Applying Experiences with Declarative Codifications of Software Architectures on COD

Position Paper

Roel Wuyts roel.wuyts@iam.unibe.ch
Stéphane Ducasse ducasse@iam.unibe.ch
Gabriela Arévalo arevalo@iam.unibe.ch
Software Composition Group
Institut für Informatik
Universität Bern, Switzerland

Abstract

This position paper presents some preliminary work we made for applying declaractive component oriented design in the context of embedded devices. We quickly describes COMES the model we develop and present how logic rules can be used to describe architectures.

1 Introduction

Software for embedded systems is typically monolithic and platform-dependent. These systems are hard to maintain, upgrade and customise, 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 [Szy98].

The goal of the PECOS (PErvasive COmponent Systems) project (Esprit project XXX) is to find solutions for component oriented development (COD). In this context we are developping Comes a Component Meta-Model for Embedded Systems. In Comes, components are encapsulations of behavior (implemented in Smalltalk, C++, C, assembly, FSA, ...). They have interfaces that consist of properties and ports, and have consistency rules that express structural integration internal to the component (for example, to check dependencies between properties). Components are connected by wiring their ports with connectors and can be wrapped in composite components. Consistency rules of the composite component can reason about the properties of the composite, but also on the connectors and the properties of the sub components.

The outline of the paper is the following: first we present our previous work in which we declaratively codified software architectures (SA) as logic programs, then

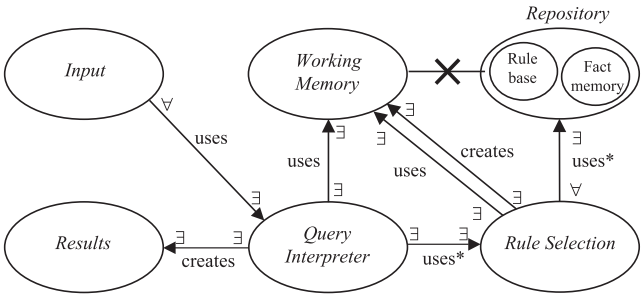

Figure 1: A software architecture for a rule-based system.

we show how such an approach is applicable in the context of component design. We then present the component model we are developping in the context of embedded devices and conclude.

1.1 Declarative Software Architectures

Our previous research in Software Architecture (SA) focussed on ways to express a SA on a high-level of abstraction while providing support to automatically verify whether source code conforms to the SA [MWD99]. Therefore we codified the SA using *software classifications* and relations between these classifications. Figure 1 shows the software architecture of a rule-based interpreter [SG96]. The circles are *software classifications*. They are connected by connectors that have *types* (such as *uses* and *creates*) and *cardinalities* (which we will not explain here as it is not relevant to components). This software architecture is expressed by logic facts and can be checked against the source code. In more detail, the model is made up from:

- *software classifications*: wrap the implementation elements, and can be seen as bags of source code elements (such as classes and methods). The items in the classification can be enumerated one by one, but are typically described using a logic program that reasons about source code. This makes it possible to, for example, group all classes in a certain hierarchy or participating in a composite design pattern.
- *relationships between software classifications* : the goal of the relationships is to connect classifications with high-level intuitive connectors such as *creates*, *accesses*, ... These connectors are actually logic programs that are mapped internally to more primitive dependencies between source code artefacts or ports [Men00].
- the *software architectures* are expressed as logic programs that enumerate specific software classifications, and the relationships between them.

- *conformance checking* of SA against the source code is done by a logic program. It checks whether the relationships hold between the software classifications. Different conformance checkers can be used of course, to implement different checks.

- *subclassifications.* Software classifications can be SA themselves instead of bags of source code items. As a result, SA can be nested in one another.

- *architectural patterns*: last but not least we defined architectural patterns, as a SA that describes a template instead of a concrete enumeration of software classifications and relationships. The definition is similar to a regular SA, but it contains logic variables that can be instantiated when the template is instantiated. Using the patterns we expressed a *rule based architecture* (a domain specific architecture for rule-based systems) and the *pipe-and-filter* architecture.

As a recapitulation, we connect entities (the software classifications) with connectors to define SA. A logic programming language is used to describe the entities, the meaning of a connector and the SA. Because of this, we automatically can express architectural patterns.

1.2 Pecos project

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. While focusing on architectural issues, it touches upon the whole software development cycle and addresses the major technological deficiencies of state-of-art component technology with respect to embedded systems by developing:

- a Component Model for embedded system components addressing behaviour specification and non-functional properties and constraints

- an interactive Composition Environment for composing embedded applications from components, validating functional (e.g., interfaces) and non-functional compositional constraints (e.g. power-consumption, code size), generating the application executable for the embedded device and monitoring their execution.

By providing a coherent approach and methodology for programming of component based embedded systems PECOS enables an efficient and competitive embedded system development.

2 Applying our declarative SA approach on COD

Before we introduce the meta-model that we use to support COD, we first want to introduce the parts we can reuse from SA, and the ones we cannot. The approach described in section 1.1 proved successful within its context. We were pleased with the logic programming approach, as it is very well suited to express relationships between

entities (both between source code entities as between software classifications), and it makes easy to define both concrete and template architectures just by using logic variables instead of logic constants.

In general, we can draw the following parallels between our approach to software architectures and the component model we are interested in:

- components can be seen as software classifications. The software classification bundles a number of implementation concepts in one place (as a result of a logic query or an explicit enumeration). The rest of the model works with software classifications (connecting them), and does not see the implementation elements inside the software classification. A component is nothing more than that: it provides an interface that can be queried, and allows you to connect it to another one.

- both models need connectors to specify relationships between components. Components need to be composed with other components to build-up a working system. The connectors in a composition specify what parts of a component are connected to what other parts of components. This is the same for software architectures or for components.

- the connectors are used to check whether relationships between components hold or not. Connectors serve at least two purposes. First of all they describe what is connected (as a form of documentation). Second they are used later on to check whether the connections are possible (like an advanced type system). This holds for both software architectures (where we checked whether a certain relation held between software classifications) and component models (to check whether two components can indeed be composed or not).

- leaf components and composite components can be used transparently, just like software architectures can be used as software classifications. In our software architecture, we could use software architectures as software classifications, thereby composing software architectures. A same concept is useful in the component-oriented programing, where it allows one to build components from leaf components or other compositions transparently.

- we would also like to have concepts such as templates and styles in the context of COD. Our software architecture was described as a logic predicate enumerating its components and their connections. An architectural template was the same as a software architecture, but used logic variables for the components or connectors that needed to be filled in later. We want to have the same mechanism in our component model, as it is a straightforward mechanism to get component templates that prescribe compositions of components that can be instantiated later on.

There are of course also differences between the context of software architectures and components, for which we have to take care in our model:

1. in our software architecture model, the interface of a software classification was part of the model. While it could be extended, this did not occur frequently. In component-oriented programming, however, we need a good concept of interface because there is a possibility that it will change more often.

2. a component has more properties than a software classification. A software classification is basically nothing but a bag of elements. Connectors in our software architecture model used a very narrow interface to interrogate the elements in the bag, which is more a white-box approach. However, the components are black-box, meaning that all the information usable by connectors has to be available from the interface.

The next section describes the Comes model (Component Meta-Model for Embedded Systems) we developed. It includes the major features from the architectural model, but makes changes to accomodate specific component issues as outlined above.

3 COMES: Component Meta-Model for Embedded Systems

This section describes the core of our component model that supports the definition and composition of components for embedded devices. We present the meta-model developed following a prototyping approach and using the experience from SA. We first give an overview of the complete model, and then we discuss all the main parts in more detail and with some examples.

3.1 COMES Overview

COMES meta-model is simple and composed by the following entities:

Component is the main entity in this model. It basically holds properties (to describe information like non-functional properties) and can be combined with other components. We distinguish between *leaf components* (that have no subcomponents) and *composite components* (that are like leaf components but also contain some connected subcomponents).

Property is the entity which lets us characterize the components. Its features are a tag and a value. The characteristics and the constraints of a component is given by a set of these pairs. The tag is the identifier of the property. Thus, we are not constrained to a fixed set of properties, and new ones can be defined according to new requirements.

Port is the place where components can be connected to other components (through connectors).

Consistency rule. Certain dependencies exist between the properties of a component. We call the act of checking if a component is valid regarding its properties: checking the consistency of a component.

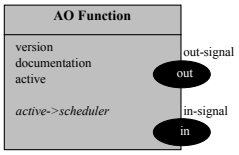

*Figure 2: A simple COMES component displayed, with three properties (*comment, version, active*), two ports (*in *and* out*), and a consistency rule (*active->scheduler*).*

Connector is a link between ports, and is used to connect components when building a composite component.

3.2 Leaf Components and their constituents

In this section we introduce *properties*, *ports*, *consistency rules* and *leaf components*. We use the *AO Function component*, shown graphically in Figure 2, as an example to make the concepts more concrete.

Property. A property is a key/value pair. The key contains the name of the property, while the value holds the actual contents of the property. Properties are used to model all kinds of static information (such as the version of the component or comment) or values of non-functional properties (to specify the memory needed for the component etc.).

For example, the AO Function component shown in Figure 2 has three properties (the values are not shown in the figure). The *version* property has a value of *1.0*, the *comment* holds a string with some textual explanation and *active* is set to *false*, since this component does not run in its own thread.

Port. A port is a quadruple consisting of a *name*, a *type* (like *event* or *signal*), an *input/output (i/o) specification* and *arguments*.

The AO Function component has two ports: an out-port and an in-port of type *signal*. As another example, suppose that we have a component A that needs to be scheduled by a *scheduler component* S, and that S wants to have the possibility to send an initialization request to A. Then we can give A a port *initialize*, of type *event*, with an i/o specification of *in* and without arguments. Likewise, S should have an out-port *initialize* of type *event* without any arguments. These ports can then be connected by a connector (as is explained later on).

Consistency rule. A consistency rule is used to check structural consistency of a component, and is expressed as a first-order logic expression over the properties of a component. It is mostly used to describe dependencies between properties (a certain property has to exist whenever there is some other property).

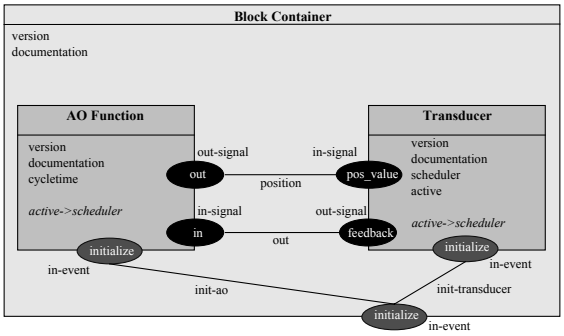

Figure 3: A composite component Block Container *that holds on to two subcomponents (*AO Function *and* Transducer*).*

In the example of the AO Function, we show one rule that is called *active->scheduler*. This rule expresses that if the component has an *active* property with value *true*, that than this component needs to have a cycletime property. In COMES this is expressed as a logic query (expressed in the reflective logic language SOUL) that is evaluated in the context of the component.

```
ifthen(componentProperty(?c, active([true])),
       componentProperty(?c, scheduler([true])))
```

Leaf Component. a leaf component consists of a *name*, a *set of properties*, a *set of ports* and a *set of consistency rules*. Hence it is basically a building block that can be connected to other building blocks while holding on to some local information. Leaf components are described in Smalltalk or logically (as we see later on when we see an example of a logic description).

3.3 Composite Components

In this section we see how components are composed into *composite components*. We introduce a composite component called *Block Container* that has its own properties and rules, and holds on to two subcomponents (the *AO Function* component that we introduced in the previous section and the *Transducer* component. This composite component is shown in Figure 3.

Connector. A connector connects ports, and consists of a *name*, a set of *out-ports*, a set of *in-ports* and a *type*. A connector is oriented (from the out ports to the in-ports). We distinguish two kinds of connectors, that differ in the i/o specification of the ports they connect:

- *passing connectors* connect out-ports to in-ports. All the ports have to be of the same type (*signal* or *event*), all out-ports have an i/o specification of *out*, and all in-ports have an i/o specification of *in*. They are typically used to model data flow between components. For example, Figure 3 shows two passing connectors between signals: the *position* and *out* connectors.
- *interface connectors* connect ports of the same i/o specification and type. They are typically used to 'publish' ports of subcomponents as ports of the composite component. The Block Container component for example has the *init-ao* and *init-transducer* connectors.

Composite component. A composite component is a leaf component that also holds on to a set of child components and a set of connectors. Whether the components are well-connected is expressed by consistency rules.

3.4 Checking a model

In this section we discuss how a model is checked once it is modeled. We also give some more examples of consistency rules.

Once an embedded system is modeled with Comes, the composition of the used components needs to be checked. This comes down to performing two kinds of checks:

1. checking the internal consistency of components
2. checking the structural integrity of the composition

When both these checks are done and succeed, the model can be used to generate template code. This falls outside the scope of this paper.

Checking the internal consistency of components is done by firing the consistency rules that are defined on components. Whenever one of such rules fails, this means that some internal property of the component is not followed. For example, in the AO Function component, there is one consistency rule used (*active->scheduler*). Therefore, when the system is checked this rule will be fired. When it is violated (meaning that the AO Function component has a property *active* set to true but no property *scheduler* or a property *scheduler* that is false), the user will be notified.

The *active->scheduler* rule is but a simple example. We also use consistency rules to express relations between non-functional requirements that are modeled in the properties. For example, when all components have a property *ramsize* with a value that indicates the amount of ram they need, we can fit any composite component with a consistency rule that calculates the combined ramsize, and makes sure it does not pass a certain value set in the *ramsize-max* property of the composite component:

```
subcomponents(?c, ?subList),
findall(?ramValue,
        and( member(?subC, ?subList),
             componentProperty(?subC, ramsize(?ramValue))),
        ?ramValueList),
sum(?ramValueList, ?calculatedSize),
```

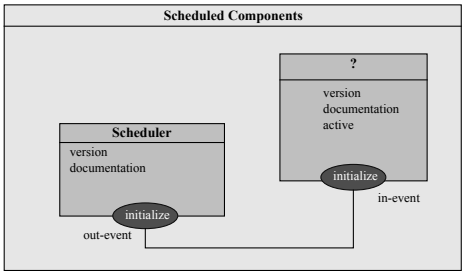

Figure 4: A template component defining a scheduler component and another component that needs to be scheduled.

componentProperty(?c, ramsize-max(?maxSize)),
smallerThan(?calculatedSize, ?maxSize).

Checking the structural integrity of the composition means that we have to check that the right connectors are used to connect the right kind of ports. This is also specified as a logic rule, that implements the behavior for passing or interface connectors. These rules can of course be changed, meaning that what it means to connect components is changed.

3.5 Template Components and Styles

To facilitate building components, we would like to have template components, that contain predefined properties, ports, subcomponents or connectors. For example, we would like every component to have at least the *version* and *documentation* properties. For this we use template components. Template components are components that have 'holes' in them that need to be filled when they are instantiated. Like in our approach to describe software architectures, we want to specify them as logic constructs containing logic variables that can be filled in at instantiation time. The result of this instantiation is then a component.

For example, consider the following template displayed in figure 4. It defines a template for a composite component *Scheduled Components* that consists of two subcomponents: a *Scheduler* and an unnamed component. The scheduler component is completely fixed: it contains no variables to be filled in. The component it schedules, however, is mostly unspecified. We only require it to have at least an in-event called *initialize*, and some properties. In its logic form, this template component is described as follows (note that we do not discuss the implementation of the used logic predicates, as this would lead us too far):

compositeComponentNamed(?c, 'Scheduled Components'),
subcomponents(?c, <?scheduler, ?t>),
componentNamed(?scheduler, 'Scheduler'),

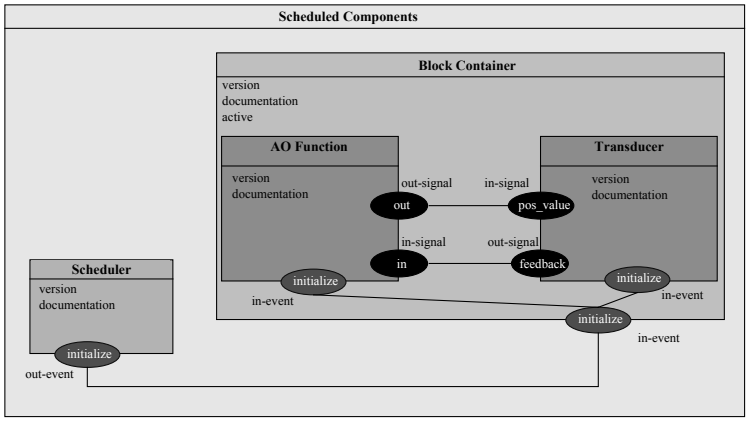

Figure 5: The template component instantiated with the Block Container component.

componentProperty(?t, property(version, 1.0)),
componentProperty(?t, property(documentation, 'demo')),
componentProperty(?t, property(active, false)),
componentort(?t, port(initialize, event, in, <>))

Then we can instantiate this template component by filling in the component to be scheduled with our Block Container component. This results in the component as shown in figure 5. Note that is also shows that a composite component (the Block Container) can be used in the place of another component (the component that is scheduled).

3.6 The Comes Tools

We have implemented this model in the object-oriented programming language Smalltalk. It contains a GUI interface to compose components with 'drag and drop' (although the connections are not shown graphically). At the moment of writing we do not yet have a GUI facilitate for template components; these have to written directly in the logic language. However, this will be adressed in the future.

The logic programming language we use is called SOUL (the Smalltalk Open Unification Language) [Wuy98, Wuy01]. SOUL is actually a reflective logic programming language. It allows us to directly reason over Smalltalk objects, this combination combines the best of both worlds: polymorphism and inheritance to express components and logic programming to express rules and relations between these components. In the future we hope to take more advantage of this by complementing the current static checks on properties etc. with dynamic checks.

4 Conclusion

We presented COMES a simple component model that we are developing in the context of the PECOS IST Project. The contributions of this paper are:

- Introduction of the Comes model for components for embedded systems. Special about this model is that it uses a logic programming language to describe component consistency rules and the meaning of connectors.
- the behavior of connectors is a form of extended type-checking, that can take properties of components into account. These properties describe static or non-functional information, that can thus be used for checking and connecting.
- the Comes implementation uses a logic programming language that reasons directly over components that are Smalltalk objects. This provides a best-of-both-worlds approach: Smalltalk is very good in wrapping things in components and the logic programming language is very good in expressing rules over and relations between components. In a pure Smaltalk approach, expressing the rules and relations would become harder and in a pure Prolog approach wrapping of components would be harder.

While explored in the context of COD, we feel that these contributions are also applicable on general COD. We relate to the workshop theme as we are building a meta-model for components in a specific context (embedded systems). Components are black-box encapsulations of some behavior (implemented in C++ or Java), and have an interface. They are connected by connectors. We try to apply our exerience in SA in the context of COD. However, these issues are also applicable to general COD.

References

[Men00] K. Mens. *Automating Architectural Conformance Checking by means of Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, 2000.

[MWD99] Kim Mens, Roel Wuyts, and Theo D'Hondt. Declaratively codifying software architectures using virtual software classifications. In *Proceedings of TOOLS-Europe 99*, pages 33–45, June 1999.

[SG96] M. Shaw and D. Garlan. *Software Architecture — Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[Szy98] Clemens A. Szyperski. *Component Software*. Addison-Wesley, 1998.

[Wuy98] Roel Wuyts. Declarative reasoning about the structure object-oriented systems. In *Proceedings of the TOOLS USA '98 Conference*, pages 112–124. IEEE Computer Society Press, 1998.

[Wuy01] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.