Non-Functional Requirements in a Component Model for Embedded Systems

Position Paper

Roel Wuyts
Software Composition Group
Institut für Informatik
Universität Bern, Switzerland
roel.wuyts@iam.unibe.ch

Stéphane Ducasse
Software Composition Group
Institut für Informatik
Universität Bern, Switzerland
ducasse@iam.unibe.ch

ABSTRACT
In this paper we describe an interesting context to study formal methods for component systems: embedded devices. The context of embedded devices is highly constrained by the physical requirements the devices have to adhere to. As a result, component models for embedded devices are not general purpose but geared towards these constrained contexts. In this paper we give the concrete setting of the Pecos project (a project with as goal component engineering for embedded devices). We describe the Pecos component model, and show possibilities where we think formal verification could be useful. We would like to use this as a very concrete example to discuss formal verification techniques.

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 [7].

The goal of the PECOS (PErvasive COmponent Systems) project (a European Esprit project) is to find solutions for component oriented development (COD) for embedded systems . In this context we are developping Comes (a general Component Meta-Model) and the Pecos Model (a specialization of Comes targeted towards embedded systems in the context of the project).

In Comes, components are black-box encapsulations of behavior. They have interfaces that consist of *properties* and *ports*, can contain subcomponents, and have consistency rules that express structural integration internal to the component (for example, to check dependencies between properties). The ports of components are connected by explicit *connectors*. 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 Pecos Model is a specialization of Comes to explicitly support components for embedded devices in the context of the Pecos project. Interesting is that this puts a lot of extra constraints on the component model. We firmly believe that this will allow us to use formal (mathematical) techniques to verify non-functional requirements of the modeled components. More specifically, we want support regarding *timing and scheduling* and *memory consumption*.

For this workshop we see ourselves as the providers of an interesting problem and problem context. We believe that the extra constraints imposed by the context of our component model make it a good example to use and assess the functionality of formal techniques. Coming from a practical, less formal discipline of software engineering and programming language design, we want to discuss with the more mathematically enclined researchers on how to come to formal support for specifying and checking components.

In the rest of the paper we introduce the specific problem context of embedded systems in more detail. Then we show the current version of the component model. Finally we enumerate the places where we think that formal techniques could help us, and discuss techniques we think are of interest to us.

2. THE EMBEDDED SYSTEMS CONTEXT
A massive shift is going from desktop applications to embedded systems, where intelligent devices taking over roles that are currently done in desktop applications. Moreover, the capabilities of embedded devices augment rapidly, and their responsibilities increase likewise. Distributed embedded devices (intelligent field devices, smart sensors) not only acquire but also pre-process data and run more and more sophisticated application programs (control functions, self-diagnostics, etc.).

The drawback of this evolution is that the software needs to follow. Here the story is less positive: the software engineering techniques that are typically employed are lacking far behind software engineering techniques for mainstream applications. Currently software for embedded devices is written in assembly or C, in a monolithic fashion, with a typical development time of two to three years. The reasons for this are two-fold. The first reason is the specific context of embedded devices (with all the constraints of power comsumption and simple hardware as a result of this). The second reason is that, up until a couple of years ago, the market for embedded devices was relatively small, and was thus neglected by the big players from desktop applications. For example, operating systems or development environments are hard to find for embedded systems.

The goal of the PECOS (PErvasive COmponent Systems) project is to apply solutions for component oriented development (COD) in the context of embedded systems. As in desktop applications, the overall goal is to have more reuse, higher quality and reduced development time. Key factor in the project is the component model to support components for embedded systems. Before we have a look at this model, we first introduce the Pecos component development process, and field devices, the embedded systems the Pecos model should support.

2.1 Pecos Process

Part of the solution of the Pecos project is a component development process. In this section we give a quick overview of this process, as this will help to introduce some choices made in the Pecos Component Model. The process consists of two main phases: the *component construction phase* and the *field device assembly* phase.

The component construction phase defines what is needed to develop a single component (that possibly contains subcomponents), instrument it (to provide information about runtime aspects of the component), and put it in the component repository. It specifies the following workflow:

- the component is created. This means defining the basic properties and the interface of the component.
- the subcomponents are filled in. If the component has subcomponents, then these subcomponents need to be selected from the repository and added to the componenent. They also need to be connected with each other.
- the component is checked. In this phase, a structural check is performed to make sure that everything is specified according to the model, and that the given information follows the rules in the model. For example, when the model specifies that a component should have a name, then this is checked at this moment. Also, when type information needs to be given it is checked that the given types exist. Or, if there are subcomponents, their connections are checked.
- generating skeleton code. When the check succeeds, meaning that the component's structure is verified, skeleton code can be generated.
- filling in the skeleton: the skeleton code has to be extended into a full working implementation.
- instrumenting the component: the component is then ready to be instrumented. In this phase it is deployed in a standard environment so that certain runtime information can be gathered. What information depends on the model. Since in the Pecos model we want to check scheduling information and memory consumption, basic figures need to be extracted. Note that we need the instrumentation because we see components as black-box abstractions where we have no idea about their internals. If this constraint is lifted, the instrumentation phase could be made simpler or even omitted. We discuss this in more detail when we discuss the non-functional checks.
- the instrumented component is then added to the component repository.

A second activity is to assemble components into field devices (the actual embedded systems that need to be modeled in the context of Pecos). This activity consists of the following steps:

- select a template for the field device that needs to be created
- select the components that need to be filled in to instantiate the field device
- connect the components
- perform structural checks on the instantiated field device
- perform non-functional checks using the information provided by the components. For example, make sure that the total power consumption of the chosen components does not exceed the limit of the Field Device, or that a schedule can be found to schedule the components.
- generate the code for the field device
- deploy the component on the actual hardware

In the next section we have a look at Field Devices, the actuall embedded systems used in the project. Then we introduce the model to support the specification and checking of these devices.

2.2 Field Devices

Field devices are embedded reactive systems. A field device can analyze temperature, pressure, and flow, and control some actuators, positioners of valves or other motors. Field devices impose certain specific physical constraints For example a TZID (a pneumatic positioner) works under the following very hard constraint: the available power is only 100 mW for the whole device. This limits severely the available CPU and memory resources. The TZID uses a 16 bit micro-controller with 256k ROM and 20k RAM (on-chip), and communicates using fieldbus communication stacks (an interoperability standard for communication between field devices). The device has a static software configuration, i.e., the firmware is updated/replaced completely, and there is no dynamic loadable functionality.

As a result from the physical constraints (especially the very harsh power consumption requirements), the runtime environment and the software are subject to the following constraints:

- One processor: all the components composing a field device are running on a single processor, that is very slow when compared to mainstream processors.
- One monolithic piece of code: after assembling the different components that compose a field device, the software for the field device forms one single piece that is deployed.
- No dynamic change: At run-time (after the field device is initialized) there is no memory allocation, nor dynamic reconfiguration.
- Single language per application: a component is created in a single language like C or C++.
- Multi-threading: field device components can be running on different threads. The scheduling is carried out by either the OS or by an explicit scheduler. However, most of the components are passive and scheduled by a central scheduler. Components that are active (that have their own thread) are typically the ones close to the hardware. They are responsible for regularly reading values from this hardware, such as the current motor position or speed.
- Components communicate by sharing data contained in a blackboard-like structure. Components read and write data they want to communicate to this central memory location.
- Some components are described by state automata. Some components have state, others are stateless because they are only representing algorithms.
- Components only offer interfaces in terms of in/out ports. The component state automata definition, or other behavorial descriptions, are not available. This is a very hard requirement, as this means that a lot of existing formal verification techniques are not usable.
- A field device architecture is fixed. It is composed by an *Analog component* controlling the overall workings of the device, a *Transducer component* that interfaces to the hardware, a *HMI component* for the Human-Machine interaction and an *EEPROM component* to store data in non-volatile memory.

3. THE PECOS COMPONENT MODEL

The Pecos Component Model is the foundation of the Pecos project. Its goal is to allow to specify and check components and Field Devices, given the constraints given above. In this section we iterate over the requirements for the model, introduce its main aspects. In the next section we then look at how formal techniques could be applied in this context.

3.1 Requirements

The goal of the Pecos Component Model is to be able to model and check a field device. More specifically, it has to allow:

- to specify individual components (that can contain subcomponents);
- to connect components;
- to assemble components into Field Devices;
- to check the structure and well-formedness of component compositions and Field Devices;
- to check non-functional requirements of Field Devices. More specifically, *timing and scheduling* of components, and their *memory consumption*;

3.2 Model Overview

In the constraints imposed by the context of embedded systems on field devices we already saw that Field Devices follow a blackboard-like achitecture. Hence, there is a central block of memory (called the *Object Manager*, or OM for short) that holds all the values that need to be passed between components in a field device. The OM is filled when the field device is initializad. At runtime, its structure does not change (as there is no allocation at runtime after the initialization). Components that need to share data to do by writing and reading from the OM.

Normally, when components would all be running in their own thread and hence in parallel, locking and synchronization of the OM would certainly be needed. However, in the specific context of a field device such a solution,(typical solution for desktop applications), is not possible. The reason is that it's too expensive in both processing power and memory consumption, and that OS facilities to support locking and synchronization are not always available or very costly. Field devices solve the problem by providing one central scheduler that sequentially schedules all components. Hence, at any moment in time, only one component has access to the OM and thus no locking is needed. Of course, this introduces other problems as well, that we will discuss in detail later on when we talk about supporting (checking) non-functional requirements.

The Pecos model builds on our experiences with supporting Software Architectures using logic programming languages [6]. The main constituents are components, ports and connectors:

- *component*: a Pecos component has a name, contains information regarding scheduling and memory consumption (see further), has a list of data ports and possibly has a list of subcomponents and connectors for these subcomponents;
- *data ports*: a data port indicates that the component provides or needs data for other components. It contains a type (of the data that will be passed, such as Float), a direction (in, out or inout),

- *connectors* connect data ports of components, and hence model a data dependency between two ports. Connectors contain the names of the component and the ports they connect

Besides this structural information, we also check some Pecos specific constraints, such as type and range information on ports. Table 1 lists all the structural checks that can be performed.

Besides the components and connectors, the Pecos model also offers a Field Device template. This is a template component that has to be instantiated with 4 concrete components. The Field Device component specifies the structure and the behaviour of a field device in such a way that its structure and semantics can be checked, and that code can be generated from it. To instantiate the field devoce, four components and their connections that have to be specified:

- *Human Machine Interface Component*: a field-device can be equipped with displays and other devices so that users can inspect or modify the behaviour from the device itself
- *Non-volatile memory Component*: the state of the component needs to be written to certain kinds of memory
- *Input-Output-Controller Component*: the data from the device component typically consist of raw values that are immediately related to the hardware contained. The function of this component is to provide an interface to the other non-hardware related components that is not hardware specific. For example, it can scale raw data from the hardware so that the display can show the value of a temperature controller in degrees Celsius.
- *Device Component*: all components that deal with the hardware are encapsulated by this component.

The result is a Field Device that can be checked for well-formedness (making sure that everytthing conforms to the structural rules) and for non-functional requirements. These last checks are the topic of the following section.

4. CHECKING OF NON-FUNCTIONAL REQUIREMENTS

The previous sections described the context of field devices and the Pecos component model to model components for field devices. However, it didn't give much information about the checking of non-functional requirements. In this section we describe what we would like to support, and what we are currently doing. We also give information about related formal work that we think could be useful (but that we not use at the moment of writing).

In the Pecos project we want to support two issues, that we have already touched upon throughout the paper: scheduling of components and memory consumption. We explain these two issues in more detail, and then have a look at opportunities we see for formal verification.

4.1 Component Scheduling

We already explained that in field devices we do not want to use regular locking of data, but instead want to schedule the components sequentially such that this is not needed. Hence, a very important aspect that needs to be checked when a field device component is instantiated is the scheduler.

More specifically, we currently instrument every individual component with information regarding its *execution time* (the time it takes to execute its behaviour once) and with information about its *cycletime*(the number of times it needs to be executed in one scheduler cycle). Using this information (combined with the information of the data dependency provided by the connectors) we are now investigating whether it is possible to derive or check a scheduler. The hardest thing to solve is that we currently identified three kinds of components: passive components, active components and event components. Passive components are straightforward to handle: they just need to be scheduled by the scheduler such that their execution and cycling information is met. Active components are more difficult. The reason is that they have their own thread that is running inside of the component. This thread is typically used to read-out values directly from hardware, such as the current speed of a motor. In the current implementation used in field devices, these values write to internal fields in the component, and when the component is scheduled the values in the internal fields are copied to the OM. Hence, active components are scheduled and handled exactly as passive components, even though they have their own thread. We are currently debating whether this is a good solution, and what would be alternatives. Event components pose the same problems as active components. They do not have their own thread, but act as event sinks that have to capture and react to events sent by certain pieces of hardware. Just as with active components, they capture an event, wait until they are scheduled by the scheduler and then handle the event.

At the moment of writing we are still investigating possible solutions to check and generate the scheduler, with probably the most interesting option to express all the scheduler constraints using Constraint Logic Programming over Real Numbers (CLP(R)), and calculate possible schedules. By the time of the workshop we will have a concrete solution for this problem, as this is currently under full development.

4.2 Memory consumption

Due to the minimal memory available in field devices, the memory occupied by a component is a crucial information. The model should support the computation of the component size and checks for component substituability.

To perform the checks, every component is instrumented with the size it needs for its code and for its data. This should then be summed and combined with the information from the blackboard.

4.3 Possibilities for Formal Verification

We are thinking to lift the constraint that components are completely black-box, and adding and using state charts as a way to describe the behavior of components. When we

Table 1: Structural Checks in the Pecos Component Model

Port	The type of the property can only be one in a fixed set (Float, Tfloat, Tscale, ...);
	The direction should be in, out or inout;
	The location of a port has to be 'static', 'dynamic', or 'nv';
	The minimum in the range is smaller than the maximum.
Component	The State can only be active, passive or 'event';
	All the numbers regarding timing and code sizes should be positive or 0..
Connector	Connectors can only connect out and in; ports;
	The types of the ports should be compatible;
	The ranges of ports should be compatible;

do this, we can think of using synchronous languages such as Esterel [1], Argo/Argonaute [5], Lustre [3], CRP [2] and combined approaches [4].

Especially Esterel seems a natural candidate to use in the context of embedded systems. It is a synchronous and imperative concurrent language dedicated to control-dominated reactive programs which are found in real-time process control, embedded systems, supervision of complex systems, communication protocols and HMI. In Esterel, programs are abstractions that manipulate input signals and generate output signals. Once programs are expressed in Esterel they can be formally proved (i.e., non-reachability of state, timing constraints), compiled to C in a compact form, and simulated. In the context of Pecos, Esterel seems particularly interesting because the size generated is suitable for field devices and, more important, timing issues and memory consumption can be verified:

- it allows the verification that given an input, the output of a program is comprised in a certain amount of cycles of the input. This means that component substitution could be verified.
- it allows different code generation schemas. The first one is boolean generation. By counting the number of instructions the exact size of a component and its exact execution time can be counted. The second is condition-based and can provide maximum execution time for a component.

Another possibility would be to look at the formalism of timed state automata, to take timing information into account.

5. CONCLUSION

In this paper we describe the context of embedded systems, for which we made a component model to specify and check Field Devices (a particular kind of embedded system). Due to the physical constraints imposed on embedded systems, a component model for embedded devices has very specific constraints: no runtime allocation, no locking or synchronization, and a simple scheduler. We describe the Pecos Component Model that we are developing, and that allows to specify and check Field Devices and their components. The most interesting aspect of the model is that we want to check certain non-functional requirements before the software for the field device is deployed in the hardware. This is still under ful development. We showed the current status of the checks, and where we suspect that formal techniques could be welcomed. In the workshop we want to discuss with people from the formal community, using our context as a test case.

6. REFERENCES

[1] G. Berry. *The foundations of Esterel.* MIT Press, 2000. Editors: G. Plotkin, C. Stirling and M. Tofte.

[2] G. Berry, S. Ramesh, and R. K. Shyamasundar. Communicating reactive processes. In ACM, editor, *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium, Charleston, South Carolina, January 10–13, 1993*, pages 85–98. ACM Press, 1993.

[3] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. In *Proceedings of the IEEE*, September 1991.

[4] F. M. M. Jourdan, F. Lagnier and P. Raymond. A multiparadigm language for reactive systems. In *Proceedings of the IEEE Internal Conference on Computer Languages*, 1994.

[5] F. Maraninchi. The argos language: Graphical representation of automata and description of reactive systems. In *Proceedings of the IEEE Internal Conference on Visual Languages*, 1991.

[6] K. Mens, R. Wuyts, and T. D'Hondt. Declaratively codifying software architectures using virtual software classifications. In *Proceedings of TOOLS-Europe 99*, pages 33–45, June 1999.

[7] C. A. Szyperski. *Component Software.* Addison-Wesley, 1998.