

Towards a Context-Driven Development Framework for Ambient Intelligence

Dennis Wagelaar

System and Software Engineering Lab

Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussel, Belgium

dennis.wagelaar@vub.ac.be

Abstract

Portable and embedded devices form an increasingly large group of computers, often referred to as Ambient Intelligence (AmI). This new variety in computing platforms will cause a corresponding diversity in software/hardware platforms and other context factors. Component-based middleware platforms offer a uniform environment for software, but they do not take away specific context differences, such as hardware resources, user identity/role and logical/physical location. Specialised component versions and/or configurations have to be made for each computing context if that computing context is to be used to its full extent. This is because the fine differences between component versions cannot be separated into finer components with the current component models. Aspect-oriented programming and generative programming technologies can be used to provide the fine-grained modularity that is necessary. In addition, the diversity of component-based platforms themselves form an extra reason for different component versions. We propose using a context-driven framework for the development of AmI components, which is based upon a gradual refinement mechanism. This refinement mechanism can cope with the course-grained differences between component models as well as the fine-grained differences between computing configurations.

1. Introduction

Nowadays, many computers are portable and embedded devices, such as PDAs, smartphones, embedded computers in cars, heating installations, etc. In the near future this Ambient Intelligence (AmI) is expected to increase [9]. This will cause a large diversity in hardware platforms - but also other factors, such as software platform, user identity/role and logical/physical location. The context [7] in which AmI software has to work varies strongly.

Within software development, there are several ways to bridge the differences in (hardware) platforms. Besides

standardised operating systems, various middleware platforms have been developed, which offer a uniform environment for software. Examples are CORBA [16], J2EE [21] and .NET [19].

These platforms do not take away the differences in e.g. memory space, I/O capabilities and processing power, however. In case we want to use a specific computer configuration to its full extent, we need specific components [24] or component configurations [12] for each computing context. The problem lies in the granularity of current component models: the fine differences between component versions cannot be separated into finer components.

Possible solutions for this granularity problem are aspect-oriented programming (AOP) [10] and generative programming [6]. With AOP, aspects that are shared between several components can be put in a separate module. With generative programming, components can be built up from sub-component templates or patterns. These technologies offer the necessary fine granularity.

In addition, the diversity of the component-based platforms themselves forms an extra reason for specific component versions. This is a very course-grained difference between components: the programming language and/or programming interface differs between platforms. All components need to be ported to the other platform.

To use the right level of granularity at any given moment, a refinement mechanism that works from design level down to implementation level is necessary. Such a refinement mechanism must be able to define several alternative refinements (based upon context data) for high-level design elements. Current object-oriented design approaches, such as the Unified Modeling Language (UML) [17] or extensions of UML [3][5][8][20][22] do not offer support for such a refinement mechanism.

We propose using a context-driven framework for the development of AmI components. The fundamental idea is to already compose the components at design level. This way, developers work with “blueprints” of components instead of the components themselves. In addition, the composition of components can be annotated with context data.

Each design-level component can have various refinements that work within specific context boundaries. Both the component configuration and the context data can be used to choose between various refinements for design-level components.

By using a context-driven refinement mechanism, an optimised implementation can gradually be obtained. The component “blueprint” approach allows for adjusting the actual component interfaces during refinement. Currently, the Object Management Group (OMG) is working on the Model-Driven Architecture (MDA) [15] standard, which uses transformations to refine a Platform Independent Model (PIM) into a Platform Specific Model (PSM). Several of these transformations chained together can be used to generate an implementation from a conceptual design. However, the MDA standard is currently very open-ended and as such does not provide a standard method yet.

An approach that can generate an optimised implementation from a conceptual design, based upon gradual refinements, is CoCompose [26]. CoCompose uses a visual, concept-based design language in which the language elements (*Concepts* and *Composites*) can contain refinements. These refinements can be expressed in terms of nested concepts and composites or as a code template expressed in a specific programming language. The code generation is currently driven by the target programming language or platform and the way the design elements are composed (interfacing, dependencies, etc.). Design Algebra [25] is used as an underlying mechanism to process and choose between alternative refinements.

In the following sections, a framework for modelling and implementing AmI component-based systems is discussed. This framework is based upon the CoCompose approach. The proposed modelling language will be discussed in section 2. Section 3 discusses a context-driven refinement mechanism for the modelling language. Finally, section 5 concludes this paper.

2. Context-aware Modelling Language

A modelling language for AmI components must be able to describe context and context dependencies as well as the components themselves. Since CoCompose is able to describe partial context data (platform dependencies and global heuristics) for its design elements, the CoCompose design language is chosen as a starting point. Instead of adding more elements to the design language in order to describe the various context aspects, we chose to simplify the modelling language to only supply a mechanism for modularisation. The actual functional design language constructs (e.g. inheritance, association and specific design patterns) can then be built on top of this modularisation layer.

2.1. Concepts

The central element in this meta-model is the *Concept*. Each element we wish to model is initially represented as a Concept. Related concepts can be connected to each other with a *Relationship*, which specifies only that two Concepts are related without any further detail on the nature of the relationship. The nature of a relationship can be made clear by using another, intermediate concept that explains the connection. Nested concepts can be used to describe concepts and their relationships in more detail. Fig. 1, for instance, shows how a “MyClass” Concept is linked to the “Parent” Concept within the “AnInheritance” Concept. This means that “MyClass” fills the parent role of an inheritance relationship.

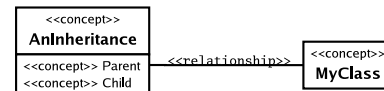


Figure 1. Representing design elements as concepts

The UML [17] language model can be mapped onto our language model, such that design elements described using UML constructs have a Concept representation. This mapping is done on the meta-model level: each UML element type (e.g. class, operation) is mapped onto a Concept representing the element type (e.g. a Concept named *Class*). Each actual design element described in UML will then map to a CoCompose concept, which inherits from the concept that represents the UML element type. A class named *MyClass*, for instance, will map to a *MyClass* Concept that inherits from a *Class* Concept.

Concepts can inherit refinements from other concepts or superimpose their own refinements on other concepts (refinements are explained in subsection 2.2).

Fig. 2 shows a design of a Breakout game, including a screen shot of the game itself. The objective of the game is to remove all the bricks at the top of the screen by hitting them with the ball. The ball must be bounced back with the paddle at the bottom, which is controlled by the player. If the ball falls down the screen (paddle has missed), the game is over. In the design, the “Breakout” Concept represents the screen, which contains a “Ball”, a “Paddle” and several “Bricks”. A Brick can observe a Ball by means of the “BallObserver” collaboration, which inherits from an “Observer” collaboration (this is not shown explicitly in the design).

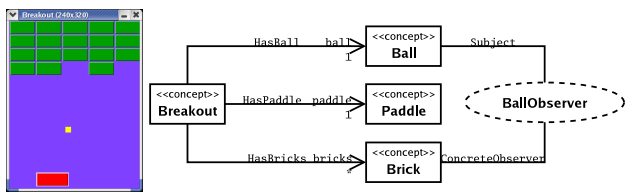


Figure 2. An example breakout game design

2.2. Refining Concepts

Concepts can have several refinements, which can be *Solution Patterns*, *Implementation Generators* or *Implementation Patterns*. Solution Patterns are template designs again described in terms of Concepts. Implementation Patterns are code templates expressed in a specific programming language and work within specific constraints. Implementation Generators are executable code adaptors that take in the generated code so far and return the adapted code that reflects their functionality. These Implementation Generators also work within specific constraints and can be used to generate code for e.g. inheritance. Concepts can reuse refinements by (1) inheriting from other concepts or (2) superimposing their refinements on other concepts.

An example Solution Pattern for a “Brick” Concept is shown in Fig. 3. A solution pattern can be applied to refine a Concept: the Concept is then replaced by an instance of the solution pattern. The “Brick” class in the Solution Pattern has a bold border, which means it is the *Default Concept* of the Solution Pattern: when applying the Solution Pattern, the Default Concept assumes the name and place of the original Concept. In the example, the “Brick” concept from Fig. 2 will be replaced by the “Brick” class from the Solution Pattern. In addition, the “Brick” class inherits from the “Panel” class, which has a Java Implementation Pattern (shown as a UML comment). Note that this Implementation Pattern (and thus the Solution Pattern) will only work within a context that supports Java and the AWT toolkit. Other Solution Patterns can be defined for the “Brick” Concept, that work within different a context, e.g. Java/Swing or Windows CE native.

Fig. 4 shows a Solution Pattern for the “Observer” concept (see “BallObserver” in Fig. 2). This Solution Pattern uses *Roles* to parametrise itself: the Roles will be filled in by Concepts in the original design. There are two Roles in this case: “Subject” and “ConcreteObserver”. Colouring is used

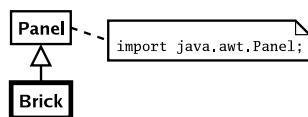


Figure 3. A Solution Pattern for “Brick”

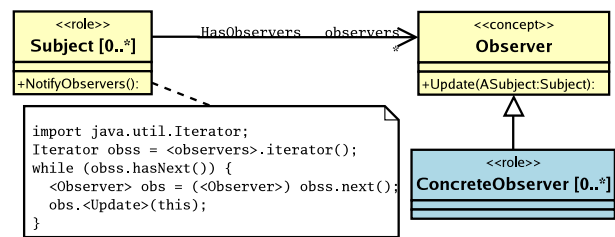


Figure 4. A Solution Pattern for “Observer”

to mark the Concepts that will be instantiated once for each time the Role with the same colour is filled. For each time the “Subject” role is filled, for instance, an “Observer” class will be generated (with its nested Concepts: “Update”, “A-Subject” and “IsSubject”¹). Note that this colouring is only necessary for roles with a multiplicity constraint other than [1..1] (in this case, both roles can be filled zero to many times – [0..*]).

The “NotifyObservers” operation has an Implementation Pattern as a Java Method. The Implementation Pattern can refer to other Concepts within the model by using <>. An Implementation Pattern can impose constraints on referenced Concepts, such as “Observer must be implemented as a Java Interface or Class”.

In order to describe generic context data, a simple *Property* model is used. A Property can contain other model elements, such as Property elements. This way, a tree-wise description of (context) properties can be made for each Model, Concept or Refinement. A special *ModelProperty* can be used to refer to an external model containing a Property tree.

If these descriptions of context properties are to be useful, we need to be able to describe constraints on these properties. A specific Concept refinement may only work within a specific context. The *Constraint* model supports simple boolean composition (and/or/not) of *ElementConstraints*. An ElementConstraint points at the model element (e.g. a Concept) on which it places the constraint. Furthermore, an ElementConstraint contains its own tree of Property elements. An ElementConstraint holds if the leaf Properties of the common sub-tree of the Property elements are also leaf properties in the original constrained element tree.

Consider the example in Fig. 5, which displays an ElementConstraint “Device” applied to a Solution Pattern “BrickSolution”. The Property tree on the left says that “BrickSolution” works within any Java platform and within a Windows CE platform, Pocket PC 2000 edition. The “Device” has a Personal Java platform. The constraint tells that “BrickSolution” should run on “Device”. In this case, the constraint holds since the com-

1 “IsSubject” is the colon that marks the type of “ASubject”

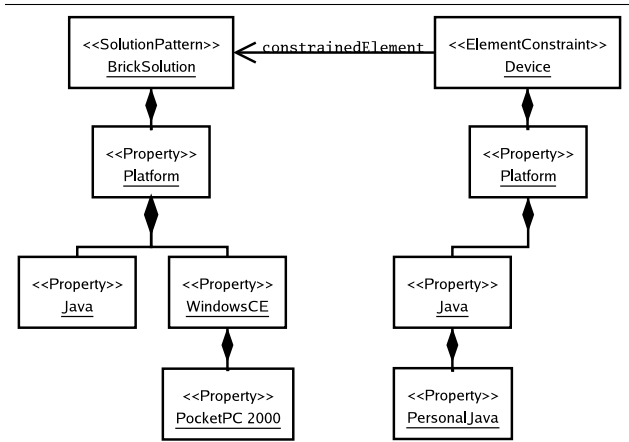


Figure 5. Example of an ElementConstraint applied to a Solution Pattern

mon Property sub-tree made up by “Platform” and “Java” has “Java” as a leaf property, which is also a leaf in the “BrickSolution” Property tree. If the “Java” Property under “BrickSolution” would have contained a Property “Java2”, then the constraint would have failed: “Java” would be the leaf in the sub-tree, but not in the original Property tree of “BrickSolution”. The meaning of a “Java2” Property under the “Java” Property is that “BrickSolution” would only have worked within a Java 2 version of Java.

3. Context-driven Refinement Mechanism

By using a simple modelling language for describing software and context properties/constraints, a refinement mechanism that is relatively simple itself can be defined. The refinement is steered by a target context description, which includes the target platform. The refinement mechanism itself is based on Design Algebra. Each Concept in a design can have multiple refinements. The design itself is made up of several Concepts that are related to each other. The combinations of possible refinements for all Concepts form the Design Space. By applying reduction operations on that space, one combination of refinements can eventually be chosen.

Instead of determining the entire design space, a more direct approach is taken. From the view of a top-level model, there are several Concepts, which have several refinements. All these refinements are annotated with Constraints and context Properties (see previous section). First, we can (recursively) eliminate all refinements of which the constraints do not hold. For example, a refinement of a certain Concept only works for J2EE. If the model is eventually translated to C#.NET, this refinement won’t work, so it can already

be eliminated. This way, non-feasible refinements are eliminated as soon as possible.

Still, many possible refinements will remain. The strategy taken here is to initially choose the refinement that best matches the context data: e.g. if a model is translated for J2ME [23] on a PDA, then the refinement with the smallest footprint is chosen first. In practise, the context information contains a user-defined heuristic, which looks for a Property that tells the footprint size. If no such Property is found, the footprint will be ignored. In a later stage of refinement, it may appear that a specific refinement constraint no longer holds. If the chosen refinement later turns out to be non-feasible, the next-best refinement is chosen, etc.

When the refinements are chosen, the Solution Pattern refinements will applied first. This results in a fully refined version of the design. Fig. 6 shows the refined version of the example design shown in Fig. 2. Note that Implementation Generators are present for all Concepts, except for “BallObserver”. This will be covered in the next step.

When all refinements have been chosen, there are still a number of Properties that have to be determined for certain Concepts, before they can be implemented. For example, if the Concept “Breakout” fills the “child” role of an “Inheritance” Concept, then “Breakout” must be implemented as a class. A Property named “Construct” could be set to “class” for the “Breakout” Concept. If “Breakout” fills roles of other Concepts, such as “HasBall” and “HasBricks”, the construct used to implement “Breakout” has to be valid for those Concepts as well. This way, other Properties, such as “Access” (private, protected, public), “Static” and “Abstract” can be set as well. Heuristics and fuzzy logic [2] can help when choosing from multiple options (e.g. choosing “interface” over “class” as “Construct” where possible).

Note that the “BallObserver” concept from Fig. 6 doesn’t have any Implementation Generators yet. Now that its Properties for implementation are determined (e.g. “Construct” is “interface”), a Concept can be chosen from the avail-

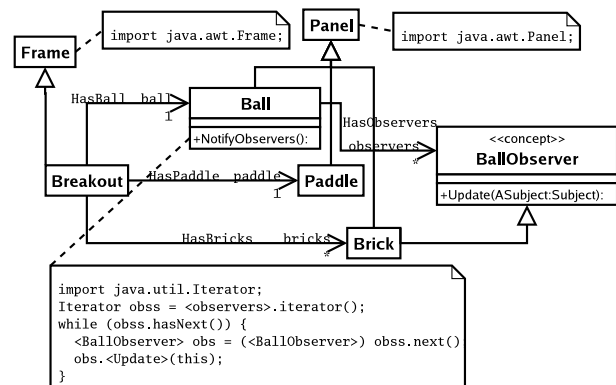


Figure 6. Refined version of Breakout

able design elements (e.g. “Class”, “Method”, “Instance”), which has an Implementation Generator that supports the determined Properties. In this case, the “Interface” concept can generate code for “BallObserver” as an interface.

When all Properties that are necessary for implementation have been determined, enough information exists to apply the Implementation Generators. Since the implementation generators for, e.g., Java have common knowledge about the structure of a Java program, they can work together on composing the separate Java elements. First, an object structure representing the Java elements is created, so the implementation generators do not have to parse back the code that has been generated by previous implementation generators. The applicable Implementation Patterns are pasted in by the Implementation Generators where necessary. When all elements are represented in this structure, code is generated for the entire structure.

4. Related Work

The idea of patterns of design elements has also been used in Theme/UML [5], which is an aspect-oriented extension of UML. It uses *composition patterns* to describe aspects. Composition patterns are template packages that are parametrised with specific UML element. The Kobra method [3] is an approach for component-based product line engineering with UML. It also uses pattern-based refinements for design elements and advocates to describe each concern in a different model. OO-Method [18] also introduces a pattern-based approach for design refinement and code generation. PRISMA [14] is a modelling approach that can be used to model context data. Both OO-Method and PRISMA use OASIS [13] as formal basis for their code generation. The above methods do not facilitate a mechanism for defining alternative refinements.

In *generative programming* [6] and *step-wise refinement* [4], *features* and feature models are used to model a family of software systems instead of a single system. Features can be optional or mandatory for a software system, depending on the presence of other features. Our language can model optional features through Solution Pattern roles, which can be left unfilled. Alternative features can be modelled by the alternative refinements.

Recently, the Object Management Group has introduced the Model-Driven Architecture (MDA) standard [15]. MDA forms an abstraction layer to specific implementation platforms. It uses model transformations to refine a high-level design (described using a Platform-Independent Model or PIM) to a platform-specific design (described using a Platform-Specific Model or PSM). Several layered PSMs can be defined to gradually refine the design. Co-Compose fits in the MDA vision in that it also uses several layered refinements, which are described using

meta-level solution patterns. These form the intermediate Platform Models (PMs) that define the transformation from a Platform Independent Model (PIM) to a Platform Specific Model (PSM).

In the context of MDA, an approach based on graph transformations has been proposed to transform UML design models to implementation [1]. UMLAUT [11] is a generic UML transformation framework, which can be used for design pattern generation and aspect weaving amongst others. The framework is built upon the UML meta-model and allows for defining your own transformations, which can be stored in a transformation library.

Our approach has its own language on which refinement transformations are applied. The actual transformations are always replacement transformations, in which the element to be refined is always replaced by its refinement. UML models can also be used within our framework via mapping. This way, our refinement mechanism can be applied to different versions of UML by creating only a new mapping.

5. Conclusion and Future Work

The proposed development framework allows for modelling components in a “blueprint” design. The framework is specifically suited for Ambient Intelligence, because it can cope with a large diversity of context variables (e.g. hardware/software platform, user id/role). An optimised implementation for a specific computing context can be automatically generated from a design. This is achieved through modelling several alternative refinements for generic design elements, called Concepts. When generating an implementation, context data and constraints are used to choose the appropriate refinements. By having modelled the refinement structure, the gradual refinement down to code level can be traced back entirely to the top-level design. If the context changes, the refinement can be back-traced as far as necessary to optimise for the new context.

Instead of defining a UML extension for modelling Aml components and context, a small subset of the UML meta-model is chosen and extended. This is done to keep the language as well as the refinement mechanism as simple as possible. In the future, a more detailed mapping from UML models to the proposed framework can be defined.

The current Property and Constraint model are very simple. These models will be extended to support different types of properties (e.g. version numbers) and constraints (e.g. “within range of”). Since new kinds of context properties and constraints may emerge continuously, an extension mechanism that allows for user-defined property and constraint types may be necessary.

The current refinement mechanism is a preliminary mechanism. It has to be specified in more detail

in order to implement it. Experiences with the CoCompose approach have shown that the refinement mechanism may suffer from scalability problems. These problems can be diminished by an optimised implementation of the refinement mechanism and by locally applying that mechanism (e.g. for one or a small group of components at a time). Also, by using selection of refinements instead of elimination, the rules can be simplified: it is acceptable to not select some working refinements, but it is not acceptable to not eliminate non-working refinements.

Tool support is necessary for the development framework to be feasible. The CoCompose approach, on which our development framework is based, already has tool support. Based on the experience gained with CoCompose, a suite of small, robust tools will be developed to support our framework.

References

- [1] A. Agrawal, T. Levendovszky, J. Sprinkle, F. Shi, and G. Kar-sai. Generative programming via graph transformations in the model-driven architecture. In *OOPSLA, Workshop on Generative Techniques in the Context of Model Driven Architecture*, Seattle, USA, 2002.
- [2] M. Akşit and F. Marcelloni. Deferring elimination of design alternatives in object-oriented methods. *Concurrency and Computation: Practice and Experience*, 13(14):1247–1279, 2001.
- [3] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wust, and J. Zettel. *Component-Based Product Line Engineering with UML*. Addison Wesley, Reading, Massachusetts, USA, Nov. 2001.
- [4] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, pages 187–197, Portland, USA, May 2003.
- [5] S. Clarke and R. Walker. Towards a standard design language for aosd. In M. Akşit, editor, *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, pages 113–119, Enschede, The Netherlands, 2002.
- [6] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison Wesley, Reading, Massachusetts, USA, June 2000.
- [7] A. K. Dey. Understanding and using context. *Personal and Ubiquitous Computing Journal*, 5(1):4–7, 2001.
- [8] J. Dong. Uml extensions for design pattern compositions. *Journal of Object Technology*, 1(5), 2002.
- [9] K. Ducatel, M. Bogdanowicz, F. Scapolo, J. Leijten, and J.-C. Burgelman. *Scenarios for Ambient Intelligence in 2010*. ISTAG, Feb. 2001. [Online] <http://www.cordis.lu/ist/istag.htm>.
- [10] T. Elrad, M. Akşit, G. Kiczales, K. Lieberherr, and H. Os-sher. Discussing aspects of aop. *Communications of the ACM*, 44(10):33–38, 2001.
- [11] W.-M. Ho, F. Pennaneac’h, and N. Plouzeau. Umlaut: A framework for weaving uml-based aspect-oriented designs. *Technology of object-oriented languages and systems (TOOLS Europe)*, 33:324–334, 2000.
- [12] M. Larsson. *Applying Configuration Management Techniques to Component-Based Systems*. Licentiate thesis, Department of Information Technology, Uppsala University, Dec. 2000. Also published as report MRTC 00/24 at Mälardalens högskola.
- [13] P. Letelier, P. Sánchez, I. Ramos, and O. Pastor. *OASIS 3.0: Un enfoque formal para el modelado conceptual orientado a objeto*. Servicio de Publicaciones Universidad Politécnica de Valencia, Valencia, Spain, 1998.
- [14] J. J. Martinez and I. R. Salavert. A conceptual model for context-aware dynamic architectures. In *Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops (ICDCSW’03)*. IEEE Computer Society, 2003.
- [15] J. Miller and J. Mukerji. *MDA Guide*. Object Management Group, Inc, May 2003. Version 1.0.
- [16] Object Management Group, Inc. *The Common Object Request Broker: Architecture and Specification*, July 2002. Version 3.0.
- [17] Object Management Group, Inc. *Unified Modeling Language Specification*, Mar. 2003. Version 1.5.
- [18] V. Pelechano, O. Pastor, and E. Insfrán. Automated code generation of dynamic specializations: an approach based on design patterns and formal techniques. *Data and Knowledge Engineering*, 40(3):315–353, 2002.
- [19] D. S. Platt. *Introducing Microsoft .NET*. Microsoft Press, 3rd edition, May 2003.
- [20] B. Rumpe, M. Schoenmakers, A. Radermacher, and A. Schrr. Uml + room as a standard adl? In F. Titsworth, editor, *Engineering of Complex Computer Systems, ICECCS’99 Proceedings*. IEEE Computer Society, 1999.
- [21] B. Shannon. *Java™2 Platform: Enterprise Edition Specification*. Sun Microsystems, Inc., July 2001. Version 1.3.
- [22] D. Stein, S. Hanenberg, and R. Unland. An uml-based aspect-oriented design notation for aspectj. In M. Akşit, editor, *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, pages 106–112, Enschede, The Netherlands, 2002.
- [23] Sun Microsystems, Inc. *Java 2 Micro Edition website*, 2003. [Online] <http://java.sun.com/j2me/>.
- [24] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley, Reading, Massachusetts, USA, 1st edition, 1998.
- [25] B. Tekinerdoğan and M. Akşit. Synthesis based software architecture design. In M. Akşit, editor, *Software Architectures and Component Technology*, Dordrecht, The Netherlands, 2002. Kluwer Academic Publishers.
- [26] D. Wagelaar and V. Jonckers. A concept-based approach to software design. In *Proceedings of the 7th IASTED International Conference on Software Engineering and Applications*, Marina del Rey, California, USA, 2003. to be published.