see Section 6.2.3).
2. The correspondence graph production is not modelled explicitly in the TRIMoS rules, but handled implicitly during runtime of the TRIMoS framework.

In the following sections the two differences are described in more detail.

6.2.1 Compound Productions

The pre- and postconditions (the left-hand and the right-hand side) of the related graph productions in the TRIMoS rules are compound into one graph. This means that the left- and the right-hand side graphs are joined into one graph to present a transformation. The pre- and postconditions are not modelled as separate graphs like in Figure 5.1. Figure 6.1 shows a *compound graph production* where the left- and the right-hand side graphs are joined into one graph. This graph carries the information needed to perform the graph transformations. Compound productions have a more compact and intuitive form in terms of UML object diagrams, when compared with classical graph productions. The differentiation between the pre- and postconditions in the compound production is done using the stereotypes known from the UML object diagrams.

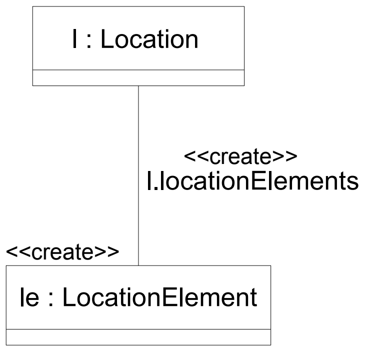


Figure 6.1: A compound graph production, the left- and the right-hand side graphs are joined into one graph.

The objects with no marking are considered to be pre-conditions (the left-hand side) of the production. The post-conditions (the right-hand side) of the production are marked with the stereotype `create`. The node *l* : *Location* on the figure is the only object in precondition graph. The edge *l.locationElements* and node *le* : *LocationElement* are the postconditions of the production.

This notion is also known from the Fujaba story diagrams. The story patterns being also graph productions contain additionally to the `create` stereotype the `destroy` stereotype. The `destroy` stereotype expresses the deletion of graph elements. The compound productions used in TRIMoS rules are *monotonic productions* (see Definition 5.5). Due to this fact, the TRIMoS rules do not deal with deletions like productions used in classical triple graph grammar rules (see Definition 5.7). Nevertheless the deletion of model elements is handled in our approach (see Section 7.5).

6.2.2 Implicit Correspondence Between the Side Models

The rule sets modelled in TRIMoS resemble pair grammars, since the correspondence graph production is not modelled explicitly. As mentioned in Chapter 5 the correspondence relation between the left- and the right-hand side graph production in TGG transformation rules has to be defined explicitly by the developer. In other words, the morphism relating the left- and the right-hand side graphs of the triple production has to be defined by hand. The TRIMoS approach has simpler TGG transformation rules without a correspondence production. There is no need for the specification of the correspondence graph transformation and relating morphisms. This is achieved by restricting the correspondence between the left- and the right-hand side transformation rules to one special case. We relate the elements of one production to the other in correspondence to the stereotypes. The preconditions of the left-hand side composite production are related to preconditions of the right-hand side composite production. The postconditions of the two productions are related respectively.

The example in Figure 6.2 shows a TRIMoS rule, which is basically the same rule as in Figure 5.2 but with two differences. First, the left- and the right-hand side productions are composite productions. Second, the correspondence graph production is not specified explicitly. Nodes (objects) with the same stereotype are related with all nodes of the same stereotype on the other side. This means that object l is related to $d1$ and objects le is related to $p1$, e, $p2$, and $d2$.

Figure 6.2 shows also attributes at the objects. For example, the object le : *LocationElement* has an attribute *name* : *String* with correspondence to the type graph in Figure 3.4. This TRIMoS rule also states that the attribute *userObject* : *Object* of the object $d2$: $DefaultGraphCell$ must have the same value $newName$ as the *name* attribute of the object le : *LocationElement*. This relation is indicated with the variable $newName$ in this TRIMoS rule. This means that TRIMoS rules relate not only nodes but also attributes supporting the change propagation between attribute values.

In fact, when considering the TRIMoS rules, we see that because of the UML object diagram notion, the edges in the diagrams are also represented by attributes in real object model structures. In other words the relations between the objects are realized using attribute values during runtime.

For example, figures 6.1 and 6.2 both illustrate the situation where the l : *Location* object has an attribute *locationElements* assigned with value le : *Location-Element*. The *locationElements* edge in the rule corresponds to the `contains` relation between the `Location` and the `LocationElement` class in type graph seen in Figure 3.4. The type graph states that `Location` object can have 0 to n `Lo-`

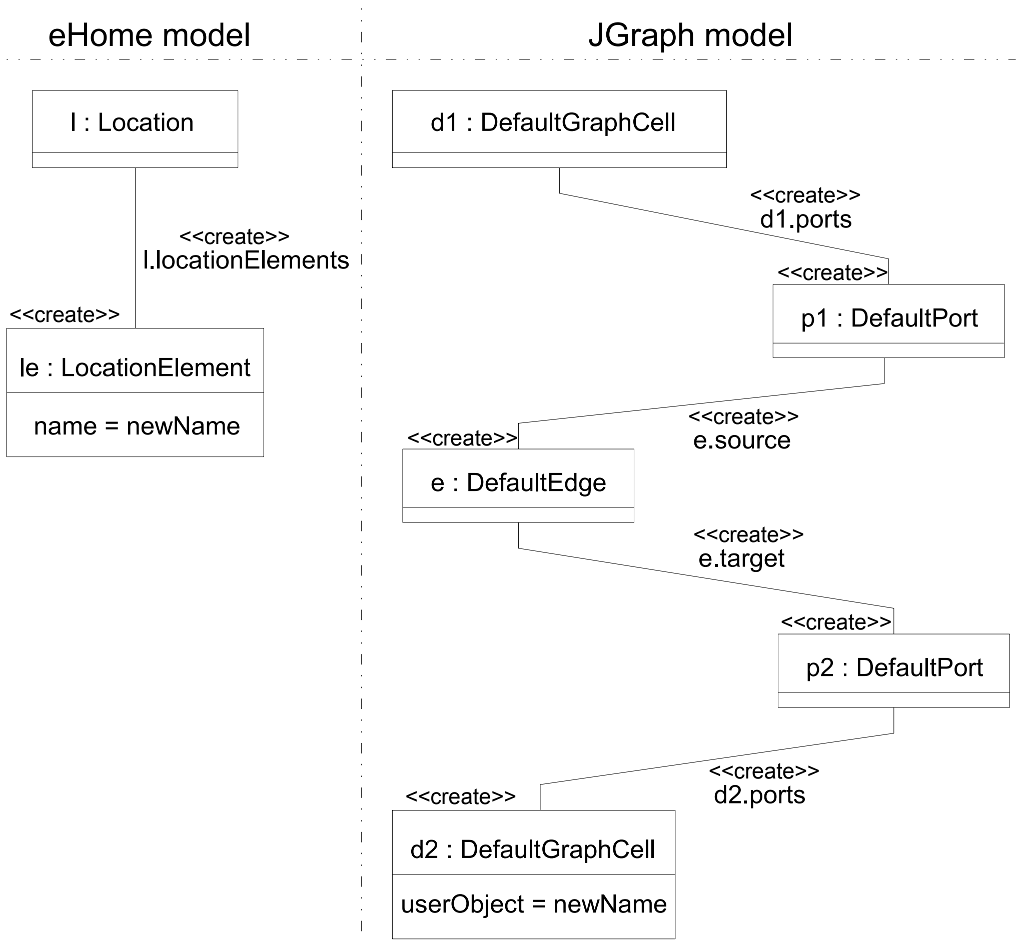


Figure 6.2: A TRIMoS rule to create a location element and corresponding JGraph structures.

`cationElement` objects as attributes via the `contains` relation. The equivalence between edge and attribute concepts enables for the TRIMoS implementation to set the correspondence relation only between the objects of the synchronized models. This also applies for the rule objects in compound productions in TRIMoS rules.

6.2.3 Rule Element Stereotypes

We already introduced the stereotype for postconditions and discussed the preconditions of the compound transformations. The preconditions are not denoted by any stereotype and the postconditions are denoted by the *create stereotype* (in figures `create`). There is also another stereotype in addition to the `create`.

The second stereotype is called *optional create* and it is denoted on the diagrams as `optional`. Figure 6.3 depicts the composite production known from Figure 6.1 and changes it by setting the stereotype `optional` for the object *le : LocationElement*. Thus, we have three types of rule elements: preconditions, postconditions (`create`), and optional create elements (`optional`). The element types apply to the objects (nodes) and as well to the attributes (edges, and attributes).

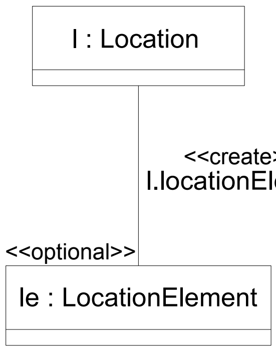


Figure 6.3: A composite production to create a location element with optional create stereotype.

The optional create stereotype has quite the same purpose and semantic as the multi objects (see Section 5.1) in the traditional graph grammar approach. This stereotype deals with the need for universally quantified operations. For example, the operation acquiring all `Location` objects a `LocationElement` object is connected to. This means in our case dealing with relations between classes with constraints `0..n` or `1..n` on both ends of the relation. The idea of the optional create (see also Section 7.4) is to model an element, which must be considered to be a part of preconditions, if it already exists in the model (is matched during precondition search) or which must be created, if it does not exist in the graph (not found for the match).

For example, the environment context of the eHome model as a type graph (see Figure 3.4) specifies the relation `contains` between the `Location` and `LocationElement` class. The relation is specified with cardinalities `1..n` for locations and `0..n` for location elements. `Location` objects can have an arbitrary number of `LocationElement` objects related to them and `LocationElements` are always connected to at least one `Location` object.

The transformation without optional create stereotype for `LocationElement` object in Figure 6.1 respects the constraint `0..n` for `LocationElement` objects. According to this transformation, it is possible to create arbitrary number of `LocationElement` objects in relation with one `Location` object. This transformation does not act correctly in the TRIMoS rule, if a `LocationElement` object is related to more than one `Location` object. In other words, if an existing `LocationElement` object with connections to `Location` objects will be connected with some other `Location` object, only the link between the objects has to be created.

The TRIMoS rule without optional create stereotype for `LocationElement` object in Figure 6.2 performs in synchronisation process as following. Every time an object from the `Location` class is set to relation with `LocationElement` object, the corresponding structures to postconditions are generated into the related JGraph model. The new JGraph edge and cell structures[3] corresponding to the post conditions are generated even, if an already existing `LocationElement` object is related to a `Location` object. This makes the related structures inconsistent.

The transformation including the optional create stereotype in Figure 6.3 solves this problem. It has the following semantic. If the object corresponding to the optional create rule element does not exist in the model (it is not found during the match search for preconditions), the rule element is handled as a part of the postconditions (the object corresponding to the element is created into the model). If the object exists in the graph (it is found during the match search for preconditions), the rule element is handled as a part of the preconditions (the object is not created to the model). The optional create stereotype is handled in detail in Section 7.4.

Thus, if the TRIMoS rule in Figure 6.2 is changed in a way that the objects $le : LocationElement$ and $d2 : DefaultGraphCell$ are specified with an optional create stereotype (as in Figure A.5), the situation described in the last paragraph will have different results. In the case an existing `LocationElement` object is related with an existing `Location` object only the objects corresponding to the postconditions (corresponding to rule elements $p1$, e, and $p2$) are created into the JGraph model.

6.2.4 Initialisation Rule

There is one special rule in TRIMoS. According to the underlying theory, TGGs consist of the set of triple productions and an initial graph (see Definition 5.8). This also applies for the TRIMoS approach. The *TRIMoS rule set* defining the *translator* can be viewed as a *TGG*, but the initial graph is defined with the help of one special rule in the grammar. We call this rule initialisation rule and it consists only of postcondition objects, i.e. objects with the create stereotype. During the start-up of the framework, the rule set is read and the initialisation rule is executed to create the initial graph. For example, the initialisation rule for the environment editor of the eHomeConfigurator tool only consist of an `Environment` class object and the `DefaultGraphCell` object. Both objects have the stereotype `create` as shown in Figure 6.4. The rules consisting of only postconditions are normally not interpreted by the TRIMoS framework.

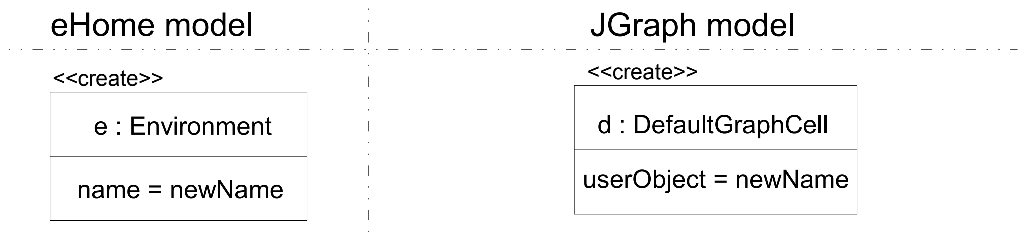


Figure 6.4: An initialisation rule for the environment editor of the eHomeConfigurator tool.

The implementation of TRIMoS can have also the objects corresponding to the initialisation rule as the input. In this way the objects are not created but just linked

[3]The JGraph API is explained in Section 7.7.1

with the correspondence graph. This is an option to link the TRIMoS framework with an application, which uses translators specified with TRIMoS rules. In this case TRIMoS considers particular objects from the application as the initial models to monitor and synchronise.

6.2.5 Semantics

The semantics of the TRIMoS rules depend on the local context of the rule application. Existing objects in the model can be matched with postconditions of different productions. The execution of the context for the TRIMoS rule is the precondition match. There are approaches like in [Wag01], which do not allow this kind of semantics, stating that every model object can be involved with postconditions of only one production. In other works an object in the model can be produced only by one production. We apply this kind of semantic because of the reactive nature of the TRIMoS framework.

Our semantics yields a smaller overhead for the rule matching task. We do not have to check if the object has already been used as a part of any production. We also have the flexibility to use different productions with intersecting contexts.

With the semantics introduced in [Wag01], there is no need for the optional create stereotype. The problem introduced in Section 6.2.3 can be solved with two productions: the production p with only pre- and postconditions (as in Figure 6.1) and production p' having un-connected preconditions as objects $l : Location$ and $le : LocationElement$. The link $l.locationElements$ is the only element in postconditions. The latter production is used in the case a new link is introduced between a given `Location` and `LocationElements` objects.

Considering the semantics of TRIMoS, these kind of rules result in a conflict. The first rule p described in Figure 6.1 is also applicable for the change if a new link is introduced between given `Location` and `LocationElements` objects. The context for the production p is the only match found for preconditions. The optional create stereotype solves this conflict. The two productions p and p' are described with only one production shown in Figure 6.3 involving the optional create stereotype. This production renders the specification of the other two productions useless by combining the semantics of both productions. Thus, the rules become more comprehensive and intuitive with an optional create stereotype.

6.2.6 Runtime Execution of the TRIMoS Rules

The TRIMoS framework can be used by application developers to create bidirectional translators by linking it with their application. In this case TRIMoS is started together with the application and will be working as a translator interpreting the TRIMoS rules specified by the application developer. The framework loads the rule set and the initial models for the left- and the right-hand sides. The developer can specify different rule sets for distinct translators.

The TRIMoS framework is developed for reactive synchronisation and synchronized models are monitored by the framework. Thus, any change in the models is discovered and localized immediately after the change appears. For every change, it is checked if it corresponds to some graph production in the TRIMoS rule set applicable for the changed model. In other words it is checked, if the change fulfils the match for the pre- and postconditions of some production applicable for the rule side, where the change appeared. With respect to the production, if the pre- and postcondition match is found for graph transformation, the corresponding generating production is executed on the other related model to simulate the change on the other side. We must emphasize the bi-directionality of the TRIMoS framework – it does not matter if the change appears on the left- or the right-hand side model.

Besides the synchronisation of the immediate changes, TRIMoS is also able to generate structures corresponding to application of the sequence of TRIMoS rules. This is the case if a larger structure is added to the model in one side and the structure corresponds to derivation of more than one production application. Therefore we can say that besides the reactive properties the TRIMoS framework has also properties known from batch-oriented TGG translators.

We see that the translational nature of the TRIMoS framework corresponds to the LR- or RL translators known from the classical TGG approach (see Section 5.2). The framework considers according to Proposition 5.1 the change on the left- or right-hand side model and then propagates the transformation using corresponding TRIMoS rule to the other side.

The implementation of TRIMoS does not use the left-local and the left-to-right production permutation results known from Proposition 5.2. The TRIMoS works in a so called rule-by-rule mode considering one rule after another in either of the two cases: translating larger structures, or synchronizing changes in the scope of one TRIMoS rule.

6.3 Summary

In this chapter, we introduced an approach for the translation and synchronisation framework called TRIMoS. This approach relies on triple graph grammar theory, but introduces restrictions and differences when compared to the classical TGG approach. The graph productions are modelled differently using just one graph and stereotypes and not two graphs for specification of the pre- and post-conditions of the production. Moreover, the TRIMoS transformation rules omit the correspondence production in the triple productions by modelling it implicitly. In the next chapter we will introduce the implementation of the TRIMoS framework and also show how it is used to create bidirectional translators between the eHome model and visualizing JGraph model.

Chapter 7

TRIMoS Implementation

In this chapter we discuss the implementation of the TRIMoS framework in detail. We will introduce the structure of the framework and principles how it performs the translation and synchronisation tasks. We will also discuss the matching algorithm. We will describe in detail implementations for: disjoint preconditions, the optional create stereotype, and the deletion of the model elements. Finally we will introduce the adapted JGraph API as an example how to adapt a programming API to be used with the TRIMoS framework.

7.1 Framework

The TRIMoS framework is used to specify bidirectional translators for object models with the help of TRIMoS rule sets. A rule set is interpreted during the runtime of the framework. It means that an application developer has to link the application with the TRIMoS framework to be able to use it. The framework should be used, if there is a need to translate structures of the given object model to the structures of some other object model. The programming APIs must be known for both models. As an example, we use in our research the eHome model API and the JGraph API.

APIs can be considered as type graphs for object models. They define the types for corresponding objects, the relations between the objects and how the object models are created during runtime. It means that the rule set defining a bidirectional translator for two different models such as the eHome model and the JGraph model, has to correspond to the programming APIs of these models. More precisely, the left- and the right-hand sides of the TRIMoS rules have to follow the corresponding APIs.

The TRIMoS framework is still a project under development. It is a working prototype, which has a perspective to become a powerful and easy to use development tool for generic translators between arbitrary Java APIs. It consists of the synchronisation core and a simple editor for the TRIMoS rules. The editor is shown in Figure 7.1. The editor helps to define a set of TRIMoS rules for a translator. The rules in set are seen on the panel on the left as a list. On the right panel side, there is a graphical editor with two panels corresponding to the left- and the right-hand side of the TRIMoS rule. The figure shows the rule for creating a location element for a location. The stereotypes of the rule elements are noted with different colours: black for preconditions, green for postconditions (create stereotype), and blue for optional create stereotype. In the figures of this thesis the stereotype notations `create` and `optional` will be used to indicate the stereotypes.

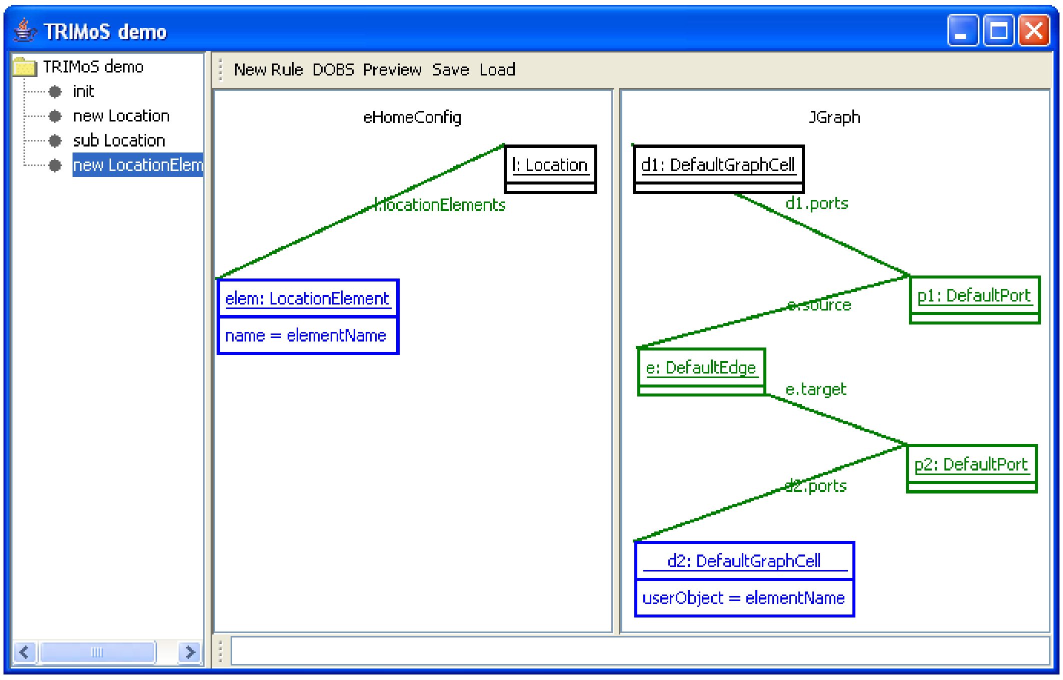


Figure 7.1: A simple editor for TRIMoS rules.

7.1.1 The Design of the Framework

The TRIMoS rule editor helps to specify a set of rules for a translator. During the start-up of the framework this set is loaded and interpreted during runtime of the framework. This subsection gives an overview of the architecture of the TRIMoS framework. The internal representation of the rules, but also the structures supporting the translation and synchronisation process are discussed. The inner data structures of the TRIMoS framework are designed with the Fujaba tool. Figure 7.2 shows the design of the TRIMoS framework.

The first part of the TRIMoS design are the necessary classes for the TRIMoS rules. The main class summarizing the common properties of the rule elements (rule objects) is `TItem` (see Figure 7.2). This class contains the attribute for the rule element's name and boolean flags signalling, if the rule object is a precondition or has the stereotype optional create. The TRIMoS rule itself is represented with the `TRule` class. The rule side is represented with the `TSide` class. One rule can have one side per synchronized model. The rule side consists of the rule elements denoted by the `TObject` class. Every rule object in the rule side has to have a distinctive name. Rule elements have also an attribute indicating what is the type (the class name) of the modelled object. There is also a `TAttribute` class modelling the attributes of the objects. This class is related to the `TObject` class and has also the field to indicate the value of the modelled attribute.

As discussed in Section 6.2.2 also the edges in the object graph, i.e. relations in the object model are modelled with attributes. If the attribute type is primitive, for example, an integer or a string, it appears on the object node it belongs to as a string pair $attrName = value$ in the rule editor. The $attrName$ is the attribute name and the $value$ the attribute value. If the attribute value is another object in the diagram, it appears as an edge between the object it belongs to and object, which is the value of the attribute. The edge is labelled $o.attrName$ in this case, where o is the object name the attribute belongs to and the $attrName$ the attribute name. For example, see Figure 7.1. The object $elem$: $LocationElement$ has the attribute $name$ with value $elementName$ and the object l : $Location$ has the attribute $locationElements$ with value $elem$.

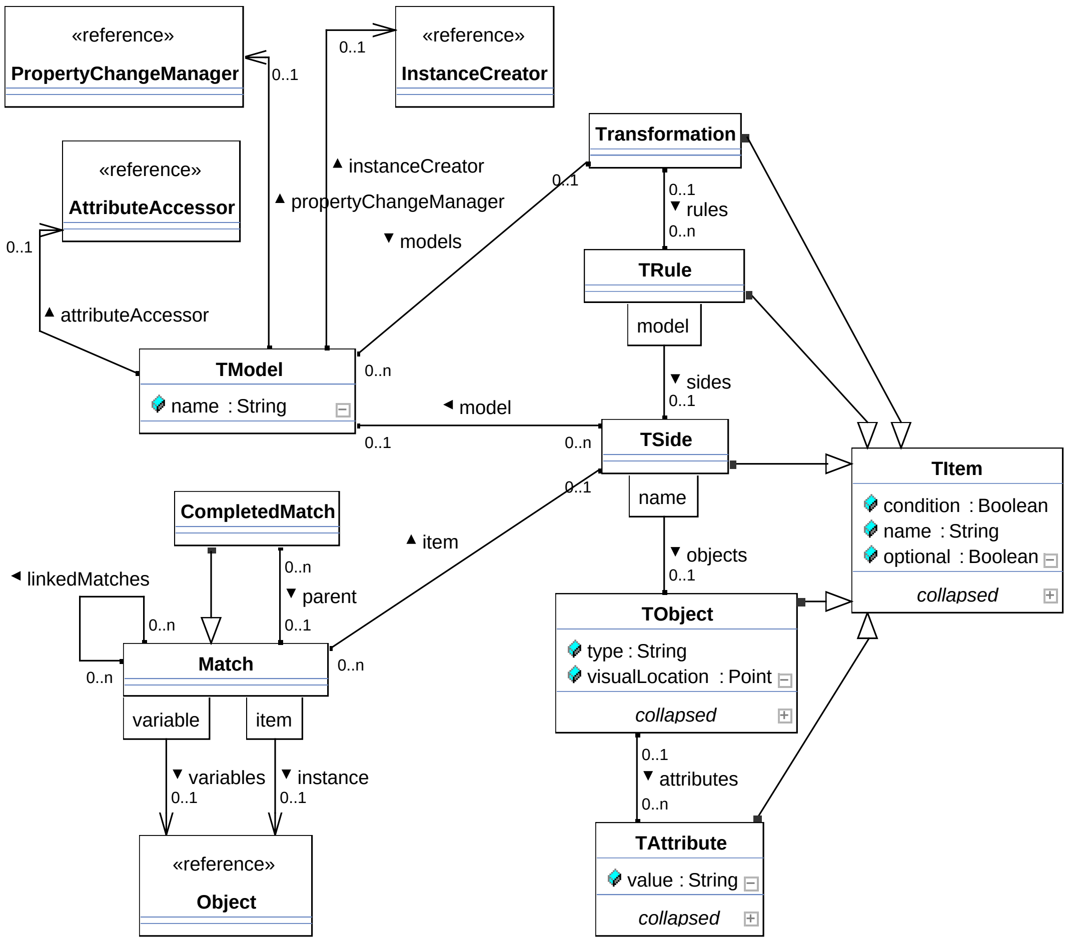


Figure 7.2: A design of the TRIMoS framework.

The rule set is modelled with the `Transformation` class. It has a relation to the `TModel` class presenting the synchronised models. A `TModel` instance aggregates all rule sides, i.e. the compound transformations which concern the model represented by the `TModel` object.

The `TModel` class also deals with the runtime structures of synchronised object models during runtime. There are three interfaces related to it: `AttributeAccessor`, `PropertyChangeManager`, and `InstanceCreator`. There interfaces are used by the TRIMoS framework to monitor or transform the synchronised models. These interfaces have to be implemented or the default implementations of these interfaces have to be sub-classed, if TRIMoS is used with an API, which does not correspond to the JavaBeans Specification [Sun97][1]. An example is given in Section 7.7, where the JGraph API is adapted to work with TRIMoS. If the synchronized API follows the JavaBeans specification, the default implementations of the interfaces can be used. These are: `DefaultAttributeAccessor`, `DefaultPropertyChangeManager`, and `DefaultInstanceCreator`.

The `AttributeAccessor` interface is used to read and write field values of the model objects. For example, in the case of checking the type validity or creation of objects. The implementing classes use Java Reflection API. The `PropertyChangeManager` manages the property change listeners used to monitor the synchronised models by assigning the listener directly to the model or delegating the model change events to the listeners. The `InstanceCreator` interface is responsible for creating and destroying the objects in the model. For example, in case of a model change on the left-hand side model, it has to be propagated to the right-hand side model. This change includes the creation or removal of objects.

Figure 7.2 also shows the classes `Match` and `CompletedMatch`. These classes have a special role in the synchronisation mechanism and we will refer to the objects from these classes in the text as *match* and respectively *completed match*. The match and completed match are objects forming the correspondence graph between the left- and the right-hand side models. The correspondence graph is constructed only during the runtime of the framework. The `Match` objects are used to match the objects corresponding to preconditions of the productions. The `CompletedMatch` objects match the objects corresponding to postconditions of the production application. The term “match matches some model object(s)” means that the match has a link to these model objects. The correspondence graph creation according to the rules is described in Section 7.1.2.

The names for the match and completed match objects are chosen with purpose. The match corresponds to the match morphism between the graph production and graph. The completed match corresponds to the elements matching the postconditions of the graph production, but expresses also the completion of the model with new objects according to the graph production.

Every match has a relation to some rule side and every match can have *linked matches* for the other side models. This applies to matches and completed matches because the `CompletedMatch` is a sub-class of the `Match` class. This results in structures where the corresponding objects in the left- and the right-hand side models are related via the chain of two matches – a match per side (see Section 7.1.2). There is also a relation between the matches and the completed matches. The matches are

[1]The accessor methods and property change listener conventions.

considered to be *parent matches* for completed matches, because matches match the preconditions of the TRIMoS rule.

Figure 7.2 does not include all important classes for the framework structure. There are classes for synchronisation management (`SyncManager`), match listeners (`MatchListener`), and matching algorithms (`Matching`). These classes are not designed with Fujaba.

Synchronisation manager is responsible for the synchronisation between models. It manages the match objects in the correspondence graph having a register for matches. Every match is registered in the synchronisation manager together with a *match listener*. The Match listener has an objective to monitor the model objects related to the match. The monitoring is done using the `PropertyChangeManager` or directly the model objects. The model objects are monitored directly if the `PropertyChangeManager` is able to set the match listener to listen directly to the model objects.

If a change appears on the object under the inspection of a match listener, the match listener triggers the matching algorithm in the `Matching` class to discover if the change corresponds to some rule. The *matching algorithm* (see Section 7.2) performs the searches using the match objects between synchronized models. If the change corresponds to some rule, the other side models are completed with the corresponding structures using again methods in the `Matching` class. These completion methods use completed match objects on the changed side to create corresponding completed match objects for the other side model. The completed matches are also used to create the objects for the other side model. The examples for the matching structures are given in the next section.

A Translator in the TRIMoS framework is a set of TRIMoS rules. Every TRIMoS rule consists of two graph productions for the left- and the right-hand side model. Thus, if a change appears on one side model, it is possible to correspond with a change on the other side model. But in addition to the synchronisation process, the rule sides also define the construction process for the corresponding side model. At least on the model parts, which are synchronized.

Every object in the synchronized model parts is monitored by some match listener. This is because, the translators transforming one model to another are defined in the way that they also cover the construction process of the synchronized parts of the models[2]. The initialisation rule (see the example on Figure 6.4) creates the initial models on both sides along with the match objects between them (see Section 7.1.2). If every object or unitary structure added to the model will be created in correspondence to some TRIMoS rule, the corresponding matches are created as well. Thus, the synchronized models are monitored completely. The next section gives an overview on the correspondence graph structures consisting of match objects during the runtime of the framework.

[2]It is usual that the entire model is synchronised

7.1.2 The Correspondence Graph During Runtime of the Framework

The correspondence graph is in between the synchronized left- and right-hand side models and is created along with the structures of these models. The example in this section will be the constructed in correspondence to the rule set for the environment editor in the eHomeConfigurator tool. The rule set is given in appendix A.

As noted in Section 6.2.4 the initialisation rule is executed at the start-up of the framework. The initialisation rule for the environment editor is given in Figure 6.4. Corresponding to the rule, there are an `Environment` and a `DefaultGraphCell` object created. Both objects are noted as postconditions in the rule. Therefore, they are matched with completed match objects. The resulting structures of the initialisation rule execution can be seen in Figure 7.3. The figure displays the structure constructed during the eHome model changes corresponding to the sequential execution of the initialisation (Figure 6.4), location creation (Figure A.3), and location element creation (Figure A.5) rules. The resulting structure is divided into three parts by the dashed line in the figure: part a) corresponds to the initialisation rule, part b) to the location creation rule, and part c) to the location element creation rule.

The initialisation rule execution creates the objects $e : Environment$ on the left-hand side; $d : DefaultGraphCell$ on the right-hand side; and $m1 : Match$, $m2 : Match$, $c1 : CompletedMatch$, and $c2 : CompletedMatch$ to the correspondence graph[3]. The side model objects are matched with completed match objects to indicate the correspondence to postconditions of the applied rule. The correspondence relation between the objects e and d is modelled with the linked completed matches $c1$ and $c2$. There are also match listeners registered together with the completed matches $c1$ and $c2$ to monitor the property changes on the respective objects e and d.

After initialisation, a location l is created and connected to the environment e. This action fires the property change event on e, which is received by the corresponding match listener related to the completed match $c1$. The match listener triggers the matching algorithm (see Section 7.2). The matching algorithm discovers that the change in the left-hand side model corresponds to the location creation rule (see Figure A.3). The change is matched by a match $m3$ and completed match $c3$. The $m3$ matches the environment object e as a precondition for the rule application and $c3$ matches the created location object l as a postcondition for the rule application.

Now, the corresponding matches and objects are created for the right-hand side JGraph model. The search for preconditions in the right-hand side is performed using the completed match objects $c1$ and $c2$. The graph cell object d is found and matched with a new precondition match $m4$. Then the $m4$ is completed by creating a completed match $c4$ as its child match. The JGraph objects $p1$, $e1$, $p2$, and $d2$ are created during completion. The created JGraph structures correspond to the right-hand side of the location creation rule (see Figure A.3).

A location element le is created as the next object into the eHome model. It is connected to the location l. For le similar actions take place like during the location

[3]In the following text we will omit the type definitions for the objects and refer to them using just the name.

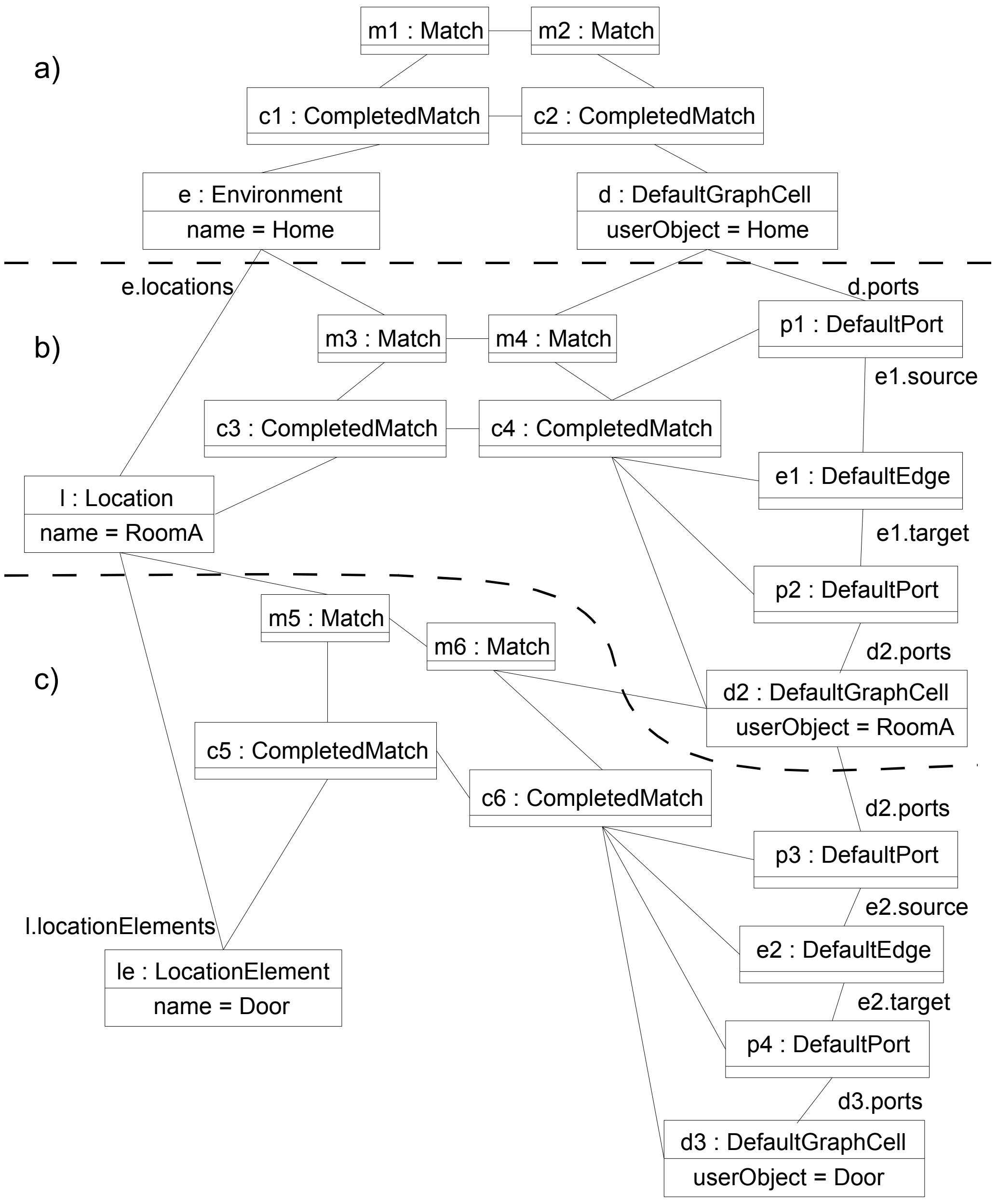


Figure 7.3: TRIMoS runtime structures for eHome model, correspondence graph, and JGraph model

creation, but following the location element creation rule in Figure A.5. This rule contains objects with the optional create stereotype. The matching algorithm is more complex in this case(see Section 7.4). The algorithm checks, if the created location element is already matched (has connections to other location objects) and if additionally to the objects $p3$, $e2$, and $p4$ also $d3$ has to be created on the right-hand side JGraph model. The mentioned objects are created because the le is a new object in this rule context. The created objects are matched with completed matches indicating the correspondence to postconditions of the applied TRIMoS rule.

Another location $l2$ is added into the eHome model with a connection to the environment e. This situation is shown in Figure 7.4. Figure 7.4 completes Figure 7.3 with structures created in this situation to the side models and correspondence graph, but omits correspondence graph structures from part a) in Figure 7.3 and completely part c). The added objects belong to the part b) in Figure 7.3.

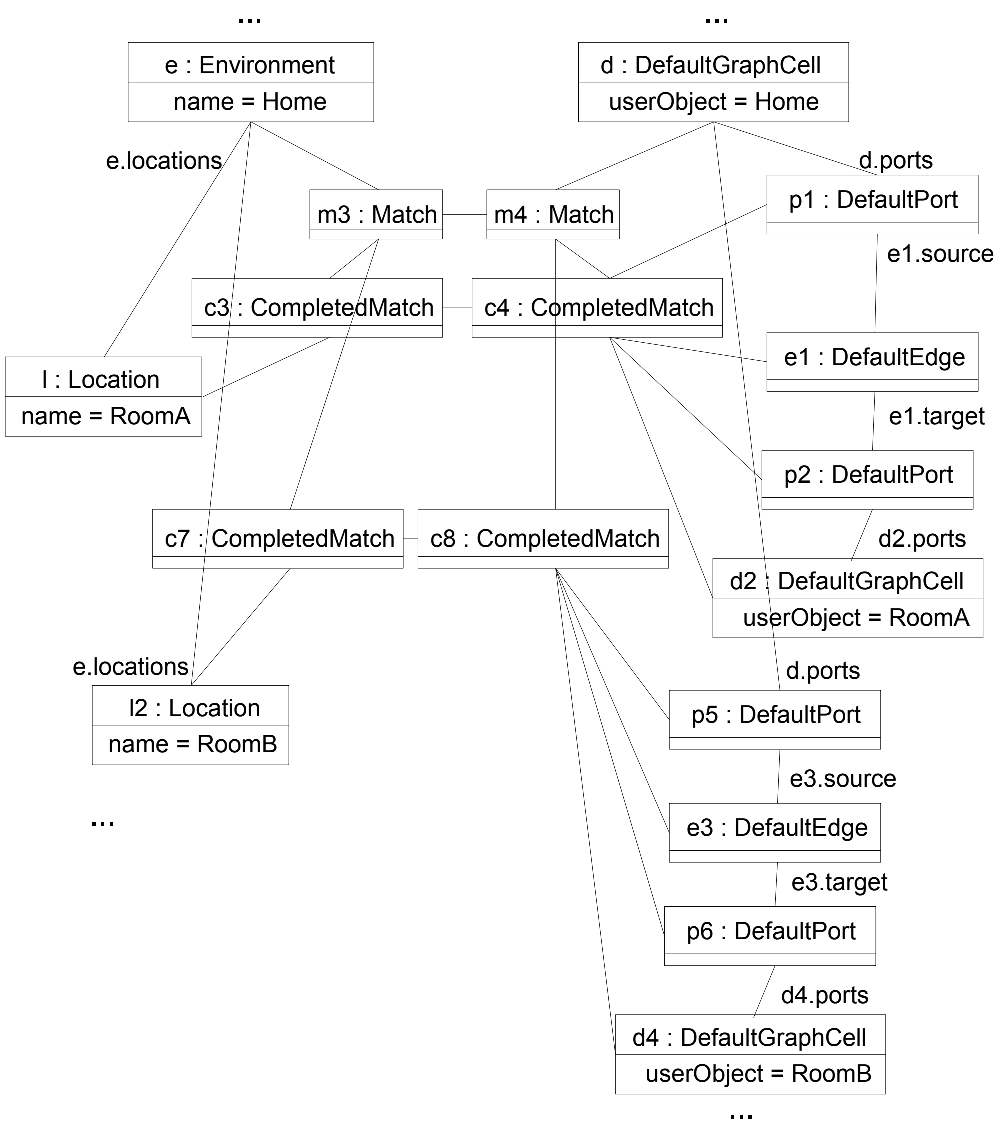


Figure 7.4: TRIMoS runtime structures for eHome model, correspondence graph, and JGraph model – second location is added.

The addition of a location to the environment is processed by the match listener related to $m3$. The matching algorithm determines two facts: the change corresponds to the location creation rule and the fact that preconditions are matched already with match $m3$. Match $m3$ was created during application of the same rule, the first time and can be reused in this matching. The created location element $l2$ is matched with the completed match $c7$. The parent match of the $c7$ will be $m3$. The preconditions for the other side are found via the $m4$. The necessary objects $p5$, $e3$, $p6$, and $d4$ are created according to the rule and matched with the completed match $c8$. The parent match of $c8$ will be $m4$.

The creation of the location $l2$ shows that the rule application in the same context (environment e as preconditions) results in the correspondence graph structure where the context is matched with one match object and objects created during every application of the rule with a new pair of completed matches. The completed matches are matching object sets in the model, which are created during every application time of any rule.

The next eHome model change connects the location element le to the location $l2$. This change corresponds to the create location element rule (see Figure A.5). This time, the location element already exists and is matched in the model. Only the link is created between the location element le and the location $l2$. Thus, according to the optional create stereotype, the location element le has to be treated during matching as a part of preconditions. Figure 7.5 illustrates the outcome of the matching and model completion steps initiated by the change in the eHome model. This figure omits nearly completely parts a) and b) visualising the environment and location creation in Figure 7.3. It presents the two locations l and $l2$, part c), and completes the part c) in respect to the last eHome model change.

The eHome model change connecting the le to $l2$ is handled by the match listener related to the completed match $c7$ (see Figure 7.4). The matching algorithm discovers that the change corresponds to the location element creation rule. It also discovers that in this case the location element le has to be considered to be part of the preconditions. Nevertheless, the location element is matched with the completed match, like postconditions (see Section 7.4). The preconditions (including the objects corresponding to the optional create elements) are also found in the JGraph side model. The missing model objects $p7$, $e4$, and $p8$ are produced. The produced elements and preconditions corresponding to the optional create stereotype rule elements are matched with new completed matches and the rest of the preconditions with a match object. The match objects are also new, because the context is different for the rule application.

There is a difference in the outcome of the last two changes in the eHome model. During the creation of the second location on the JGraph side there are four objects created $p5$, $e3$, $p6$, and $d4$ to represent the edge between the graph cell representing the environment and a new cell representing the new location. The rule A.3 is applied for this change. When connecting the given location element with the new location, the JGraph side is completed with only three objects $p7$, $e4$, and $p8$. These objects are used to draw an edge between the cells representing the new location and the location element. The rule A.5 is applied for this change. The difference is caused by the fact that location element creation rule contains the optional create rule element.

In the next section we will give an overview of the matching algorithm used by the TRIMoS framework. The matching routines are discussed in detail in the following sections.

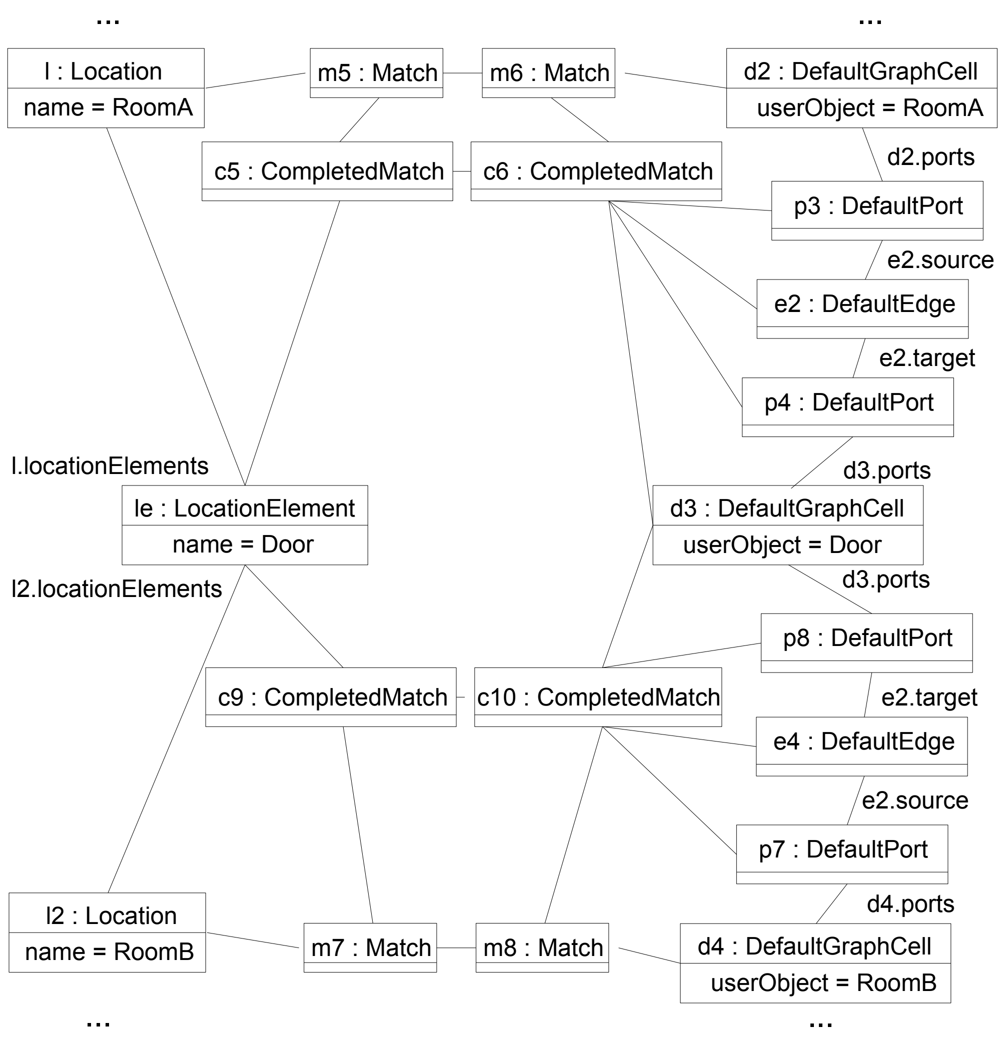


Figure 7.5: TRIMoS runtime structures for the eHome model, correspondence graph, and JGraph model – location element is connected to the second location.

7.2 Initial Matching Algorithm

The TRIMoS framework synchronises the models by constantly monitoring them. The synchronisation is done symmetrically with respect to the left- and the right-hand side model. The synchronised change events are considered to happen in the left-hand side of the model. Thus, in the following text we discuss the LR-translations assuming that RL-translations are performed in exact the same manner.

The synchronisation is performed with the help of match listeners, which listen to the changes in the left-hand side model. Every match object is paired with one match listener, which listens to property change events on the model objects matched by the match object. For example, considering Figure 7.3 there is a match listener paired with match $m3$ listening to the change events of the environment object e or a match listener paired with the completed match $c4$ listening to the change events on objects $p1$, $e1$, $p2$, and $d2$.

If an object changes in the model, the matching algorithm is started by a match listener which discovered the change. The matching for the rule application is searched locally, close to the change. The matching algorithm itself resides in the `Matching` class and it is used to analyse the model structure related to some match.

The initial (simple) matching algorithm developed in early stages of the TRIMoS framework is the following:

- Phase 1. In the model where the change occurs (the left-hand side) it is considered, if the change matches some rule:
    1. the changed context is matched with all rules to see if the preconditions of the rule match,
    2. the postconditions of the rules are also checked to discover if the change corresponds to the production, i.e. if the rule matches on the left-hand side together with its preconditions and postconditions,
    3. the the objects corresponding to the pre- and postconditions are matched with new completed match,
    4. the completed match is split into match matching precondition objects and completed match matching postcondition objects.
- Phase 2. After the match is found on the left-hand side, the matches are found and completed on the right-hand side – i.e. the changes on the left-hand side model are propagated to the right-hand side model:
    1. On the right-hand side the corresponding preconditions are searched for the left-hand side precondition objects. The corresponding match is created if not found.
    2. The found match is completed with the completed match on the right-hand side, i.e. the objects corresponding to the rule's right-hand side composite production's postconditions are created in the right-hand side model.

7.3 Disjoint Preconditions

TRIMoS also deals with the rules where the preconditions in compound production are not a connected graph. This is the case if a simple link creation between already existing objects has to be considered as a graph production. The link creation can correspond to the creation of a larger structure in the other side containing objects and links between them. For example, Figure 7.6 shows a rule where objects $l1$ and $l2$ form a precondition graph for the left-hand side and $d1$ and $d2$ respectively for the right-hand side. The postcondition link $l1.locations$ on the left-hand side corresponds to a larger structure on the right-hand side indicated with the stereotype `create`. This kind of a rule would be necessary, if the eHome model structure would require that every location object has a link to an environment object. Then, sub-location construction is defined only by creating links between given locations in the model.

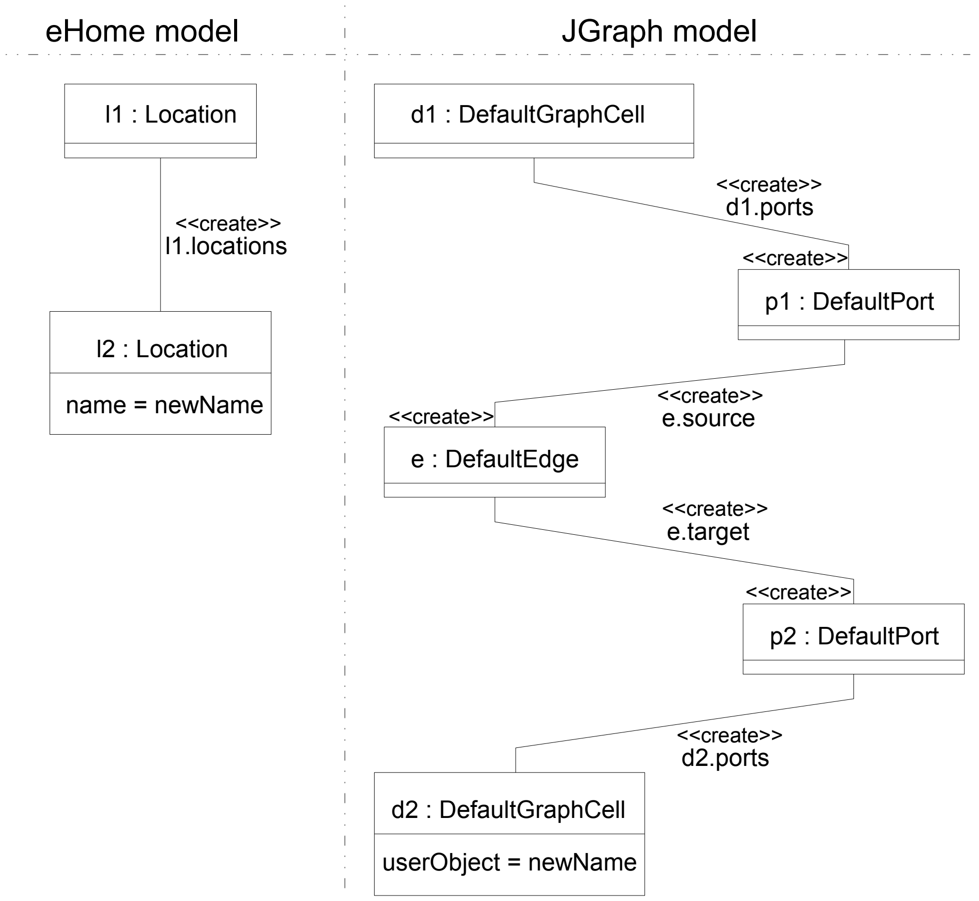


Figure 7.6: TRIMoS rule with disjoint preconditions.

The simple matching algorithm given in Section 7.2 is not capable to handle TRIMoS rules using compound productions with disjoint preconditions. It is because one precondition object must be a postcondition attribute for another precondition object ($l1$ is an attribute of $l2$ according to the production in Figure 7.6). In other words, the link between the two precondition objects is not a precondition link. The initial matching algorithm can not search for a precondition match via postcondition links in the right-hand side model. This is because of a simple reason: the links do not yet exist. We enhance the simple matching algorithm to deal with disjoint preconditions.

The simple matching algorithm copes with several precondition objects in the rule side, if they are interconnected with precondition links. In a simple case, the preconditions compose a connected graph for the matching algorithm. The matching algorithm chooses one object from the connected precondition graph. It finds all objects and attributes, if these are available to satisfy the entire rule. The simple matching algorithm does not find the match for precondition objects which are not interconnected with precondition links.

The first phase of the matching algorithm is not modified. The searching for the match on the left-hand side model can be done by following the postcondition links also in the disjoint preconditions case. A completed match is created during the execution of the first phase. It is splitted after the matching step into the match and completed match to match precondition and postcondition objects correctly.

The search for precondition objects for the right-hand rule side is changed in the second phase of the matching algorithm. The preconditions can be searched only in a connected graph so far. In the current case, the preconditions are not connected. Thus, we introduce a *candidate set* for precondition objects. The candidate set consists of objects which are suitable candidates for precondition objects, i.e. the candidate set is a container object having links to all of the candidates for the precondition objects. In terms of graphs, the candidate set is a common parent for possible precondition objects completing the precondition graph to be connected.

We use Figure 7.7 to illustrate the following discussion on the disjoint preconditions and the candidate set. This figure extends Figure 7.4 by visualizing a change where location $l2$ is connected as a sub-location to the location l. The change corresponds to the TRIMoS rule in Figure 7.6. Figure 7.7 shows the match objects $m9$ and $c11$ created during the first phase of the matching algorithm. According to the TRIMoS rule with disjoint preconditions in Figure 7.6 the preconditions l and $l2$ are matched with the match $m9$. The completed match $c11$ has no relation to model objects because the rule has nothing but a link between the location objects as postcondition. The objects painted with dashed lines are created during the second phase of the matching algorithm. This will be discussed below.

The candidate set for a precondition search on the right-hand side is constructed in the following manner: the completed match on the left-hand side is used to start searching the candidates, because this completed match represents the change on the left-hand side model (see $c11$ in Figure 7.7). First, the parent match of the completed match is acquired (match $m9$ in Figure 7.7). Second, the instances of the parent match are acquired (objects l and $l2$ in Figure 7.7). These instances correspond to the disjoint precondition objects for the rule application on the left-hand side model. For every instance the completed matches are acquired (completed matches $c3$ and $c7$ in Figure 7.7). For these completed matches the linked completed matches for the right-hand side model are acquired (completed matches $c4$ and c8 in Figure 7.7). The instances of the completed matches on the right-hand side are added to the candidate set (objects $p1$, $e1$, $p2$, $d2$, $p5$, $e3$, $p6$, and $d4$ in Figure 7.7 are added to the candidate set).

After constructing the candidate set, the right side of the given rule is changed. A new precondition rule object is added to the right side of the rule, namely the rule object corresponding to the candidate set. This new rule object has links to all the precondition rule objects in the rule. The changed rule side corresponding to the TRIMoS rule in Figure 7.6 is shown in Figure 7.8. The objects in the right-hand side model suitable for the preconditions in the right-hand rule side are searched with the help of the changed rule and the constructed candidate set.

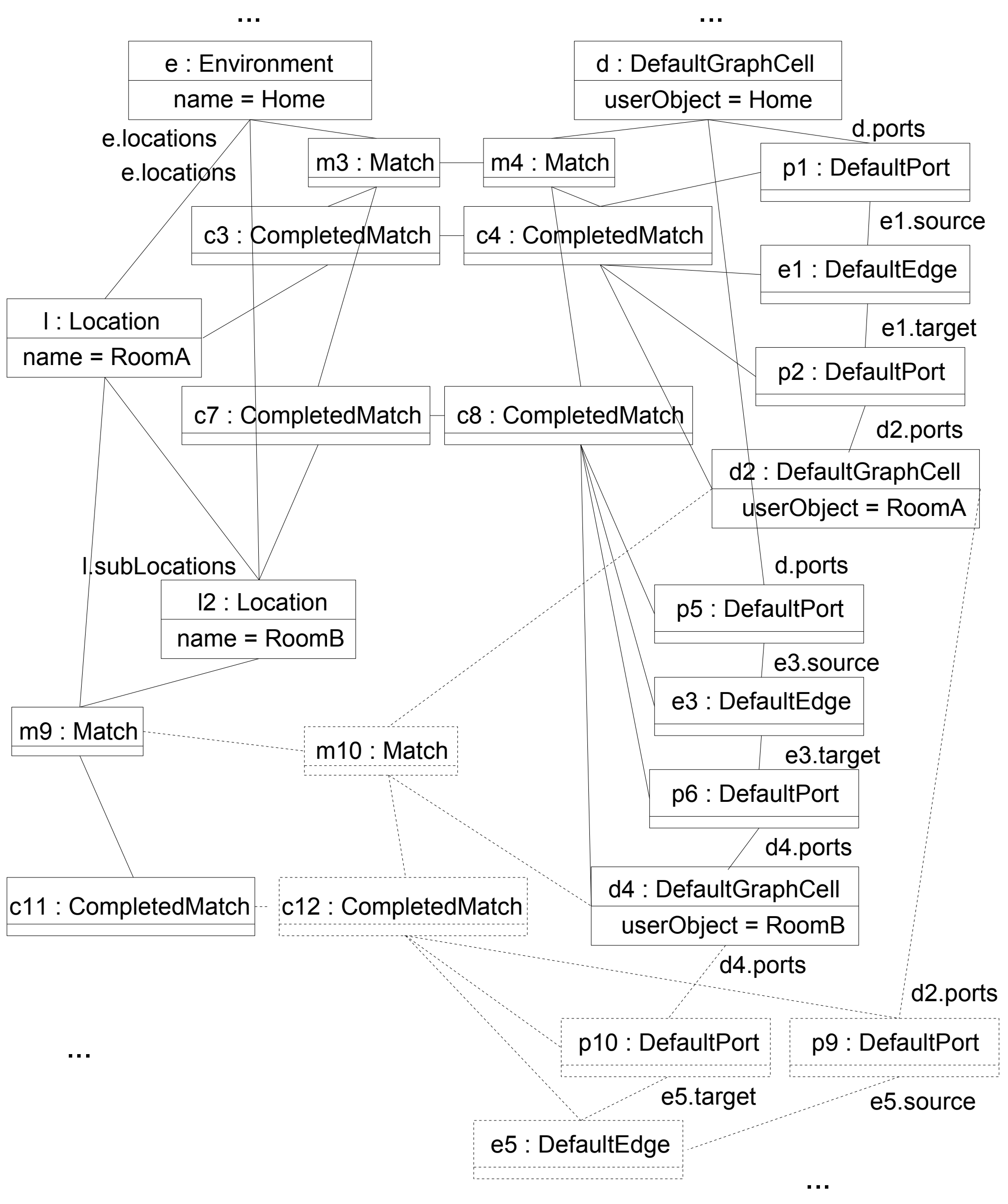


Figure 7.7: Application of the TRIMoS rule with disjoint preconditions on the eHome model instance

If the search is successful, the found match on the right-hand side is linked to the left-hand side precondition match. The right-hand side match matches the preconditions on the right-hand side model although, the preconditions are not forming a connected graph in the right-hand side model (objects $d2$ and $d4$ are matched with $m10$ in Figure 7.7). The matching algorithm proceeds as usual by completing the found match on the right-hand side with completed match for postconditions (completed match $c12$ is created in Figure 7.7) as well as a corresponding postcondition structure in the right-hand side model (objects $p9$, $e5$, and $p10$ are created in

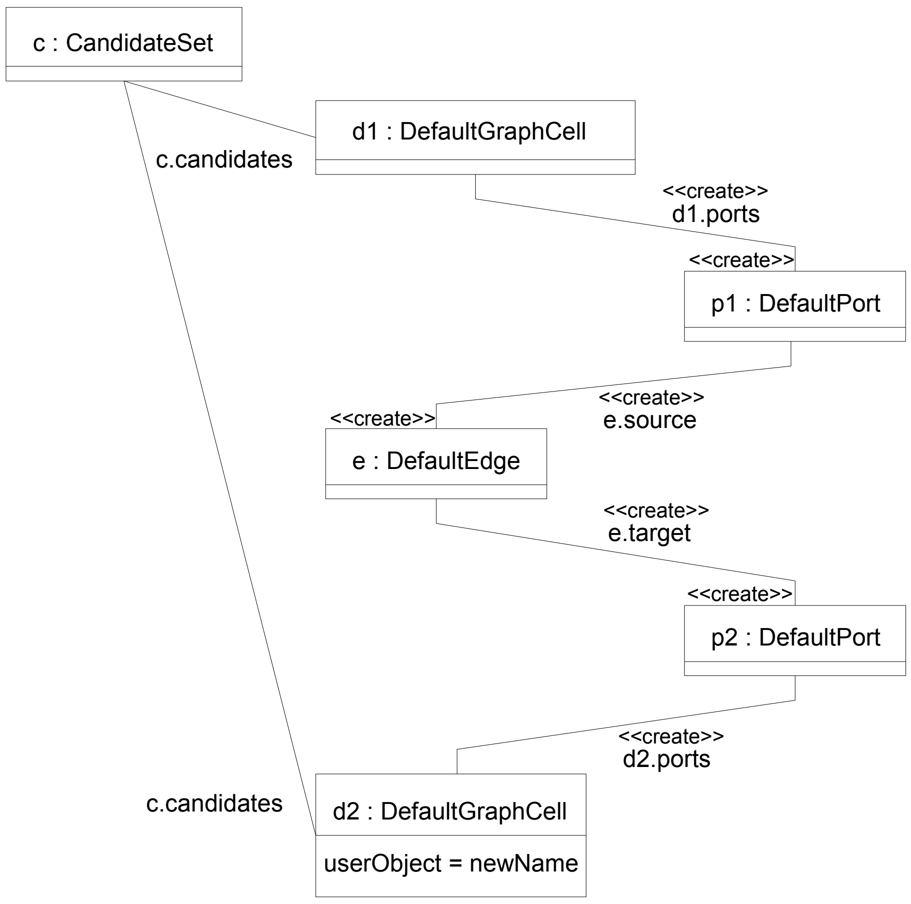


Figure 7.8: A rule side is complemented with the rule object for the candidate set

Figure 7.7). The completed match created during the completion of the right-hand side model is linked with the initial completed match on the left-hand side. The changes of the matching algorithm are summarized in the Phase 2 step 1 of the current matching algorithm in Section 7.6.

7.4 Optional Create Stereotype in the Rules

The motivation and reasoning for the optional create stereotype is given in Section 6.2.3 as well as an example for the TRIMoS rule with optional create objects is depicted in Figure A.5. Section 6.2.5 also explains why the compound production with optional create as in Figure 6.3 is not equal with a combination of the following two rules. Both rules have instead of optional create objects: first, postcondition objects (the object *le* in Figure 6.1); and second, precondition objects (the object *le* in Figure 6.3 being precondition and not optional create).

The initial matching algorithm does not deal with the TRIMoS rules with optional create stereotype objects. We refine the algorithm to cope with this stereotype. The implementation of the optional create stereotype introduces a greater computational complexity for matching than other stereotypes. The search for the objects which might satisfy the optional create stereotype rule elements to complete the match on the other side is more complex. This will be discussed below.

The first phase of the matching algorithm is not changed. The example of the change corresponding to the rule with optional create objects can be found in Figure 7.5. After the location element le is connected to the location $l2$ the matches $m7$ and $c9$ are created to mark the correspondence of the change to the location element creation rule in Figure A.5. At this point the objects $m8$, $c10$, $d4$, $p7$, $e4$, and $p8$ do not exist yet.

The algorithm proceeds with the second phase by searching the preconditions on the right-hand side. The preconditions for the rule are found (graph cell $d4$ in Figure 7.5) and matched with a match (match $m8$ in Figure 7.5), which is linked to the match on the left-hand side (matches $m7$ and $m8$ are linked in Figure 7.5).

On the right-hand side the objects corresponding to optional create rule elements are treted as postcondition objects, if the rule is applied first time. The instances corresponding to rule objects with optional create stereotype have to be created similarly to the initial matching algorithm together with the objects corresponding to the postconditions of the rule. This is done if the location element le is connected to location l in Figure 7.5.

If objects corresponding to the optional create stereotype already exist, they are not created, but treated as precondition objects. Hence, the second phase of the matching algorithm must be modified in the part where completed matches are created and completed on the right-hand side. For these means the search is performed similar to the preconditions search for rules with disjoint preconditions to find correct preconditions on the right-hand side model.

As mentioned before, in the second phase the preconditions for the right-hand side production are found and matched with a match (the match $m8$ in Figure 7.5). There is an empty completed match created on the right-hand side in correspondence of the completed match on the left-hand side (the completed match $c10$ is created and linked with $c9$, at this point $c10$ has no links to objects $d3$, $p8$, $e4$, or $p7$ yet in Figure 7.5). The next steps differ from the initial matching algorithm. If the rule matched on the left-hand side has elements with optional create stereotype, the candidates for objects corresponding to optional create rule elements (in this case preconditions) on the right-hand side are searched using the completed match on the left-hand side (the completed match $c9$ in Figure 7.5). This is also a difference to the search in the case of disjoint preconditions. The search to find objects matching the optional create rule elements on the right-hand side is not performed using the parent match of the completed match on the left-hand side, but using the completed match itself.

First, the instances (the location element le in Figure 7.5) are acquired from the completed match on the left-hand side (the completed match $c9$ in Figure 7.5). All the completed matches related to the instances are found on the left-hand side (the completed matches $c5$ and $c9$ in Figure 7.5). The corresponding linked completed matches are found on the right-hand side (the completed matches $c6$ and $c10$ in Figure 7.5). The instances of the completed matches on the right-hand side are considered as candidates for optional create objects in the role of preconditions of the rule application on the right-hand side (the objects $p3$, $e2$, $p4$, and $d3$ in Figure 7.5, the completed match $c10$ has no references to post conditions yet). These instances are added to the candidate set as candidates for preconditions.

After the candidates are found on the right-hand side for instances corresponding to optional create rule elements, a temporary match on the right-hand side is created. This temporary match is used to find the preconditions satisfying the rule with optional create elements. The temporary match is the copy of the match (the match $m8$ in Figure 7.5) on the right-hand side, because this match is already matching the normal preconditions of the rule application.

The rule used for matching is also changed. The optional create rule objects are set to be preconditions and a candidate set rule object is added having links only to the rule elements, which had the optional create stereotype before. The change is similar like in Figure 7.8, but the candidate set rule object c is linked with optional create rule objects (which were changed to precondition objects).

Then the rule is used to search the preconditions and the candidate set is used as a root object to begin the search. If the candidate set (containing objects $p3$, $e2$, $p4$, and $d3$ in Figure 7.5) matches the corresponding candidate set rule object, the suitable combination of the candidates are added to a temporary match to represent preconditions (the graph cell $d3$ in Figure 7.5 is linked with a temporary match). Thus, if the search is successful, the temporary match contains the normal preconditions for the rule application, but also the instances which correspond to the optional create rule objects. If this matching fails the temporary match is destroyed.

After the search the rule is restored and the candidate set is removed from the temporary match. At this point the preconditions are found (the objects $d4$ and $d3$ in Figure 7.5) to complete the matching by creating the missing model objects for the right-hand side model. If temporary match is not destroyed, the new completed match on the right-hand side will have the temporary match as its parent match (the completed match $c10$ in Figure 7.5 has the temporary match as its parent).

The completed match is completed without objects corresponding to the optional create rule objects (the objects $p7$, $e4$, and $p8$ are created and linked with completed match $c10$ in Figure 7.5). If the temporary match is destroyed, the new completed match will have the match representing normal preconditions as its parent. The completed match is completed also with objects corresponding to the optional create rule objects (this is the case when $p3$, $e2$, $p4$, and $d3$ are created in Figure 7.5).

If the temporary match is used, and it is set as a parent for the new completed match, the instances corresponding to optional create objects are moved from the temporary match to the new completed match, because they do not represent normal preconditions (the graph cell $d3$ is not linked with temporary match, but with completed match $c10$ in Figure 7.5). Thus, the model instances corresponding to the optional create rule elements are always matched with completed matches. The summary of the changes are given in Phase 2 steps 1 and 2 of the matching algorithm in Section 7.6.

7.5 Deletion of Model Elements

A synchronisation mechanism must handle also removal of the objects from synchronised models. The initial matching algorithm does not handle the synchronisation of deletions. It is extended by a validation step, which checks if the matches are still consistent with the model side they are related to. Every deletion of some object from the model results in fired property change events by the objects related to the destroyed object.

The change event is caught by some match listener. The match listener triggers a matching algorithm to validate the match it is related to. The matching algorithm performs a validation check on the match. We check if the TRIMoS rule corresponding to the match object still applies on the objects related to the match, i.e. if the changed model part corresponds to the rule side. The match is removed if it does not have relations to the objects it should have. The linked matches are also removed. The objects related to the linked matches are also removed, if they are not matched by other completed matches. The changes summarizing the validation phase of the matching algorithm are given Section 7.6.

This kind of a deletion mechanism considers the TRIMoS rule as a unit of deletion. If deletion appears on the left side destroying a matching of some rule, the corresponding structures matching the rules right-hand side are removed completely from the right-hand side model. The objects related to some other rule application (matched with other matches) are not removed from the right-hand side model. This kind of deletion behaviour can also be seen as a direct result of the monotonic compound productions used in TRIMoS rules.

7.6 Matching Algorithm

- Validation Phase. The match / completed match is validated. If it is not valid:
    1. the instances of the linked matches are removed if they are not matched with some completed match,
    2. the linked matches are removed,
    3. the match is removed,
    4. if the match is a completed match, the parent match is removed, if the parent match does not have any other completed matches as its children.
- Phase 1. In the model where the change occurs (the left-hand side), it is considered, if the change matches some rule. No candidate set is needed since the search is done on this side also using the postcondition links for the preconditions and the objects corresponding to optional create are treated also as postconditions:
    1. the changed context is matched with all rules to see if the preconditions of the rule match,
    2. the postconditions of the rules are also checked, i.e. if the rule matches on the left-hand side together with its preconditions and post conditions to discover if the change corresponds to the production,

3. the objects corresponding to the pre- and postconditions are matched with a new completed match,
4. the completed match is split into a match matching precondition objects and a completed match matching postcondition objects. (The instances corresponding to the optional create rule elements are matched with the completed match.)

- Phase 2. After the match is found on the left-hand side, the matches are found and completed on the right-hand side – i.e. the changes on the left-hand side model are propagated to the right-hand side model:

  1. On the right-hand side the corresponding preconditions are searched for the left-hand side precondition objects. The corresponding match is created if not found. In the case of rule with disjoint preconditions:
     (a) the right-hand side rule is changed to contain a `TObject` for the candidate set which has links to all possible precondition rule objects;
     (b) the candidate set is composed for preconditions searching over the matches of the left side and linked matches on the right side;
     (c) the right-hand side model objects are matched with the help of candidate set. If the match for preconditions is found, it is completed in the next step.
  2. The found match is completed with the completed match on the right-hand side, i.e. the objects corresponding to the rule's right-hand side composite production's postconditions are created in the right-hand side model. In the case of the rule involving optional create rule elements:
     (a) a temporary match is created for preconditions. It is a copy of the match found in Phase 2 step 1. The variables from the completed match created in Phase 1 step 4, are copied into the variables of the temporary match.
     (b) the right-hand side production of the TRIMoS rule is changed to contain a `TObject` for a candidate set, which has links to all matched once rule objects, these objects are also set to be precondition objects.
     (c) The candidate set is composed for the right-hand side model instances which might correspond to optional create rule objects.
     (d) The temporary match is complemented with necessary preconditions corresponding to the optional create objects by matching the changed rule against the candidate set. If matching fails the temporary match is removed.
     (e) If matching in the previous step is successful, the temporary match is assigned to be the parent of the new completed match
     (f) The new completed match is completed, i.e. the objects corresponding to postconditions of the rule are created into the right-hand side model.
     (g) The objects and variables corresponding to the optional create objects are moved down to the newly completed match. (The instances

corresponding to the optional create rule elements are matched with the completed match.)

(h) The clean up step. The temporary match is removed, the changed rule is restored. The match found in Phase 2 step 1 is set to be the parent match of the newly completed match on the right-hand side.

7.7 Adaptation of the JGraph API

TRIMoS is used to synchronize the eHome model instance and JGraph structures. Therefore, the interfaces: `PropertyChangeManager`, `AttributeAccessor`, `InstanceCreator` are implemented for the JGraph API. The eHome model does not need specific implementations for these interfaces, because it follows the JavaBeans specification [Sun97] design guidelines and property change management conventions. The JGraph API does not follow the JavaBeans specification – it has non-consistent getter and setter methods, list attributes and its own change event management mechanism. These differences have to be encapsulated into the adaptive classes implementing the mentioned three interfaces to use JGraph API with TRIMoS.

7.7.1 JGraph API

The JGraph API [Com] is an open source API providing elaborate features to create visual graphs and most importantly to customize them. The developer has the freedom to develop a graph implementation with specific behaviour and visualisation using JGraph. JGraph offers a set of useful features and examples to do that. This API offers additionally default implementations which can be easily extended to meet the specific needs of an application. JGraph is designed to be also a component in the Java Swing framework, thus it is easy to use it together with Java Swing API for visualisation in graphical user interfaces.

The `JGraph` class represents the graph. This class extends the Swing JComponent class being the visualisation component for the GUI development. This class links the JGraph API with the Java Swing API. The `GraphModel` interface represents the graph model for the JGraph class. The graph model is the actual graph consisting of nodes, ports, and edges. Edges can be connected with ports, which are connected to the nodes. The interface `GraphCell` represents nodes in the graph, the interface `Edge` respectively the edges, and the interface `Port` respectively the ports. All these interfaces have default implementations: `DefaultGraphModel`, `DefaultGraphCell`, `DefaultEdge`, and `DefaultPort`.

The JGraph API has also one drawback – it is quite complicated to use. For example, to create an edge between two graph cells, the following steps have to be performed. A graph model, an element set and an attribute map are created. Two graph cells are created and added to the element set. The attributes for both cells are inserted into the attribute map. Two ports are added to the graph cells. The edge object is created and added to the element set. The edge attributes are inserted into the attribute map. A connection is created to connect the edge with ports. Finally, the elements, attribute map, and connection set are added to the graph model. If the graph model is added to the `JGraph` component, the two graph cells and the edge between them are visualised on the `JGraph` component.

7.7.2 Adaptation of the JGraph API

We have to encapsulate the above described complex procedures to create JGraph cells and edges with simpler procedures. This enables us to specify simple compound productions usable in TRIMoS rules as in Figure A.3. The JGraph API is adapted for TRIMoS with implementing the following classes:

1. `TrimosGraphModel` extends and customises the JGraph `DefaultGraphModel` class;

2. `JGraphAttributeAccessor` implements the TRIMoS `AttributeAccessor` interface and adapts the accessor routines to read and write the fields in the graph model. For example, the attribute *ports* at the `DefaultGraphCell` does not exist in JGraph API, but is still used in TRIMoS rules (see Figure A.3). Accessing of the *ports* attribute is redirected to the routines adding elements to the graph model and reading the attributes of `DefaultGraphCell` objects.

3. `JGraphInstanceCreator` implements the TRIMoS `InstanceCreator` interface. It adapts the instance creation and removal from the graph model;

4. `JGraphPropertyChangeManager` implements the `PropertyChangeManager` interface from the TRIMoS framework. It adapts the graph change event mechanism of JGraph to be used by match listeners.

We consider the work amount of adapting the JGraph API considerably smaller than programming the translators between the eHome model and JGraph structures. In both cases the developer has to study the JGraph API. We suggest that, the JGraph API should be used to adapt it for the TRIMoS framework, so the translators can be specified with TRIMoS rules.

7.8 Summary

This chapter gave an overview of the implementation of the TRIMoS approach, introduced in Chapter 6. We handled the general design of the TRIMoS framework, the structures created during runtime, and also the implementation of matching algorithm to consider essential rule types and stereotypes used in TRIMoS rules. We also discussed the JGraph API and adaptations to use the JGraph API with TRIMoS. The next section gives an overview on future work. It outlines how the TRIMoS framework can be refined and improved.

Chapter 8

Future Work

The future work is discussed for three different fields: eHome systems, TRIMoS framework, and triple graph grammars.

8.1 eHome Systems

8.1.1 Refinement of the SCD-process

We consider the most promising research direction for the SCD-process refinement, the application of *parametric contracts* [Reu01] in the automatic configuration phase. The eHome services are developed as software components having a black-box specification. They offer a public interface for the system developer but no insight into the mechanisms implemented inside the eHome service. The eHome services developed in the scope of the eHomeConfigurator project are of course open source. Still, in perspective of the SCD-process for eHome systems, it is important that no re-programming of the services takes place. The SCD-process is a software configuration and not a development process.

Parametric contracts introduce so called grey-box approach for software components. The black-box component is coupled with a parametric contract, a white-box description of the software component. The parametric contract contains the information about the provided and required services inside the component. This information can be used for the architectural dependency analysis, automatic component adaptation, or quality of service predictions. One more important result on this field is that parametric contracts can be generated directly from the software component's code.

Parametric contracts are typically specified using the finite state machine formalism. The latest research results in the parametric contracts field [RHH05] introduce the graph grammar based formalism and define the compositionality of parametric contracts. These results are important for the eHome field because of two application areas: the refinement of the automatic configuration phase of the SCD-process and the development of eHome services.

The automatic configuration in the SCD-process considers currently the functional composition of the eHome services. It works practically with the semantic labels of service functionalities. The automatic configuration does not perform architectural analysis or resource resolution on the software component level. To improve the automatic configuration is the point, where application of parametric contracts could prove extremely useful.

Parametric contracts could be used by the SCD-process to perform automatic configuration of the eHome services regarding the architecture of the services, used communication protocols, and resource requirements. This kind of process would be able to compose eHome services provided by different parties requiring no communication and development overhead. This would also allow to detect conflicts between the services already during the configuration phase of the SCD-process.

8.1.2 Development of eHome Services

eHome services are component based. The black-box nature of eHome services protects the intellectual property of the service creator. Still, there is a conflict between the re-usability of the components and information requirement for the reuse to take place. The motivation behind the parametric contracts is to provide an appropriate level of the information needed for the reuse of software components.

Currently, the eHome services are developed by-hand. We see two advancement possibilities for the development process for eHome services. First, the application of generic techniques in eHome service development. Second, pairing the service development with the generation of parametric contracts for the services.

We used the Fujaba tool suite to develop our eHome model. The same tool was used to develop the automatic configuration routines for the SCD-process. We consider that eHome services could also be developed entirely with Fujaba. Fujaba is a graph rewriting system, thus it might be possible that the graph grammar based approach of parametric contracts could be included in the tool suite. This would allow the eHome service developer to create the services using model-driven development techniques and generate automatically the parametric contracts out of the service component's source code.

8.1.3 The eHomeConfigurator Project

The future work on eHomeConfigurator project should include the tool support development for the SCD-process refinements, the development of the open-source eHome services, and the support for the research done in the eHome group.

The eHomeConfigurator tool in the project can be considered as a platform to develop end-user applications for the eHome field. For example, to support eHome business, there is a need for a special tool for service providers for the service specification. The customers requesting an eHome should have a simple tool to specify their home environment. The companies offering the eHome deployment service should have a tool for the configuration of the eHome system and automatic deployment. They should also have a tool generating automatic installation instructions for the hardware in eHome. The deployment editor in the eHomeConfigurator and DOBS could be together a source base and a test-aid for the development of the graphical user interface for the eHome system or simple interfaces for the inhabitants of the eHome.

Considering the organizational aspects, the eHomeConfigurator project should find more industrial partners besides the inHaus Duisburg [inH05]. The project should find the means to introduce itself for the open-source community. This way we could find enthusiastic developing contributors interested in the eHome field. Our project should also be linked with the Amigo project [Ami05], which is an Integrated Project (IP) funded by the European Commission in the Sixth Framework Programme (see Section 9.1.1).

8.1.4 Security Aspects in eHome Systems

The topics security and privacy are not thoroughly researched in the eHome field. These topics are essential, because this computer-science field deals with the most private aspect of the every-day life – the home environment. Also the eHome should be the inhabitant's castle and not a pitfall. The eHome system must be reliable and secure. These systems can not malfunction and can not enable unauthorized access to the home environment and inhabitants privacy.

The eHomeConfigurator project comprises several security risks. The first risk is introduced by the design of the eHome model (`EhService` interface in the model). The second risk lies in the way how current eHome services work with the eHome model. There could be a possible attack designed on the eHome system using a malicious eHome service. This service might compromise the eHome model instance in the eHome system or shut down essential services on the service gateway. The eHome model instance is still unprotected and the resource management does not consider the authorisation aspects. This problem should be addressed as soon as possible, to make a next step towards an eHomeConfigurator project which satisfies the security requirements in reality.

8.2 TRIMoS

The further development of the TRIMoS framework should head in the direction to create a well tested and easily usable TRIMoS API. There is still work to be done like bug-fixing, performance testing, and writing the documentation. The TRIMoS system should also be optimized considering the memory usage and the synchronisation mechanism itself. The matching algorithm can be optimized for example, to use better strategies for rule selection during the matching.

The TRIMoS framework could be integrated with REclipse project [Pro05]. This project would provide an advanced TGG editor (TGGEditor module of the REclipse framework) for the TRIMoS rule specification. The current TRIMoS editor has limited features and a primitive graphical user interface. TRIMoS could also be used as a basis for research on TGG's discussed in the next section.

The model element deletion mechanism currently implemented in TRIMoS (see Section 7.5) considers the TRIMoS rule as a unit for deletion. We could study the application of the `destroy` stereotype for TRIMoS rules. This stereotype is used in Fujaba story diagrams to indicate transformations removing elements from the model. This development would of course conflict with the monotonic nature of TGG productions.

There is also a non-determinism problem to be addressed in TRIMoS. The framework does not implement any fail-safe, if rules are conflicting in the interpreted rule set. There could be some approach based on heuristics to solve this problem. The solution through user interaction is not appropriate, because TRIMoS is a system for reactive transformations. The user interaction based conflict resolution would introduce a communication overhead between the application employing the TRIMoS and the end-user of this application.

8.3 Triple Graph Grammars

The TRIMoS approach introduces simpler triple graph grammar rules resembling by appearance the pair grammars. The rules use more compact compound productions to specify the rule's left- and right-hand side. The third graph production in between is not modelled explicitly. It is fixed to one particular form and is created during the runtime of the TRIMoS framework.

Because of the implicit modelling of the correspondence graph, TRIMoS rules lack the expressiveness of the classical TGG approach. Still, they are applicable in the graphical user interface development. They can also be used for synchronisation of object models having similar structure. The TRIMoS approach adds an aspect of simplicity for TGG theory.

The general TGG approach should adopt the idea of optional correspondence production in the TGG rules. This is important in the perspective of simpler TGG rules. The specification of the correspondence graph should be implicit for simpler transformation cases. The correspondence transformations could be defined explicitly for more complex cases. The systems implementing the TGG approach should be able to interpret the rules with implicit and explicit correspondence productions.

Triple graph grammars are used as a basis for integration, translation, and synchronisation frameworks. These frameworks deal with transformations and translations of model structures. But models often employ behaviour which has no impact on the model's structure. There are functions using the model as an input and implementing scientific or statistical calculations, or calculations for status reports. These kind of functions or methods in the synchronized models could also be related with each other. Therefore, another research direction for TGG could be the linking of arbitrary method calls, which do not affect the model structure.

8.4 Summary

In this chapter we gave an overview on the ideas for future research and work. The ideas cover two computer science areas: the eHomes and triple graph grammars. Both fields are relatively young having a lot of research work ahead. We think that the ideas presented in this chapter could be very interesting and valuable direction indicators to discover the unknown. The next chapter discusses the related work for these two fields.

Chapter 9

Related Work

This chapter handles the related work to this thesis. We will discuss projects similar to the eHomeConfigurator project and the tool support for eHome processes. We will discuss the work related to the development tools we used in the eHomeConfigurator project and also related work for the TRIMoS approach.

9.1 eHome Systems

The field of smart homes, i.e. eHomes, is a sub-area of the ubiquitous computing research field. The term ubiquitous computing was introduced by Mark Weiser in [Wei91]. It denotes a computer science field researching possibilities to apply computing technologies everywhere in the everyday environment using micro controllers and sensor networks. In this case, the computing technology would be as common in our environment as “writing”. For example, we currently see text written everywhere: on posters, on pieces of paper, or on door signs. Ubiquitous computing would introduce micro controllers into our environment, into doors, floors, objects in our living space, to get the information from the environment. The important factor of ubiquitous computing is that it is a *calm technology* [WB98], because the computing technology in the environment should not be in the centre of the users attention, but run in the background, offering the information on demand.

9.1.1 Intelligent Home Projects

There are numerous projects dealing with eHome systems. The Intelligent House Duisburg Innovation Center (inHaus) [inH05] is one example of a project with big industrial partners, such as Sony, Viessmann, or Volkswagen. This project’s main research is done in the inHaus facility in Duisburg, which includes a residential home, a workshop, a networked car, and a networked garden. This is a complete test site for eHome systems. inHaus is also a very valuable partner of our eHome group, offering possibilities to test the eHome scenarios and tools developed within the group. There already have been successful tests performed in cooperation between the eHome group and inHaus in the residential home of the inHaus project.

Two other similar projects of a smart home environment are the industrial project T-Com Haus [T-C05] and the privately financed project 213 Smarthouse [Sch05a].

A promising project funded by the European Commission is Amigo [Ami05], aiming to develop open, standardized, interoperable middleware, and intelligent user services for the networked home environment. The Amigo project unites different partners such as Philips Research, France Telecom, Fraunhofer IMS, or Microsoft. This project has a budget of 24 million euros.

Additionally, there is The Adaptive House project [Moz98], relying heavily on the artificial intelligence technologies. This project tries to create an adaptive home environment, which tracks the inhabitant's behaviour using different sensors, cameras, and microphones. The neural-network based home system tries to render the user interfaces useless. It controls the home environment relying entirely on the assumptions of inhabitant's current and future behaviour.

The difference between the eHomeConfigurator project and projects mentioned above is that the eHomeConfigurator project focuses only on the software development process and more importantly software configuration process for eHome systems. The only hardware platform developed in the eHomeConfigurator project are the two eHome demonstrators, for testing purposes. These demonstrators are miniature eHome models, with minor construction costs for the materials and hardware.

There are also other eHome processes related to the SCD-process. The eHome group also researches business processes surrounding the SCD-process [Kir05]. Additionally, the eHome group researches the problematics of virtual eHomes and transport processes for inhabitant profiles[1].

9.1.2 Tool Support for eHome Systems

Tool support in the field of eHome systems focuses on graphical user interfaces and on easier development of eHome services. When considering the eHomeConfigurator tool and the SCD-process, there seems to be no other approach dealing with software configuration for eHome systems in the process level.

For example, the following two approaches support user interface development for eHome systems. The open source SUPPLE toolkit [GW04] developed at the University of Washington. It deals with the automatic generation of user interfaces for display devices used in ubiquitous environments. SUPPLE uses decision-theoretic optimization to render an interface from an abstract functional specification, an interchangeable device model, and an user model.

The SmartHome User Interface project [vDY96] creates an intuitive web-based interface to an automated household. The idea of the project is to create an interface for the household, which is available over the Internet all over the world. The SUPPLE and the SmartHome User Interface projects are very similar to work done in the eHome group on interactive user interfaces in eHome systems [Gab04].

The Equip Component Toolkit (ECT) [GT04] was created for development of applications in ubiquitous computing fields, involving designers and users in the development process. Users are supported by graphical tools, which provide various representations of the running environment, plus facilities for monitoring, and

[1]The virtual eHome and inhabitant profiles are new topics handled in the eHome group. There are no publications yet.

(re)configuration of the applications. This toolkit shares with the eHomeConfigurator project the idea of lowering the costs of smart home systems. ECT introduces cheaper and generic ways to develop and reconfigure services running in an ubiquitous computing environment. Nevertheless, ECT introduces a software development process, and not an automated software configuration process like the SCD-process.

9.2 Model-Driven Development

It is common for the scientific research of a field that many ideas are reinvented. This is also true in the case of software engineering, when looking at the idea of generating executable code according to models. For example, the modelling language SDL [EHS97], known from telecommunication systems, existed already in early seventies. It is a specification language for the telecommunication area. It incorporates ideas of model-oriented specification and executable specifications. As second example, software product line development processes [FI] are using a two step approach: first designing a language for specifying family members and secondly defining the automatic process to produce family members.

The ideas of modelling and automatic code generation can also be found in graph rewriting systems such as PROGRES [Sch91] developed at our Department of Computer Science 3 of the RWTH Aachen University, in the late eighties. The system consists of the PROGRES language and the PROGRES programming environment. The language is a strongly typed specification language for complex data structures. The programming environment consists of a syntax-directed editor and a code generator for C-code. The Fujaba tool suite [KNNZ99] was developed in the late nineties initially by people active also in the PROGRES development. Fujaba uses UML modelling and generates fully executable Java code.

The recent release from OMG [Obj02] is called MDA: the development process which also fully relies on modelling and code generation. Despite of the loops in the software engineering research history, it is clear that software engineering is moving towards the idea of model-driven development. The TRIMoS framework developed in the scope of this thesis is also a tool supporting model-driven development techniques.

9.3 Triple Graph Rewriting Systems

The theoretical background in Chapter 5 gave an overview on the underlying theoretical concepts for the TRIMoS framework. The TGG implementations and approaches can be divided into two main categories: batch-oriented and incremental rewriting systems. Batch-oriented systems transform an input graph to another graph defined by a different graph grammar than the input graph. The incremental systems allow translation of the two graphs step-wise, by starting the translation when demanded by the user, analysing the two graphs, and generating the missing structures. TRIMoS is an incremental synchronisation framework, but more specifically a reactive system. This means that TRIMoS propagates the transformations in between synchronized models automatically after the transformation takes place.

One of the earliest implementations of the TGG translation approach was implemented in the scope of the IPSEN project [Nag96]. IPSEN was a graph-based integrated software engineering environment, including implementations for the graph transformation and the TGG translation approaches [Sch94].

There is an incremental TGG approach for integration of engineering tools developed in our department [BHW05]. This approach considers the documents as the communication medium between the engineering tools. Thus the integration between the tools means that changes are propagated between the tools via a set of inter-dependent design documents, which have to be kept consistent with each other. This approach includes user interaction for conflict resolution of rules[2].

A very similar approach to the TRIMoS framework is described in [Wag01]. This TGG-based system can be used to transform models. Like TRIMoS, it is Fujaba-related, does not perform detection of conflicting rules, and does not include user interaction. The big difference between the TRIMoS framework and this approach is the semantics of the rules (see Section 6.2.5) and that the TRIMoS framework is reactive synchronisation framework.

9.4 Summary

In this chapter we discussed the related work to the following topics: the eHomeConfigurator project, the model-driven development tools we use in the project, and the TRIMoS framework. The related work in the field of eHome systems targets heavily the hardware and infrastructure development for eHomes. The software engineers in this field deal mostly with eHome service development. Our research group concentrates on the general software engineering process for eHome systems. We try to break the cost barrier in the eHome market created by the expensive software development processes. We are replacing the software development with software configuration process for eHome systems. This is a new research approach in the eHome field.

The Fujaba tool suite used in the eHomeConfigurator project has proven itself to be of a great value. This tool has been an inspiration to research the model-driven development techniques to solve the problems at hand. We have developed a TRIMoS framework to create bidirectional translators for object models. This TGG based framework differs with its reactive nature, compact and simple rules from other TGG based approaches. The TRIMoS framework adds the reactive translation and implicit correspondence graph aspects to the TGG field of computer science.

[2]The non-determinism problem described in Chapter 5.

Chapter 10

Conclusion

10.1 Summary

This thesis is a contribution to eHome systems and model-driven development in the field of computer science. The work with eHome systems gave us an eHome model and the work in the model-driven development field resulted in the development of the triple graph grammar based TRIMoS translation and synchronisation framework. The efforts in these two fields are interconnected in the development of the eHomeConfigurator tool supporting the eHome model instance's transformations during the SCD-process. The TRIMoS framework was developed to address the translator problem that occurred while implementing the eHomeConfigurator tool.

This thesis begins with an introduction and motivation for the research in two fields. First, the problematics in eHome systems and low-cost specification, configuration, and deployment process (SCD-process) for eHome systems are introduced. This is followed by a discussion of the eHome model, its technical solution and structure. The transformations and life cycle of the eHome model instance are regarded in the perspective of the SCD-process.

Chapter 4 on tool support for the SCD-process describes the development of the eHomeConfigurator tool and how this tool is put into use during the SCD-process. The structure of the tool and implementation of modules developed in the scope of this thesis are discussed in more detail. The discussion explains the translator problem in the context of the Specificator module development.

Before considering the solution approach for the translator development problem, the graph theory behind the solution approach is introduced. The solution approach is called TRIMoS and it is dealt with in two parts: the approach itself with introduced additions to the triple graph grammar theory, and the implementation of the TRIMoS framework. The discussion on the implementation of the framework concentrates on the design of the framework and matching algorithm, used to interpret the TGG transformation rules.

The thesis ends with a discussion on the future and related work, and this conclusion.

10.2 Relevance

The main results of this thesis are the eHome model, the eHomeConfigurator tool and the TRIMoS framework.

10.2.1 eHome Model

The eHome model is a model reflecting eHomes in reality. The model was refined during the past two years and has proved itself as an appropriate model for eHome systems. The eHome model provides a structure, which covers the general aspects of eHomes. The derived eHome model instance contains the specific aspects of a particular eHome. The general and specific aspects facilitate the low-cost SCD-process for eHome systems which helps to break the price barrier on the eHome mass market. The eHome model is an enabling factor for the SCD-process and for the context aware eHome services.

10.2.2 eHomeConfigurator

The eHomeConfigurator tool was developed to support the SCD-process and the eHome model instance transformations during the process. The eHomeConfigurator is a proof of the applicability of the eHome model and validity of the SCD-process to be used for eHome systems. Furthermore, with its open-source nature, the eHome-Configurator tool offers a good source-base for future tool development.

10.2.3 TRIMoS

The TRIMoS framework is a powerful TGG-based approach designed according to the requirements of software developers. This framework marks an important step towards simpler and more intuitive triple graph grammar rules. It uses a familiar UML notation, more compact and comprehensive compound productions, and requires no explicit modelling of the correspondence productions in TGG rules. These features facilitate a broader application of the TGGs in model-driven development because of better acceptance among software engineers.

10.3 Outlook

The future work on the eHomeConfigurator project and the TRIMoS framework may have several directions. One research direction for the eHomeConfigurator project could be the application of parametric contracts in the configuration phase of the SCD-process and in the eHome service development. Parametric contracts provide the means for service architecture analysis, composition analysis and automatic adaptation of eHome services during the SCD-process. This brings the automatic configuration of the SCD-process from the semantic label level to the level of the software components and provides reliable configurations with the guaranteed quality of service. The application of parametric contracts also facilitates the generic eHome service development.

The eHomeConfigurator tool itself is a promising basis for further tool support development for eHome systems. It can be used as a basis for user interfaces or service development, configuration, and deployment tools. The tool can also be extended to support distributed eHomes and virtual eHomes. The application of the parametric contracts and further development would extend the eHomeConfigurator to general software configuration tool supporting, for example, the processes in the automotive field, or embedded systems.

The work on TRIMoS framework might continue developing a TRIMoS Java API applicable in the software industry. This requires additional testing, optimization, and development work. The core of the TRIMoS framework could be integrated into the REclipse project [Pro05] to employ a better rule editor. The TRIMoS can also be extended to consider optionally the rules with explicit correspondence production to offer the complete expression power of the TGG rules, while maintaining the simplicity of the TRIMoS rules.

Appendix A

TRIMoS Rules for the Environment Editor

This appendix gives a TRIMoS rule set for the environment editor translator of the eHomeConfigurator tool. The type graph for the rules is in Figure 3.4. The initialisation rule has less elements as it should have in real application. For example, on the left-hand side, there should be a `DataHolder` object having a link to the $e : Environment$ object to indicate that the environment object is registered in data holder. On the right hand-side there are missing objects for the `EnvironmentEditorPanel`, the `JGraph` and the `TrimosGraphModel` classes. There should be a link between the $d : DefaultGraphCell$ object and the `TrimosGraphModel` object.

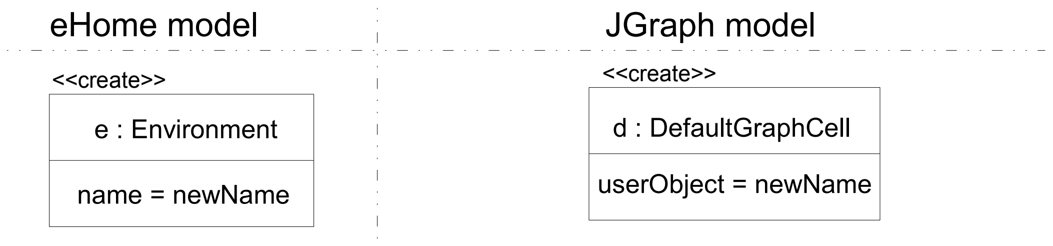


Figure A.1: An initialisation rule for the environment editor translator.

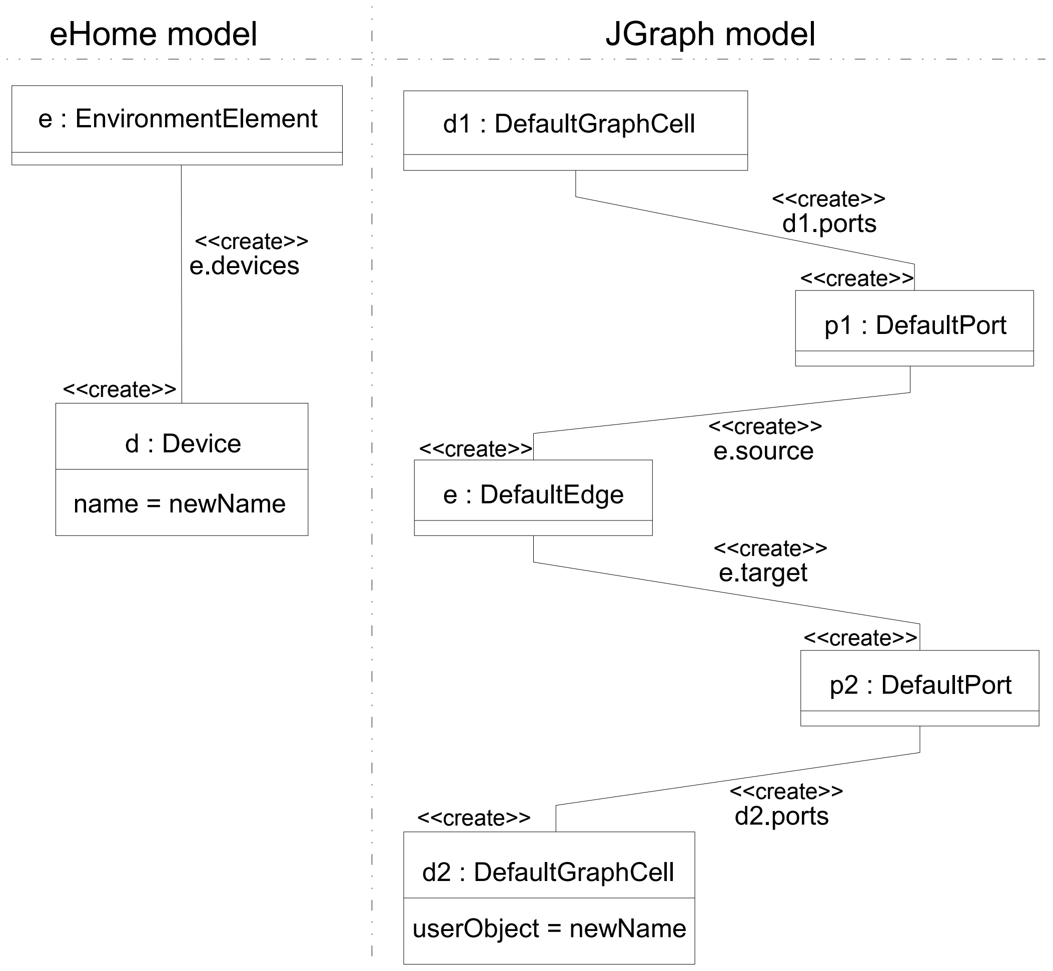


Figure A.2: A rule rule to create devices.

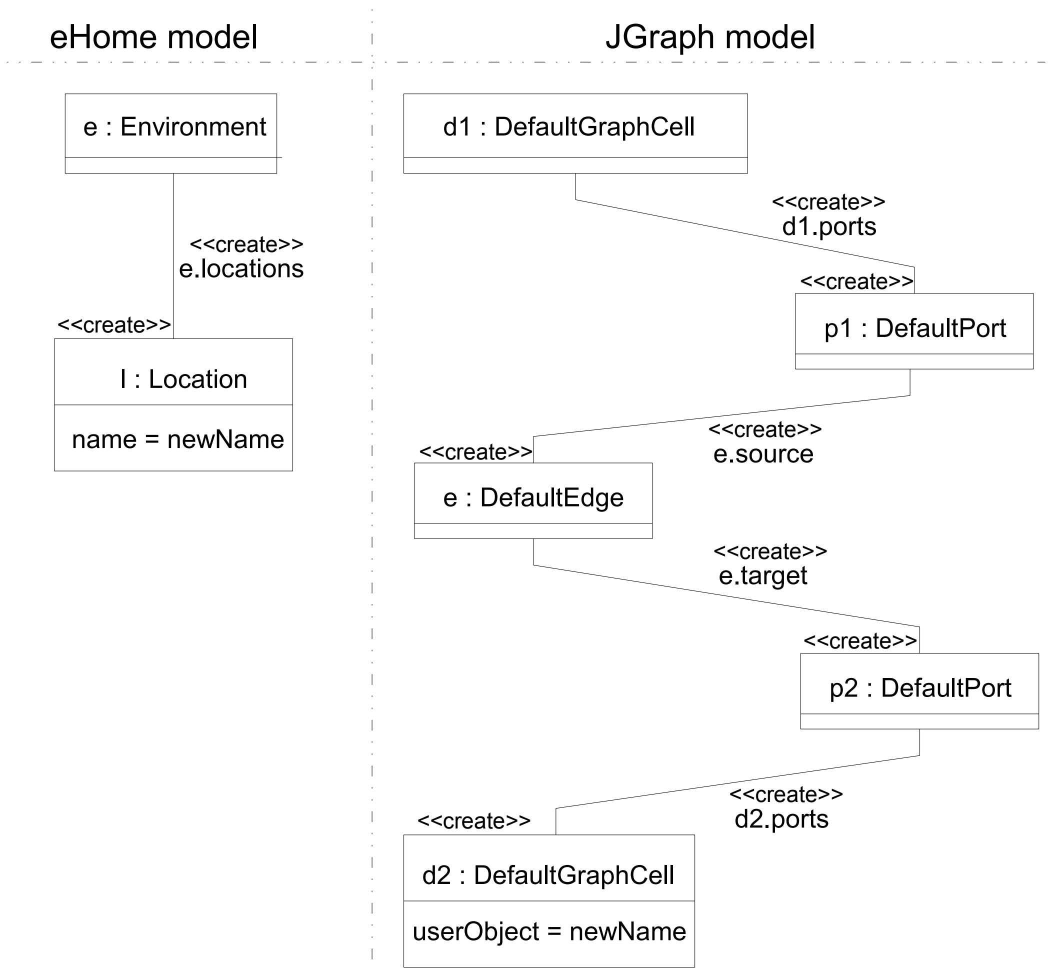


Figure A.3: A rule to create locations.

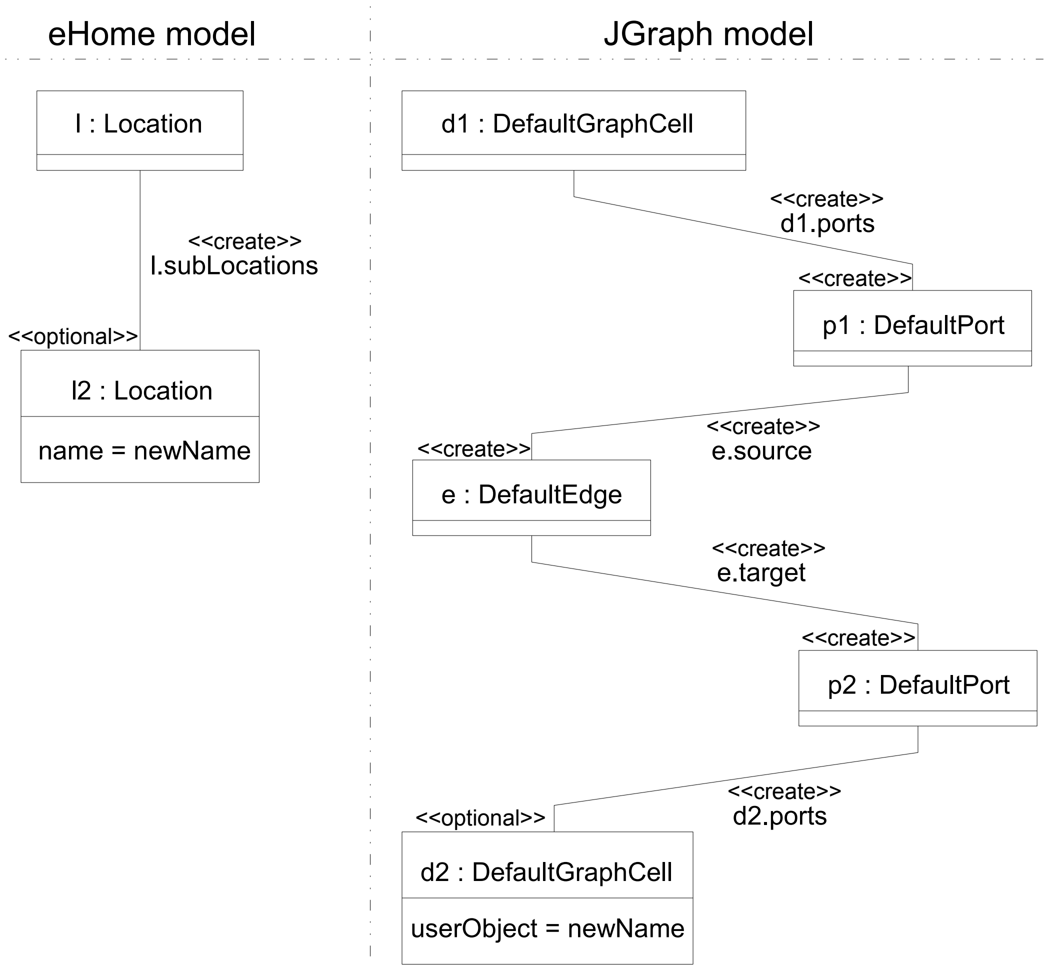


Figure A.4: A rule to create sub-locations.

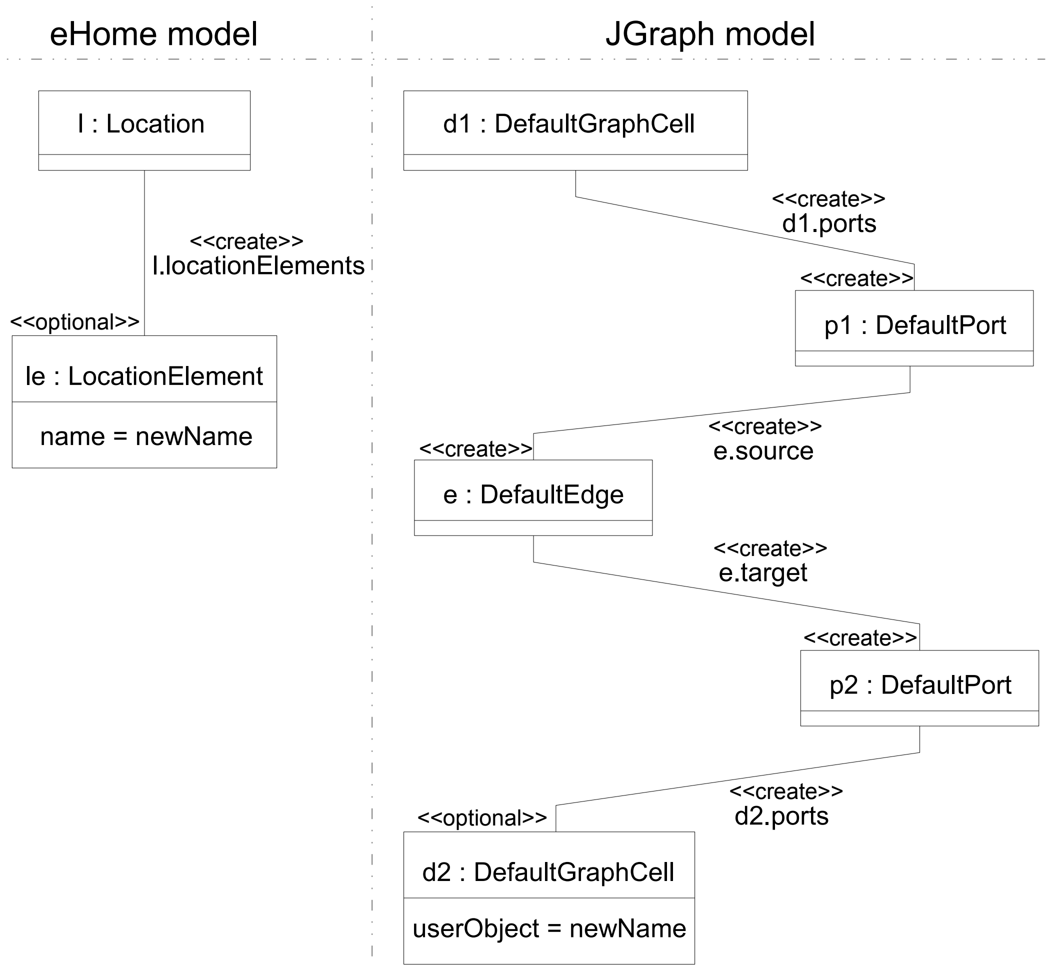


Figure A.5: A rule to create location elements.

List of Figures

Bibliography

[Akh05] Arash Akhoundi. Verteiltes Automatisches Deployment von eHome-Konfigurationen. Master's thesis, Rheinisch-Westfälische Technische Hochschule Aachen (RWTH), March 2005.

[Ami05] Amigo. Ambient Intelligence for the networked home environment. http://www.hitech-projects.com/euprojects/amigo/index.htm, 2005.

[BHW05] Simon M. Becker, Thomas Haase, and Bernhard Westfechtel. Model-based a-posteriori integration of engineering tools for incremental development processes. *Software and Systems Modeling (SoSyM)*, 4(2):123–140, May 2005.

[BvHH$^+$03] Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. OWL Web Ontologie Language Reference. http://www.w3c.org/TR/2003/PR-owl-ref-20031215/, December 2003.

[CG01] Kirk Chen and Li Gong. *Programming Open Service Gateways with Java Embedded Server Technology.* Addison-Wesley Professional, 2001.

[Com] JGraph Community. JGraph swing component. http://jgraph.com/jgraph.html (01.06.2004).

[EHS97] J. Ellsberger, D. Hogrefe, and A. Sarma. *SDL - Formal Object-Oriented Language for Communicating Systems.* Prentice Hall, 1997.

[EPS73] H. Ehrig, M. Pfender, and H.J. Schneider. Graph-grammars: an algebraic approach. In *Proceedings IEEE Conf. on Automata and Switching Theory*, pages 167–180. Iowa, 1973.

[FF04] Ira R. Forman and Nate Forman. *Java Reflection in Action.* Manning Publications, October 2004.

[FI] Fraunhofer-IESE. Product line software engineering (PuLSE). http://www.iese.fhg.de/PuLSE/.

[FNTZ98] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language. In G. Engels and G. Rozenberg, editors, *Proc. of the* 6th *International Workshop on Theory and Application of Graph Transformation*

(TAGT), Paderborn, Germany, LNCS 1764, pages 296–309. Springer Verlag, November 1998.

[Gab04] Peter Gabriel. Interaktive Benutzeroberflächen für eHome-Systeme. Master's thesis, Rheinisch-Westfälische Technische Hochschule Aachen (RWTH), June 2004.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[GNU99] GNU Project. GNU Lesser General Public Licence. `http://www.gnu.org/copyleft/lesser.html`, February 1999.

[GT04] Humble J. Greenhalgh C., Izadi S., Mathrick J. and I. Taylor. ECT: A toolkit to support rapid construction of ubicomp environments. In *In Proceedings of the Sixth International Conference on Ubiquitous Computing (UBICOMP'04)*, September 2004.

[GW04] Krzysztof Gajos and Daniel S. Weld. SUPPLE: automatically generating user interfaces. In *IUI '04: Proceedings of the 9th international conference on Intelligent user interface*, pages 93–100, New York, NY, USA, 2004. ACM Press.

[GZ02] Leif Geiger and Albert Zündorf. Graph based debugging with fujaba. *Electronic Notes in Theoretical Computer Science*, 72(2), 2002.

[Hec05] Reiko Heckel. Graph transformation in a nutshell. In Jean Bezivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2005.

[inH05] inHaus Duisburg. Innovationszentrum Intelligentes Haus Duisburg. `http://www.inhaus-duisburg.de` (22.6.2004), 2005.

[Jäg00] Dirk Jäger. UPGRADE - A Framework for Graph-Based Visual Applications. In Manfred Nagl, Andy Schürr, and Manfred Münch, editors, *Proceedings Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE'99)*, volume 1779 of *LNCS*, pages 427–432, Kerkrade, The Netherlands, September 2000. Springer.

[JW03] Michael Jeronimo and Jack Weast. *UPnP Design by Example.* Intel Press, May 2003.

[Kir05] Michael Kirchhof. *Integrierte Low-Cost eHome-Systeme – Prozesse und Infrastrukturen.* 2005. to appear.

[Kli04] Markus Klinke. Instanziierung von eHome-Diensten. Master's thesis, Rheinisch-Westfälische Technische Hochschule Aachen (RWTH), May 2004.

[KNNZ99] T. Klein, U. Nickel, J. Niere, and A. Zündorf. From UML to java and back again. Technical Report tr-ri-00-216, University of Paderborn, Paderborn, Germany, September 1999.

[Kno04] Knopflerfish OSGi. 2004. Knopflerfish Website.

[KNS04] Michael Kirchhof, Ulrich Norbisrath, and Christof Skrzypczyk. Towards Automatic Deployment in eHome Systems: Description Language and Tool Support. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, Proceedings, Part I*, volume 3290 of *LNCS*, pages 460–476. Springer, 2004.

[Kre04] Ingo Kreienbrink. Klassifikation und Suchstrategien in eHome-Szenarien. Master's thesis, Rheinisch-Westfälische Technische Hochschule Aachen (RWTH), June 2004.

[KWB03] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise.* Addison-Wesley, April 2003.

[Mal05] Adam Malik. Informationserfassung für automatisches Deployment von eHome-Systemen. Master's thesis, Rheinisch-Westfälische Technische Hochschule Aachen (RWTH), February 2005.

[McB04] Brian McBride. An Introduction to RDF and the Jena RDF API. `http://www.hpl.hp.com/semweb/doc/tutorial/RDF_API/index.html` (07.05.2004), 2004.

[Moz98] Michael Mozer. The neural network house: An environment that adapts to its inhabitants. In *Proceedings of the American Association for Artificial Intelligence Spring Symposium on Intelligent Environments*, pages 110–114. AAAI Press, 1998.

[Nag96] Manfred Nagl, editor. *Building Tightly Integrated Software Development Environments: The IPSEN Approach.* LNCS 1170. Springer, 1996.

[NFM00] N. Noy, R. Fergerson, and M. Musen. The knowledge model of Protege-2000: Combining interoperability and flexibility, 2000.

[Nor03] Ulrich Norbisrath. Softwaretechnik-Projektpraktikum im Hauptstudium: Werkzeugunterstützung zur eHome-Dienstespezifikation, 2003. `http://www-i3.informatik.rwth-aachen.de/teaching/0304/swt-pp/index.html`.

[NSM04] Ulrich Norbisrath, Priit Salumaa, and Adam Malik. eHomeConfigurator. `http://sourceforge.net/projects/ehomeconfig`, 2004.

[NSSK05] Ulrich Norbisrath, Priit Salumaa, Erhard Schultchen, and Bodo Kraft. Fujaba based tool development for eHome systems. In *Proceedings of the International Workshop on Graph-Based Tools (GraBaTs 2004)*, volume 127 of *Electronic Notes in Theoretical Computer Science*, pages 89–99. Elsevier, 2005.

[Obj02] Object Management Group, Inc. `http://www.omg.org` (24.02.2005), 2002.

[Pra71] Terrence W. Pratt. Pair grammars, graph languages and string-to-graph translations. *ournal of Computer and System Sciences*, 5(6):560–595, 1971.

[Pro] Prosyst Software AG. mBedded Server 5.x. `http://www.prosyst.de/solutions/osgi.html` (03.03.2005).

[Pro05] Project group REclipse. A Reverse Engineering Framework for Eclipse. `http://www.reclipse.org/`, 2005.

[Reu01] Ralf H. Reussner. *Parametrisierte Verträge zur Protokolladaption bei Software-Komponenten.* Logos Verlag, Berlin, 2001.

[RHH05] Ralf Reussner, Jens Happe, and Annegret Habel. Modelling parametric contracts and the state space of composite components by graph grammars. In *FASE*, pages 80–95. Springer-Verlag, 2005.

[RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modelling Language Reference Manual.* Addison Wesley Longman, February 1999.

[Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations.* World Scientific, 1997.

[RWT] RWTH Aachen University of Technology, Department of Computer Science III. eHome Group. `http://www-i3.informatik.rwth-aachen.de/ehome`.

[Sch91] Andreas Schürr. *Operationelles Spezifizieren mit Programmierten Graphersetzungssystemen.* PhD thesis, RWTH Aachen, 1991. Dissertation RWTH Aachen.

[Sch94] Andy Schürr. Specification of graph translators with triple graph grammars. Spinger Verlag, 1994.

[Sch03] Christian Schneider. CASE Tool Unterstützung für die Delta-basierte Replikation und Versionierung komplexer Objektstrukturen. Master's thesis, Technische Universität Braunschweig, April 2003.

[Sch05a] Matthias Schmidt. Smarthouse. `http://www.schmidt213.de`, 2005.

[Sch05b] Tim Schwerdtner. Strategien zur informationserfassung für ehome systeme. Master's thesis, Rheinisch-Westfälische Technische Hochschule Aachen (RWTH), 2005.

[Skr04] Christof Skrzypczyk. Beschreibungssprache für eHome-Konfigurationen. Master's thesis, Rheinisch-Westfälische Technische Hochschule Aachen (RWTH), April 2004.

[Sun97] Sun Microsystems, Inc. JavaBeans Specification. Specification, August 1997. `http://java.sun.com/products/javabeans/docs/spec.html` (19.5.2004).

[T-C05] T-Com. T-Com Haus. `http://www.t-com-haus.de/` (26.9.2005), 2005.

[vDY96] Rolf Raven Elmar van Dijk and Fredrik Ygge. Smarthome user interface: Controlling your home through the internet. *ISES*, 8, 1996.

[Wag01] Robert Wagner. Realisierung eines diagramm-übergreifenden konsistenzmanagement-systems für uml-spezifikationen. Master's thesis, University of Paderborn, 2001.

[WB98] Mark Weiser and John Seely Brown. The coming age of calm technology. In Peter Denning and Robert Metcalfe, editors, *Beyond Calculation: The Next Fifty Years in Computing*, pages 75–86. Springer, 1998.

[Wei91] Mark Weiser. The computer for the 21st century. *Scientific American*, 265(3):94–104, 1991.

[X1004] X10. Protocol Specification. `http://www.marmitek.com/en/basisimages/x10_protocol.PDF` (30.5.2005), 2004.

[Zün99] Albert Zündorf. FUJABA (From UML to Java and Back Again). `http://www.fujaba.de`, 1999.

Index