Rule-driven component composition for embedded systems

Thomas Genßler
Program Structure Group
Forschungszentrum Informatik
Haid-und-Neu-Straße 10-14
76131 Karlsruhe, Germany
+49-721-9654 620
genssler@fzi.de

Christian Zeidler
Corporate Research
ABB Germany
Speyerer Str. 4
D-69115 Heidelberg, Germany
+49-6221-59 6259
zeidler@decrc.abb.de

ABSTRACT
We present in this paper an approach to *correct-by-construction* software composition based on the use of non-functional properties of the involved components and a set of constraints and design rules over those properties. We focus on the domain of software for embedded devices although most of the presented concepts can also be extended to component-based software development in general. We believe that software development for embedded devices would benefit a lot from the component-based approach. However, software for embedded devices usually has to fulfill much stronger reliability and correctness requirements than conventional software. This calls for appropriate techniques and approaches to ensure the correctness of the software being built. We propose to use first order predicate logic to check statically verifiable properties design rules. Furthermore, we support the specification of contracts which will be checked dynamically.

Keywords
Static composition checking, components, software for embedded devices

1 Introduction

Component-based software engineering is quickly becoming a mainstream approach to software development. According to Components, Objects and Development Environments: 1998, International Data Corporation the expected turnover increase will be a factor of five from 1997 to 2002. At the same time there will be a massive shift from desktop applications to embedded systems. The PITAC report describes this as the phenomenon of the disappearing computer. More and more traditional IT systems will move from visible desktop computers to invisible embedded computers in intelligent apparatus. Furthermore, industrial automation systems become increasingly decentralized, relying on distributed embedded devices (intelligent field devices, smart sensors) to not only acquire but also pre-process data and run more and more sophisticated application programs (control functions, self-diagnostics, etc.). As a consequence of these facts, one can expect that component-based software engineering for embedded systems will be a key success factor for the software industry in the coming decades.

But the state-of-the-art in software engineering for embedded systems is far behind other application areas. Software for embedded systems is typically monolithic and platform-dependent. These systems are hard to maintain, upgrade and customize, and they are almost impossible to port to other platforms. Component-based software engineering would bring a number of advantages to the embedded systems world such as fast development times, the ability to secure investments through re-use of existing components, and the ability for domain experts to interactively compose sophisticated embedded systems software. Visual techniques have been proven to be very effective in specific domains like GUI software composition. Composition of embedded systems software still has a long way to go to reach that level. At the very least, users would benefit greatly from the effective use of visual techniques for providing feedback in the development process (during design, composition, installation, and during runtime validation). Unfortunately component-based software engineering cannot yet be easily applied to embedded systems development today for a number of reasons. Up to now, the mainstream IT players did not pay much attention to the (so far) relatively small embedded systems market and consequently did not provide it with suitable technologies or off-the-shelf software (such as operating systems). From a technical point of view, these choices were justified by considering the major characteristics of embedded devices, such as limited system resources (CPU power, memory, etc.) and man machine interface functionality, the typically harsh environmental conditions, and the fact that the development and target systems are not the same.

The rapidly growing market share of embedded systems is changing the equation and making investment in component-based software engineering for embedded systems not only viable but also essential. Vendors of embedded devices would benefit by being able to offer scalable product families, whose functionality could be tailored by flexible composition of reusable building blocks. These families are differentiated by the performance of the hardware and the provided functionality, but are based on re-use of many identical software components. All this requires that the embedded systems software be modular and composed of loosely coupled, largely self-sufficient, and independently deployable software components.

The project frame for this paper is the project PECOS - Pervasiv Component Systems, which is funded by the European

Community. Figure 1 illustrates PECOS main objectives, The goal of PECOS is to enable component-based software development of embedded systems by providing an environment that supports the specification, composition, configuration checking, and deployment of embedded systems built from software components.

There are many challenges we address in the PECOS project goals. In this paper we focus on the aspect of component composition. We first give an overview on the development process with component before we present an approach to *correct-by-construction* software composition.

2 Development with Components

The specific domain of embedded systems implies specific restrictions. To cope with the resource limitations is one domain specific problem. Another one is to support development of real-time application assembled out of components. This is a challenge by itself and a topic of investigations of the last decades. The most prominent approaches are RoseRT by Rational [11] and Rhapsody by Ilogix [8].

Both of them apply the event base programming style and support implementation based on state automata, but do not consider reuse or component orientation as their major drivers. Therefore they start with UML-like specification and extend the definition tools with functionality that provides code generation for an specific target. Both approaches do not consider neither component model definition nor architecture, beyond the event based communication of capsules [11] or active objects [8]. Composition of applications out of components and active reuse support by appropriate repository implementation is not offered adequately either.

In order to make component-based software engineering happen, not only for field devices, and to achieve a reduction of development costs and time by reuse of established and proven components, it is not enough to solve only one of the presented obstacles. An overall approach for the development of component-based embedded software is needed.

As we believe this approach has to comprise several main features as depicted in Figure 1, which we have categorized in five groups and describe below. In a first outline the identified groups should concentrate of the following issues:

Component model:

- addresses non-functional properties and constrains such as worst-case execution time and memory consumption
- allows to specify efficient functional interfaces (e.g. procedural interfaces)
- allows to specify architectural styles that describe components connections and containment relations
- allows for code generation and controlled component adaptation when architectural styles are applied to components (source language or generative components)

Component-based architecture for field devices:

- a framework for field devices that is expressed as standard interfaces, components, and architectural styles
- is based on field bus architecture
- express compile-time optimization abilities, which could be applied during target code preparation

Repository:

- storage and retrieval of components during analysis, design, implementation, and composition
- stores components and architectural styles according to the component model including interface descriptions, non-functional properties, implementation (potentially for different micro controllers), support scripts for composition environment, test cases
- supports component versioning

Composition Environment:

- supports composition techniques (visual or script based)
- checks composition rules attached to architectural styles in order to verify that a component configurations meets their constraints
- performs component adaptation and code generation for the application
- supports definition of composition rules, which in an subsequent step could be compiled to architectural styles description

Run-time Environment:

- provides an efficient implementation model for components
- addressing the constrains for field devices: low available memory, implementation possibly necessary in C or optimized C^{++}
- supports the approach to compile a component-based design into a optimized firmware for the embedded device, thus having no run-time environment beside the RTOS (Real-time operating system)
- allows for a hardware and RTOS independent implementation of components (e.g. by an RTOS abstraction layer [13])

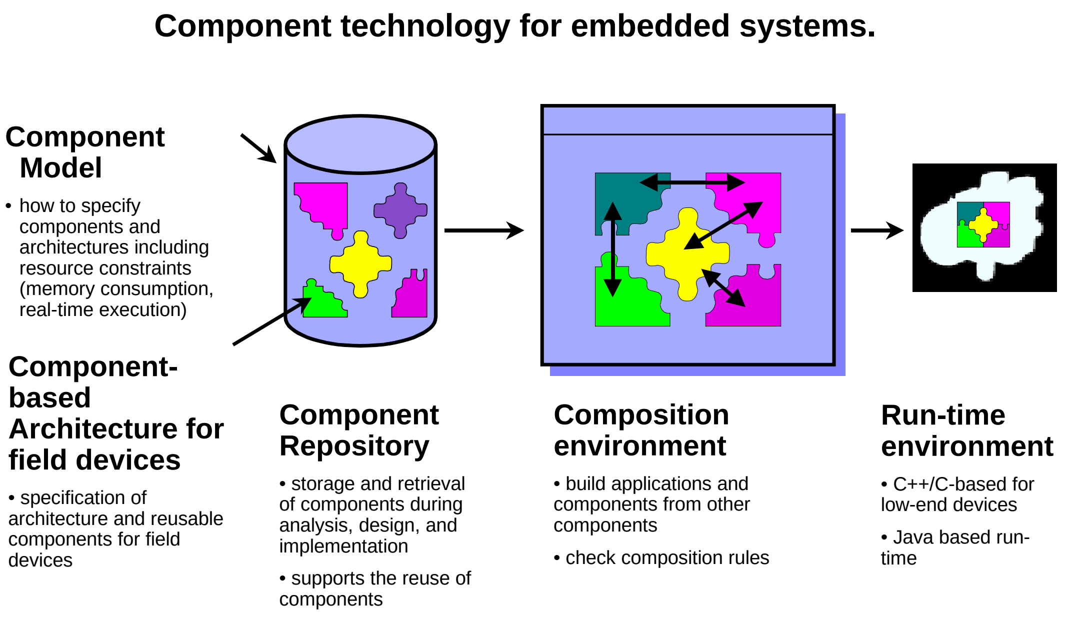
Component technology for embedded systems.
Component Model
• how to specify components and architectures including resource constraints (memory consumption, real-time execution)
Component-based Architecture for field devices
• specification of architecture and reusable components for field devices
Component Repository
• storage and retrieval of components during analysis, design, and implementation
• supports the reuse of components
Composition environment
• build applications and components from other components
• check composition rules
Run-time environment
• C++/C-based for low-end devices
• Java based run-time

Figure 1: PECOS main targets

Based on these five categories, which make up the major ingredients for a component-based systems (CBS) development. All of them are currently poor developed or even absent. For the ongoing discussion in this paper we concentrate on the development process with components. Where a prerequisite is a sound component specification and a composition support that the developer can trust, since it provides him some consistency and certification support for the result of his composition work.

3 Our Approach: correct-by-construction Software composition

The most critical part of component-based system construction is the composition process. Most future defects of the system being built arise from mismatches and inadequacies of the composed components. Thus, the static verification of the correctness of component composition is a very crucial and important task in the development process.

In traditional software development processes, however, static correctness checks are usually reduced to syntactical checks or simple semantics checks like type-checks. But often these simple checks are not enough to discover defects in the system that are caused by structural, functional or non-functional inadequacies of the composed components. Those defects are hopefully discovered during testing. If not, it can become very costly to correct them, especially in the area of embedded devices where software is usually stored in ROM.

The goal of our work is to provide more powerful means to support a *correct-by-construction* software development process. For this purpose we introduce the notion of *rules* as means to provide stronger correctness checks than mere syntactical checks or type checking. Rules represent statically checkable constraints. We informally define the term *static correctness* as follows: A composition is statically correct if

- it is syntactically correct **and**
- the system complies with the used static type system **and**
- all rules are fulfilled

However, some of the constraints on a system can not be expressed, and thus not be checked, statically. In order to increase dynamic correctness we use contracts to express constraints such as pre-, post-conditions and invariants.

The remainder of this chapter is split into two parts. In the first part we introduce our current component meta-model which we partly implemented in our composition compiler prototype PECOS-CoCo. This meta model focuses on modelling component systems in the field of software for embedded devices. Thereafter we concentrate on how this meta-model supports development with components in this application domain, especially how it helps to ensure correct component composition. We show, how rules can be used to better support a *correct-by-construction* development process. We also demonstrate how we use contracts to specify dynamically checkable constraints.

Component Meta-Model

The proposed component meta model serves to model component-based systems in the area of software for embedded devices. However, most of the concepts can easily be mapped to conventional component-based systems.

A central entity in our model is a **component**. Components model stateful entities of computation, i.e., the actual pieces of software in the target language of an embedded software system. A component has a *unique identifier* and a *set of* *properties*. Components support single inheritance. Components can be composed out of other components. Components have a concept of instantiation. Figure 2 shows a component definition in our composition system.

Properties of components are distinguished in general purpose properties and pre-defined properties with certain semantics. The latter are explicitly modelled as meta-model elements. The following shows a list of pre-defined properties of a component:

- **Ports**: Ports are distinguished in signals and events/handler. While signals represent pure data transfer from one component to one or more other components, events and handlers serve to invoke functionality upon a component. An event corresponds to the invocation of functionality while handler correspond to the declaration of functionality.[1] Contracts can be assigned to a port (see below for description of contracts).

- **Rule references**: Rules can be attached to components. When the system is checked, those rules must hold.

- **Super component**: A component may have exactly one super component (single inheritance).

- **Description**: A documentation string.

General purpose properties are either mandatory or optional. They can be used to specify non-functional properties of components such as worst-case memory consumption,[2] needed cycle time etc. Mandatory properties must be set when the component is composed. A property can either be set in the component or for a concrete instance of that component. In the first case, this property is used for all component instances if the value is not changed for an instance while in the second case the property value belongs to the instance setting it.

Connecting components means connecting their ports or, in other words, establishing a communication link between components. Communication, however, can happen in different ways, e.g., synchronously/asynchronously or via method call or message passing. To abstract from the concrete communication mechanism, we us connectors. **Connectors** represent meta programs that generate or transform code in the target language in order to glue the pieces of code together. Connectors have a unique identifier. They are stateless and can not be instantiated for obvious reasons. [3] A connector takes a set of source and target ports as parameters. A more detailed discussion of connectors as meta programs can be found in [2].

[1]At target language level, an event corresponds to a method call while handler are mapped onto method definitions.

[2]In the given application domain (software for embedded devices), such information is usually available.

[3]Connectors basically represent code generators. Thus it makes no sense to instantiate them.

```
component Actuator  extends FunctionBlock{
 ports:
  // signals
   public signal in int p1{
   pre:
    p1>0;
  };
  // event handlers, events are
  // specified similarly
   public handler execute(){
   pre:
    // the signal p1 must have been initialized
    p1 ==  valid
  };
 properties:
     mandatory code : string =
        "/codebase/Actuator.java";
    //the component is active and
    //needs to be scheduled by a scheduler
     mandatory active : boolean = true;
    //worst case execution time=30ms
     mandatory executionTime : int = 30;
    // must be set when composing this component
     mandatory  threshold: int;
 description:
   "Description of Actuator"
}
```

Figure 2: Example of a CoCo Component

A **composition** specifies how components are interconnected. Compositions declare a fix number instances of components and define their configuration. Furthermore, a composition specifies how the ports of those instances are wired, i.e., which connector is used for connecting which ports. The expression `sequentialMultiCast-MethodCall(s.execute() -> a1.execute(), a2.execute());` in figure 3 states that the actuator components `a1, a2` be connected to the scheduler component `s`, using method call communication. Since their are two communication sinks (`a1.p1, a.2.p1`, those methods are called sequentially. The connector `sequential-MultiCastMethodCall` generates the respective code fragements, i.e., the method invocations in the scheduler component.

Note that instances can not be created dynamically but only statically, i.e., via declaration in the instances list. However, in the application domain we focus on (embedded software) this is not really a limitation.

Compositions can occur as part of a composite component or at top level (system composition). Figure 3 shows an example of a composition. **Rules** and **contracts** specify constraints over a component or a composition. A rule specifies constraints over one or more components in terms of predicates over component properties. Rules only refer to statically available information like properties, connector or component identifiers and the like. Thus they can be checked statically as they do not refer to information that is only available at runtime. Rules are distinguished in

```
composition{
  instances:
    a1: Actuator;
    a2: Actuator;
    s: Scheduler;
  configuration:
    a1.threshold = 20;
    a2.threshold = 30;
    s.cycleTime = 100; // 100 ms cycle time
  wires:
    sequentialMultiCastMethodCall(
     s.execute() -> a1.execute(), a2.execute());
    // use the standard signal connector
    a1.p1 -> a2.p1;
  rules:
    systemHasScheduler() and
    existsOnlyOneScheduler() and
    allActiveComponentsAreScheduled() and

    // check if the sum of the worst
    // case execution time is lower
    // than the cycle time of the scheduler.
    sumExecutionTimeLTCycleTimeOfScheduler(
     [a1,a2],s
    );
}
```

Figure 3: Example of a Composition

- Consistency Rules: Consistency rules can be attached to a certain component. They check constraints concerning the properties of this particular component such as *”if the component has property X it must also have property Y”*.
- Composition Rules: Composition rules express constraints over a composition. Those constraints range from simple structural constraints to architectural styles, i.e., *”Each component in the system is either passive or it is active and scheduled by a scheduler”*.

Contracts, on the other hand, may refer to dynamic information, i.e., the current values of signals. Thus, they can not always be checked statically but often the checking must be deferred to runtime. Contracts are distinguishes in pre-, post-conditions and invariants. Contracts can only be assigned to ports. Refer to figure 2 for an example of contracts.

The last important entities in our meta model are **packets**. Packets define the structure of the actual data being transferred between components.

Correct Component Composition

The above meta model serves to describe components and their relationships and provides also a basis for code generation, i.e., the creation of code skeletons or the generation of glue code. On top of that it especially focusses on ensuring correctness of component composition. In contrast to traditional approaches we do not only apply syntactic checks or simple semantics checks (like type checking) but also validate constraints over non-functional properties (i.e., structural, runtime requirements) of a component. To do so, we extended the concept of components with the notion of functional and non-functional properties. Consistency and composition rules provide as with a means to reason about the static correctness of a component or composition. We are now ready to refine our definition of static correctness of a composition as follows:

A composition is statically correct if

- it is syntactically correct **and**
- the system complies with the used static type system **and**
- for all component instances of a composition holds: there exists no mandatory property of the component of the respective instance that is not set to a certain value – either at component level or at instance level **and**
- the consistency rules of all involved components are fulfilled **and**
- all composition rules of the respective composition are fulfilled

```
% knowledge base
% components
 component('Actuator').
 component('FunctionBlock').
 component('Scheduler').
% inheritance
 extends('Actuator', 'FunctionBlock').
% ports
 signal('Actuator',['public'], 'in', 'p1').
 handler('Actuator','public','execute', []).
 event('Scheduler','public','execute', []).
%properties
 property('Actuator','mandatory',
    'code','/codebase/Actuator.java').
 property('Actuator','mandatory',
    'executionTime', 30).
 property('Actuator','mandatory',
    'active', 'true' ).
 property('Actuator','mandatory',
    'threshold', 'void').
 property('Scheduler','mandatory',
    'isScheduler', 'true').
 property('s','mandatory',
    'cycleTime', 'void').
```

Figure 4: Prolog knowledge database for an ”Actuator”

To prove the static correctness of a system, we first apply the usual syntactical and type analysis techniques. After that we check if all mandatory properties are set to a valid value. In order to check the fulfillment of consistency and composition rules that are attached to a certain component or a composition we employ first order predicate logic restricted to the form of Horn clauses. We map the knowledge about the entire system or parts of it as well as the rules onto terms in this logic. Structural information, such as component names, component inheritance relationships as well as knowledge about component properties and their values are mapped onto ground terms (or facts) while rules are mapped on predicates and functions. As we only use Horn clauses, this information can easily be translated into Prolog. In our composition environment prototype we use a Prolog engine ([16]) to perform the actual correctness check. Figure 4 shows the Prolog knowledge base derived from the component specification above.

Consistency and composition can as well easily be mapped to Prolog terms. The correctness check can then be reduced to a Prolog goal containing a conjunction of all rules that must be fulfilled. Figure 5 shows, how composition rules can be translated to Prolog.

Contracts, on the other hand, can in general only be checked at runtime. For each contract we generate the appropriate check code for dynamic checking. The code for checking, for example, pre-conditions and invariants for an event handler is then always executed upon method entry of the corresponding method. On method exit, post-condition checks are invoked. This facilitates testing as violations of such constraints are detected during runtime. However, some of those contracts could also be checked statically, if they only used statically available information. The static evaluation of contracts remains, however, subject of further investigations.

Discussion of our approach

The proposed approach allows for powerful static correctness checks at composition time. The applicability of rules ranges from checking simple properties of a component or composition to enforcing architectural styles. Our approach is, to a great extent, language independent. Although we currently only support Java, we plan to incorporate language support for C and C^{++}.

As mentioned before, we do not support dynamic creation of component instances. While this allows for a number of static predictions on the behavior of the system it also limits the class of systems we are able to deal with. However, in our main application domain, this is not a real problem as embedded systems usually prohibit dynamic object creation anyway.

Beyond checking of static properties, one could also consider to extended our approach to dynamic properties using program analysis techniques. However, this would come at the price of losing some language independency at the model level.

4 Related work

Several approaches to the composition of software out of components have been proposed in the literature. An important contribution to this issue stems, without doubt, from the field of software architecture systems [1, 14, 15, 3]. Architecture systems introduce the notion of components, ports, and connectors as first class representations.

Ensuring the correctness of software composition at the construction time has been addressed in literature in a number of different ways. In [4] the authors introduce the notion of *micro-components*. Micro-components represent programming language idioms. Micro-components have assigned contracts and requirements. When being composed those contracts and requirements are statically checked using first order predicate logic. However, non-functional requirements and composition rules are not considered.

[6] focusses on the interactions between (distributed) components. In this paper the authors introduce an semi-automatic approach to interaction protocol checking. The main idea of this approach is to use so-called *program nets*, an subclass of algebraic Petri-nets to model the interaction behavior of components. The program nets of components can then be composed and liveliness and correctness properties can be checked with the known restrictions. Other approaches to interaction compatibility checking can be found in [9] (modelling of dynamic interaction protocols in terms of *regular types*) [17] (regular expressions to define interaction protocol which are used for runtime checking) and others.

Object Constraint Language (OCL) is another approach to put more semantics information into software model. OCL is a precise, textual language for expressing constraints over elements of an UML model like pre- and post-conditions, invariants as well as navigation paths in object graphs. However, until recently there have been few attempts to provide tool support for checking OCL constraints. Approaches to the specification of a precise semantic for OCL in order to enable tool support can be found among others in [12, 7]. Available OCL tools include IBM's free OCL parser [5], the OCL compiler (generates code for evaluating OCL constraints at runtime) [10] and others.

5 Conclusions and Future Work

In this paper we have introduced an approach to *correct-by-construction* software development with components. It is limited to static system construction for embedded systems domain, but introduces handling of non-functional properties and the notion of statically verifiable construction rules.

Our future work will focus on the extension of our realization towards support of C and C^{++} as well as the support of dynamic applications. A challenge we definitely will try to face is the incorporation of contracts, which could be proved during the composition time. This for sure needs data flow analysis and will be language specific.

Another area of interest is interaction protocol checking among components. Protocol checking can be reduced to the language inclusion problem which is only decidable for regular languages. However, there have been approaches to extend interaction protocol checking to special context-free call sequences. We will investigate how far we can adopt protocol checking techniques for our our approach. All those extensions are planned to be supported by tool prototypes in order to demonstrate the relevance and applicability for industrial environments.

```
systemHasScheduler :-
 exists(I,instanceProperty(I,'mandatory','isScheduler', 'true')),
 write_ln('systemHasScheduler passed successfully.').
existsOnlyOneScheduler :-
 existsOneSolution(I, instanceProperty(I,'mandatory','isScheduler', 'true')),
 write_ln('existsOnlyOneScheduler passed successfully.').
allActiveActuatorsAreScheduled :-
 forall(instance(I,'Actuator'),(instance(S,'Scheduler'),exists(C,wire(C, S,'execute',I,'execute')))),
 write_ln('allActiveActuatorsAreScheduled passed successfully.').
sumExecutionTimeLTCycleTimeOfScheduler :-
 findall(T, (instance(I, 'Actuator'), instanceProperty(I,'mandatory','executionTime',T)), Set),
 sumList(Set,Res),
 existsOneSolution(I, instance(I, 'Scheduler')), instanceProperty(I,_,'cycleTime', CycleTime),
 Res =< CycleTime,
 write_ln('sumExecutionTimeLTCycleTimeOfScheduler passed successfully.').
```

Figure 5: Composition Rules in Prolog

6 Acknowledgement

The work presented in this paper is part of the ongoing research project PECOS granted by the European Commission under IST Programm IST-1999-20398.

REFERENCES

[1] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–49, July 1997.

[2] U. Aßmann, T. Genßler, and H. Bär. Meta-programming Greybox Connectors. In Richard Mitchell, Jean Marc Jézéquel, Jan Bosch, Bertrand Meyer, Alan Cameron Wills, and Mark Woodman, editors, *Proceedings of the 33th TOOLS (Europe) conference*, pages 300–311, 2000.

[3] Paul C. Clements. A survey of architecture description languages. In *Int. Workshop on Software Specification and Design*, 1996.

[4] Agustin Cernuda del Rio, Jose Emilio Labra Gayo, and Juan Manuel Cueva Lovelle. Itacio: a component model for verifying software at construction time. http://www.sei.cmu.edu/cbs/cbse2000/papers/06/06.html, 2000.

[5] IBM Application Development. The object constraint language. http://www-4.ibm.com/software/ad/standards/ocl.html, 2001.

[6] T. Genßler and W. Löwe. Correct Composition of Distributed Systems. In *Proceedings of the 31st TOOLS conference*, 1999.

[7] A. Hamie, J. Howse, and S. Kent. Interpreting the Object Constraint Language. In *Proceedings of Asia Pacific Conference in Software Engineering*. IEEE Press, July 1998.

[8] Ilogix. Rhapsody of ilogix. http://www.ilogix.com/fs_prod.htm, 2000.

[9] O. Nierstrasz. Regular types for active objects. In *Proceedings OOPSLA'93*, pages 1 – 15. ACM, 1993.

[10] University of Dresden. The OCL compiler. http://dresden-ocl.sourceforge.net/, 2001.

[11] Rational. Rose for real-time. http://www.rational.com/products/rosert/index.jsp, 2000.

[12] Mark Richters and Martin Gogolla. On formalizing the UML object constraint language OCL. In Tok-Wang Ling, editor, *Proc. 17th Int. Conf. Conceptual Modeling (ER'98)*. Springer, Berlin, LNCS, 1998.

[13] Douglas Schmidt. Tao. http://www.cs.wustl.edu/ Schmidt/TAO.html, 2000.

[14] M. Shaw and D. Garlan. *Software Architecture – Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[15] Mary Shaw, Robert DeLine, D.V. Klein, T.L. Ross, D.M. Young, and G Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, pages 314–335, April 1995.

[16] SWI. Swi prolog. http://www.swi.psy.uva.nl/projects/SWI-Prolog/, 2001.

[17] Jan van den Bos and Chris Laffra. PROCOL – A Parallel Object Language with Protocols. In *Proceedings of the OOPSLA '89 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 95–102, October 1989. Published as ACM SIGPLAN Notices, Proceedings OOPSLA '89, volume 24, number 10.