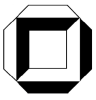


Universität Karlsruhe (TH)
Research University · founded 1825

TECO

Telecooperation Office (TecO)
Institute for Telematics
Faculty of Informatics
Universität Karlsruh (TH)

Diploma Thesis

Modeling Ubiquitous Systems

by

cand. inform.
Konrad Miller

Supervisor:
Prof. Dr. Wilfried Juling

Supervising Research Assistant:
Dipl.-Inform. Till Riedel

Day of Completion: 05/15/2009

I hereby declare that this thesis is my own original work which I created without illegitimate help by others, that I have not used any other sources or resources than the ones indicated and that due acknowledgment is given where reference is made to the work of others.

Karlsruhe, May 15th 2009

Contents

Deutsche Zusammenfassung

Die vorliegende Diplomarbeit beschäftigt sich mit dem Designprozess Ubiquitärer Systeme. Ubiquitäre Systeme folgen einem anderen Paradigma als gewöhnliche Systeme. Es werden Prinzipien verwendet die aus dem Gebiet der Verteilten Systeme stammen. Der Fokus Ubiquitärer Systeme wird sehr stark auf den Kontext ausgelegt, indem die Verarbeitung des gegebenen Kontextes zu einem Mehrwert für den Nutzer führen und nicht das System selbst die Aufmerksamkeit des Nutzers auf sich ziehen soll.

Diese Zielsetzung führt zu veränderten Anforderungen im Designprozess gegenüber dem Designprocess bei herkömmlicher Software. Ein zentraler Punkt ist die Möglichkeit die Funktionsweise Ubiquitärer Systeme zu überprüfen, da die komplexe Struktur zu einer erhöhten Anzahl potentieller Fehlerquellen führt. Auch Fragen bezüglich Merkmalen wie der Skalierbarkeit, Robustheit und Latenz des Systems können nur mittels sorgfältiger Analysen beantwortet werden.

Die Arbeit wendet sich an Personen im Gebiet der Softwaretechnik und an Entwickler Ubiquitärer Systeme. Sie basiert auf Technologien aus dem Bereich der Modellgetriebenen Softwareentwicklung.

Der Designprozess ubiquitärer Systeme ist eine komplexe Aufgabe, die viel Zeit in Anspruch nimmt, da Aspekte, die zu Fehlern und Problemen führen leicht übersehen werden können und somit potentiell viele Iterationszyklen, die den gesamten Designprozess beinhalten, durchgeführt werden müssen.

Ziel der Arbeit ist es diesen Prozess zu verbessern, indem auf Domänenebene gearbeitet wird und ein Großteil von Sourcecodes automatisch generiert wird. Dies gilt vor allem für Sourcecode verschiedener Analyseplattformen, die dem Test des Designs auf gute Erfolgsaussichten dienen. Somit soll eine schnelle Fehlererkennung auf verschiedenen Ebenen, zu einem frühen Zeitpunkt im Entwicklungsprozess, ermöglicht werden.

Im Gegensatz zu dem Gebiet herkömmlicher Software, die auf einem Rechner basiert und nicht auf verteilten Systemen, fand im Bereich verteilter Systeme bisher kaum Forschung im Bereich der Modellgetriebenen Softwareentwicklung statt.

Bisherige Ansätze auf diesem Gebiet haben einen deutlich anderen Fokus als die vorliegende Arbeit. Dieser liegt z.B. auf dem automatische Verteilen einer Anwendung auf verschiedene Knoten oder dem Generieren von Code für Middlewares, allerdings nicht auf der verbesserten Analyse während der Design Space Exploration und der Generation von Zielplattform-code.

Während der Bearbeitung der Arbeit wurden bestehende Systeme analysiert, mit dem Ergebnis dass eine neue Domänenspezifische Sprache zur Beschreibung Ubiquitärer Systeme benötigt wird welche es ermöglicht die zwei zentralen Komponenten dieser Systeme, Kontext und Verteiltheit, zu modellieren. Darauf wurde ein Metamodel für UbiML, einer solchen graphischen Sprache, entworfen. UbiML hat das Ziel, diese Domäne flexibel genug zu beschreiben um weit anwendbar zu sein und präzise genug zu sein um leicht transformierbar zu bleiben. Zusätzlich wurden ein Editor für diese Sprache und eine Transformation zur Netzwerksimulatorplattform GloMoSim implementiert.

Die Arbeit wurde qualitativ untersucht. Dazu wurde eine beispielhafte Design Space Exploration durchgeführt. Im Rahmen dieses Beispiels konnte funktionierender GloMoSim-Code erzeugt werden. Da es sich bei der in GloMosim verwendeten Programmiersprache Parsec um einen C-Dialekt handelt liegt die Vermutung nahe, dass es auch möglich ist auf diese Weise Zielplattform-Code zu gewinnen. Anhand diesem und anhand der Modellierung eines weiteren realistischen Beispiels konnte gezeigt werden, dass UbiML flexibel und ausdrucksstark genug ist um Ubiquitäre Systeme zu beschreiben während sie präzise genug ist um zu prozeduralen Quellcode transformiert zu werden. Die Entwickler werden während des Entwurfs Ubiquitärer Systeme nicht nur dahingehend unterstützt dass größere Teile von Sourcecode für z.B. Simulationen generiert werden können, sondern zusätzlich durch die Möglichkeit das Design einfach zu refaktorisieren. Diese Eigenschaften machen UbiML zu einer guten DSL um eine Design Space Exploration für Ubiquitäre Systeme durchzuführen.

1. Introduction

Nowadays it is hard to imagine to work without the aid of computers. The vision of Ubiquitous Systems is to advance further and support us in our daily lives while becoming practically invisible to us [Weis99]. The new paradigm is to have multiple small, ubiquitous computers surrounding us, configuring themselves instead of few big ones that need our attention. This new paradigm requires changing development practices to cope with the arising challenges. One does not develop software for a single platform anymore, but for a potentially very heterogeneous system of nodes with a lot of new points of failures. Today it is not possible to easily explore the design space of Ubiquitous Systems in respect of errors and implementation.

1.1 Problem Definition

The design process of Ubiquitous Systems, especially the design of large scale Wireless Sensor Networks (WSN) has been practically based on trial and error until now.

Systems are often designed and tested in a small-scale manner in a controlled environment. The results gained from testing those unrealistic prototype systems are then used to simulate a large deployment of the system with the intent to test its scalability, often with poor results concerning predictability of the real system's behavior. An example project in which this caused major problems is described in [TMSEFH06]. In the project a system has been implemented which enables vehicles to automatically enter a motor-way coordinating the entrance with the nearby vehicles which are already on the road via an ad-hoc network. This system works well with a small number of involved cars, but failed when field-tested in a realistic environment on an American freeway.

Many different analysis methods and platforms exist and they can be used to study various details of a designed system. A major weakpoint of the current design process is however, that all of the analysis platform specifications like

simulation-code, the definition of Petri-Nets or Process-Algebras as well as the code for the actual prototype and product itself are all hand coded separately. In general, only a small portion of this code can be reused.
This makes a design space exploration very cumbersome with long-lasting iteration cycles.

1.2 Objectives

The ultimate goal would be to create a Meta Model and all necessary tools to describe Ubiquitous Systems and their instantiation in a way that allows transformation to all useful analysis platforms and code generation for all reasonable target platforms. With a huge repository of well designed components and protocols at their fingertips, software engineers could easily explore the design space of their targeted product and focus on the design instead of the implementation.

The first and one of the most important steps on the way to reach this vision is to create a well reasoned Meta Model, a domain specific language (DSL) for describing Ubiquitous Systems. This is the main objective of this thesis.

1.3 Methodology

Model Driven Software Development (MDSD) pursuits a similar vision. Therefore, an evaluation whether formalisms and frameworks used in the field of software engineering can be reused or modified to support modeling Ubiquitous Systems will be done beforehand.

To show the applicability of the DSL to Ubiquitous Systems a simple Ubiquitous System will be instantiated. The applicability to other, different types of Ubiquitous Systems, will be shown also.

Since the main goal of the DSL is the possibility to transform its instances to adequate target models, a possible transformation to simulator code will also be discussed.

1.4 Contribution

The main contribution of the thesis is a structural and functional analysis of Ubiquitous Systems, and the design of a domain specific language (DSL), namely *UbiML*, which can be used to describe Ubiquitous Systems in a way that makes transformation to different analysis platforms and ultimately the target platform possible. A graphical editor for the DSL was developed using the Eclipse Graphical Modeling Framework.

To demonstrate the functionality of UbiML a simple Ubiquitous System was instantiated using the developed graphical editor. Furthermore, a transformation of the instance to a simulation platform (GlomoSim) is described.

In short, this thesis provides a basis for Model Driven Software Development in the field of Ubiquitous Systems.

1.5 Thesis Outline

The rest of the thesis is structured in the following way: Chapter 2 gives the reader background information about Ubiquitous Systems and Model Driven Software Development. In Chapter 3 the requirements that UbiML needs to meet and possible use of existing meta models, domain specific languages and tools are elaborated after describing the current and a desired design process for Ubiquitous Systems in more detail. It also points out related work and their capabilities and limitations leading to central aspects in the design of a Domain Specific Language for Ubiquitous Systems. Chapter 4 gives a high level description of the semantics of such a Language, UbiML, which was designed in the context of this thesis. The following chapter 5 describes how the implementation of UbiML was done in Eclipse GMF. UbiML is evaluated in Chapter 6. A use case of how a UbiML enabled design process may look like and the description of a transformation to GloMoSim can also be found there. The thesis concludes with chapter 7, where the proposed solution and its evaluation is summarized. This chapter also points out limitations of the solution and future work to be done.

2. Background Information

This chapter provides background information about Ubiquitous Systems and their current design and verification cycle in general. Furthermore, it introduces the field of Model Driven Software Development (MDSD) with their primary nomenclature.

2.1 Ubiquitous Systems and its Nomenclature

The term *Ubiquitous System* was coined in 1988 by Mark Weiser in [Weis99] where he foresaw Ubiquitous Computing to enhance "computer use by making computers available throughout the physical environment, while making them effectively invisible to the user". *Embodied Virtuality* refers to this "process of drawing computers out of their electronic shells" [Weis99]. Computer devices become smaller and smaller, they are embedded into our live and are disappearing from our visual perception.

Mark Weiser also coined the term *Calm Technology*. It refers to technologies that let the user chose which information shall be in the center of his attention without losing peripheral awareness of other information. The goal is to stay focused without neither missing important other information nor being distracted by them. A good real-life example of a *Calm Technology* is a water level indicator of an indoor plant. If information about the plant's need for water is wanted, one can check the water level indicator, but if the current task is unrelated to the plant's need for watering, the indicator is unobstructive enough not to draw attention to itself. Ubiquitous Systems shall support humans in their every day life without actively using them or needlessly disrupting their user.

A recent vision in this field is the *Internet of Things*. It refers to networking many objects from the everyday life such as household appliances and to equip those objects with their own intelligence and possibility to communicate with other objects in their environment with the goal that those objects organize their own processes autonomously and collaborately.

Typical characteristics of Ubiquitous Computers include their low computing power and memory, finite energy due to the use of batteries and small form factor. Furthermore, usually unreliable wireless communication with limited network bandwidth and a peer to peer/mesh-like, dynamic and ad-hoc topology is used. Computation is preferably done collaboratively in the network but can also be done on a dedicated processing machine or sink where the data is accumulated.

Recently, a numerous variety of such systems has been designed, simulated and deployed by the research community. They range from Wireless Sensor Networks (WSN) like Collaborative Business Items [DRBS+07] to wearable computers providing an augmented reality [SMRL+97]. Older systems, like the cellular telephone are already integrated in our lives.

Most Ubiquitous Systems can be separated into four functional parts: sensing, processing, networking and acting. The *sensing* and *processing* parts detect the user's intention. *Networking* is used to be able to collaborate with other nodes and the *acting* part supports the user in some task. A simple example would be dialing a telephone number (*sensing*) and *processing* the pressed buttons to detect that the user tries to call somebody, followed by the *action* of establishing a connection to the other person's telephone. *Networking* is used to establish the call and to transmit the speech itself.
A more sophisticated example would be reading data from sensor nodes like light intensity, temperature or the pressure applied to chairs to detect a conference in a room with the intent to indicate the conference on a display at the conference room's door in order to avoid disturbance.

The network traffic in such systems differs significantly from the one observed in the Internet due to the very different usage scheme [Feng04]. Often, Ubiquitous Systems contain sensor nodes which only forward collected data to other nodes or a sink. This data can be transformed in different ways along the path to the sink. Examples include compression and data aggregation. The usage of gateways is also common. Data storage can be done in many different places like on the node itself, on a gateway or in a sink. The design space is huge and its solutions differ greatly in their performance [Mill08].

The following paragraphs explain Ubiquitous System related terms and also give an overview over the most important techniques used in the design of Ubiquitous Systems.

2.1.1 Context and Real World Awareness

Context Awareness is the key concept of Ubiquitous Systems. Many different authors have defined the contextual properties of Ubiquitous Systems.

Bill Schilit defines the three important aspects of *context* to mobile computing [ScAW94] as:

- **Connectivity:** What resources are nearby?
- **Social situation:** Who you are with?
- **Location:** Where you are?

Day et al. [ADBD+99] suggest the following three very similar categories for context:

- **Computing context:** This category contains mainly hardware related properties, but also properties that are generated by the hardware with the goal to compute or communicate. Typical examples are processors, memory size, input/output devices and network connectivity.
- **User context:** The context related to interaction between artefacts is located in this category. User context includes properties like the own location, the location of nearby systems and people and the social situation.
- **Physical context:** This category contains the contectual properties that are not the Ubiquitous System itself or its users, but are generated by the environment. Examples for physical properties are the noise level in a certain location, lighting and temperature.

Krogstie's categories [KMPE05] (cf. figure 2.1) are more fine grained:

- **Spacio-temporal context:** Properties like time, movement speed and location.
- **Environmental context:** Similar to Day's Physical Context and Shilit's Connectivity, but also includes nearby services and the network connectivity which is placed in the User and Computing contexts by Day and Shilit.
- **Personal context:** Physical and mental state of the user. Tired, angry and sleeping are examples.
- **Task context:** Current goals, tasks and actions.
- **Social context:** Information about relatives and friends and the users current role or status (i.e. "watching TV" or "working"). Sensitivity of the system to the user's identity is drawn from the social context.
- **Information context:** Available global and local information space.

Context Awareness refers to the idea to use information about the *context* of a system to adapt its behavior [ScAW94].

Context is usually drawn from sensors in combination with classification (figure 2.2). One example of a *Context Aware* system is the "Intelligent Home" [eal99]. Another example would be a different rendering of web pages dependent on the viewing device. To achieve that, one could write a Java-script which queries the browser type and adapt the rendering to the target browser.

A *sensor* is a piece of hardware with the ability to sense a physical phenomenon such as heat, light or position and transfer it into a signal which can be further manipulated and worked with. The procedure is called *sensing*.

The discovery of a phenomenon is called *detection*. Often detection is done via sensor readings in combination with classification techniques.

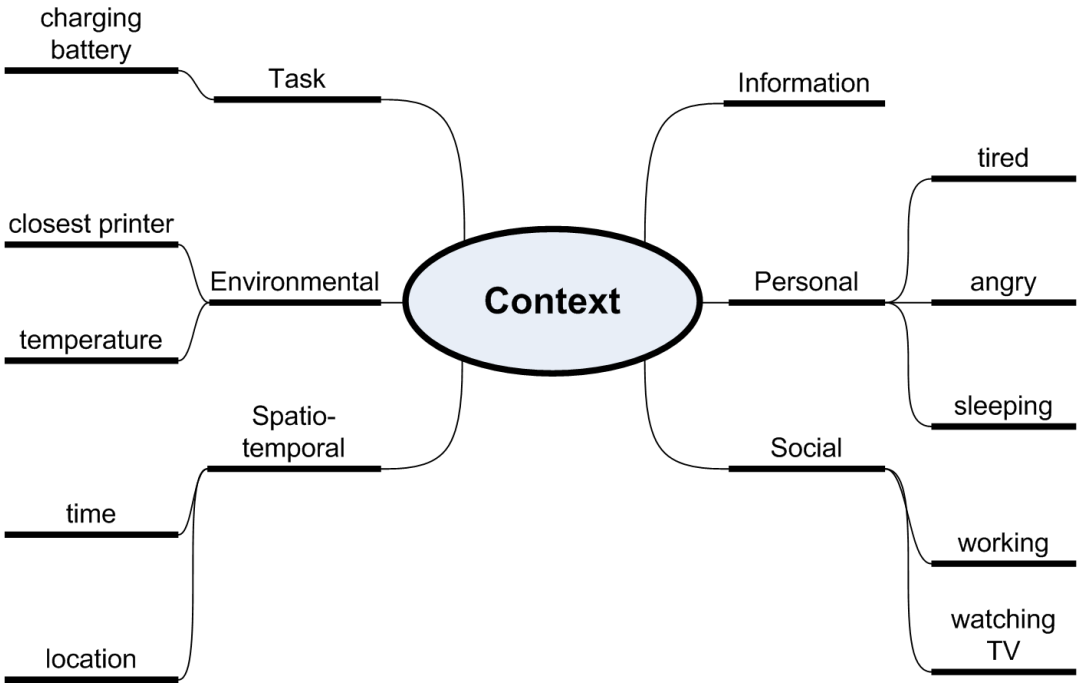


Figure 2.1: The six categories for context defined by Krogstie.

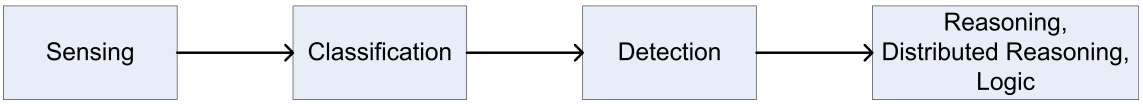


Figure 2.2: Context Acquisition with Sensors.

There are many ways to classify data with different advantages and drawbacks. The used algorithms are mainly results of research in the field of Artificial Intelligence and Machine Learning.

A subset of frequently used classification techniques includes:

- Decision Trees
- Artificial Neuronal Networks
- Bayes Filters
- Fuzzy Logic
- Logistic Regression (Maximum-Likelihood-Estimation)

Inaccuracy of measurements and the dependence on seemingly orthogonal or unknown parameters make classification a hard task. All those classificators are actually estimators of the context that led to the measured inputs with imprecise outputs.

Tracking produces a series of measurements over time to estimate the state of a physical entity. One example for tracking is producing a movement vector from multiple position triangulations of an object. A sample application would be the tracking of packages at a parcel delivery service.

2.1.2 Artefacts

An *Artefact* is an object — a mobile device which a human can interact with. Interaction can be direct as it is the case by pressing buttons mounted on an object. It can also be indirect through the detection of context.

A typical example of an artefact is a physical item equipped with a *sensor node*. *Sensor Nodes* contain at least one sensor and computing/communication functionality which usually includes a processor, memory, power supply (battery) and a network radio to communicate the own state to other artefacts.

Common sensor node hardware ranges from embedded computers with 400MHz Intel XScale processor, 64MB SDRAM, 32MB flash memory, IEEE802.11b WiFi and a 6V DC battery with 105Ah total current draw like an eXtreme Scale Stargate [Aror05] to small UC Berkeley Mica Mote using a AtMega128 microcontroller running at 4MHz and 4KB RAM/128KB flash memory available [HiCu02].

2.1.3 Networking

Since Ubiquitous Systems contain many subsystems that need to be connected with each other to be able to cooperate, networking is an essential part for Ubiquitous Computers. Most techniques in Ubiquitous Systems are the results of wireless sensor network and mobile ad-hoc networking research in general.

In systems with a highly dynamic topology due to mobile nodes it is often preferable to address a service, or data (*Data Centric*) instead of a network address (*Address Centric*). The use of Data Centric Addressing can ease the communication in Ubiquitous Systems. One is often interested in the context of a certain location, not in the context of a certain node number.

Querying the temperature of node #23 would be Address Centric, while one would query which nodes exceeded 25°C recently or the temperature of surrounding nodes in a Data Centric approach.

Data Centric routing can be achieved through anycast protocols by assigning an anycast address to each service/data-type in the network and by using special anycast routing protocols [Proc99].

2.1.4 Execution

The fundamentals of execution in Ubiquitous Systems come from the field of Distributed Systems. An important aspect of execution in Ubiquitous Systems is context based execution, which refers to the paradigm that the execution flow may change based on the context it is executed in.

Data is preferably processed in the network, close to the point where it is generated as opposed to using a dedicated data sink and processing machine — which is typical for WSNs.

In-Network Processing can have a big impact on scalability by avoiding or mitigating bottlenecks, e.g. on network bandwidth through aggregation or compression. These and other vantages also impact energy usage and performance of the complete system.

Collaboration between the artefact that share a common context is used to offer services concerning this context. An example would be offering temperature distribution information in the context of a room using multiple temperature sensors on different artefacts. Due to very limited computing resources, data is often processed collaboratively by many nodes using distributed algorithms.

Another reason for *Collaborate Processing* is that sometimes information can only be extracted from multiple nodes' data and no node could provide the data alone. One example for this case would be triangulation, where one uses three distance readings from spatially different reference points to calculate the position of a target object. The Implementation of Collaborative Processing Algorithms is usually complex.

A *state* is a distinguishable configuration of sensor informations which can depend on previous sensor configurations. Transition to a state can be purely internal, but it can also trigger an action in the Ubiquitous System which might lead to further state changes.

2.2 Model Driven Software Development

Model Driven Engineering (MDE) separates the specification of system functionality and technology, enabling integration and interoperability into/between different platforms via transformations and code generation. The development approach is to specify an abstract model and generate program code from it.

Definition 2.1 Model

"A formal representation of entities and relationships in the real world with a certain correspondence for a certain purpose." *Stachowiak* [Kühn05]

The important aspects are:

- **mapping:** based on a real entity
- **abstraction:** only describe important details, remove everything else
- **isomorphism:** conclusions hold for the real world
- **pragmatics:** the model is usable in place of the original for some purpose

---

The first tools to support this development paradigm were developed in the 1980s and became generally known as *Computer-Aided Software Engineering* (CASE) tools. Unrealistic expectations, a lengthy training process, weak repository controls and inadequate standardization [Vari96] are some of the problems why CASE wasn't widely accepted by the software development community.

Model Driven Software Development (MDSD) is an advancement of the CASE approach. It uses *Domain Specific Languages* (DSL) to describe a problem domain. A modeling language which can be used in arbitrary contexts (domains) — a general purpose modeling language — is either of infinite size or as problem-unspecific as a conventional 3GL programming language. It would be of no more

use than modeling the problem in the programming language itself. *Domain Specific Languages* aim to model a domain in a more concise, Platform Independent way. The result is a so called Platform Independent Model (PIM). DSL instances can then be transformed into target models like code of a higher programming language. This target model is referred to as Platform Specific Model. This process is depicted in figure 2.3.

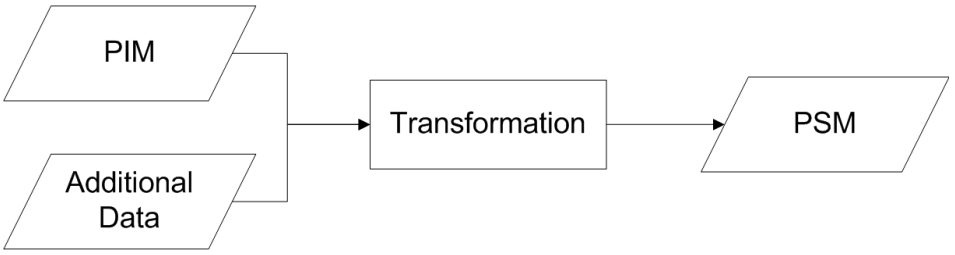


Figure 2.3: The relationship of Platform Independent Model (PIM), Transformation and Platform Specific Model (PSM). The PIM is transformed to PSM with the help of additional information like source code snippets.

The main goal of a Domain Specific Language is the ability to express a particular type of problems or solutions precisely and easily, thus the improvement of the software design process and the software product. In MDSD, the model (specified in a DSL) is the primary artefact. It is what is explicitly specified, developed and versioned — not the target platform source code. The model is used for documentation, communication, analyses, reasoning and code generation.

MDSD aims to improve [vDKV00]:

- reliability and testability
- maintainability
- portability
- reusability
- documentation and communication
- optimization and validation at domain level
- development speed

DSLs and MDSD in general are not a silver bullet, though. There are some disadvantages compared to the traditional development approach. The most important ones are that firstly more time and effort needs to be undertaken specifying the system. Secondly, transformations need to be written or adjusted which often requires a reference implementation. Furthermore, it is not realistic to assume that all target code can be generated automatically, some code will always need to be hand-coded for each target platform.

A well known example of a DSL is the Extended Backus-Naur-Form (EBNF) which can be used to formalize context-free grammars. A grammar specified in EBNF

can then be transformed into a parser by a tool called Bison [biso]. The parser, written in C-code, accepts the specified language. The target model in this example would be the programming language C and the running binary parser a runtime instance of the model.

Model Driven Architecture is a MDE initiative launched by the Object Management Group (OMG) in 2001. It provides guidelines how to define models at various levels of abstraction (figure 2.4).

Definition 2.2 Meta Model

A Meta Model is the language definition of a model [Kühn05]. It describes which parts a model consists of and is used to build valid models.

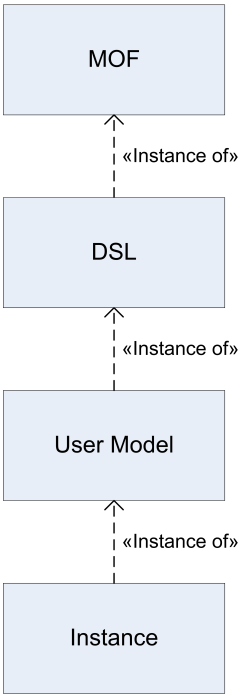


Figure 2.4: The OMG defines 4 layers of abstraction. The Meta Object Facility (MOF) is an instance of itself. The Domain Specific Language (DSL) is an instance of the MOF and describes the User Model which in turn describes an Instance of the problem domain.

The model layers suggested by the OMG are:

- **M3 (MOF):** The Meta Object Facility (MOF) is a Meta Meta Model and describes how to specify Meta Models (definition 2.2) — it also describes itself and is therefore also an instance of itself. Ecore (cf. section 5.1.1) is a MOF dialect.
- **M2 (DSL):** A Domain Specific Language is an instance of the MOF and is build to describe instances of the modeled domain. In the example this is the EBNF.
- **M1 (User Model):** The User Model describes a concrete problem and its solution in the specific domain modeled by the DSL. In our example this is a concrete grammar written in EBNF.

- **M0 (Instance):** The Instance is the the lowest level of the modeling hierarchy. It can be executed or transformed into an instance of a target model. This is a running instance of the actual parser in the Bison example.

2.2.1 Model-To-Model Transformations

Model-To-Model Transformations are an essential part of MDSD. They enable the separation of the modeled specification from the functionality which is provided in the target model.

Definition 2.3 Transformation
The transformation rules that describe how to automatically generate target models from source models are referred to as a Model-To-Model-Transformation [MeCG05].

The source and target models in the definition can each also be a single model. Of capital importance for a Model-To-Model-Transformation is that it is a computable function. Figure 2.5 depicts the relationship between the Model Hierarchy and a Model-To-Model-Transformation to the C-Programming-Language.

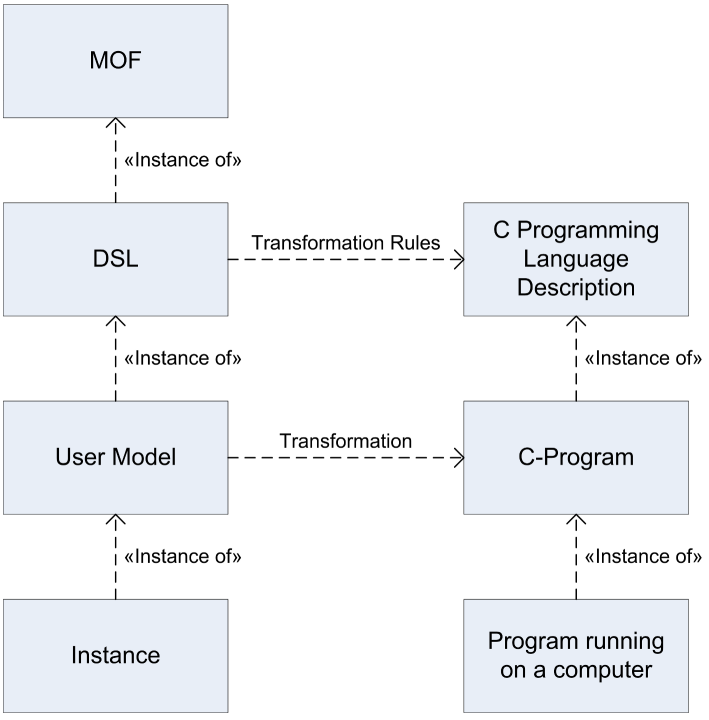


Figure 2.5: A sample Model-To-Model Transformation. The boxes to the left depict the abstraction layers in modeling as defined by the OMG. The boxes on the right side show a sample platform-specific target model — the C-programming-language. Transformation rules are defined on the DSL-layer but the actual transformation is performed on the User Model.

2.3 Summary of this chapter

This chapter gave background information about two topics that are of great importance for the rest of the thesis. First, Ubiquitous Computing was introduced and its key terms were pointed out and explained. Then, the reader was given information about Model Driven Software Design with its history, strengths and weaknesses. The term Domain Specific Languages was defined and the transformation to analysis models and source code identified as its main purpose.

3. Analysis

This chapter first identifies important goals in the design of Ubiquitous Systems that ought to be reached (section 3.1). In section 3.2 it compares the requirements of a design process for Ubiquitous Systems with the requirements of a design process for traditional software systems. A Domain Specific Language is only useful if it can be transformed to useful target models. Sections 3.3, 3.4 and 3.5 give an overview over interesting models for Ubiquitous Systems. Section 3.6 gives an overview over three important Meta Models which are used for software or systems design. Finally, in section 3.7 the chapter compares the introduced models and MDSD-Platforms with the goals of this thesis.

3.1 Design and Verification of Ubiquitous Systems

As Ubiquitous Systems are massively distributed systems with dynamic collaboration over error-prone wireless links among their nodes and hard resource-, particularly energy-constraints, their design can be very challenging. Their design must take this into account and find a balance between orthogonal design parameters. Typical orthogonal design parameters are price, radio distance, reliability and form factor.

Design criteria or principles include:

- **robustness:** Failures, especially false alarms and service outages discourage the use of a system. A low error probability and an adequate response to errors is essential. In Ubiquitous Systems it is much harder to accomplish this due to the many additional points of failures that exist in such systems.
- **longevity:** The maintenance overhead must be minimal. Mechanical survival of hardware and sufficient battery capacity for the systems to run for months or years is desirable.

- **form-factor:** It should be able to embed the needed hardware without drawing too much attention to it. The system should become invisible to the user if it does not need attention.
- **low latency:** The time to reach a certain state or until a certain action is performed must be adequate.
- **scalability:** The system should still function efficiently if the system size increases by several orders of magnitude. It should also be able to handle incremental deployment and usage.
- **interoperability:** The system should be able to operate in a heterogeneous environment.

All of those design criteria have been in the attention of the research community for a long time and many different approaches exist to address them. As everywhere else there is “no free lunch” here either, every optimization always has drawbacks which can influence other subsystems in a way that the “optimization” hurts the overall performance instead of helping it. One example to back this is that removing redundant network traffic might help latency and scalability while sacrificing robustness. Moreover the choice of a certain design decision is often influenced by the usage characteristics of the system and data processed by the system — these characteristics might be different than anticipated or might change in the future, retrospectively leading to a poor choice.

3.1.1 The Present Design-Process of Ubiquitous Systems

Today the design of Ubiquitous Systems is driven mainly by choosing technologies and hardware that seem right for the target application and then the effort to build the system upon this hardware. The development cycle might do multiple iterations on every level in the design process and each cycle takes a lot of time since the entire system must be implemented and tested before the next cycle begins (figure 3.1).

Another approach is to implement the target system in a simulator and do the testing there. Although the focus of this technique should lie on the design of the protocols, a lot of time goes into the implementation instead of the design.

Addressed tests are generally done under unrealistic conditions with a very small subset of features, a small number of nodes and unrealistic environment. The predictability gained from those isolated tests is basically zero [Kobb08].

3.1.2 Desired Design-Process of Ubiquitous Systems

The aspired design process is a top down approach as depicted in figure 3.2. The business case, as the goal of the design process, stands in the center of attention. The first and foremost goal is to formalize it, along with the requirements, constraints, features, and logic that are indispensable to reach the design goal. This is what the Domain Specific Language is used for.

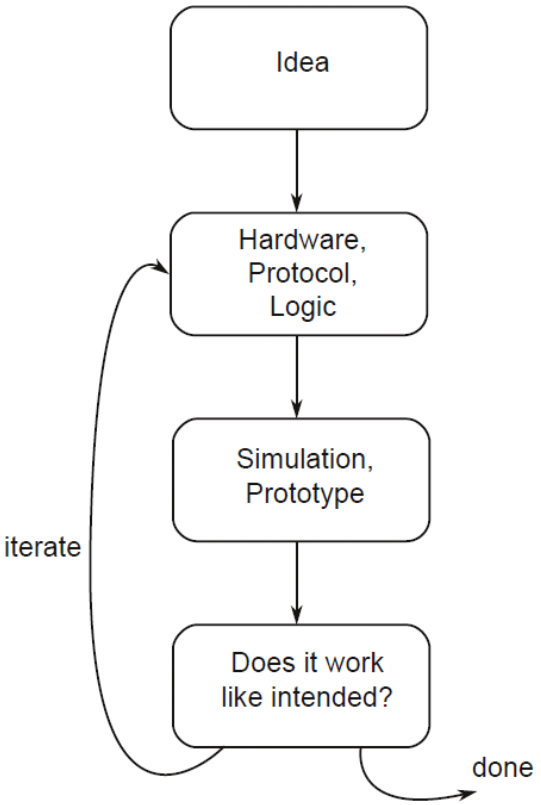


Figure 3.1: Present Design-Process of Ubiquitous Systems. The iteration entails re-writing big portions of simulation and target platform code.

After this the model is fix and the design space exploration can take place. From now on only the hardware and protocols change. Different hard and software components can be evaluated by parameterizing the model in different ways; model-checking and simulation code are generated — rapid deployment can take place by platform code generation.

The iteration cycle is much faster and the developer gains the possibility to easily try out new ideas and to learn about unapparent tradeoffs. Higher startup costs (figure 3.3) are caused by the significant learning effort needed to use a DSL and the required tools. However, one can argue that high-level languages and their respective tools would also require a significant amount of time to learn. These costs are compensated by a higher product quality and lower total cost.

3.2 Requirements of Ubiquitous Systems

The design and the requirements of Ubiquitous Systems differ greatly from that of conventional Software Systems. In short, the main difference is that context and mobility play a fundamental role in Ubiquitous Systems while they play hardly any role in traditional software engineering. This makes existing requirement engineering and software development methods inadequate for Ubiquitous Systems [KMPE05]. Following is a description of the most important properties of Ubiquitous Systems that need to be dealt with when designing a Domain Specific Language.

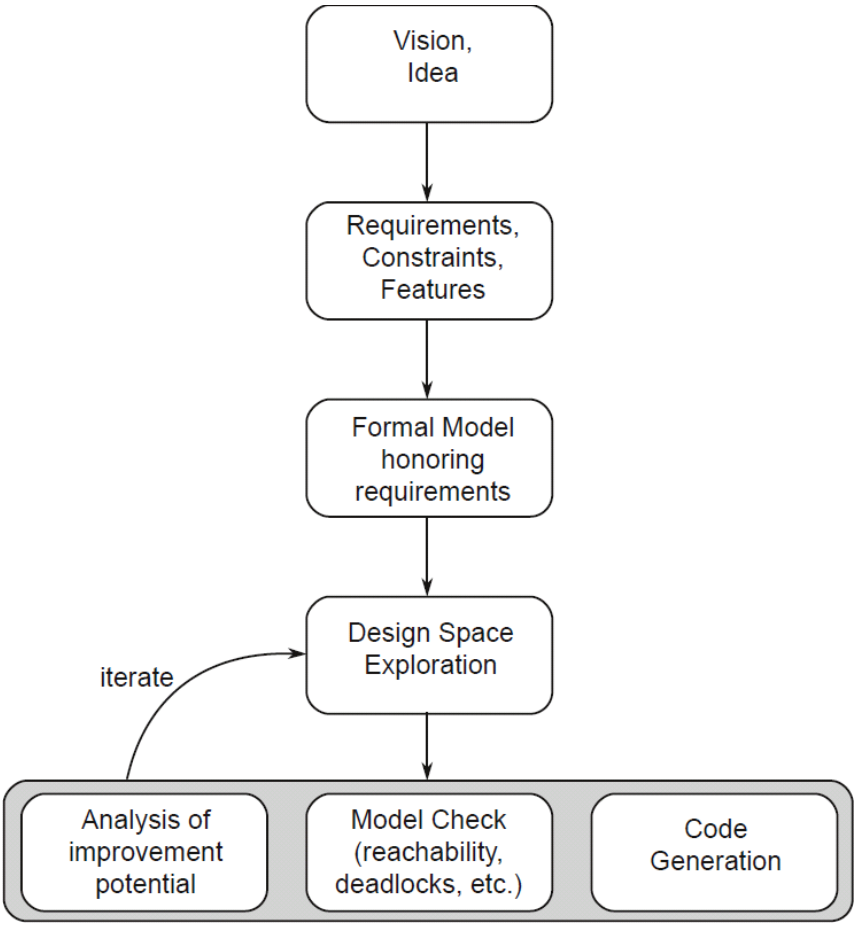


Figure 3.2: Desired Design-Process of Ubiquitous Systems. Specifying requirements and the creation of a formal model requires of significant amount of time. Once the model is created, the iteration step can be done quickly by using code generators.

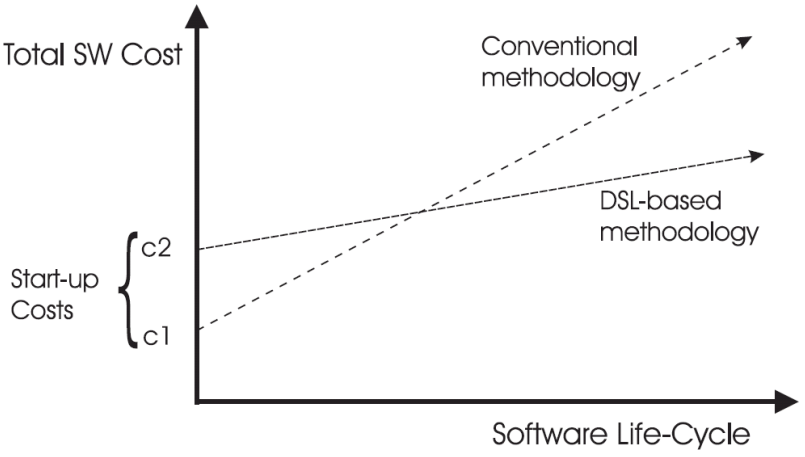


Figure 3.3: Cost comparison between conventional and DSL enabled development [Huda98]. The Model Driven Software Development approach has a higher startup cost (c2) compared to the startup cost of the conventional development paradigm (c1) due to the time needed to formalize requirements and to specify models and transformations. The higher startup cost is accompanied by a lower cost during the life-cycle of the product and often pays off in the longrun.

3.2.1 Dynamic Environment

There are many different environmental parameters that can be dynamic in a particular Ubiquitous Subsystem. The most common one is dynamic location due to mobility of a subset of the autonomous nodes. This mobility leads to the dynamic behavior of a number of other parameters. For nodes with fixed location, a relative location shift can be caused by the mobility of other nodes. The change in node density can then cause the bandwidth to vary dramatically — a very high density may lead to many collisions and thus a very low bandwidth while a very low density may cause network partition. A change of relative positions also has a significant impact on the user environment.

Another major dynamic parameter can be the target platform. It may not be known in advance and it might change over time of the product's life-cycle as new hardware is engineered [FiSa01].

3.2.2 Modeling Context

Context has already been defined in section 2.1.1. The quality of persistence of contextual properties is very diverse. Sensed context is biased heavily by noise and sensing errors for example while user-supplied data is rather reliable but outdated. The context model needs to take this heterogeneity into account.

Kolos-Mazuryk compared different ways to model context in his Ph.D. thesis. In [KMEW06], he suggests that a context model should consist of *Context-attributes*, the specification how these attributes are *changed*, the *reaction* of the service to changes and *activities* which cause attribute changes. Moreover the *actors* which perform the activities and furthermore *interdependencies* between context attributes. UbiML, the DSL designed in this thesis, models context very close to this suggestion — details are given in section 4.2.

3.3 Formal Models

This section gives an introduction to two important *Formal Models* which can be used to analyze different aspects of Ubiquitous Systems. Generally, static and analytical analyses can be easily performed on formal models. But execution and simulation sometimes possible as well, e.g. with Petri-Nets.

3.3.1 Process Calculi

Process Calculi, also known as *Process Algebras*, can be used to formally model concurrent systems — their interactions, communications and synchronizations. Algebraic laws which can be used to manipulate and analyze the model are also provided [Hill05]. Most Process Calculi agree on basic operators, namely sequencing, nondeterministic choice, and parallel composition. They differ mainly in the mathematical representation or extend those basic operators to make other analyses possible [Lutt06].

A model which is formalized in a Process Algebra can be transformed into a graph which can then be verified. Common analyses performed on the graph are [Hill05]:

- reachability analysis
- deadlock- and livelock-freedom
- matching of specification and implementation
- model checking

Well known examples of Process Calculi are Tony Hoare's *Communicating Sequential Processes* (CSP) and Robin Milner's *Calculus of Communicating Systems* (CCS). CSP and CCS both lack timing information and can thus make no assumptions about performance. *Stochastic Process Algebras* close this gap by associating timing information and relative probabilities of state-changes with each activity. A widely used representative of this class of Process Algebras is Jane Hillston's *Performance Evaluation Process Algebra* (PEPA). The expected rate at which an activity is performed is denoted as a negative exponentially distributed random variable. Assumptions about throughput and response time can be drawn from PEPA. It also makes predictions such as how long it takes to reach a certain state. [Baet05]

The main problem in the analysis of large systems using Process Calculi is state space explosion. The following approaches try to mitigate its effects [Hill05]:

- **Model simplification:** replace submodels with simpler submodels via weak isomorphisms and strong bisimulation.
- **Model aggregation:** abstract from submodels by finding equivalence relations and consolidate them to fewer states.
- **Conversion to Product-Form:** Models in product form are solvable efficiently.

Important Process Calculi that emphasize on mobility and on wireless links are the π-*Calculus* [MiPW89] and the *ambient calculus* [CaGo98].

Many tools exist to analyze the discussed target models. Typically static analysis is performed on Process Calculi, but some tools are also capable of performing simulation on the calculus. PRISM belongs to this category. Code Generation can usually not be done from a Process Calculus.

For the PEPA Process Algebra, the *PEPA Workbench* is such a tool. It can construct a deviation-graph and Continuous Time Markov Chains which can then be analyzed further. The *PEPA State Finder* is capable of finding all states in PEPA models. Petri-Nets can be constructed from PEPA with the *Imperial PEPA compiler*. A very popular tool to do probabilistic model checking is an open-source tool called *PRISM Model Checker*. It supports model checking on Continuous Time Markov Chains, Discrete Time Markov Chains and Markov Decision Processes against properties expressed in continuous stochastic logic [Hill05]. PEPA models can be imported into PRISM via an included PEPA to PRISM compiler. The *Möbius Modeling Platform* offers different ways to describe models and multiple techniques to solve and analyze them. It was developed for studying availability, reliability, and performance of networks. It supports Petri-Nets, Markov-Chains and Stochastic Process Algebras as source models.

3.3.2 Petri-Nets

Petri-Nets generalize deterministic finite state automatons to create a simple way in which concurrency can be modeled formally. They offer a graphical notation (figure 3.4) that consists of *places*, *transitions*, *tokens* and *directed arcs* which form a bipartite graph. Arcs connect places and transitions, but never arcs with arcs or transitions with transitions. Places may contain a non-negative number of tokens. To *fire* is the term that describes that a transition takes place — the involved tokens are transferred to the places connected to the firing transition by outgoing arcs. Transitions fire at undetermined but discrete times if all places that are connected via incoming arcs contain at least one token — multiple transitions can fire in the same Petri Net at the same time simultaneously which models concurrency in the system.

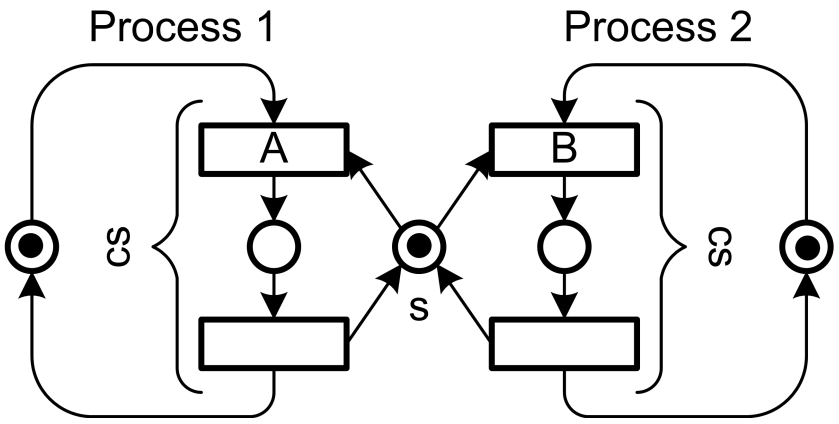


Figure 3.4: Simple Petri-Net synchronizing two processes. Transition A and B cannot fire simultaneously due to semaphore (s), effectively allowing only one process to be in its critical section (cs).

Coloured Petri-Nets (CPN) combine regular Petri-Nets with the possibility to define data types and the use of variables.

There are also extensions that add timing informations to Petri-Nets. In *Timed Petri-Nets*, transitions can be associated with timing information. *Stochastic Petri-Nets* (SPN) add nondeterministic time to transitions which are expressed as a exponentially distributed random variables.

A whole variety of behavioral and structural properties can be analyzed on Petri-Nets using static analysis techniques. Some important ones are [Mura89]:

- **Reachability:** Are there states that can never be reached?
- **Boundedness:** Is the number of tokens bounded for all places? Since places usually represent buffers, this is equivalent to the question if it can be guaranteed that no buffer overflow will take place.
- **Liveness:** Is there, at all times, at least one transition that can fire? This is equivalent to deadlock freedom.
- **Reversibility:** Can the network get back to the initial marking? Together with Reachability this implies livelock freedom.

- **Synchronic Distance:** How closely related are events?
- **Controllability:** Is every marking reachable from any other marking?
- **Consistency:** Is there a marking and a firing sequence in which all transitions occur at least once?

Petri-Nets can be transformed to simulation-code and real platform independent code as well.

3.4 Simulation Models

Simulators which can be used to model and make predictions about distributed systems like Wireless Sensor Networks and Ubiquitous Systems in general have been developed in the past. Unfortunately the analysis is time consuming and one needs to run many different simulations with different parameters to get statistical significance. On the other hand Simulators are insensitive to state space size and thus lack the main downside of Process Calculi.

Adrian Genaid compares different simulation platforms and analyzes their applicability for simulating Ubiquitous Systems in [Ried09]. The most important simulators which can be of use when designing Ubiquitous Systems are introduced in the following.

3.4.1 DEVS

An important representative on this class of simulator models is the Discrete Event System Specification (DEVS) formalism (figure 3.5). DEVS allows for modeling distributed discrete event systems modularly. Every module is a black box which transforms an input signal to a transformed output signal. This represents the module's interaction with the environment. Modules can be hierarchically made of other modules [ASSDGC07].

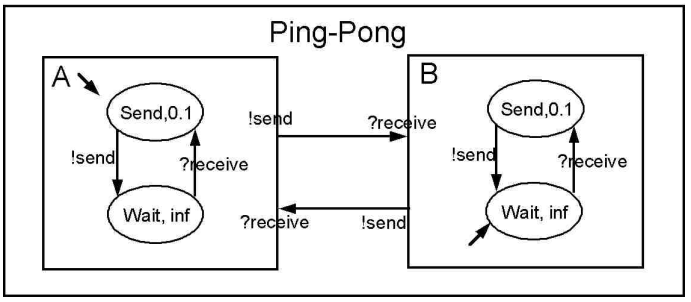


Figure 3.5: A simple DEVS Model describing a Ping-Pong game [Hwan07].

Antoine-Santoni et al. have used a DEVS-based simulator to analyze a Wireless Sensor Network used for environmental monitoring with focus on the network management of the system [ASSDGC07].

3.4.2 Ptolemy

Ptolemy is a component assembly framework with a graphical user interface and a rich simulation backend. The key concept is the definition of models of computations which describe the interaction between concurrent system components. Multiple different such models of computations can be used in a single simulation. The components themselves are called actors in Ptolemy. Their behavior can be specified in Java. The described system can then be analyzed using many different so called domains for properties like dataflow or real-time characteristics. [Lee]

Vergil

Vergil is an extensible, graphical user interface which can be used to enter Ptolemy models. Figure 3.6 shows a block diagram representation of a system which plots a set of differential equations.

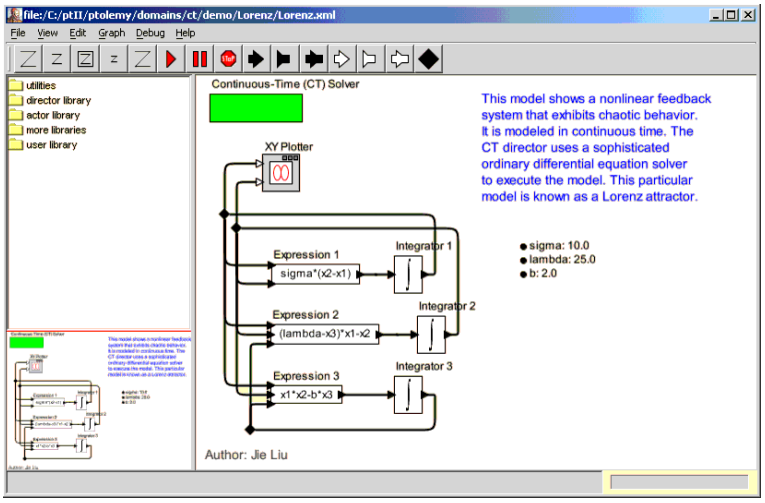


Figure 3.6: Vergil, a graphical user interface for Ptolemy.

Viptos

Visual Ptolemy and TinyOS (*Viptos*) [ChLZ06] is a graphical development and simulation environment build upon Ptolemy. It which can be used to describe embedded systems using block and arrow diagrams in combination with nesC [GLBW+03] (a C-Programming-Language dialect) code. The result can then be simulated using the Ptolemy domains and also in the TinyOS simulator (TOSSIM) [LLWC03].

3.5 Network Simulators

Network Simulators are useful to test protocols in a controlled environment. Network Simulator Code is not a sophisticated model for static analysis. It can however be emulated to be tested on real hardware, e.g. with NS-2.

Network Simulators that can be used to simulate Mobile-Ad-Hoc networks are generally also suited to be used for the networking part of Ubiquitous Systems. Three common Network Simulators are introduced in the following paragraphs.

3.5.1 Network Simulator 2

The *Network Simulator 2* (*NS-2*) is a very popular discrete event network simulator that is well-suited for wireless networks (but also wired networks). It actually comes with a specific class for mobile ad-hoc networks called *MobileNode*. NS-2 can also be used as an emulator, which makes it possible to try out implemented protocols on real networks, or on a network that consists of a mixture of real and simulated nodes. NS-2 is often used to do small-scale simulations. It does not scale well and has a very large memory footprint which makes simulations with more than a few hundred nodes impossible. Small scale-simulations can be undertaken very fine-grained and accurately [Di C03].

3.5.2 GloMoSim

The *Global Mobile Information Systems Simulation Library* (*GloMoSim*) [ZeBG98] is another simulator for wireless mobile ad-hoc networks. Unlike NS-2, GloMoSim was designed to support large-scale simulations which can be run in distributed environments (SMP, Clusters). It is written in *Parsec*, a simulation language that can be used to describe parallel execution of discrete simulation models in a C-like syntax. *GloMoSim* is, as well as NS-2, structured similarly to the ISO/OSI-Reference-Model which makes it possible to easily exchange layers without modifying other parts of the simulation code. Many popular protocols on different layers, like IPv4 with AODV routing or UDP, are already implemented and can be used off the shelf. GloMoSim also includes some basic movement models. The Random-Waypoint-Model and a Trace-Based-Model are two examples. GloMoSim will be used as the target simulation platform in the evaluation in chapter 6.

3.5.3 OMNeT++

OMNeT++ [KSWW+08] is a component based, open source network simulator written in C++. It consists of hierarchically nested modules to reflect the structure of the real system. The modules are implemented as coroutines and thus appear to be running in parallel. Although it also scales well, it is not very well suited for Ubiquitous Systems, since it lacks implementations of important protocols like the ad-hoc routing protocols AODV and DSR.

3.6 Software-Engineering Meta Models

There is a row of general purpose modeling languages and domain specific languages that seem to be well suited for modeling Ubiquitous Systems. This section introduces the most promising ones.

3.6.1 Unified Modeling Language

The *Unified Modeling Language* (*UML*) is a graphical, general purpose modeling language that is used mainly to model object oriented software. It is standardized by the OMG and through ISO/IEC 19501. UML does not strictly bind a semantic to the defined syntax, but it enables developers to communicate and document their software models more easily. Furthermore, some tools are able to generate

programming language code stubs. Constraints can be expressed in the *Object Constraint Language* (OCL).

UML is very large and complex. It takes a considerable amount of time to learn it and even after having fully understood it, UML does not help much modeling a specific problem domain. UML does not embody any domain knowledge in the field of Ubiquitous Systems.

3.6.2 Systems Modeling Language

The *Systems Modeling Language* (*SysML)* is a UML2 profile for systems engineering. It contains a subset of UML2's diagrams with extensions (figure 3.7) and some modifications to fit the needs that system model developers have.

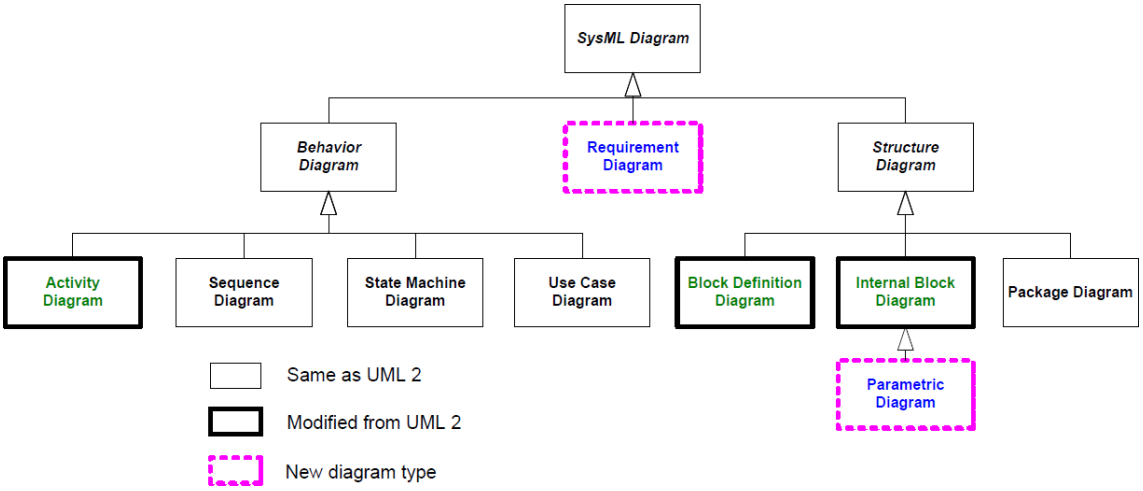


Figure 3.7: SysML: Modifications applied to UML2[FrMS08]

SysML is also standardized by the OMG and, in contrary to UML, it provides semantics. Dynamic in the model can be expressed but dynamic of the modeled entities themselves can unfortunately not. Ubiquitous Systems are highly dynamic though; one example would be mobility. Another reason against modeling Ubiquitous Systems with SysML is, that it — like UML — doesn't contain any domain knowledge.

3.6.3 Pervasive Modeling Language

The *Pervasive Modeling Language* (PervML) is an MDA-approach specifically designed to be used for modeling pervasive systems. It consists of different views and models that form a Platform Independent Model. Requirements and abstract functionality are specified in the *Analyst-view*. The *Architect-view* takes this specification and enriches it by describing how the functionality can be implemented in soft- and hardware. UML State Transition Diagrams are used to specify the behavior of the different components. The PIM can then be transformed into Java-code which can then run on a SOA-Middleware. [Ried07]

The major downside of PervML is, that it is only transformable to a specific target model which is not very well suited for small embedded devices. It moreover does not allow for transformation to any of the numerous different analysis platforms which are very helpful for a reliable and fast design space exploration.

3.6.4 Visual Robotics Development Kit

The *Visual Robotics Development Kit* (VRDK) was designed to be a high-level graphical modeling language for developing applications in Ubiquitous Computing scenarios [KWUB06]. The User Models can than be translated to different programming languages. Currently, there are transformations available to C and C#. The graphical model supports commands that are modeled closely after imperative programming languages. Commands like conditional forks, loops and branches as well as the evaluation of formulas are part of the language, but also commands to describe the parallel execution of different threads. [Ried07]

As described in section 3.2, a very important aspect of Ubiquitous Systems is their extensive interaction with context. Unfortunately, VRDK does not take this into account. The only context that one can access is the location which is not sufficient for complex Ubiquitous Systems.

3.6.5 The Palladio Component Model

The *Palladio Component Model* (*PCM*) is a component-based software engineering (CBSE) framework. It was built to allow predictions about performance like response time, throughput and resource utilization. Hardware details, usage profile and other influence factors on performance are taken into account by specifying them in a parametric way [BeKR07]. The main components PCM contains are a domain specific language implemented in EMOF/EMF (cf. section 5.1) and a performance simulation. Component behavior can be expressed via arbitrary stochastic distribution functions.

Unfortunately, PCM cannot easily be extended by further analysis capabilities due to major deficiencies in the Meta Model. The architecture modeled with PCM is static. Connections cannot move, meaning components cannot dynamically change with whom they are connected. This also limits the possibility to move towards or away from resources, or changing the context of components which is essential in Ubiquitous Systems. Another drawback of PCM is, that it can model one influence factor only. However, more than one factor can be important when trying to model context.

3.7 Deriving Design-Elements for UbiML

Many different Models have been explained in the sections 3.3 through 3.6.

Process Calculi were introduced as a formal Model. Since they are hard to write directly, Process Calculi are rather a target model than a source model in a Model Driven Software Development process. They can be used to answer several questions about i.e. deadlock- and livelock-freedom or the queuing behavior of the network.

Petri-Nets on the other hand could be either a source and a target model in a MDSD process for Ubiquitous Systems. Unfortunately there is no concept to deal with context in Petri-Nets as only processes are modeled. Moreover there is no domain knowledge about Ubiquitous Systems in Petri-Nets as they were

originally created to describe chemistry processes. A transformation to computer-processes is easily possible as the focus lies on describing processes already. This does not only make Petri-Nets an attractive target model, but also an intermediate model in which the procedural part of Ubiquitous Systems could be expressed in.

DEVS introduces a very good concept signals, signal-flows and for describing the interaction with context intuitively. From an abstract point of view, DEVS describes signals and their transformation to other signals. In DEVS there is no dissociation between the description of the systems behavior which is very helpful when exploring the design space of a system.

The Ptolemy framework offers a very rich analysis backend and is very flexible in the way it can model things. On the other hand it is very complex and was not build to describe the domain of Ubiquitous Systems. In fact it is so flexible that every process or system can be modelled.

The introduced network-simulators NS-2, GloMosim and OMNeT++ are specified in general purpose programming languages and are thus entirely unsuitable as source models for a MDSD process. They can be a very useful target model in such a process while studying network protocols in the designed Ubiquitous System though.

UML and the UML profile SysML were also introduced. UML is a general purpose modeling language and is as such too powerful and hard to learn while containing no domain knowledge for being a valid source model candidate. SysML in contrary to UML does contain domain knowledge in the field of engineering. Unfortunately, it is not possible to express a dynamic system with SysML. Only dynamic behavior in a static system can be expressed. Both, UML and SysML are not good candidates for target models either because the analysis capabilities of those languages are very limited.

PervML is coming close to a good source model. Its concepts and implementation focuses very strongly on Java and Java middlewares instead of describing the system neutrally to allow different transformations. VRDK comes even closer, but it in turn does not allow to describe and interact with different types of context but only covers the location.

A MDSD-process similar to the one in the Palladio Component Model is desirable in the field of Ubiquitous Systems, although with a different focus. PCM is already a well developed MDSD-process with a well defined MOF. It focuses on performance modeling. While verification lies in the center of attention in the design of Ubiquitous Systems, performance takes a subordinate role. Business Processes and the integration of the new systems, their semantics and in particular their failure probability — ultimately the price — are subject to modeling rather than software components and their performance.

Another major difference between the described systems and a desired system for modeling Ubiquitous Systems lies in the modeled entities. Ubiquitous Systems are supposed to be integrated into the workflow and hence depend greatly on interaction with humans and other unknown, dynamic systems. Moreover, Ubiquitous Systems are, unlike usual software, used in highly dynamic environments. Mobility, environment changes — and context in general — play a major role.

In a nutshell, a new Domain Specific Language which is capable of easily and comprehensively describing Ubiquitous Systems needs to be created with the following properties:

- Clear Meta Model and a Model Driven Development workflow
- Containment of domain specific knowledge in the field of Ubiquitous Systems
- Possibility to cope with very dynamic systems
- Easy description of various types of context
- Possibility to transform to analysis-models
- Separated Models for modeling the behavior of the system and for it's instantiation

3.8 Summary of this chapter

This chapter first described the design decisions that need to be made when developing Ubiquitous Systems. The typical current design process and a visionary, desired design process were depicted. Existing solutions that might have been used for creating the desired design process were analyzed with the conclusion that a new Domain Specific Language needs to be created. The chapter concluded with guidelines how such a Model must look like.

4. The UbiML Meta Models

This chapter describes the domain specific language UbiML which is the main contribution of this thesis. The language is divided into two Models, the Artefact Model and the Instantiation Model. The former describes all entities in a Ubiquitous System along with their behavior and influence on one another. The latter is used to describe instances of the modeled artefacts and their parameters, i.e. start positions in a simulation and the number of sensor nodes.

The major novelty to existing systems, like the Palladio Component Model, is the very flexible way in which artefacts, their interaction with each other and the interaction with context are modeled.

4.1 The UbiML Workflow

There are formal and informal requirements that UbiML needs to meet. The foremost requirements are the ability to easily specify Ubiquitous Systems and the influences that affect the operation of the System, most importantly the context 3.2.2 and interaction between subsystems.

As described earlier in section 2.1 the typical Ubiquitous System consists of sensing, processing, acting and networking. Mainly these tasks and their properties and constraints need to be easily expressible in UbiML.

UbiML should support developers and engineers in analyzing and documenting their design as well as in the communication with colleagues and in the implementation of the desired system. Moreover, it should be possible to create repositories that make rapid development possible by the reuse of previously designed and analyzed subsystems. Another requirement is that UbiML is flexible enough to be extended to use new target platforms — it should be as independent from specific target models as possible without degenerating to a general purpose modeling language that does not contain any domain knowledge.

When using UbiML to design Ubiquitous Systems, the workflow (figure 4.1) is as follows. After the vision becomes clear the driving force behind the project creates

a high-level description of functionality, features, requirements and constraints. Engineers then take this description, choose hardware subsystems and protocols that seem adequate to reach the vision and map them, along with an anticipated target environment and usage, to an UbiML Model specification. At this point the designed system can be tested rapidly with many different analysis platforms to find errors in the specification of protocols and identify inappropriate design decisions. When the design seems to be solid, code generation can take place. Since mostly not all of the target platform code can be generated, some hand-coded parts need to be added at this point as well. A hardware prototype can be soldered and if the prototype passes all tests in the “real world” the initial development phase is completed.

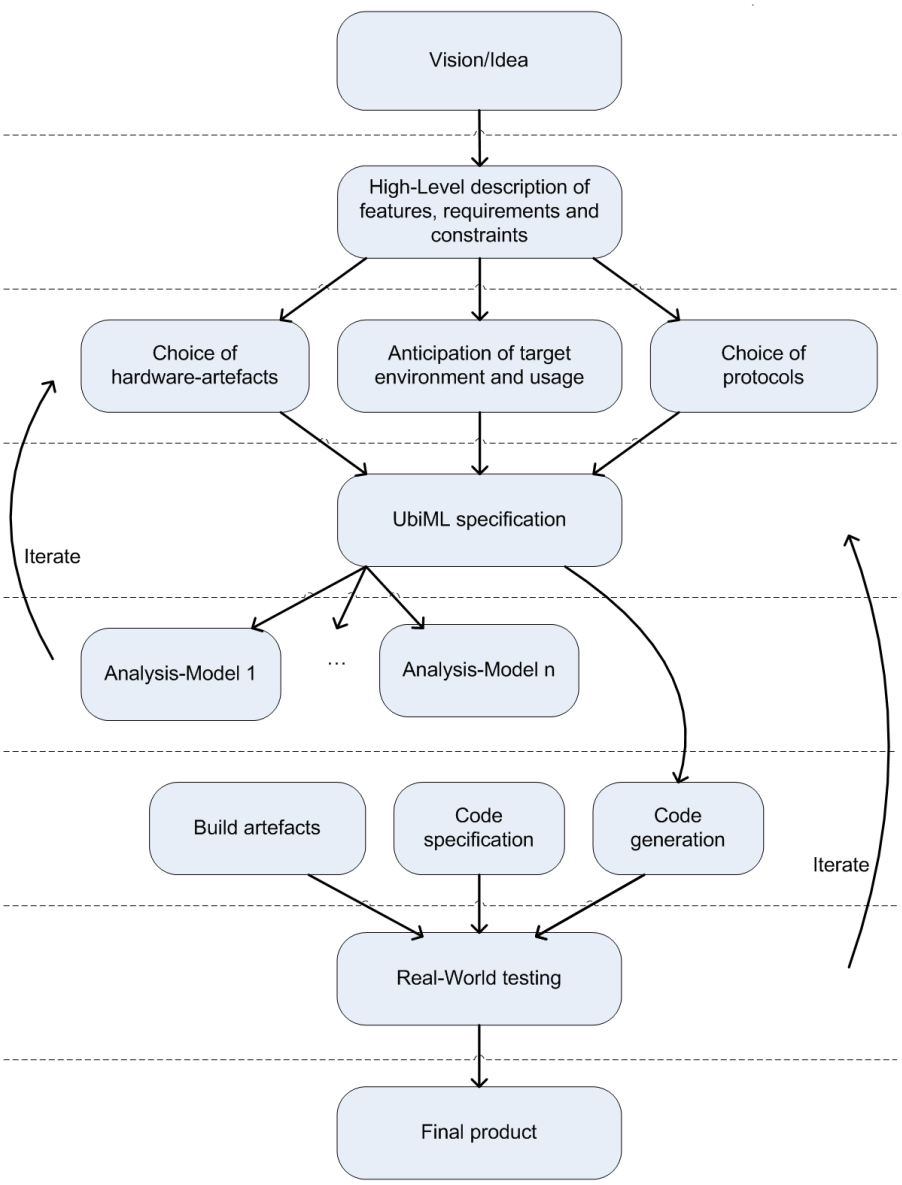


Figure 4.1: Workflow when designing Ubiquitous Systems with UbiML.

Figure 4.2 depicts the tools and models involved from designing and implementing UbiML to its usage. A detailed description of the Eclipse GMF-Tools is given in section 5.1.

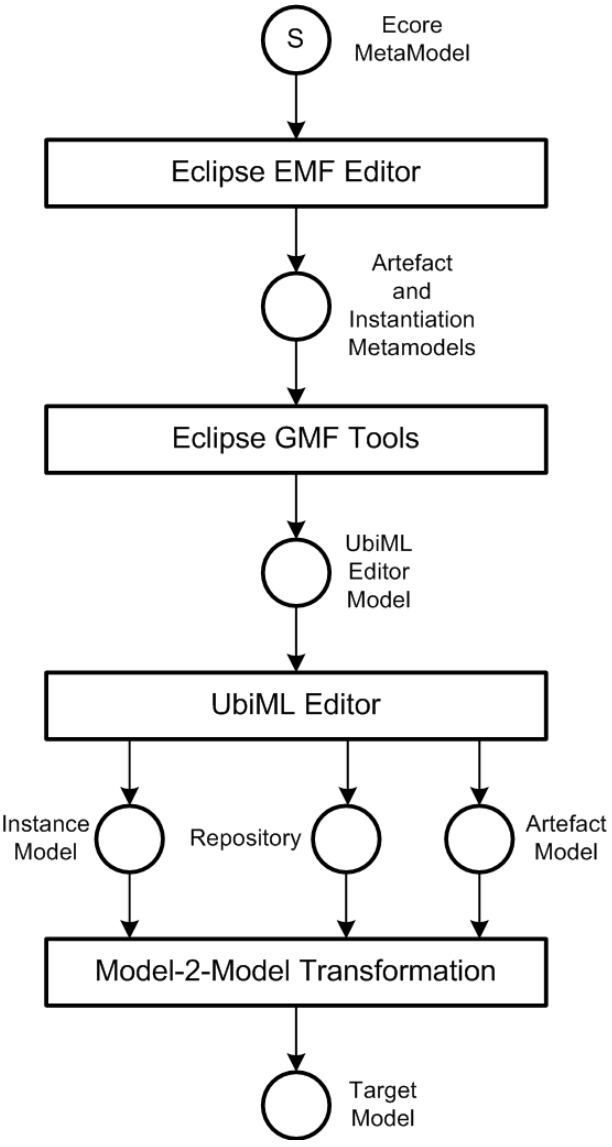


Figure 4.2: Tools and Models in the UbiML enabled workflow. The Instance Model, Artefact Model and Repository reference each other. The GMF-Tools need additional models that define i.e. shapes and tools (cf. section 5.1).

4.2 Artefact Meta Model

The Artefact Meta Model describes the entities in a Ubiquitous System as well as communication, context and constraints. A simplified version of the Meta Model is depicted in figure 4.3, the full Meta Model can be found in figure 5.6.

The Meta Model contains the following conceptual components that will be discussed in greater detail in this section:

- Artefacts
- Computation Models
- Values
- Signals
- Signal Transformations
- Timers
- Assertions

4.2.1 Artefacts

All physical things are modeled as *Artefacts*. Artefacts have a name and can be assembled from other Artefacts — so called *Sub-Artefacts*. Artefacts are the used to structure a modeled entity. It can be thought of as a casing that experiences a context and that contains hardware to run code and to interact with the environment. Typical examples that fall into this category are sensors, radios and a processor which runs the program logic of a system. Sub-Artefacts allow structuring a system for better clarity.

Sub-Artefact Connections

An Artefact is connected to a Sub-Artefact with a directed vertex — the Sub-Artefact Connection. Artefacts and their Sub-Artefacts must form a directed acyclic graph.

4.2.2 Computation Models

Dependent on the connection-type to an Artefact, the Computation Model represents an algorithmic representation of the context model (section 3.2.2) which affects the connected Artefact or it represents a process (i.e. a protocol) that runs on a processor integrated into the Artefact.

The semantic difference of computation and context is reflected in the UbiML Model by defining that context is always passive, while processing can trigger the flow of data. Context can only be interacted with by actively reading it through a processing Computation Model. As a consequence of this, Computation Models which represent context cannot be connected to Timers (section 4.2.6).

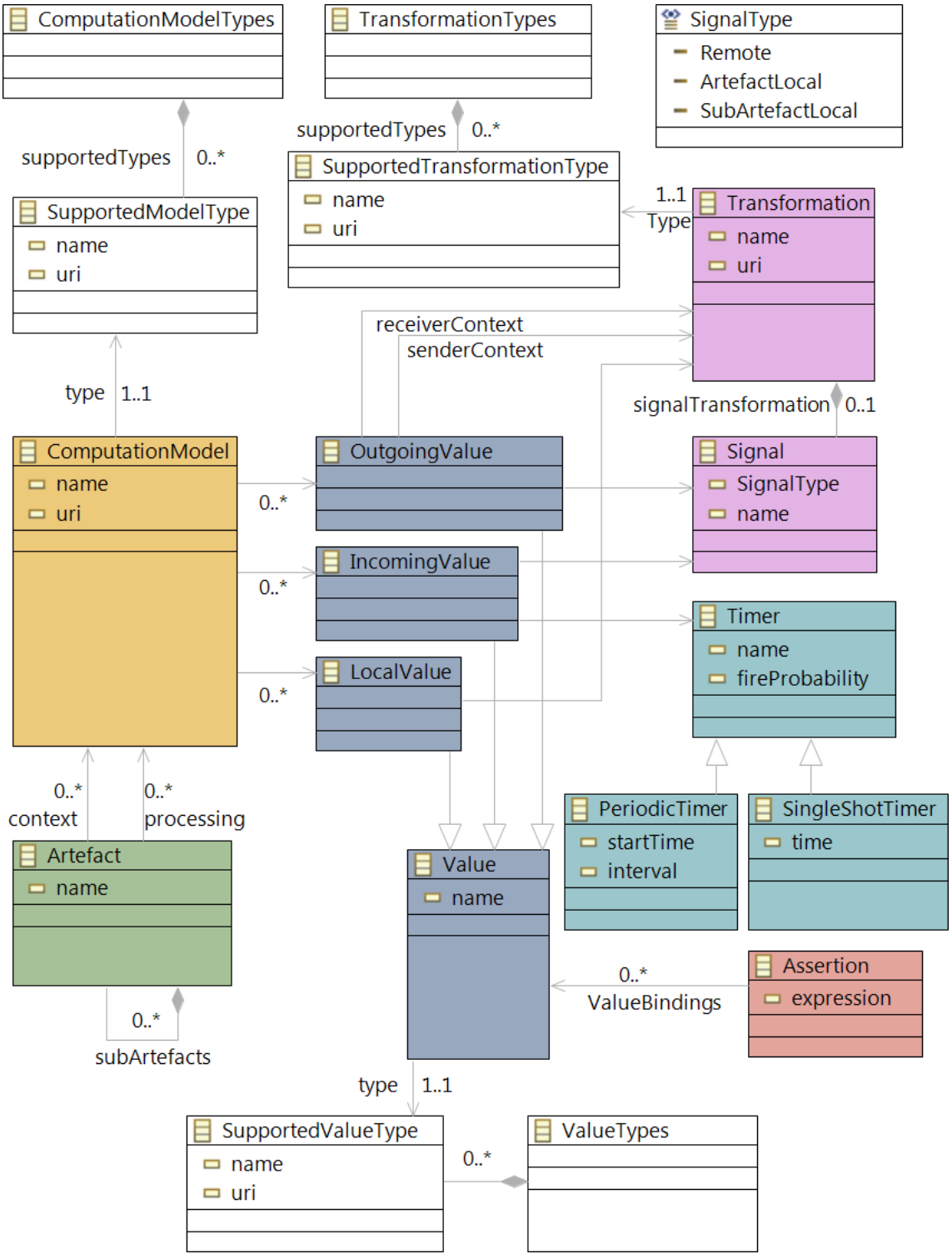


Figure 4.3: Simplified version of the Artefact Meta Model. Connections between components are shown in a simplified form. The Canvas-Root-Node and its connections have been omitted. The full Meta Model can be found in figure 5.6.

The concept of the Computation Model is very close to the DEVS formalism, which as been introduced in section 3.4.1. As in DEVS it is also possible to couple Computations by assigning the outputs of different Computation Models to another Computation Model which can reason on a higher level (see section 4.2.3 for an explanation how inputs and outputs of Computation Models are handled in UbiML).

The representation of the actual computation that takes place in the Computation Model is not specified by the Meta Model and can be chosen in the User Model (cf. 4.2.8). Which representation is appropriate, depends greatly on the aspired target models. I suggest to use generic, graphical representation of algorithms where applicable. Two examples that should work well for many different target models are *Finite State Automata* and as already elaborated in section 6.5 also *Petri-Nets* (cf. section 3.3.2). Target platform source code could be written here as well if supported by the transformation.

Context Connections

Computation Models can be connected to Artefacts by *context connections* in which case the attached Computation Model is an algorithmic representation of its (passive) context. Context in terms of UbiML is the computationally derivable, external state which is a representation of the contextual properties previously defined in section 2.1.1. An example is a function of time which can model the temperature profile of a room over the course of a day.

Processing Connections

If a Computation Model is connected to an Artefact by a *processing connection*, the Computation Model represents a process running on the Artefacts processing unit.

4.2.3 Values

The interface of context and computations (Computation Models) are *Values*. Values can be bound to arbitrary Computation Models. In this binding the name, which can be used to access the Value in the particular Computation Model, is declared. There exist three types of Values: Local, Outgoing and Incoming Values. The type defines the usage. *Local Values* are never used outside of a single Computation Model and therefore cannot be bound to multiple Computation Models. They can be used to hold internal states — variables that are needed internally in an algorithm across calls. Local Values are analogous to static variables in C or private member variables in C++. *Outgoing Values* are used for transferring computation results to other Computation Models and *Incoming Values* are their counterpart that are used to access Outgoing Values.

Outgoing Values can never be read directly, but have to always be read via a Signal (section 4.2.4). This forces the creator of the User Model to think about and model transmission errors and inaccuracies between contextual properties and their sensor readings.

Signal Transformations (section 4.2.5) may also be connected to Local Values which may be used for storing internal state across transformations. Furthermore,

Transformations can directly access the contextual properties (the Outgoing Values of contextual Computation Models) of the sender and receiver of a Signal. This is i.e. useful to apply a distance based fading model to radio signals. This is the only place where a foreign Outgoing Value can be accessed directly.

Local Value Bindings, Outgoing Value Bindings, Incoming Value Bindings

Local-, Outgoing- and Incoming-Values are bound to Computation Models by the Local-, Outgoing- and Incoming-ValueBinding connection, respectively. The connection is also associated with a bindingName — a string which can be used as an identifier to access the connected Value.

4.2.4 Signals

Outgoing Values and Incoming Values are connected by *Signals*. Signals correspond to radios, shared memory and sensor sampling in the real world. There are different types of Signals to reflect the different possibilities in the visibility of emitted signals. Signals can have the types *SubArtefactInstance*, *ArtefactInstance* and *Remote*. *SubartefactInstance* Signals are emitted to all Computation Models of the current Sub-Artefact-Instance. They are not visible to other Sub- or Super-Artefacts of the Instance. Signals of the type *ArtefactInstance* are visible to the entire Artefact-Tree of the Instance, but not to other Instances. *Remote* Signals are potentially transmitted to all Instances in the System.

Signal Sender Connections

Outgoing Values can be connected to a Signal with SignalSenderConnections representing the fact that the value that is assigned to the OutgoingValue through the OutgoingValueBinding is sent using this Signal.

Signal Receiver Connections

Incoming Values can be connected to a Signal with SignalReceiverConnections. OutgoingValues which are connected to the same Signal are copied into the IncomingValue on send.

4.2.5 Signal Transformations

Signals can be further associated with a Signal *Transformation*. The *Transformation* reflects the fact, that sensor-sampling and data-transfers are always error prone and transferred values may change on the way. It is basically a stream transformation — an algorithm or a function of contextual properties. One example could be to specify Wireless-Radio-Signal-Fading as a Rayleigh distribution-function based on the positions of the communicating nodes. Signal Transformation handling can also be handed off if supported by the analysis platform — to a simulator for example. This is done in the example User Model in section 6.2.

Transformation Connections

Transformations are associated with Signals by connecting them with a TransformationConnection. OutgoingValues which are sent to IncomingValues through the associated Signal are then modified by the Transformation.

Sender- and Receiver-Context Connections

Signal Transformations may need to access the context of the sender and/or the receiver of a Signal. In that case the needed contextual OutgoingValues are bound directly to the Signal Transformation with SignalContext- or ReceiverContext-Connections. Each such connection is also associated with a bindingName which is used as an identifier to access the OutgoingValue.

Transformation Data Connections

Signal Transformations may also be connected to LocalValues. A LocalValue is bound to the Transformation by connecting them with a TransformationData Connection. The bindingName Attribute is used as an identifier to access the LocalValue. The LocalValue can used to store internal Transformation-State across calls.

A LocalValue must not be connected to both a Transformation and a Computation Model at the same time.

4.2.6 Timers

Not only Signals can be the source for Incoming Values. Another possibility is to connect *Timers* to Incoming Values. Timers are needed as triggers. When describing a Sensor-Node with the function to detect fires for example, one could use Timers to model polling of a temperature sensor or smoke detector. Another example are timeouts in network protocols. There are two different types of Timers: Periodic Timers and Single-Shot Timers. *Periodic Timers* are appropriate to model polling while *Single-Shot Timers* are the better choice for timeouts. Periodic Timers are instantiated with a fire-probability, start-time and a fire-interval while Single-Shot Timers are instantiated with fire-probability and fire-time.

Timer Receiver Connections

A Timer is connected to an IncomingValue which receives the timer-event via TimerReceiver-connections.

4.2.7 Assertions

Assertions can be bound to arbitrary (even Local) Values and are used to formulate logical assumptions about the system in a *Linear Temporal Logic* [Emer90]. These Assertions are not meant to be transformed to constructs like Assertions in the C-Programming-Language — Assertions across local memory boundaries would not work well there. The main reason for Assertions in UbiML is verification of the modeled system in the design-phase. One could formulate invariants, logical and precision checks as Assertions that need to hold during simulations and/or formal model analysis.

Assertion Binding Connection

The Values in the Assertion-expression are bound to an Assertion through the AssertionBindingConnection. The connection carries a bindingName which represents the identifier which is used to access the bound Values.

4.2.8 Types

All Values, Signal Transformations and Computation Models must have a Type.

Supported Value-, Transformation-, Model-Types

The Type is declared by attaching a SupportedValue-, SupportedTransformation- or SupportedModelType respectively to the entity with a Value-, Transformation-, ModelTypeConnection.

Value Types, Transformation Types, Model Types

There must be exactly one ValueTypes, TransformationTypes and ModelTypes node in the User Model. The Value-, Transformation- and Model-Types must be connected to their Value-, Transformation- or ModelTypes Node via a Supported Value-, SupportedTransformation- and SupportedModelTypeConnection respectively.

4.3 Instantiation Meta Model

While the Artefact Model specifies which parts the modeled system consists of, the purpose of the Instantiation Model (figure 4.4) is to parameterize one instance of the system. The Artefact Model describes which Artefacts exist and the Instantiation Model specifies how many of that Artefact exist and where. While the Artefact Model specifies what constants and variables the User Model contains, the Instantiation Model specifies their values or initial values.

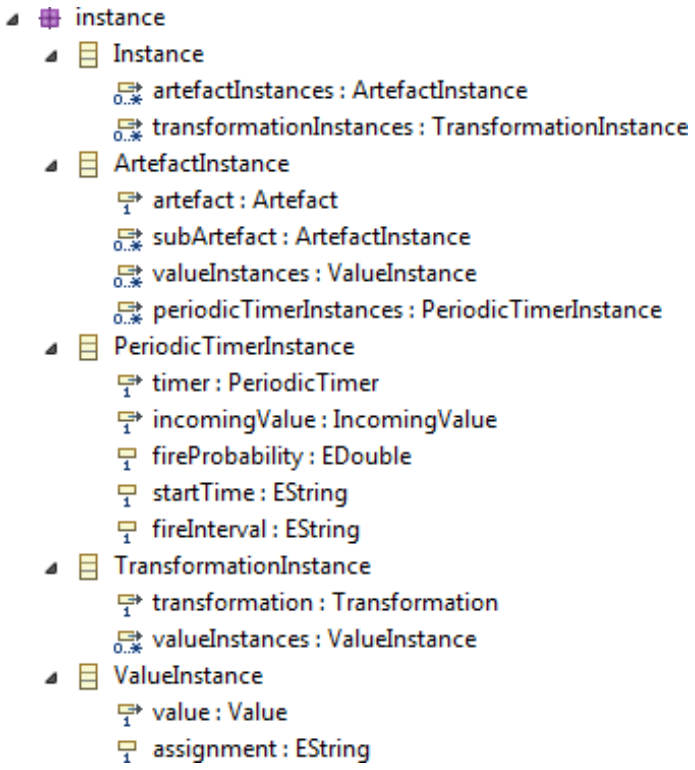


Figure 4.4: The Instantiation Meta Model in the Ecore Diagram Editor.

The Instantiation in UbiML is done in a tree-like editor. Every Root-Artefact-Node (that is an Artefact that no other Artefact is connected to by a Sub-Artefact-Relationship) can be a top-level node in the Instantiation Model. All free values and timers of that Attribute-tree can then be set using Attribute child elements of that top-level node. Every Value element contains a reference to the instantiated model-element and a string value which can be converted to the correct type of the value. An Artefact-tree with Artefact-, and Attribute-sub-nodes represents an instance. If multiple, equal sub-trees need to be specified, the subtree needs to be duplicated.

Transformations and their values can also be instantiated as top-level nodes.

4.4 Summary of this chapter

This Chapter introduced the workflow when designing Ubiquitous Systems using UbiML. It also gave a high-level overview of the components that UbiML is made of. UbiML is a domain specific language that aims to improve the shortcomings in the current design process of Ubiquitous Systems by giving the designer a graphical language in which dynamic context can be easily expressed in.

5. Implementation of a Graphical Editor for UbiML

The *Eclipse Modeling Framework* (*EMF*) is an open source framework for modeling structured data and generating a reflective, dynamic editor for that data. A tree-based editor is generated through a transformation of the the EMF Model to Java source code. The EMF Model is an instance of the Ecore metamodel. The editor is then capable of validating, querying, changing and serializing instances of the data model.

The graphical equivalent to EMF is the *Graphical Modeling Framework* (*GMF*). It is internally based on EMF and the 2D rendering component *Graphical Editing Framework* (*GEF*). GMF consists of six different models that, together, are transformed to a graphical editor. Not all of the models have to be written by hand. Different wizards help with most of the steps.

openArchitectureWare (oAW) is a workflow engine which makes it possible to orchestrate different models and transform them into target models in possibly multiple steps. It also contains languages to describe those transformations.

This chapter describes an implementation of the UbiML Artefact-Metamodel in Eclipse GMF after describing the Graphical Modeling Framework with its models and openArchtectureWare in general. An oAW based transformation of UbiML to GloMoSim will be discussed in chapter 6.

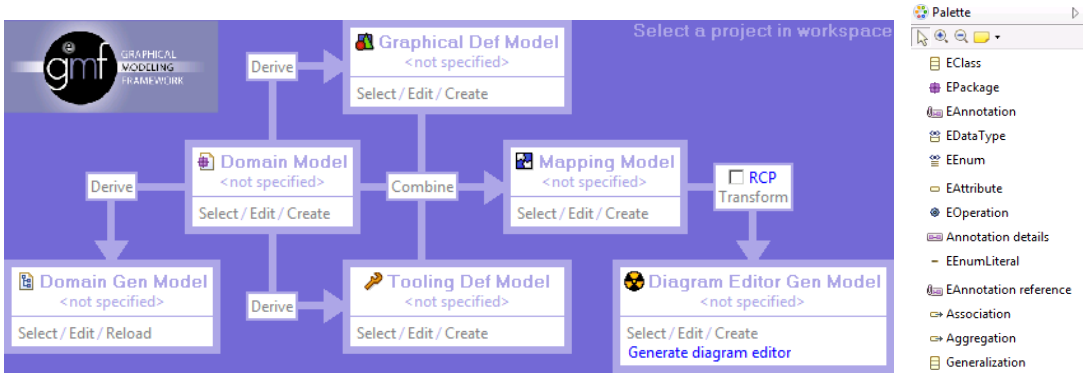
5.1 Eclipse Graphical Modeling Framework

The Graphical Modeling Framework (GMF) provides a generative component and runtime infrastructure that supports the development of graphical editors for different Meta Models in Eclipse. Typical examples for such editors, targeted by the GMF Project, are UML and Business-Process Editors.

GMF consists of a collection of models. Each model describes a different aspect of the targeted graphical editor.

The GMF models are:

- **EMF Domain Model:** This is the actual model for the DSL. It is expressed in Ecore.
- **Domain Gen Model:** Generated from Domain Model, generates Java code needed for the other transformations.
- **Tooling Definition Model:** Defines the tool-palette that is used to create a model-instance in the generated editor (figure 5.1b).
- **Graphical Definition Model:** Defines the visual representation of the diagram elements. It includes shapes, decorations, graphical nodes and connections.
- **Mapping Definition Model:** Maps the tooling and graphical definitions to the corresponding model elements.
- **Diagram Gen Model:** Is generated from the Mapping Definition Model and is transformed to the actual graphical editor for the model.



(a) (b)

Figure 5.1: (a) The Eclipse GMF-Dashboard depicting the interrelationship of the GMF Models. (b) EMF Tool-Palette which can be used to specify Ecore-Models in a graphical notation.

The first step of a simplified workflow (cf. also 5.1a) for creating such an editor is to create a Domain Model, the Meta Model of the language that the editor is written for. The Domain Model contains the non-graphical information and is the result of a problem analysis. The actual semantics are defined there. The Domain Model can be expressed in different languages like UML2, XML and Ecore. In the next step, the graphical representation for diagram nodes and links in the resulting editor is defined in the Graphical Definition Model. The definition of the so called Tooling Definition Model comes next. It specifies how the Tool-Palette (figure 5.1) looks like. The Mapping Model which is defined then, combines the Domain Model, Graphical Definition Model and Tooling Model by defining which elements belong together (figure 5.5). For every element of the Domain Model, the Mapping Model defines which tool creates the element and how this particular Domain Model element looks like (which element of the Graphical Definition

Model describes appearance of this element) in the editor. After the definition of those models, the graphical editor is generated and can be enhanced by editing the generated plug-in code.

A more detailed description of the different models involved in the process follows.

5.1.1 Domain Model

The Domain Model can be expressed in different Modeling-Languages. One of them is Ecore (figure 5.2).

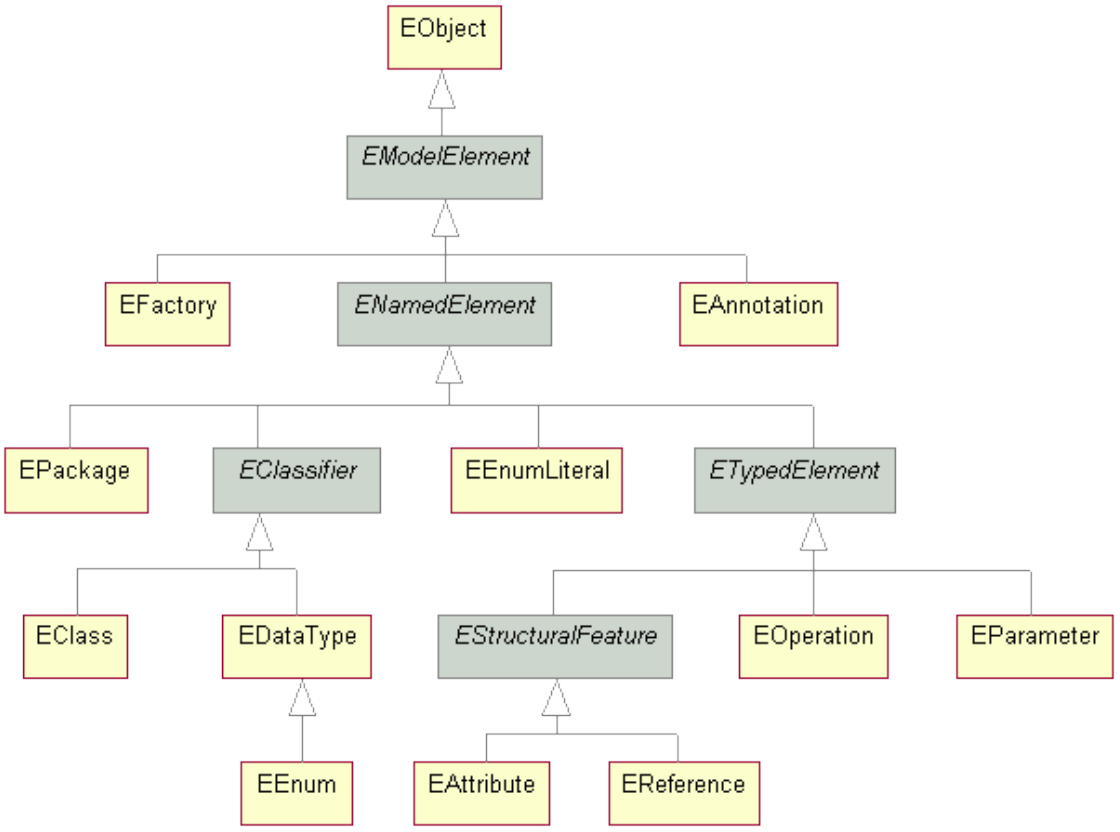
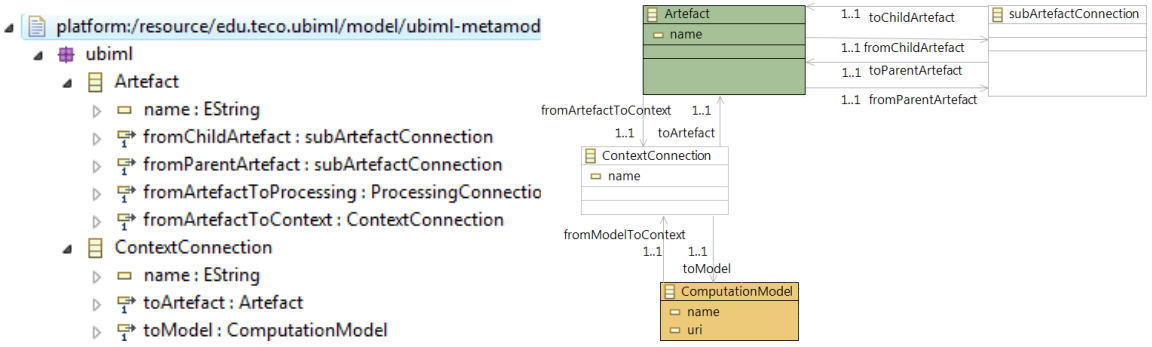


Figure 5.2: Ecore-Metamodel [Co+ot]. Domain Models are specified using these Entities. The most frequently used elements for specifying UbiML's Metamodel were EClass, EEnum, EAttribute and EReference.

This model is the actual semantic core of the Editor. Ecore can be written and modified using the Eclipse Modeling Framework (EMF). EMF can be used for code generation based on a structured data model. This structured data model is expressed in Ecore which is why EMF provides an editor (figure 5.3a) for it. The Ecore Model can be serialized as XMI and instantiated as a runtime model in Java. Newer versions of GMF have the possibility to create a graphical representation of the Domain Model and edit it in an UML-like notation (figure 5.3b).

5.1.2 Graphical Definition Model

The Graphical Definition Model defines the visual representation for diagram nodes, links and labels in the editor. The model is very flexible and supports different shapes, line-styles, fonts and spatial attributes.



(a) (b)

Figure 5.3: Different representations of the same Artefact Model. (a) Eclipse Modeling Framework's "Sample Reflective Ecore Model Editor" (b) Graphical-Representation of an Ecore Model

All visible objects are defined in the "Figure Gallery". Each different graphical element consists of a "Figure Descriptor" node which contains a hierarchy of child-elements that describe the way it looks like.

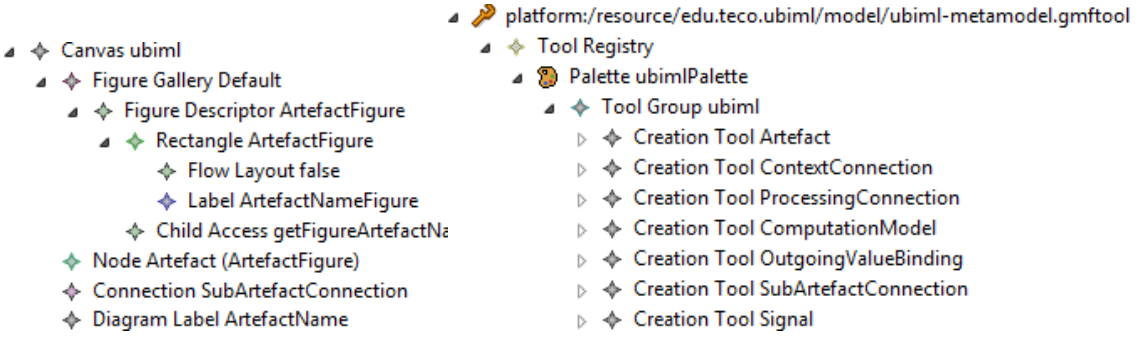
Sibling nodes of the Figure Gallery represent different node, connection and label types and reference elements in the Figure Gallery. Elements in the Figure Gallery are referenced through "Child Access" elements. This makes figures accessible by multiple nodes and thus reusable.

Figure 5.4a depicts how a simple Graphical Definition Model could look like.

5.1.3 Tooling Definition Model

The Tooling Definition Model defines which tools exist in the Tool-Palette (figure 5.1b) and how they are grouped. The Tool-Palette can also contain items for zooming in and out of the canvas and selecting items.

GMF provides a tree-like editor 5.4b, the GMFtool Model Editor, to make the specification easier.



(a) (b)

Figure 5.4: (a) Graphical Definition Model in the "GMFgraph Model Editor" (b) Tooling Definition Model in the "GMFtool Model Editor"

5.1.4 Mapping Model

The Mapping Model links up the Tooling Definition, Graphical Definition and Domain Model (figure 5.5).

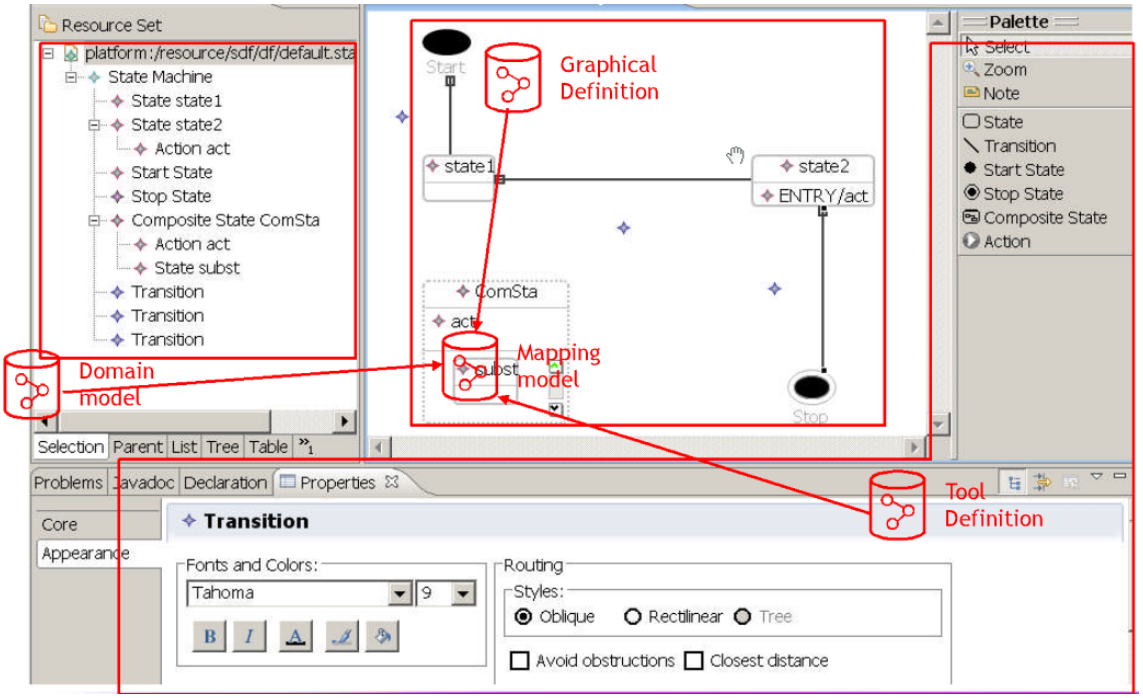


Figure 5.5: The Mapping Model links Tooling Definition, Graphical Definition and Domain Model together. [TiSh]

It can be created and edited using the “GMFmap Model Editor” which is very similar to the model Editors for the Tooling Definition, Graphical Definition and Ecore Models.

Every User Model that ought to be used as a basis for a GMF Editor needs to have a Canvas-Root-Node which represents the canvas of the editor. The nodes that have an association with that root-node can be drawn directly on the canvas.

To add one of these nodes to the editor, a “Top-Node-Reference” needs to be added to the Mapping Model with the association as the “Containment Feature” of the Top-Node-Reference. Then, a “Node Mapping” child needs to be created and the according Ecore-element specified as the “Element”. The correct Tool and Diagram Node (appearance) is chosen here as well. The contained features, like labels for instance, can be added by creating children to the “Node Mapping”, such as the “Feature Label Mapping”.

Links (connections between nodes) can be added to the editor by adding a “Link Mapping” node to the Mapping Model. When links are represented as classes in the User Model, their source and target references can be chosen by the “Source Feature” and “Target Feature” fields of the “Link Mapping”. The containment and element, as well as the tool and diagram link can be chosen here as well.

It is also possible to nest diagram elements inside each other.

5.1.5 openArchitectureWare

openArchitectureWare (*oAW*) is a modular MDSD workflow engine implemented in Java. Workflows, which specify generation and transformations of models, can be expressed in a XML-based language. oAW can work on EMF Models, but also on others like UML2, XML or JavaBeans. openArchitectureWare contains several languages that can be used to do model transformations or to formulate constraints. The most important ones are:

- **Xpand:** Model-To-Text Transformations — handwritten code can be intermixed with generated code.
- **Xtend:** navigate source models to do a Model-To-Model Transformation.
- **Xtext:** Generate Text-Model-Editors using a language similar to EBNF.
- **Check:** Define constraints.

The implementation of a Transformation from UbiML to GloMoSim which has been written in the context of this thesis makes extensive use of Xpand and Check. Some source-code excerpts are shown in the section 6.3.

5.2 Implementation of the Artefact Meta Model

The Meta Model for UbiML's Artefact Model is depicted in figure 5.6. The Canvas-Root-Node and its connections are hidden for the sake of clarity. Connections from the Canvas-Root-Node to all other classes exist in the User Model.

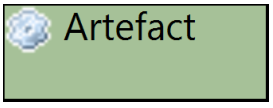
All classes that represent links in the editor, have been modeled as EClasses and are colored white with gray lines. Types, Type collections and Enumerations are also colored white, but with black lines.

The Incoming-, Outgoing-, Local-Values (dark blue) all inherit from the Value class. This makes the specification of Value-Types and Assertions easier. The Periodic-, and Singleshot-Timers do also have a common super-class, to prevent re-specification of their common arguments.

Not all of the diagram elements have been implemented in the editor, as not all parts are needed for basic functionality. The following figures depict the graphical representation of UbiML's elements that have been implemented. A reference to the according section in chapter 4.2 where it is explained is also given for each of the depicted elements.

A full example of how the graphical notation of UbiML looks like can be found in section 6.2.

5.2.1 Artefacts



Single Artefact with the name "Artefact".

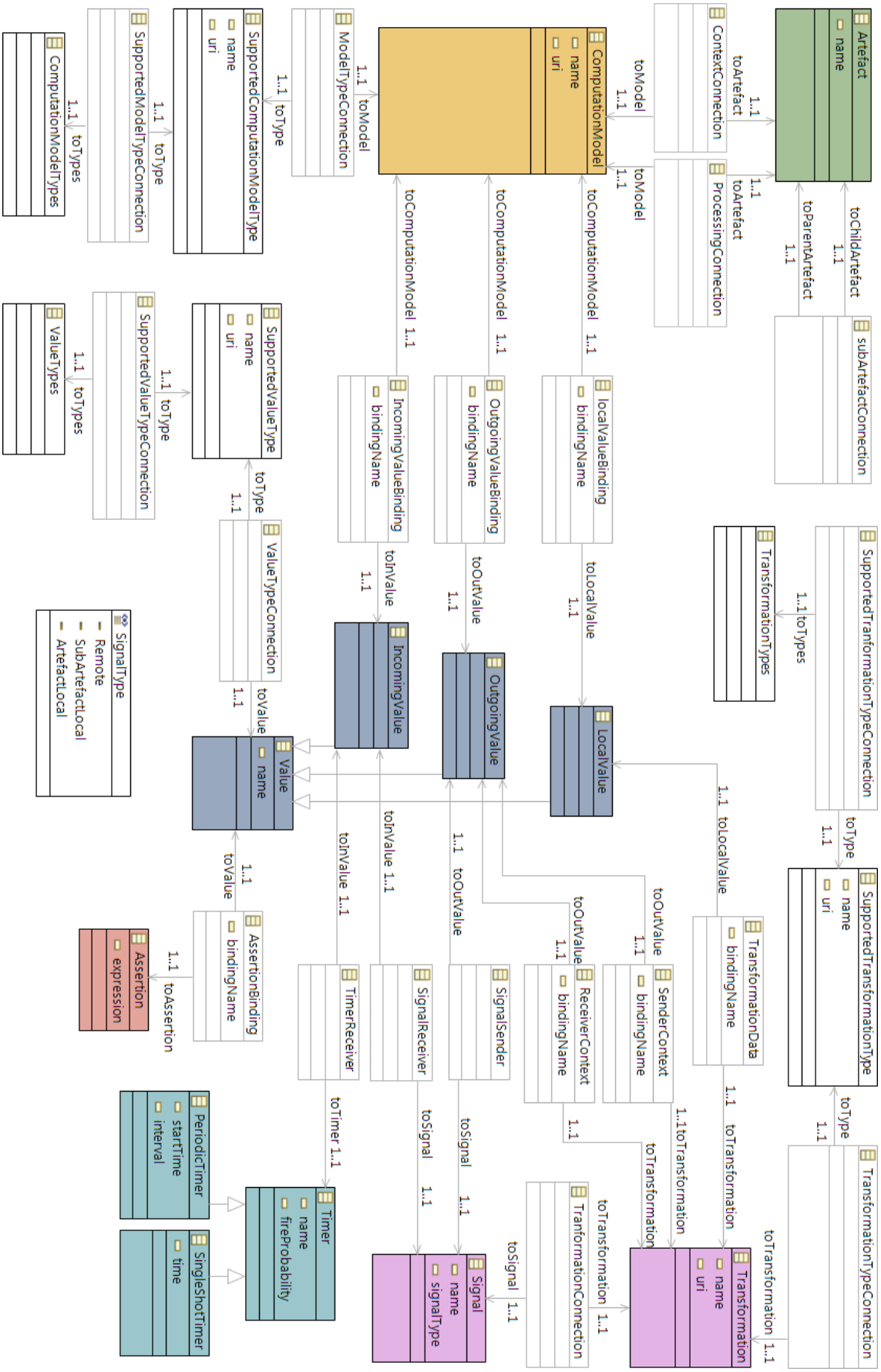
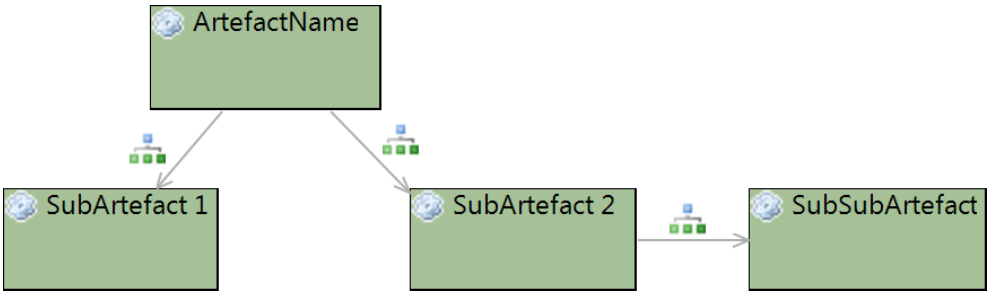


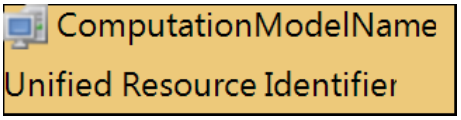
Figure 5.6: Artefact Metamodel (without canvas-root-node and its connections)

SubArtefact-Connections

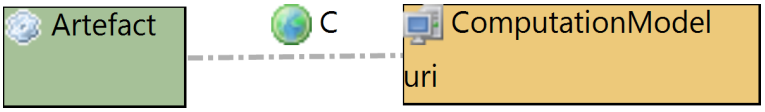


Artefacts and Sub-Artefacts connected to each other by Sub-Artefact-Connections. Semantic description can be found in section 4.2.1.

5.2.2 ComputationModels

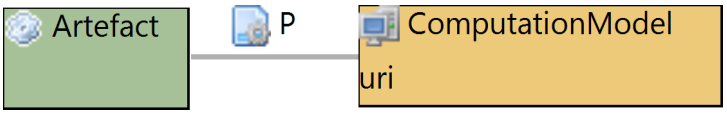


ContextConnection



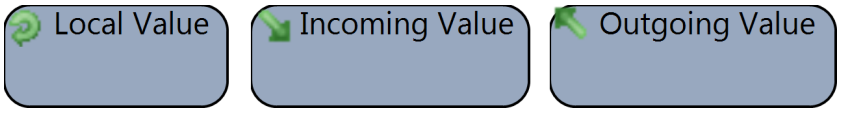
Artefact connected to a ComputationModel with a ContextConnection.

ProcessingConnection



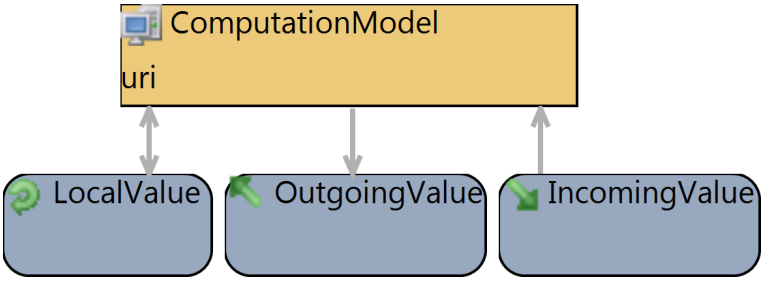
Artefact with a ComputationModel attached to it through a ProcessingConnection. Semantic description in section 4.2.2.

5.2.3 Values



The three Values: Local Value, Incoming Value and Outgoing Value.

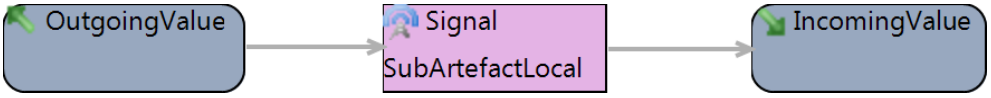
Local Value Bindings, Outgoing Value Bindings, Incoming Value Bindings



The connections used for binding Values to a Computation Model.

Semantic description in section 4.2.3.

5.2.4 Signals



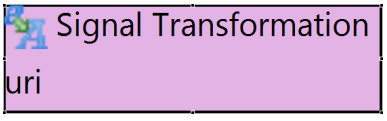
OutgoingValue and IncomingValue connected by a Signal. The Signal is of the SignalType “SubArtefactLocal” which means that the Signal is only delivered to this Sub-Artefact in the own instance. The OutgoingValue and IncomingValue must have the same root-Artefact. No Transformation is attached.

Signal Sender and Signal Receiver Connections

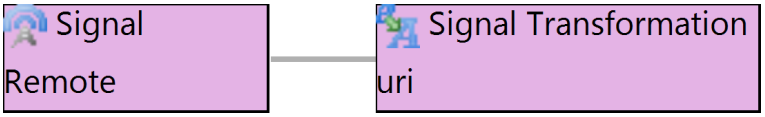
An OutgoingValue connected to a Signal with a SignalSenderConnection. The IncomingValue is connected to the Signal with a SignalReceiverConnection.

Semantic description in section 4.2.4.

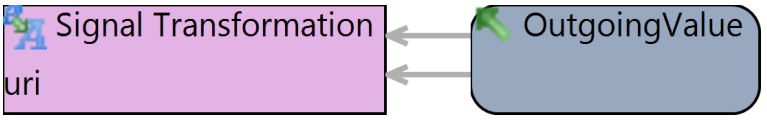
5.2.5 Signal Transformations



Transformation Connections

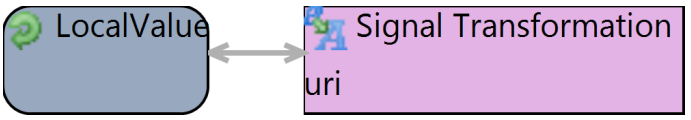


Sender- and Receiver-Context Connections



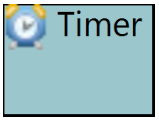
Transformation accessing the OutgoingValues of sender and receiver.

Transformation Data Connections

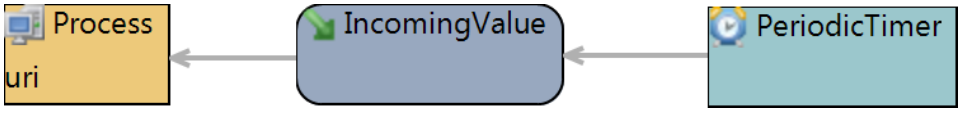


Semantic description in 4.2.5.

5.2.6 Timers



Timer Receiver Connections



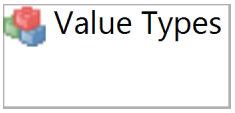
A Process which is triggered by a PeriodicTimer. The Timer is connected to the IncomingValue through a TimerReceiverConnection.

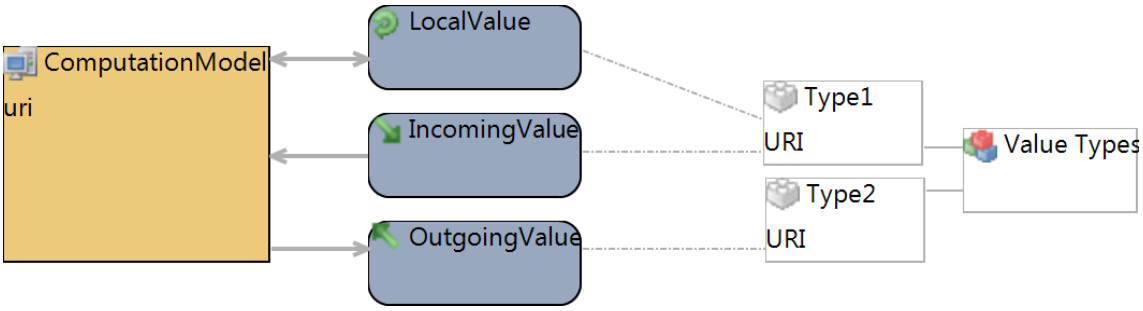
Semantic description in section 4.2.6.

5.2.7 Types

Supported Value Types

Value Type Collection





A ComputationModel with three different Values attached to it. A LocalValue, IncomingValue and OutgoingValue. The former two are of type “Type1”, while the latter is of “Type2”.

Semantic description in section 4.2.8.

5.3 Implementation of the Instantiation Meta Model

The Instantiation Meta Model is located in a sub-package of the Artefact Meta Model. It does not have an association to the canvas-root-node and is therefore not visible in the graphical artefact editor.

Instead of instantiating the Artefacts in graphical notation, leading to a cluttered model, the instantiation is done in a tree-like editor. The Instantiation Model is reflecting the Artefact Model that lives in the same file. It is therefore possible for the editor to check the validity of the instantiation and aid in the instantiation process by finding unbound values.

5.4 Summary of this chapter

This chapter first gave a detailed description of the Eclipse Graphical Modeling Framework and its models. Then a simple proof-of-concept implementation of UbiML’s Artefact Model and Instantiation Model was depicted.

6. Evaluation

This chapter provides a qualitative evaluation of the Domain Specific Language UbiML. The first section explains the evaluation methodology that is used to evaluate the graphical language. It is followed by three related examples to show how a design space exploration could look like. Moreover a transformation of the presented examples to the Global Mobile Information Systems Simulation Library (GloMoSim) is presented and analyzed.

In section 6.4 a more elaborate description of the real Ubiquitous System scenario "Hazardous Goods" is given and it is sketched how the scenario could be modeled in UbiML.

The chapter concludes with a summary of the evaluation results.

6.1 Evaluation Methodology

Unfortunately, a quantitative evaluation of this work is not possible, since no comparable framework exists that UbiML can be benchmarked against. It could only have been benchmarked against hand-writing code. Furthermore, only a single Model-To-Model Transformation, a transformation from UbiML to GloMoSim, exists. Due to those restrictions, different qualitative properties of UbiML are evaluated instead.

The main hypothesis being evaluated is that UbiML is expressive enough to model the vital parts of Ubiquitous Systems easily and unambiguously. This is done by creating and analyzing User Models for two typical types of Ubiquitous Systems and describing how commonly used parts can be expressed in UbiML. It is shown that the typical features of Ubiquitous Systems can be modeled clearly and precisely. A transformation to the network-simulator GloMoSim is also depicted to show the applicability of transformations for a rapid and reliable design space exploration.

6.2 UbiML User Model: Monitoring the Position of Autonomous Robots

The goal of the first User Model example is to describe an ad-hoc network of autonomous robots that need to monitor each others work to increase the reliability that they perform a given task. This scenario has been thoroughly described in [Mill08]. Simulation results for various communication protocols that can be used for monitoring autonomous robots can also be found there.

A simple example could be a set of autonomous vacuum-cleaning robots with the task of collaboratively cleaning a big house or construction robots deployed on the moon's surface to autonomously build a moon-station.

A developer aims to explore different communication-paradigms and optimizations in respect to scalability, latency, accuracy and robustness. Those parameters are not completely orthogonal and their interdependencies are not obvious. The developer decides that the network simulator GloMoSim will answer most of his questions concerning the performance of the used protocol.

Instead of developing a simulation from hand, he could design a simple UbiML Model. In the following I describe three possible iteration steps of a design space exploration for the vacuum robot scenario.

6.2.1 Periodic Single-Hop Broadcasting

A graphical representation of a possible UbiML User Model to reflect the design of an autonomous vacuum cleaning robot is depicted in figure 6.1. Every robot periodically broadcasts his current position to the other robots. When receiving a position-message, the robot saves the received position in a log for later use in the algorithm that determines where it needs to vacuum next. A GPS position reading is always imprecise. The variation between the real position and the robots GPS reading is introduced by the "GPS-Inaccuracy" Signal-Transformation. In our simple model this inaccuracy is modeled through a uniformly distributed random deviation within a certain interval. The maximum deviation is given through the Local Value "inaccuracy" which is set in the Instantiation Model.

The UbiML Model can be instantiated, as shown in figure 6.2. The Computation Model and Signal Transformation stubs (i.e. movement.pc, gps_transformation.pc) can then be set by creating text files with the given name. They contain the actual protocol logic, while the graphical notation only represents the data flow. Listings 6.1 through 6.6 illustrate how little code a developer has to write for basic functionality. The generated portion of source-code — which would have had to be written by hand if UbiML were not used — would have been more than 750 lines of code.

```
pos.x = nodePtr->mobilityData.next.x;
pos.y = nodePtr->mobilityData.next.y;
```

Listing 6.1: *movement.pc* — We don't implement our own mobility model, but use GloMoSim's Random-Waypoint mobility model and access the simulators current position stored in mobilityData. Notice that pos is the binding-name defined in the graphical UbiML notation.

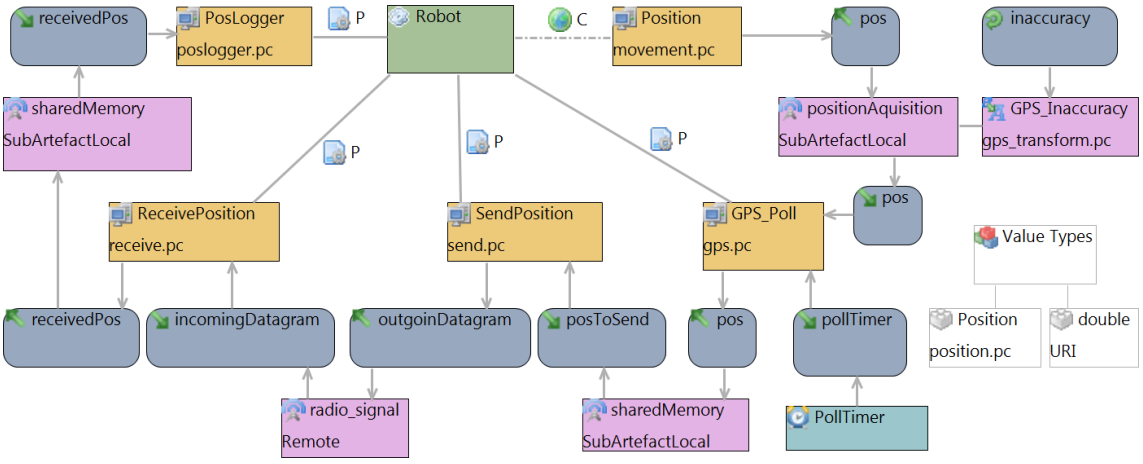


Figure 6.1: User Model for a scenario in which mobile, autonomous robots broadcast their position periodically and log the positions received by other robots. The broadcasts are triggered by the PollTimer. The data type of the used Values (dark blue) are set by associations with the nodes connected to the "Value Types"-node. Those connections are hidden for reasons of clarity.

```
outgoingPos = posFromGPS;
```

Listing 6.2: *gps.pc* — outgoingPos and posFromGPS are the binding names from the graphical UbiML User Model. outgoingPos has been declared before our stub and will be passed to the SendPosition Computation Model after the stub without further ado. posFromGPS has been declared with the correct type prior to the stub and been initialized to the position which was processed by the GPS-Transformation to model the inaccuracy caused by using a GPS.

```
double deltax, deltay;
deltax = 2.0*inaccuracy*pc_erand(senderNodePtr->seed);
deltay = 2.0*inaccuracy*pc_erand(senderNodePtr->seed);
in.x = in.x - inaccuracy + deltax;
in.y = in.y - inaccuracy + deltay;
return in;
```

Listing 6.3: *gps_transform.pc* — The Value "in" is passed to the Signal Transformation function and needs to be transformed and returned. The transformation which is applied here is to change the value by a uniformly distributed random amount between 0 and the local value "inaccuracy". pc_erand is a function defined by GloMosim and returns the required uniformly distributed random double value between 0.0 and 1.0.

```
Position *mypos = (Position*) pc_malloc( sizeof(Position) );
*mypos = posToSend;
```

```
outgoingDatagram.data = (void*)mypos;
```

Listing 6.4: *send.pc* — posToSend and outgoingDatagram are the named bindings for our Incoming Value and Outgoing Values. All that needs to be done is allocating memory for the position and adjusting the data pointer in the datagram to point to the allocated memory. The GloMoSim function to send the datagram is called after the stub.

```
receivedPos = *((Position*)(data->data));
```

Listing 6.5: *receive.pc* — We only need to copy the datum from the datagram to receivedPos. Notice that we do not need to free the memory allocated in send.pc since this is done by GloMoSim.

```
printf("[%d] logged Position(%lf, %lf)\n",
    nodePtr->nodeAddr, data.x, data.y );
```

Listing 6.6: *poslogger.pc* — All received positions are written to the console and can be processed after the simulation i.e. if the output has been redirected into a file during the simulation.

The instantiated model can then be transformed into a GloMoSim simulation and run to determine the parameters the developer is interested in. Section 6.3 gives a detailed description of how this Model-To-Model Transformation from UbiML to GloMoSim-code can be done.

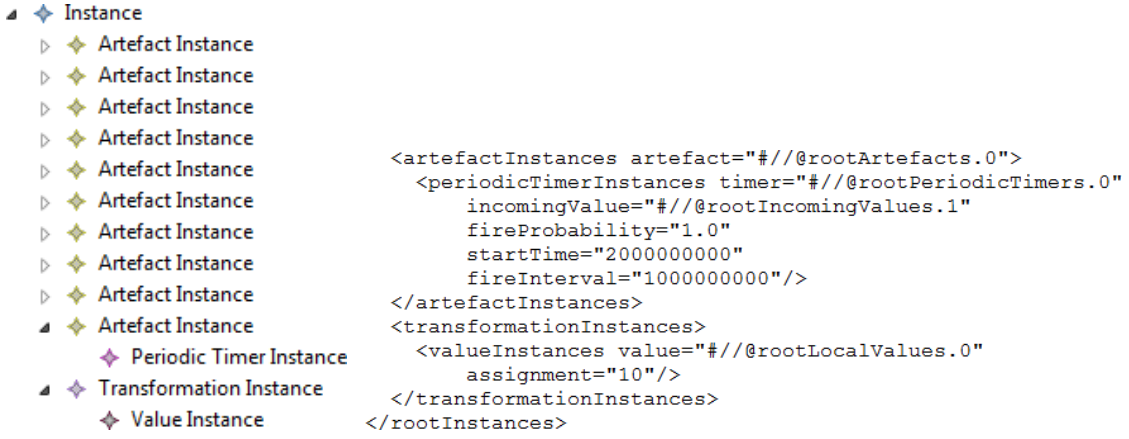


Figure 6.2: Example instantiation of the first autonomous vacuum cleaning robot User Model. (a) The instantiation part of the User Model in the Sample-Reflective-Model-Editor. Ten Instances of the Robot are declared, each with the same PeriodicTimer settings, but mutually different timers could have been set at this point. A maximum inaccuracy of 10 meters is set for the GPS-Inaccuracy Transformation. (b) A part of the same Instantiation in XMI. The timings are given in nanoseconds, thus the shown PeriodicTimer is set to start 2 seconds after the beginning of the simulation and will fire every second with a probability of 1.

6.2.2 Multi-Hop-Broadcasting

After analyzing the simulation results, the developer notices, that many position-broadcasts only reach a part of the autonomous robots and concludes, that this must be due to the single-hop broadcast in combination with far distances between the robots. He decides to change the communication-protocol by adding the capability to do multi-hop broadcasting on application-layer. Modifications need to be done in different parts of the simulation code, but using UbiML the code can be refactored easily.

To avoid that position-updates circulate infinitely in the network, a message identification number (ID) is generated for every update. If a robot receives an update with a new ID, it will forward the update to the PosLogger and then broadcast it to all nodes in its range. If the ID has been received before, the packet is ignored. To indicate that the broadcasted position is not the own but the position of another robot, the source robot's address is also included in the message.

Figure 6.3 depicts the refactored UbiML Model in graphical notation.

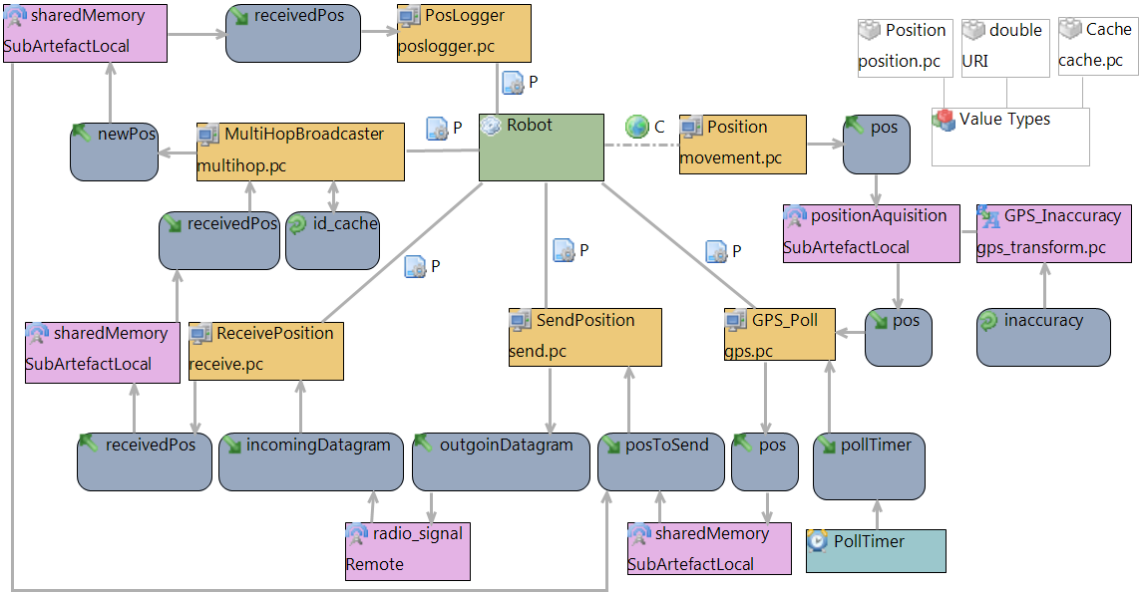


Figure 6.3: User Model for the adapted autonomous vacuum-cleaning robots example with the goal to utilize multi-hop broadcasting. Received positions are now run through the new Computation Model "MultiHopBroadcaster" where the decision is made if the position is new, in which case it is passed on to the PosLogger and repeated for other nodes which are out of reach for the original sender. Duplicates are dropped silently in the MultiHopBroadcaster. A new data type "Cache" has been added to hold a fixed number of recently received IDs. Received IDs are hashed into the cache "id_type", owned by the "MultiHopBroadcaster" — duplicates are detected through this cache.

After changing the graphical UbiML User Model, the stubs need to be changed to reflect the new semantics of the model. A very unsophisticated version of the MultiHopBroadcaster could be written as shown in listing 6.7.

```
if( (cache[pos.id%1024].id == pos.id) &&
```

```
    (cache[pos.id%1024].addr == pos.addr) )
    return; // ID is in cache, nothing else to do

cache[pos.id%1024] = pos; // put ID in cache
newPos = pos;
```

Listing 6.7: *multihop.pc* — Notice that the Position data type was changed to carry an ID and source-address. The code was omitted for brevity. Calculating a modulus of the ID as a hash function may not be very collision resistant, but should suffice for this demonstration.

6.2.3 Holding Back Minor Updates

Using application-layer multi-hop-broadcasting, the number of packets generated by the protocol increases dramatically. The developer notices this and needs to find a way to mitigate this effect. He decides to change the protocol again. The protocol shall only broadcast a position-update if the robots position has changed more than a certain threshold to the position broadcasted before. Again, due to the use of UbiML the changes are trivial and easily done.

The GPS_Poll Computation Model is changed to implement the intended behavior. A new LocalValue “lastPosSent” is added to hold the last position that the robot has broadcasted. Another LocalValue (minDeviation) determines the minimum Deviation from the last broadcasted position that is needed for the protocol to send an update. minDeviation is set in the Instantiation Model. The refactored UbiML User Model is depicted in figure 6.4. The gps.pc stub is changed to the code shown in listing 6.8.

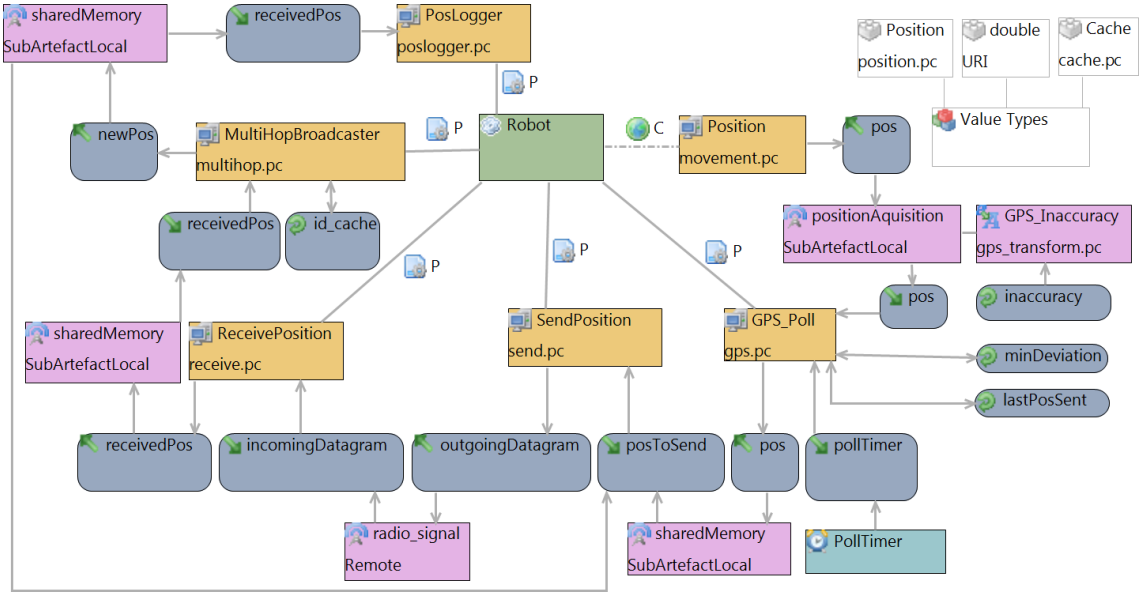


Figure 6.4: Another iteration of the vacuum-cleaning robots UbiML Model. Notice the two additional LocalValues attached to GPS_Poll. Sent positions are saved in “lastPosSent” and updated positions only broadcasted if the distance to the current position is greater than “minDeviation”.

```
if( (sqr(lastPosSent.x - posFromGPS.x) +
     sqr(lastPosSent.y - posFromGPS.y))
       <  sqr(minDeviation) )
    return; // nothing to do, only minor change in location

// store and broadcast new position
lastPosSent = posFromGPS;
outgoingPos = posFromGPS;
```

Listing 6.8: *gps.pc* — An adapted version of the stub to reduce redundant position updates.

6.3 Code-Generation: A Transformation to GloMoSim

A prototypical transformation to GloMoSim has been implemented for this thesis and has been successfully used to create compiling and fully working GloMoSim simulations from the examples introduced in section 6.2. The next section introduces some implementation specific details of the GloMoSim network-simulator — a high-level introduction has been given in section 3.5. Then, in the following section the most important XPand-constructs used to write the transformation are explained. A description how UbiML User Models can be transformed to GloMoSim, including some XPand source-code excerpts follows.

6.3.1 GloMoSim

To create a simple GloMoSim application which can be run on a node in GloMoSim one has to create at least three functions. One function which is called on creating the application on the node, used for initializing data structures, called ${appname}Init where ${appname} denotes the name of the application. The second function has the purpose to clean up dynamic data structures and write log-files — it is called ${appname}Finalize. The third function which is needed is the handler function for incoming messages from the Transport-Layer and for Timers, it is called ${appname}.

```
void ${appname}Init( GlomoNode* );
void ${appname}Finalize( GlomoNode* );
void ${appname}( GlomoNode*,Message* );
```

Listing 6.9: Manditory functions when creating a GloMoSim application. ${appname} denotes the name of the application. Data structures needed by the application can be initialized in ${appname}Init. They can be cleaned up in ${appname}Finalize. ${appname} is called on a new message (from transport or timer).

6.3.2 openArchitectureWare XPand

openArchitectureWare's XPand transformation language has already been mentioned in section 5.1.5. This section gives a short overview over the most important XPand statements. All XPand statements are enclosed in French quotation marks — the so called guillemets.

File

All text generated between the file statements will be written into a file with the given filename.

```
<<FILE ``filename''>>
    Hello World!
<<ENDFILE>>
```

Listing 6.10: Create a new file with the name “filename” and write the text “Hello World!” into it.

Define

Define statements can be called with Expand and define what code shall be generated for a certain Domain-Element.

```
<<DEFINE definitionName FOR Element>>
    ElementName = <<Element.name>>
<<ENDDEFINE>>
```

Listing 6.11: Element has an attribute called name, it will be printed after the text “ElementName = “.

Expand

Definitions can be called with Expand. Targets can be single elements or for instance all elements of a certain type. Define statements can contain multiple Expand statements.

```
<<EXPAND definitionName FOREACH Element>>
```

Listing 6.12: Expand code for an element of the model.

Foreach

Foreach allows iteration over lists in the User Model.

```
<<FOREACH this.connection AS c>>
    <<c.name>>
<<ENDFOREACH>>
```

Listing 6.13: Print all connection-names of the current node (“this”).

If-Else

If-Else constructs allow conditional transformation.

```
<<IF this.connection>>
    empty
<<ELSE>>
    <<EXPAND print FOREACH this.connection>>
<<ENDIF>>
```

Listing 6.14: Print “empty” if there are no connections, otherwise expand each connection with the “print” definition.

Rem

REM statements enclose a comment, which is removed from the file before doing the actual transformation.

```
<<REM>>Comment<<ENDREM>>
```

Listing 6.15: A comment in XPand.

6.3.3 From UbiML to GloMoSim

We will generate the following files:

- **transformation.h:** Header file containing all Signal-Transformation definitions.
- **transformation.pc:** Parsec file containing the implementation of all Signal-Transformations used in the simulation.
- **artefact_${artefactname}.h:** Header file for the each root-Artefact (that is an Artefact without parents) which contains all Computation Model definitions.
- **artefact_${artefactname}.pc:** Parsec implementation of all Computation Models in the Artefact ${artefactname}.
- **mytypes.h:** Header file containing the common data-types.

transformation.h

First the file is created and typical header and footer are written to prevent duplicate inclusion, then the rest of the file is expanded in transformation_h:

```
<<DEFINE transformations_h FOR Map>>
    <<FILE "transformations.h">>
    #ifndef TRANSFORMATIONS_H
    #define TRANSFORMATIONS_H
    #include "mytypes.h"
    <<EXPAND transformation_h FOREACH rootTranformations>>
    #endif
    <<ENDFILE>>
<<ENDDEFINE>>
```

Listing 6.16: Create the transformation.h file. The transformation-function-headers are expanded in the transformation_h definition.

transformations_h defines one function declaration for each transformation in the User Model:

```
<<DEFINE transformation\_h FOR Transformation>>
    <<fromTransformationToSignal.toSignal.fromSignalToRecv
    .toInValue.fromValueToType.toType.first().name>>
    transformation\_<<name>>
    ( <<fromTransformationToSignal.toSignal.fromSignalToSender
```

```
    .toOutValue.fromValueToType.toType.first().name>> in,
    GlomoNode *senderNodePtr, GlomoNode *receiverNodePtr );
<<ENDDEFINE>>
```

Listing 6.17: Follow the path in the model from the transformation to Incoming Value which will receive the transformed signal to determine the correct return type. Then to the same for the Outgoing Value of the sender to determine the type of the "in" parameter — the Value which will be transformed.

transformation.pc

The previous paragraph illustrates how the transformation function-names are generated. This must also be done in this file. The implementation for each function is generated as follows:

First, create variables for the sender and receiver contexts and call the respective context-functions to retrieve the current context-values piping the return value through the according transformation.

Second, create the local values as static variables and initialize them to the value given in the Instantiation Model. Static variables are instantiated only at the first call and keep their value across calls.

When this is done, create a new block containing a #include "«uri»" statement. This is where the source-code, specified in the file with the filename given in the User Model, is included.

```
Position
transformation_GPS_Inaccuracy( Position in,
                               GlomoNode *senderNodePtr,
                               GlomoNode *receiverNodePtr )
{
        assert(senderNodePtr && receiverNodePtr);
        {
                // Local Values
                static double inaccuracy = 10;
                {
                        #include "gps_transform.pc"
                }
        }
}
```

Listing 6.18: There are no senderContext or receiverContext connections so no code is generated at this point, the local value "inaccuracy" has been bound to the name set in bindingName (which was also "inaccuracy) and initialized to the value given in the Instantiation Model (10). Then the gps_transform.pc file is sourced as defined in the transformation's uri attribute in the User Model.

artefact_${artefactname}.h

In our vacuum-cleaning robot example (section 6.2) the created file is called “artefact_robot.h”. The standard header and footer to prevent duplication are inserted first. Then a structure is created to hold the Artefact’s internal state — it contains all local values. The Value-names are prefixed by the path from the root Artefact to the Computation Model, that is a underscore separated enumeration of the element-names on the way from the root Artefact to the Local Value. Moreover, a function declaration is created for every Computation Model in the Artefact. The function-names use the same prefixing scheme as it was used for Local Values. Two examples of this prefixing-scheme can be found in listing 6.19.

artefact_${artefactname}.pc

The function-implementations are created very similarly to the transformation functions. First, the attached Values are bound to the names given in the ValueBinding and initialized by calling the according functions. Then the uri is included. Contextual Computation Models return the Outgoing Value at this point. Procedural Computation Models in turn call the next Computation Model in the dataflow with the OutgoingValue as an argument.

The ${artefactname}_init() function is generated to initialize the Local Values in the Artefact’s internal state structure. The ${artefactname} function (cf. listing 6.19) must handle all Timers and remote Signals sent to the Artefact. A switch statement with a case block for each Timer and remote Signal is created, calling the according Computation Model. The payload-data is passed to the function in case of Signals.

```
void Robot(GlomoNode *nodePtr, Message *msg)
{
  GlomoRobot *clientPtr = getRobot(nodePtr);

  switch(msg->eventType)
  {
  case MSG_APP_FromTransport: // -> Received "Remote" Message
  {
    GlomoRobotMessage *data
        = (GlomoRobotMessage*) GLOMO_MsgReturnPacket(msg);
    assert(data);

    switch (data->msgType)
    {
    case RADIO_SIGNAL:
      Artefact_Robot_processing_ReceivePosition(nodePtr, data);
      break;
    }
    break;
  }

  case MSG_APP_TimerExpired: // --> "Timer" Expired
  {
```

```
    AppTimer *timer = (AppTimer*) GLOMO_MsgReturnInfo(msg);
    assert(timer);

    switch (timer->type)
    {
    case _ROBOT_POLLTIMER:
      Artefact_Robot_processing_GPS_Poll(nodePtr, timer);
      setTimer(nodePtr, _ROBOT_POLLTIMER, 1*SECOND);
      break;
    }
    break;
  }

  GLOMO_MsgFree(nodePtr, msg);
}
```

Listing 6.19: Message handler function which was generated by a fully automatic transformation from the example UbiML User Model (figure 6.1) to GloMoSim. All Timers and Remote Signals are handled here.

mytypes.h

This file contains all common data-structures. As the current implementation has only prototypical character it only supports different SupportedValueTypes. One structure is created for each SupportedValueType.

6.4 UbiML User Model: Hazardous Goods

A high level overview of the purpose, functionality and design of the CoBIs Hazardous Goods scenario has been given in section 6.4.1. A description of the hardware, components, protocols and logic is given in the following sections. It is also demonstrated how they can be modeled using UbiML.

6.4.1 Overview

The “Collaborative Business Items” (CoBIs) project was carried out by the Telecooperation Office (TecO) from the Universität Karlsruhe (TH), SAP, Infineon, Ambient Systems, Twente and Lancaster University in collaboration with BP in 2006. Its goal was to enhance business processes through the usage of autonomous sensor nodes (figure 6.5). One usecase for the CoBIs project was dealing with the storage of hazardous chemical goods.

Management and monitoring of drums containing the chemicals and enforcing storage regulations through alarms hence avoiding dangerous situations due to improper storage and handling were main objectives in this usecase.

To achieve this, Particle *sensor nodes* [DKBZ05] (figure 6.6) were attached to each drum in the project. Storage regulations were pushed onto the nodes from a back-end system and the monitoring was done autonomously and *collaboratively* by the nodes themselves. Alarming the workers of improper storage and providing

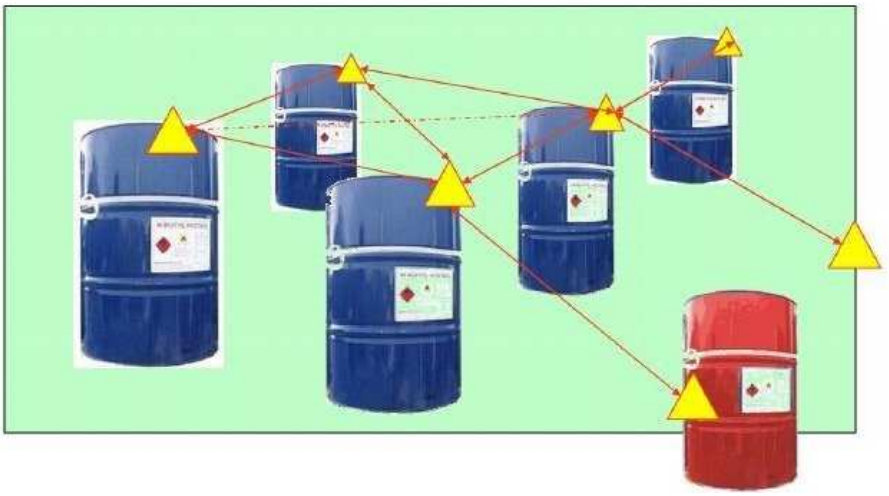


Figure 6.5: Ad-Hoc communication in the Hazardous Chemical Goods usecase [HSDR06]. Sensor nodes which have been attached to drums containing chemical goods autonomously monitor their state with the goal to enforce storage regulations.

Figure 6.6: Particle sensor node clip and drum with mounted Particle

storage information was done on-site via the drums sensors and off-site via the back-end system in management consoles.

The described system was deployed on 21 storage drums in a field test at the BP Saltend Chemical Plant in England. Three storage areas in two buildings were created for the trial. Two of the storage areas (Location A and B) were used for actual storage while Location C was used as an initial/temporal storage area. The different storage areas are depicted in figure 6.7.

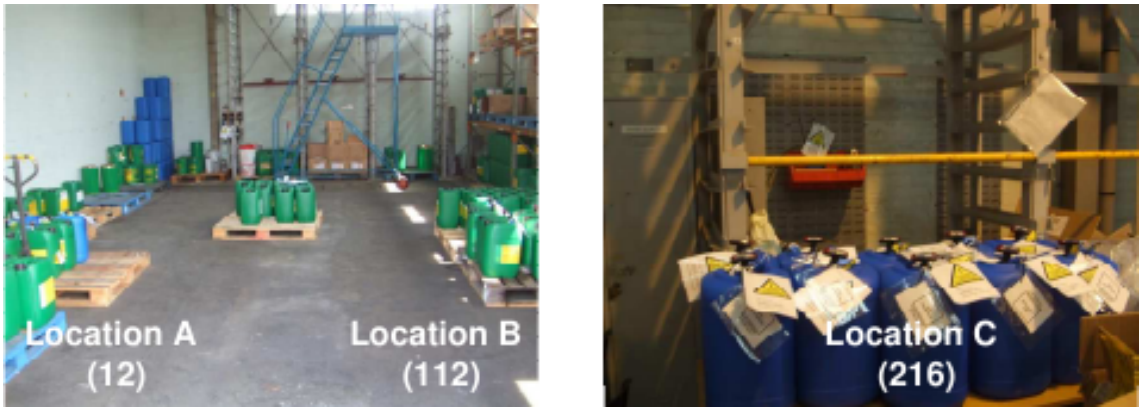


Figure 6.7: The Storage Locations which have been set up for the CoBIs Hazardous Goods field-test [HSDR06]. Location A and B were used to simulate final storage locations while Location C was used for initial or temporal storage.

The drums contained two different chemicals which cannot be stored together. Workers interacted with the system in three ways. Technical support staff managed the devices. They attached sensor nodes to the drums and replaced them when they broke. Warehouse workers interacted with the system by moving drums into and out of storage areas and obeying alarms triggered by the system. Safety Officers monitored the drums off-site and created/changed storage regulations. The interaction is depicted in a use-case diagram diagram in figure 6.8.

This introductory scenario poses a good example, because it contains many typical and challenging aspects of Ubiquitous Systems. The system contains autonomous, context-aware artefacts (the particles) which interact and collaborate in detecting illegitimate storage conditions. The interaction with different workers on- and off-site is another central part of the system.

A more detailed description of hardware, software, process, field test setup and storage regulation and analysis can be found in the following sections.

6.4.2 Hardware

The system consists of four major functional parts. The *Particle* sensor-nodes are attached to the chemical drums. They communicate with *Particle Gateways* via an 868MHz radio link using a proprietary protocol. Particle Gateways forward the data received by Particles to a LAN and emit an infrared beacon signals with location information. The *Gateway Server* provides a compatibility layer that makes it

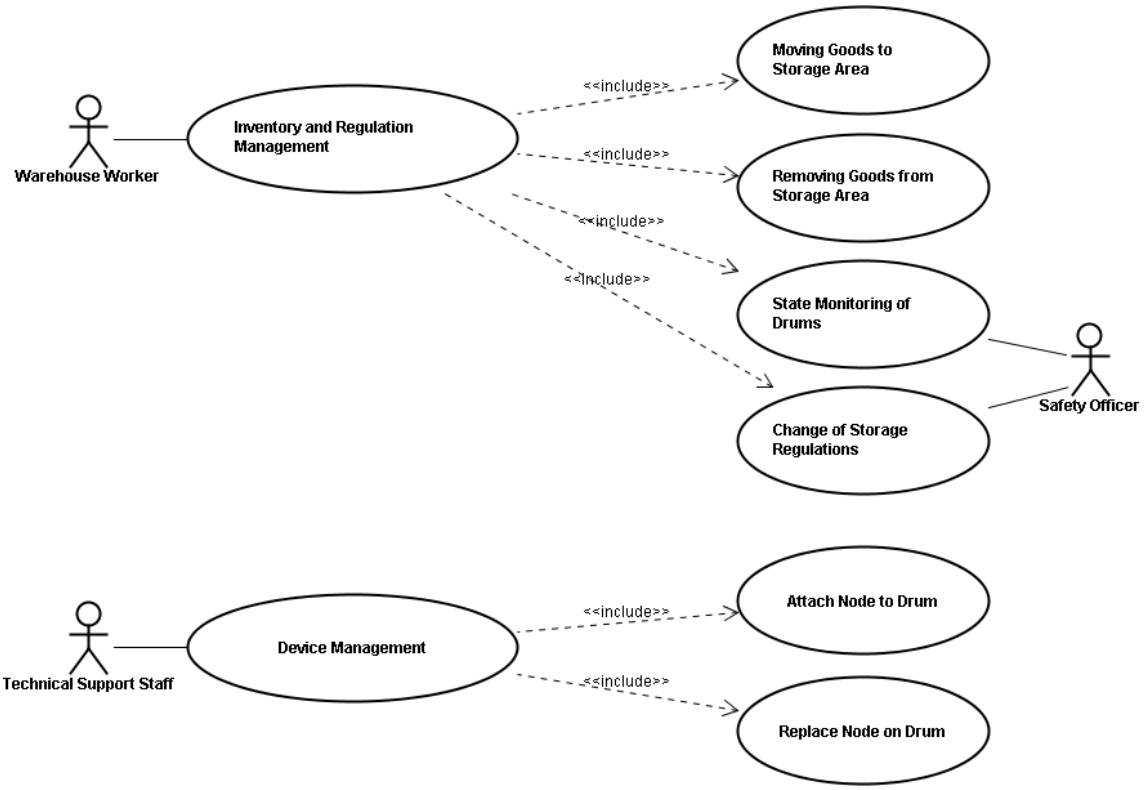


Figure 6.8: CoBIs' Hazardous Chemical Goods usecase diagram [HSDR06]

possible to access the sensor-nodes without settling on a certain node-type. Information can be pushed to the Particles from the Gateway Server and higher-level information about the Particle-network can be queried. The *Application Server* is also connected to the LAN and monitors the system. It can push new storage regulations through the Gateway Server.

Particle Sensor-Nodes

Particle sensor-nodes were packaged in portable rear lights for bicycles (figure 6.6). Additionally to the radio, they contain an infrared sensor to read the IR-Signal containing location information which is emitted by the Particle Gateway in each location. They also have a temperature sensor and a LED build in. A duty cycle of one second was configured, meaning that Particle nodes wake up every second to make sensor readings and to communicate with other Particles and the Particle Gateway.

Particle Gateways

Particle Gateways were off-the-shelf WiFi routers manufactured by Asus which were equipped with an integrated Particle node in order to make communication with the sensor-nodes possible. Furthermore, an infrared emitter was added. The routers Linux-based operating system was customized to run the required protocols.

Gateway and Application Servers

Standard PC-Hardware was used for the Gateway and Application Servers. They are connected to the network via Ethernet.

Hardware Setup

The hardware was set up as depicted in figure 6.9. A total of 21 drums were equipped with Particle sensor nodes. The contents of the drums are listed in table 6.1. Every one of the three storage locations (cf. figure 6.7) is equipped with a Particle Gateway. The Gateways at storage locations A and B are connected by a WiFi, while storage location C is connected through wired Ethernet. The Application Server and Gateway Server are running on separate machines in a fourth location.

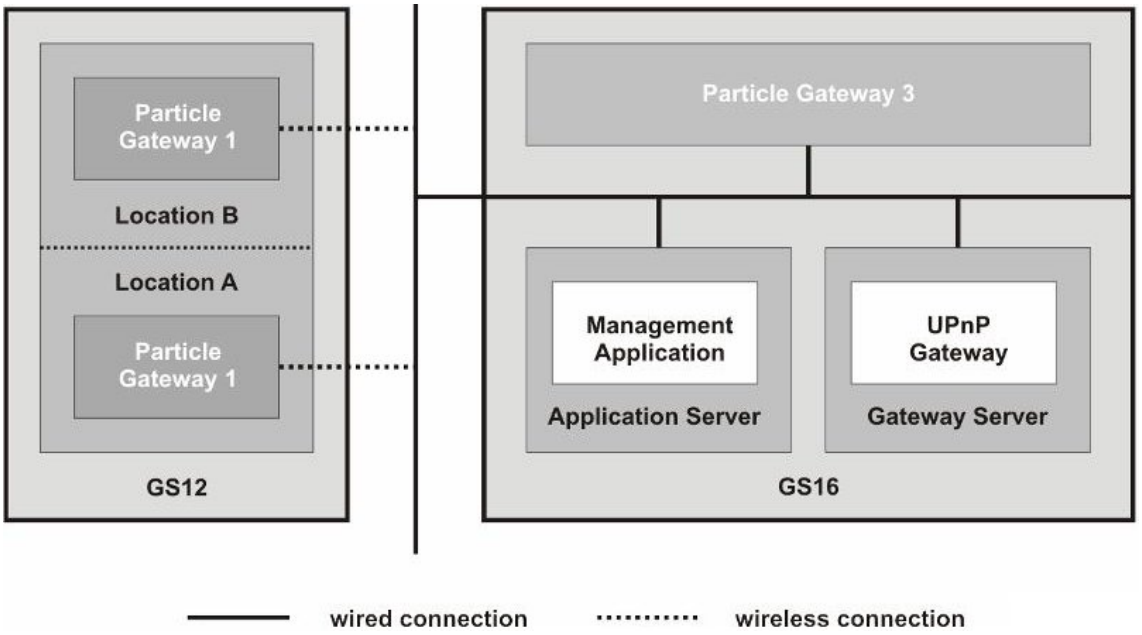


Figure 6.9: Hardware setup of one of the CoBIs Hazardous Goods trials in Saltend 2007. Particle sensor-nodes are spread over the three storage locations. Every storage location has a Particle Gateway installed relaying messages from the sensor-nodes to the backend LAN. There are 2 PCs connected to the backend LAN. One of them is running a UPnP Gateway software which represents the services offered by the Particles as UPnP devices. The other one is running a Management Application written by SAP. [DRBS+07]

6.4.3 Software

A short high-level description of the functionality that was implemented in software on the different hardware-platforms is given in the next paragraphs. The storage regulation and related alerts and notifications are also enumerated. All information is taken from the two rolling reports on the application trials which have been carried out during the project [HSDR06] and [DRBS+07].

Particle Sensor-Nodes

Particle sensor-nodes contain logic to check the imposed storage regulations collaborately and to communicate sensor-readings, alerts and notifications to the back-end.

Particle Gateways

The Particle Gateways basically acts as a network router between the WSN and the backend LAN.

Gateway Server

The Gateway Server is capable of representing the Particle Nodes services as UPnP-Devices thus making their discovery and control easy for the Management Applications on the Application Server.

Application Server

The Application Server contains an SAP management application (SAP WebAS) and uses the Gateway Server to communicate with the Particle sensor-nodes.

Storage Regulations

For the trial only very few and simple storage regulations were put into place. The regulations concerning the storage temperature and the maximum number of drums of a kind stored in the same location (limit) are listed in table 6.1. The only additional rule was that water must not be stored together with either Energear or Energol. All other combinations were allowed.

Liquid	Available	Limit	Min. Temp.	Max. Temp.	Max. Time in unknown Location
Autran	6	5	0 °C	40 °C	∞
Energear	3	3	0 °C	40 °C	600 s
Energol	2	3	0 °C	40 °C	120 s
Water	10	∞	0 °C	28 °C	∞

Table 6.1: Storage regulations for the chemical drums in the Hazardous Goods field-test. Limit denotes the maximum number of drums of that kind that is allowed to be stored together.

Alerts and Notifications

If storage regulations are violated, the system reacts with the alerts and notifications listed in table 6.2. If a particle receives an alarm, it broadcasts that alarm as well — this “alarm infection” has the goal to increase the chance that the Particle Gateway notices the alert.

6.4.4 Modeling the Hazardous Goods scenario

The following paragraphs explain how the hardware components that have been described before, could be modeled using UbiML.

Nr.	Monitoring Rule	Type	Description
1.	Storage Limit	Alert	The total amount that is allowed to be stored for a certain chemical is exceeded.
2.	Incompatible Goods	Alert	Two chemicals that are prohibited to be stored in proximity are placed too close to each other.
3.	Min. Temp. Exceeded	Alert	The temperature of the node is below the minimum temperature defined for that chemical.
4.	Max. Temp. Exceeded	Alert	The temperature of the node is above the Exceeded maximum temperature defined for that chemical.
5.	Temporary Storage Time	Alert	A drum has been located in a temporary storage Exceeded area, i.e. an area without a location beacon, for too long.
6.	Node Failure	Alert	The node is close to failing, e.g. because of low battery power.
7.	Location Change	Notification	The drum is moved, i.e. the location has changed. The ID of the new location is sent to the back end.
8.	Voltage Change	Notification	The voltage of the battery changed. The new value is transmitted.
9.	Temperature Change	Notification	The temperature changed. The new value is transmitted.

Table 6.2: Alerts and Notifications which can be sent if storage regulations are violated.

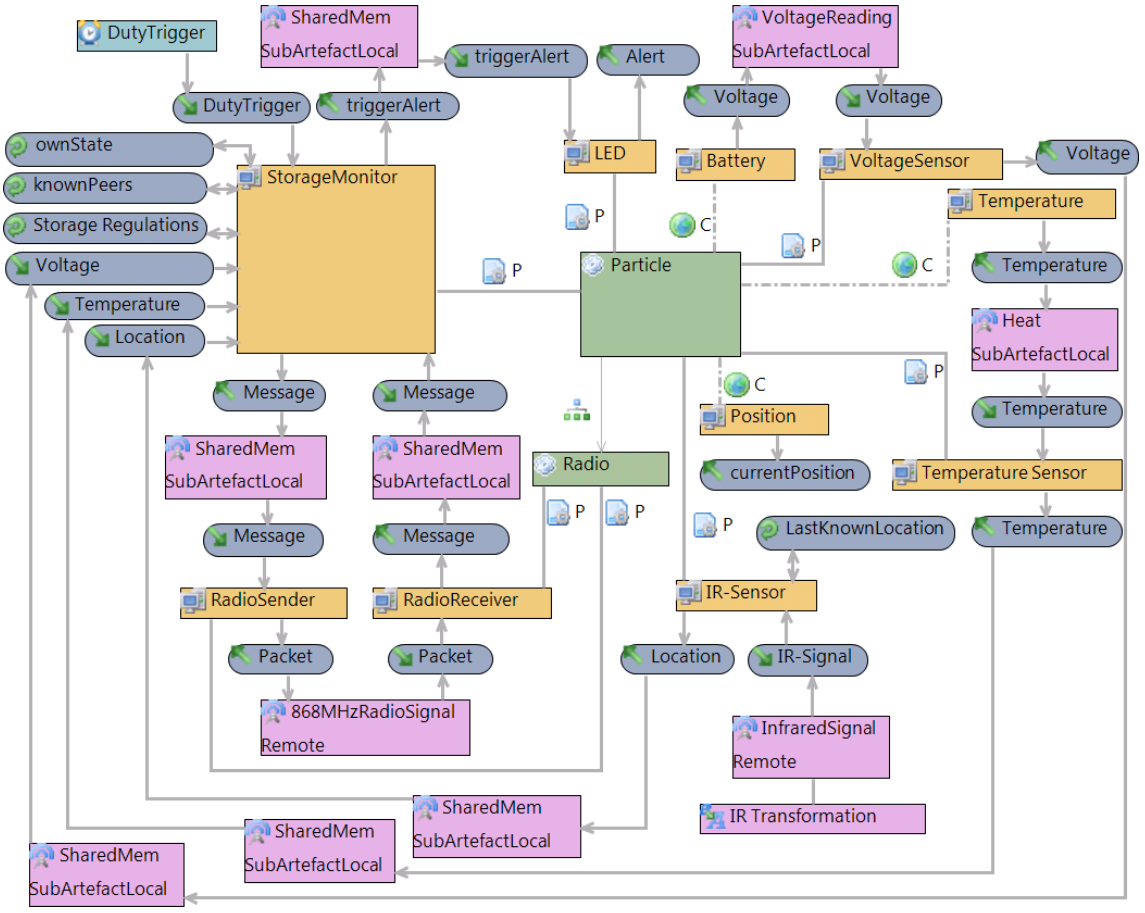


Figure 6.10: A UbiML User Model for a Particle sensor-node. The node is only active after being triggered by the DutyTrigger Timer. When active, it tries to fetch the current location from the IR-Beacon sent by the Particle Gateway. It also reads the temperature and voltage sensors and communicates with reachable peers to verify that the current storage situation complies with the regulations and with the Particle Gateway to provide informations to the Management Server.

Particle Sensor-Nodes

Particle sensor-nodes are designed to autonomously check that their storage situation complies with the storage regulations that are currently in place. They can warn the local worker that a violation has been detected by blinking the build-in LED and send the alerts and notifications listed in table 6.2. Particles can be modeled in UbiML as shown in figure 6.10.

An implementation of the Computation Models and data types is not given for this example because GloMoSim does not support the use of different node types and different signals in one simulation. The implementation would need stationary and mobile nodes as well as four different signals (Infrared, 868 MHz, WiFi and Ethernet).

The information needed to write implementations for a flexible target model exist in the User Model though. Alerts 1 and 2 can be detected through collaboration using the 868 MHz Radio. Alerts 3 and 4 can be caught using the temperature sensor. The time in an unknown location can be traced by putting the last time a beacon was received in the “ownState” Local Value. This makes detecting Alert 5 easy. Alert 6 can be checked for by reading the Voltage Sensor. All conditions in which notifications have to be sent, can be detected by storing sensor readings in the “ownState” Local Value and comparing the stored value with the current sensor readings.

Particle Gateways

A Particle Gateway has mainly two tasks. First, to inform the Particles about their location. This is done by broadcasting an infrared beacon with the according information. Second, to bridge the management LAN with the Particle network. This is done in UbiML by simply emitting the data in all received packets on the other network. The proposed User Model is depicted in figure 6.11.

Gateway and Application Servers

The Gateway and Application Servers are regular PCs with Ethernet connections to the management LAN. Since they have no sensors or similar but only a single interface to context — the network, UbiML does not help much designing those systems. It does not hurt either, though, figure 6.12 depicts a User Model for the Gateway and Application Servers.

6.5 Evaluation Results

An evaluation of the UbiML Meta Models and its tools has lead to the following results. First of all UbiML is a model as defined in definition 2.1 as all four properties that constitute a model are given. UbiML is a *mapping* of a system in the real world to an *abstract* description. It is moreover *isomorph* to the real world in some respects as the analysis platforms it is transformed to allow conclusions which hold in the real world and would not be possible otherwise (*pragmatics*).

UbiML can be very helpful for designing Ubiquitous Systems. Several properties have been named in section 2.2 which are allegedly improved by using a MDSD

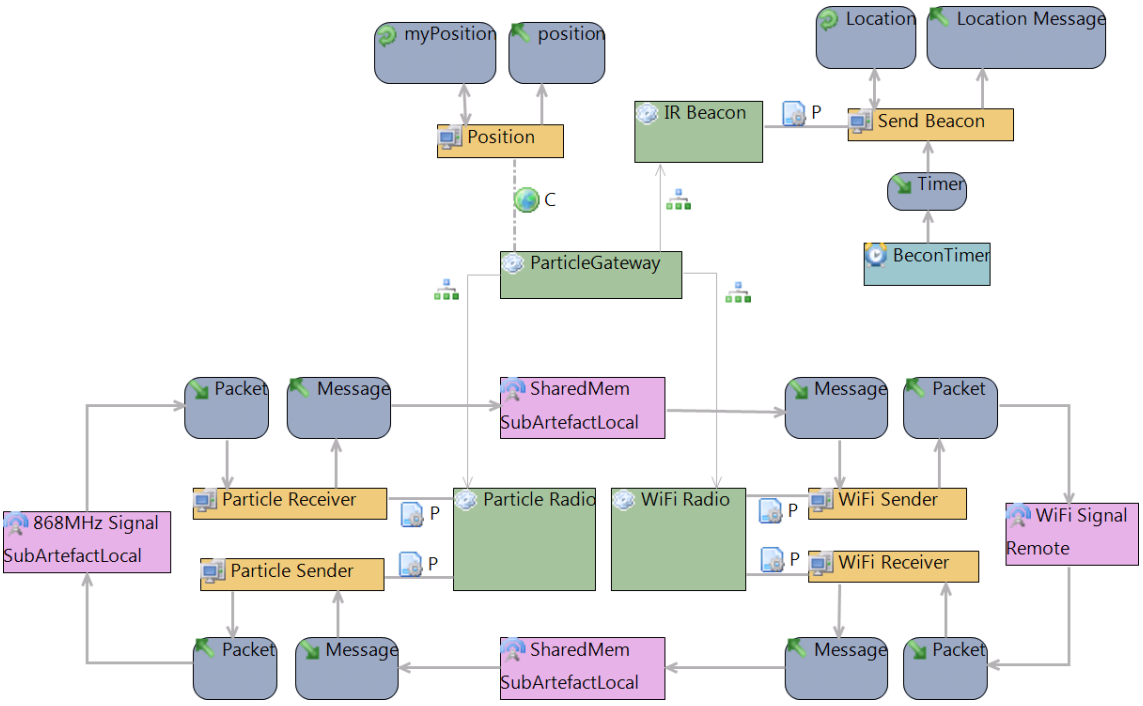


Figure 6.11: UbiML User Model for a Particle Gateway. The IR-Beacon broadcasts the programmed Location whenever triggered by the Beacon-Timer. The Position context was added because it might be needed i.e. for fading models. The Particle Radio transfers the message that was received to the WiFi sender and vice versa which bridges both networks.

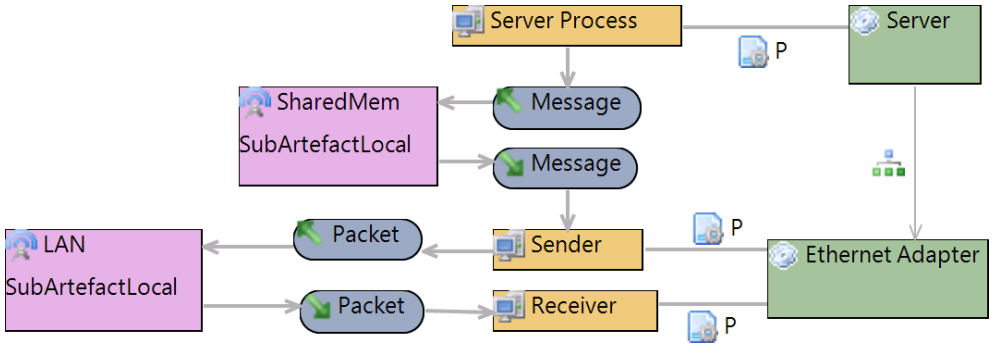


Figure 6.12: The Gateway and Application Servers are regular PCs which only interface with the other systems via their Ethernet link. The Server Process represents the actual program running on the machine. Details about Values have been omitted.

approach. The evaluation shows that most of these properties significantly improve by an UbiML enabled design. The testability, maintainability, reusability, development speed and maintainability are such properties.

Especially the first example has shown, that modeling with UbiML leads to a clear model of envisioned system while reducing the number of lines of code to a minimum. It was also shown in this example that UbiML supports the developer in refactoring his design leading to a fast and easy design space exploration. When progressing to a new iteration of the models only very few changes had to be done in UbiML and the source code stubs. Although no repository support has been implemented yet, the creation of a new User Model can be done very quickly.

CoBIs Hazardous Goods, the second example, made it clear that UbiML enables system designers to model even complex Ubiquitous Systems which interact with many different kinds of contexts (Temperature, Location, Dutycycle, Infrared etc) easily and precisely. When modeling the server-components it became evident though, that UbiML is not well suited for designing regular client-server software for PC-Hardware. This field of application was not intended for UbiML though and remains the domain for general purpose modeling tools or component based systems like PCM.

The amount of code that can be reused when switching between target models has not been evaluated. Only a single transformation has been implemented, a transformation to GloMoSim, where all Computation Model and Transformation stubs were implemented using target model code directly. Since GloMoSim applications are implemented in Parsec — which is a C-dialect — I anticipate that it will be very easy to write a transformation to C-code which can be compiled to the actual target platform.

In a nutshell one can say, that most of the code for interfaces, accessing context and communication can be generated when using UbiML to design a system. One can concentrate on the implementation of the actual functionality. In combination with other models this part could also be generated as UbiML does not forbid but encourage the use of i.e. automatons for the specification of program logic.

6.6 Summary of this Chapter

This chapter presented that UbiML is well suited for enhancing the design process of Ubiquitous Systems. First it was shown how a design space exploration for a system with autonomous robots could look like. Compiling and working simulation code was generated for the simulation platform GloMoSim. Only a few lines of code have had to be changed between the different designs. Then the code generation from UbiML to GloMosim was explained in detail. For better comprehensibility, a short introduction to GloMosim and XPand was given beforehand. Moreover a UbiML User Model for a real Ubiquitous System was depicted and it was shown that UbiML is expressive enough to describe complex systems.

7. Conclusion

This last chapter demonstrates that the objectives that have been defined in chapter 1.2 have not only been met but have been excelled.

A domain specific language, namely UbiML, has been designed in the context of this thesis as a result of analyzing many different existing modeling concepts and tools. It has a clear Meta Model, Model Driven Development workflow and has the capability to be transformed to analysis models. A strong emphasis has been set to modeling dataflows and the easy descriptions of the very dynamic nature of different types of context.

UbiML consists of two Meta Models — the Artefact Meta Model which can be used to model subsystems and the interrelationship between subsystems and the Instantiation Meta Model which can be used to describe an instance of the previously modeled parts.

Furthermore, a UbiML editor has been implemented for the thesis in Eclipse GMF and a transformation from UbiML to GloMosim has been created in openArchitectureWare's XPand transformation language.

Modeling realistic examples of Ubiquitous Systems, it has been shown, that UbiML is flexible and expressive enough while remaining concise enough to be transformed into procedural programing language source code. The designers of Ubiquitous Systems are not only supported by the generation of large portions of simulation-code, but also in the refactoring process which is needed exploring the design space of systems.

7.1 Future Work

The main question which remains unanswered is if all useful target analysis platforms are reachable by transformations from UbiML. This question will have to be answered in future research. Moreover, it is a open question if UbiML is easy to

learn, understand and use for developers that wish to design a Ubiquitous System.

The UbiML Editor has only prototypical character and has to be improved in many different ways in order to be usable for real projects. There is currently no repository implemented.

Only one transformation to a target model has been implemented for this thesis. Being similar (a C programming language dialect) to real target platform code it can be assumed that a transformation to real target platform code can be easily done. Nevertheless, many more transformations to different analysis models have to be implemented for UbiML to unfold to its full potential.

Bibliography

[000] *Diagnosis and diagnosability analysis using process algebra*, 2000.

[ADBD+99] Gregory D. Abowd, Anind K. Dey, Peter J. Brown, Nigel Davies, Mark Smith und Pete Steggles. Towards a Better Understanding of Context and Context-Awareness. In *HUC '99: Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*, London, UK, 1999. Springer-Verlag, S. 304–307.

[Aror05] A. Arora et al. ExScal: elements of an extreme scale wireless sensor network. *Embedded and Real-Time Computing Systems and Applications, 2005. Proceedings. 11th IEEE International Conference on*, Aug. 2005, S. 102–108.

[ASSDGC07] T. Antoine-Santoni, J. F. Santucci, E. De Gentili und B. Costa. Modelling & simulation oriented components of wireless sensor network using DEVS formalism. In *SpringSim '07: Proceedings of the 2007 spring simulation multiconference*, San Diego, CA, USA, 2007. Society for Computer Simulation International, S. 299–306.

[Baet05] J. C. M. Baeten. A brief history of process algebra. *Theor. Comput. Sci.*, 335(2-3), 2005, S. 131–146.

[BeKR07] Steffen Becker, Heiko Koziolek und Ralf Reussner. Model-Based performance prediction with the palladio component model. In *WOSP '07: Proceedings of the 6th international workshop on Software and performance*, New York, NY, USA, 2007. ACM, S. 54–65.

[biso] Introduction to Bison. http://www.gnu.org/software/bison/, accessed 2009/03.

[CaGo98] Luca Cardelli und Andrew D. Gordon. Mobile Ambients. In *FoSSaCS '98: Proceedings of the First International Conference on Foundations of Software Science and Computation Structure*, London, UK, 1998. Springer-Verlag, S. 140–155.

[ChLZ06] Elaine Cheong, Edward A. Lee und Yang Zhao. Viptos: A Graphical Development and Simulation Environment for TinyOS-based Wireless Sensor Networks. Technischer Bericht UCB/EECS-2006-15, EECS Department, University of California, Berkeley, Feb 2006.

[Co+ot] IBM Corporation und andere. API for the Ecore dialect of UML. http://download.eclipse.org/modeling/emf/emf/javadoc/2.5.0/org/eclipse/emf/ecore/package-summary.html, accessed 2009/04.

[CoZe88] A.I. Concepcion und B.P. Zeigler. DEVS formalism: a framework for hierarchical model development. *Software Engineering, IEEE Transactions on*, 14(2), Feb 1988, S. 228–241.

[Di C03] G. A. Di Caro. Analysis of simulation environments for mobile ad hoc networks. Technischer Bericht, Dalle Molle Institute for Artificial Intelligence, 2003.

[DKBZ05] Christian Decker, Albert Krohn, Michael Beigl und Tobias Zimmer. The particle computer system. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*, Piscataway, NJ, USA, 2005. IEEE Press, S. 62.

[DRBS+07] C. Decker, T. Riedel, M. Beigl, L.M. Sa de Souza, P. Spiess, J. Muller und S. Haller. Collaborative business items. *Intelligent Environments, 2007. IE 07. 3rd IET International Conference on*, Sept. 2007, S. 40–47.

[eal99] V. Lesser et al. The intelligent home testbed. In *In Proceedings of the AutonomyControl Software Workshop*, 1999.

[Emer90] E. Allen Emerson. Temporal and modal logic. 1990, S. 995–1072.

[Feng04] Leonidas Guibas Feng Zhao. *Wireless Sensor Networks - An Information Processing Approach*. Elsevier. 2004.

[FiSa01] Anthony Finkelstein und Andrea Savigni. A Framework for Requirements Engineering for Context-Aware Services. In *In Proc. of 1 st International Workshop From Software Requirements to Architectures (STRAW 01*, 2001, S. 200–1.

[FrMS08] S. Friedenthal, A. Moore und R. Steiner. OMG Systems Modeling Language (OMG SysML) Tutorial. http://www.omgsysml.org/INCOSE-2008-OMGSysML-Tutorial-Final-revb.pdf, last accessed 2009/03, 2008.

[GLBW+03] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer und David Culler. The nesC language: A holistic approach to networked embedded systems. *SIGPLAN Not.*, 38(5), 2003, S. 1–11.

[Grou] Object Management Group. Unified Modeling Language (UML) Specification: Infrastructure version 2.0. http://www.omg.org/docs/ptc/03-09-15.pdf, accessed 2009/03.

[HiCu02] Jason L. Hill und David E. Culler. Mica: A Wireless Platform for Deeply Embedded Networks. *IEEE Micro*, 22(6), 2002, S. 12–24.

[Hill05] Jane Hillston. Tuning Systems: From Composition to Performance. *Comput. J.*, 48(4), 2005, S. 385–400.

[HSDR06] S. Haller, L.M.S.d.S Souza, C. Decker und T. Riedel. Collaborative Business Items Rolling Report on Studies and Trials. Technischer Bericht, SAP, TecO and Lancaster University, 2006.

[Huda98] P. Hudak. Modular domain specific languages and tools. Jun 1998, S. 134–142.

[Hwan07] Moon Ho Hwang. FD-DEVS PingPong example. http://en.wikipedia.org/wiki/File:FD-DEVS_PINGPONG.JPG, accessed 2009/05, 2007.

[Inte94] International Organization for Standardization. *Open Systems Interconnection Basic Reference Model*, 1994. ISO/IEC 7498-1:1994.

[KMEW06] L. Kolos-Mazuryk, P. A. T. van Eck und R. J. Wieringa. A Survey of Requirements Engineering Methods for Pervasive Services. Freeband A-MUSE deliverable D5.7a TI/RS/2006/018, Enschede, March 2006.

[KMPE05] L. Kolos-Mazuryk, G. J. Poulisse und P. A. T. van Eck. Requirements Engineering for Pervasive Services. In *Second Workshop on Building Software for Pervasive Computing. Position Papers., San Diego, California, USA*. No publisher, October 2005, S. 18–22.

[Kobb08] S. Kobbe. Konzeption und Evaluierung eines energieeffizienten Sensornetzwerks anhand realer Sensorknoten. Diplomarbeit an der Universität Karlsruhe (TH), 2008.

[KSWW+08] A. Köpke, M. Swigulski, K. Wessel, D. Willkomm, P. T. Klein Haneveld, T. E. V. Parker, O. W. Visser, H. S. Lichte und S. Valentin. Simulating wireless and mobile networks in OMNeT++ the MiXiM vision. In *Simutools '08: Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), S. 1–8.

[Kühn05] Thomas Kühne. What is a Model? In Jean Bezivin und Reiko Heckel (Hrsg.), *Language Engineering for Model-Driven Software Development*, Nr. 04101 der Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2005. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[KWUB06] Mirko Knoll, Torben Weis, Andreas Ulbrich und Alexander Brändle. Scripting Your Home. In *LoCA*, 2006, S. 274–288.

[Lee] Edward Lee. Ptolemy II. http://ptolemy.eecs.berkeley.edu/ptolemyII/index.htm, accessed 2009/05.

[LLWC03] Philip Levis, Nelson Lee, Matt Welsh und David Culler. TOSSIM: accurate and scalable simulation of entire TinyOS applications. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, New York, NY, USA, 2003. ACM, S. 126–137.

[Lutt06] Bas Luttik. What is algebraic in process theory? *Electr. Notes Theor. Comput. Sci.*, Band 162, 2006, S. 227–231.

[MeCG05] Tom Mens, Krzysztof Czarnecki und Pieter Van Gorp. 04101 Discussion – A Taxonomy of Model Transformations. In Jean Bezivin und Reiko Heckel (Hrsg.), *Language Engineering for Model-Driven Software Development*, Nr. 04101 der Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2005. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[Mill08] Konrad Miller. Evaluierung von Monitoringstrategien in Ad-Hoc-Netzen. Studienarbeit an der Universität Karlsruhe (TH), 2008.

[MiPW89] Robin Milner, Joachim Parrow und David Walker. A Calculus of Mobile Processes, Parts I and II. Technischer Bericht -86, 1989.

[Mura89] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4), Apr 1989, S. 541–580.

[ORMS] Architecture Board ORMSC. Model Driven Architecture (MDA) - A Draft with annotations of issues to resolve. http://www.omg.org/docs/ormsc/01-04-01.pdf, accessed 2008/07.

[Proc99] Proc. Conference on Information Sciences and Systems (CISS). *Anycast Routing for Mobile Services*, 1999.

[Ried07] Till Riedel (Hrsg.). Mobile und verteilte Systeme, 2007.

[Ried09] Till Riedel (Hrsg.). *Mobile und verteilte Systeme*, Band 6: Seminar im WS 2008/09 der *Interner Bericht / Universität Karlsruhe, Fakultät für Informatik ; 2009,3*. Universität Karlsruhe, Fakultät für Informatik, Karlsruhe. 2009.

[ScAW94] B. Schilit, N. Adams und R. Want. Context-aware computing applications. *Mobile Computing Systems and Applications, 1994. Proceedings., Workshop on*, Dec 1994, S. 85–90.

[SMRL+97] Starner, Mann, Rhodes, Levine, Healey, Kirsch, Picard und Pentland. Augmented reality throuh wearable computing. *Presence: Teleoper. Virtual Environ.*, 6(4), 1997.

[TiSh] A. Tikhomirov und A. Shatalin. Introduction to the Graphical Modeling Framework. http://www.eclipsecon.org/2008/index.php?page=sub&id=337, accessed 2009/04.

[TMSEFH06] M. Torrent-Moreno, F. Schmidt-Eisenlohr, H. Fussler und H. Hartenstein. Effects of a realistic channel model on packet forwarding in vehicular ad hoc networks. *Wireless Communications and Networking Conference, 2006. WCNC 2006. IEEE*, Band 1, 0-0 2006, S. 385–391.

[Vari96] Various. *Development and Acquisition Booklet*. Federal Financial Institutions Examination Council. 1996.

[vDKV00] Arie van Deursen, Paul Klint und Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6), 2000, S. 26–36.

[Weis99] Mark Weiser. The computer for the 21st century. *SIGMOBILE Mob. Comput. Commun. Rev.*, 3(3), 1999, S. 3–11.

[ZeBG98] Xiang Zeng, Rajive Bagrodia und Mario Gerla. GloMoSim: a library for parallel simulation of large-scale wireless networks. In *PADS '98: Proceedings of the twelfth workshop on Parallel and distributed simulation*, Washington, DC, USA, 1998. IEEE Computer Society, S. 154–161.