A data-centric approach to composing embedded, real-time software components*

Roel Wuyts[a],Stéphane Ducasse[a], Oscar Nierstrasz[a],

[a]{wuyts} {ducasse} {nierstrasz}@iam.unibe.ch
Software Composition Group
Institut für Informatik
Universität Bern, Switzerland

Abstract

Software for embedded systems must cope with a variety of stringent constraints, such as real-time requirements, small memory footprints, and low power consumption. It is usually implemented using low-level programming languages, and as a result has not benefitted from component-based software development techniques. This paper describes a *data-centric component model* for embedded devices that (i) minimizes the number of concurrent tasks needed to implement the system, (ii) allows one to verify whether components meet their deadlines by applying Rate Monotonic Analysis (RMA), and (iii) can generate and verify schedules using Constraint Logic Programming (CLP). This model forms the foundation for a suite of tools for specifying, composing, verifying and deploying embedded software components developed in the context of the PECOS project.

1. Introduction

Component-based software development (CBSD) is quickly becoming a standard approach to develop mainstream software systems. Most component models and systems, such as (D)COM, .Net, JavaBeans [17] and CCM [24], however, target the development of mainstream desktop applications with abundant hardware resources. None of these existing models explicitly addresses the constraints imposed by small embedded devices. Although CBSD would bring numerous advantages to the embedded systems world, such as shorter development times, and the ability to secure investments through reuse of existing components, in practice, standard component models are too heavyweight to be applied to embedded systems development.

In this paper we outline an approach to CBSD for embedded systems developed in the context of the European IST project PECOS[2]. PECOS enables CBSD for small embedded systems by providing an environment that supports the *specification*, *composition*, *configuration checking*, and *deployment* of embedded systems built from software components. The PECOS approach was applied in the context of *field devices*, small embedded devices developed by the prime contractor of the project, ABB. The central element of the approach is a data-centric component model that makes it possible to check non-functional requirements of component compositions.

The PECOS component model is unusual in that PECOS components have very simple interfaces consisting solely of data ports. Components are either *active*, *event* or *passive*, and interaction occurs only where data ports are connected. This design choice makes it particularly easy to reason about synchronization and timing aspects of components. First of all, it enables timing analysis of component compositions using rate monotonic

*In *Journal of Systems and Software — Special Issue on Automated Component-Based Software Engineering*, vol. 74, no. 1, 2005, pp. 25-34.

[2] *Pervasive Component Systems*, IST-1999-20398. ABB Corporate Research Centre, Ladenburg (DE), Research Centre for Information Technologies (FZI), Karlsruhe (DE), the Software Composition Group (SCG) at the University of Bern (CH), and Object Technology International (OTI), Amstelveen (NL).

Figure 1. Pneumatic positioner (TZID) field device.

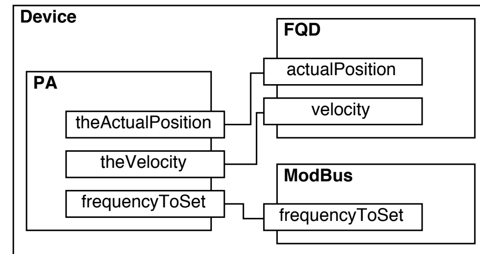

Figure 2. A canonical field device containing three subcomponents: Device and ModBus (active), PA (passive), and FQD (event).

analysis, a technique for verifying that a set of tasks can meet their deadlines. Second, it enables schedule verification of component compositions using a constraint logic programming approach.

The next section describes the context of field devices, their specific constraints and the solutions proposed by the PECOS project. Then Section 3 presents the PECOS component model. Sections 4 and 5 then address the issues of the timing verification and schedule verification of compositions. Section 6 discusses the model and some related work. We conclude with a few remarks on the current status of this work.

2. Field Devices: Context and Challenges

Rather than attempt to develop a CBSD approach for the embedded systems domain in general, the PECOS project focussed on the specific challenges and demands of *field devices*. In this section we introduce field devices, consider the problems they pose for CBSD, and outline how PECOS tackled these problems.

2.1. Field Devices

Field devices are small reactive, embedded systems that make use of sensors to continuously gather data, such as temperature, pressure or rate of flow. An example of a typical field device is the TZID pneumatic positioner (Figure 1), used to control pneumatic actuators attached to valves. Such devices have the following characteristics:

- The available power is only 100 mW for the whole device. This limits the choice of suitable CPUs. The software architecture is driven by the *fieldbus architecture*, a common standard for linking devices from different manufacturers. Fieldbus stack implementations from third party suppliers are used and need to be integrated.

- Field devices use one dedicated processor, optimized for overall power consumption with a speed ranging from just a few MHz to several tens of MHz. The processor typically has some RAM (around 20 kilobytes), non-volatile memory and a great deal of I/O on-chip, but often lacks extra functionality like a floating point unit.

- Parts of the software impose real-time constraints, such as the control loop, and the execution of fieldbus function blocks.

- The device has a static software configuration, *i.e.,* the firmware is updated or replaced as a whole, without dynamic loadable functionality. There is also no requirement for dynamic memory allocation.

- Many field devices are used in safety critical areas, such as chemical plants and therefore require costly certification procedures. In order to minimize the chance of certification failures, thorough testing is needed.

2.2. A Canonical Field Device

In order to validate CBSD for embedded systems, the PECOS project developed both hardware and software for a demonstrator device with characteristics very similar to the TZID field device. This device is concerned with setting a valve at a specific position between open and closed. Figure 2 shows a component Device containing three connected subcomponents that collaborate to set the valve position. A control loop is used to continuously monitor and adjust the valve.

ModBus is an *active* component *i.e.*, with its own thread of control. It is responsible for interfacing to a piece of hardware called the *frequency converter*, which determines the speed of the motor. The frequency to which the motor should be set is obtained from the PA component. ModBus outputs this value to the frequency converter using the ModBus protocol (hence its name). The ModBus component requires its own thread, because it blocks waiting for a (slow) response from the frequency converter.

FQD (Fast Quadrature Decoder [18]) is an *event* component that is triggered when an event is raised from the motor. This component abstracts from a micro-controller module that does FQD in hardware. It provides the PA with the velocity and the position of the valve.

PA is a *passive* component that implements the control loop to set the valve to a certain position. It holds the desired position of the valve, reads the current state of the valve from the FQD component, computes the new frequency for the motor controlling the valve and passes this to the ModBus component. This repeated adjustment and monitoring constitutes the control loop.

This example is used throughout the paper to illustrate the different concepts that are introduced.

2.3. The PECOS Approach

PECOS develops a CBSD approach that abstracts from best practice in field device software development. It is non-intrusive, and evolutionary, rather than radical, and consists in the following elements:

- *Component Model*: the component model is the foundation for the tools. It abstracts from best practice in software design for field devices, and supports reasoning about non-functional constraints (timing, scheduling and memory consumption).

- *Component Description Language (Coco)*: Coco implements the PECOS component model and is used to specify components and field devices.

- *Code Generation*: Coco specifications are used to generate code skeletons.

- *Runtime Environment (RTE)*: Generated code targets the RTE, which abstracts from the real-time operating system used.

- *Verification*: When components are composed, functional and non-functional aspects of the composition can be verified. *Composition rules* express constraints over compositions, such as dependencies between components. Non-functional checks include timing analysis, schedule generation and verification, and checks on memory consumption.

This paper only presents the component model and the timing verification. Detailed information on the other items can be found in [7] or on the PECOS website mentioned before.

3. The PECOS Component Model

The PECOS component model has been designed with the particular requirements of field devices in mind, and reflects best practice in this domain. For this reason the model differs substantially from typical component models, and is much simpler in many respects.

3.1. Design choices

The following design choices for the PECOS component model directly reflect best practice in the field device domain:

- *Cyclic behavior*: Each component is responsible for a single task which is repeatedly executed. However, the model also supports aperiodic event or active components.

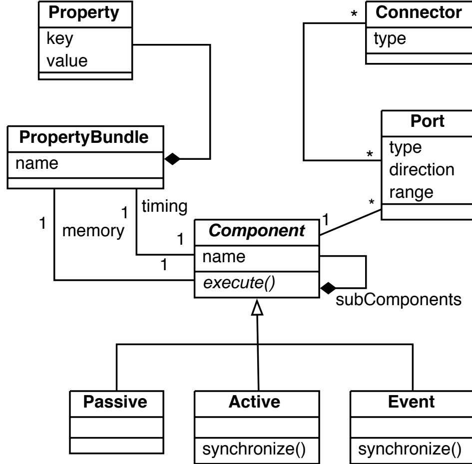

Figure 3. Overview of the PECOS Component Model.

- *Fine-grained components*: Components encapsulate a single task and export a data interface. They communicate by means of shared data. The result is a fine-grained model that makes it possible to minimize concurrency and check timing constraints.

- *Threading*: Components may be *active*, *i.e.*, have their own thread of control, *event* (to model the capture of an event), or *passive*.

- *Minimize critical sections*: Contention for shared resources must be minimized due to the high cost of locking data on the platforms used.

- *Separate scheduler*: Control flow is separately specified by a scheduler for composite components.

3.2. Static Structure

The PECOS field device component model [3] defines a vocabulary of *components*, *ports*, *connectors*, and the rules governing their composition. The model is illustrated in Figure 3.

A *component* is a computational element with a *name*, a number of *property bundles* and *ports*, and a *behavior*. There are *active* components with their own thread of control, *passive* components without their own thread of control, and *event* components, whose behavior is triggered by a hardware or software event. The *behavior* of a component consists of a procedure that reads and writes data available at its ports, and may produce effects in the physical world. This behavior is modeled as the entry point labelled exec. A *port* represents data which is read or written by a component, and which may be shared with other components. Each port is characterized by its *name*, its *type*, a *range* of possible values, and a *direction* (in, out or inout) indicating whether the component reads, writes, or reads and writes the data.

A *composite component* is specified by connecting selected ports of its internal subcomponents (that can be composites themselves). A *connector* describes a data-sharing relationship between ports. It has a name, a type, and a list of ports it connects. Only compatible ports may be connected [3]. Composite components need to specify a *schedule* for executing their subcomponents.

A *property* is a tagged value. The tag is an identifier, and the value is typed. A *property bundle* is a named group of properties. Property bundles are used to characterize aspects of components, and are used to add meta-information to the model, such as information regarding timing or memory usage.

In the example of Figure 2, FQD is an event component, PA is a passive component and ModBus is an active component. The composition itself (the Device component) is an active composite component. FQD has out ports *actualPosition* and *velocity*, connected to in ports of PA (*theActualPosition* and *theVelocity*). The out port *frequencyToSet* of PA is connected to the in port of the same name on ModBus.

3.3. Execution Model

We now consider the dynamic aspects of the model, in particular synchronization and scheduling.

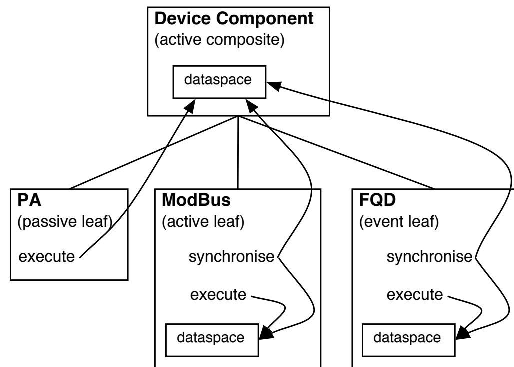

Figure 4. Tree view of the example.

3.3.1. Synchronization

The model imposes a hierarchical structure on compositions with a single composite component at the top level. Since this component controls the whole application, it is always active. The top-level component can be interpreted as a *tree* of simple and composite components, some of which are active, and others which are not.

The total number of threads in an application is the sum of the number of active components and the number of event components. Each passive component is fully under the control of the (unique) active component that contains it. The behavior of a component is given by its predefined exec procedure, which can be empty. The complete behavior of a composite component consists of the generated schedule which executes the passive components it contains and its own exec procedure in some given sequence.

The only difficulty is posed by the *critical sections* where separate threads read or write shared data. Contention occurs precisely wherever the external ports of an active component (whether simple or composite) are connected to the ports of any other component. In order to strictly limit the possible interference between concurrent threads, all of the ports belonging to a single thread, *i.e.*, an active component and all the passive components it contains, are grouped together as a single *dataspace* for that thread. Ports that are shared between threads are *replicated*, and explicitly synchronized by a sync procedure which locks the shared ports and updates the out-of-date copies. These sync procedures constitute the only critical sections in an application.

A composite component therefore not only schedules the exec procedures of its passive subcomponents and its own exec procedure, but also the sync procedures of its active and event subcomponents.

This is illustrated in Figure 4. PA is passive and thus uses the dataspace provided by Device. The FQD and ModBus components have their own dataspace, used by their exec procedure. Their sync procedure has access to both dataspaces so that they can be synchronized. Note that any passive subcomponents of FQD would use the dataspace of FQD. Any passive subcomponents of PA would use the dataspace provided by Device.

The synchronization semantics of the PECOS component model has been formalized [19] using Petri nets [20].

3.3.2. Scheduling

We can now summarize scheduling as follows:

- A composite component must provide a schedule for its direct subcomponents, *i.e.*, which specifies when to run its own exec procedure, the exec procedures of its passive subcomponents and the sync procedures of its active and event subcomponents.
- Each active and event component executes its own exec procedure and that of any subcomponents in a single thread of control.
- Passive components are scheduled by their parents.
- Synchronization procedures for active and event components are scheduled by their parents.

Figure 5 illustrates a sample run of a field device over a period of 120 milliseconds. Since the period of the device is 60 ms the figure shows two periods in the runtime of the device. Because there are two active components and one event

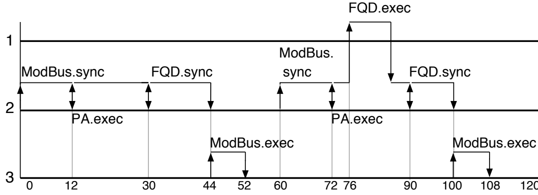

Figure 5. Possible Execution trace for the example.

component in the example, there are three concurrent threads: one for the Device component (Task 1), one for the ModBus component (Task 2), and one for the FQD component (Task 3). Assume that the priority of task 1 is higher than the priority for task 2, which is higher than the priority of task 3.

In the first period of task 2, the sync procedure of the ModBus is executed, followed by the exec procedure for the passive PA component and the sync procedure for the FQD component. When they are finished executing, the lower priority task 3 executes the exec procedure of the ModBus component. Since there is no event that triggers the exec procedure for the FQD component, nothing happens in task 3. In the second period of the device more or less the same happens except that an event triggers the exec procedure of the FQD component. That interrupts the behavior in task 2, since it has a higher priority. When it is finished, task 2 resumes, followed by task 3.

3.4. Checking Non-Functional Requirements

One of the key challenges for the PECOS component model was to support timing analysis to verify whether a component composition is schedulable and can meet its deadlines. Two possibilities exist to do this. Either the internal behavior of components can be modeled (for example using CSP [12] or Petri nets [20]) or the components can be instrumented at runtime to obtain execution times. The latter approach was chosen, since it was well-understood and corresponded to best practice.

The input for tools checking non-functional requirements is always a concrete component composition where the property bundles of components contain runtime information. This runtime information consists of the worst-case execution times of components, and optionally the maximum blocking times for active or event components.

Two *complementary* checks are performed using this information: *rate monotonic analysis* (RMA) and *schedule verification*. The RMA verification ensures that the overal component composition has time enough to run. Schedule verification ensures that within a task, activities can be scheduled sequentially. Both checks are complementary, since the RMA verification works on the level of the complete device, but can by itself not ensure that the schedule for a task is feasible. The schedule verification can ensure that the schedule for a task is feasible, but not that there is enough time to run all the activities (taking other tasks and task-switching overhead into account).

The next two sections explain the two complementary checks in detail and applies them to the example field device.

4. Timing Verification

Given a component composition, rate monotonic analysis (RMA) is used to check whether all the components involved in the composition meet their deadlines. This section briefly introduces RMA, and then shows the mapping from the component model to RMA and how this is used in practice.

4.1. Rate Monotonic Analysis

Rate monotonic analysis (RMA) [23] consists of a number of simple, practical techniques to generate or verify schedules for a set of real-time tasks. RMA algorithms assign a fixed priority to each task and assign higher priorities to tasks with shorter periods. It then provides different Theorems to check whether tasks can meet their deadlines depending on whether the tasks are (i) periodic and independent, (ii) mixed periodic and aperiodic, or (iii) interacting [23]. For PECOS in-

(1) each resource has a ceiling priority defined as the highest priority of all the potential tasks that use the resource. (2) A task gets the lock on a resource if this resource is not locked. The task then inherits the resource's ceiling priority plus one. It can then proceed with the execution of the critical section. If the lock could not be acquired, the task is blocked up to the point where the blocking task unlocks the resource and retrieves its assigned priority.

Figure 6. Rules employed by the highest-locker protocol.

teracting tasks are needed, because of the `sync` procedures of active and event components.

The difficulty with interaction is that high priority tasks should be minimally delayed by lower priority tasks when both are contending for the same resources. But when there are different tasks with different priorities that can freely lock resources, the periods where tasks of a higher priority are blocked by tasks of a lower priority become unpredictable. This situation is called *unbounded priority inversion*. Since the blocking times become unpredictable, no timing verifications can be done.

Therefore RMA assumes that the implementation uses real-time synchronization protocols that have two important properties: freedom from mutual deadlock, and bounded priority inversion, where at most one lower priority task can block a higher priority task. Examples of such protocols are the *priority ceiling protocol* or the *highest-locker protocol* (shown in Figure 6).

When such real-time synchronization protocols are used, a theorem known as *RMA theorem 4* (see Figure 1) can be used to check whether a set of interacting tasks meets its deadlines. Note that the other RMA theorems (one to three) either assume non-interacting tasks or only provide crude approximations.

A set of n periodic tasks using an appropriate real-time synchronization protocol will always meet its deadlines, for all task phasings, iff

$$\forall i, 1 \leq i \leq n,$$

$$min_{(k,l) \text{in} R_i} \sum_{j=1}^{i-1} C_j \left\lceil \frac{lT_k}{T_j} \right\rceil + C_i + B_i \leq lT_k$$

where C_i, T_i, and B_i are the *worst-case execution time*, *period* and *blocking time* for a task i, and

$$R_i \quad = \quad \left\{(k,l) | 1 \leq k \leq \ i, l = 1, ..., \left\lfloor \frac{T_i}{T_k} \right\rfloor \right\}$$

Table 1
RMA Theorem 4

4.2. RMA based Timing Analysis

To use RMA to verify a component composition a mapping is needed from components to RMA tasks. The example used is the same as described in Section 2.1 and used in the rest of the paper, but extended with runtime figures. This is shown in Figure 7. It is the input used for both the RMA analysis and the schedule analysis discussed in Section 5.

Mapping component behavior to tasks.

With a passive component P, a (periodic) task is associated that has a worst-case execution time, period and deadline as defined by Ps timing property bundle. With an active or event component A, two tasks are associated: a task T_{sync} for the `sync` procedure and a task T_{exec} for the `exec` procedure. The worst-case execution time, period and deadline of T_{sync} and T_{exec} are given by the sync and exec part of the timing bundle of A.

Note that RMA only allows tasks that are periodic, an inherent difficulty when applying RMA in a context where aperiodic tasks exist. To solve this problem lots of research is done in mapping aperiodic behavior to periodic tasks [23, 15]. Proposed solutions differ in desired response time, overal optimality of the scheduling, the ability to recuperate slack time left by non-utilized aperiodic tasks, and run-time overhead of the decision

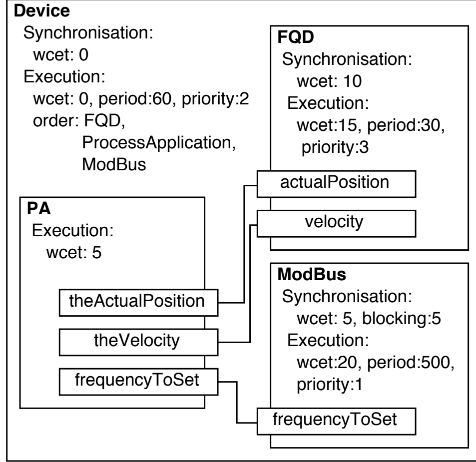

Figure 7. The example from Figure 2 extended with the runtime information for the components. All numbers indicate milliseconds, *wcet* stands for worst-case execution time.

algorithm. One class of solutions are *sporadic servers*, that have as advantage that they have a straight-forward implementation with very little runtime overhead. A sporadic server is equivalent to a regular periodic task from a theoretical point of view and thus fully compatible with RMA algorithm. In the example there is one aperiodic component, ModBus, of low priority that we map to a sporadic server task with a long deadline of 500 ($taskT_5$).

Assigning Priorities to tasks.

Then priorities need to be assigned. RMA specifies that the shorter the period (worst-case execution time) of a task, the higher its priority. For tasks with the same period, arbitrary priorities are chosen so that the tasks can be ordered according their priority. So when mapping the model to RMA the timing bundles are processed to make sure that these conditions hold. In the example all the priorities are given, so no priorities have to be chosen.

Deriving blocking times.

All tasks need blocking times. Tasks associated with passive components always have a blocking time of 0. But tasks for active or event components can indicate the time they need to lock the data space during synchronization or execution. When those blocking times are not given, worst-case blocking times are derived using the priorities and worst-case execution times. To explain how blocking times are extracted, assume the following setup: A is a composite active or event component that has n subcomponents. The sync procedure of A is called A_{sync}. The exec procedure is called A_{exec}. Every subcomponent has procedures that are scheduled by A. Call these procedures S_1 to S_m, where $m \geq n$ (since some of the subcomponents can be composite components). Now call S the set consisting of A_{sync} and S_1 to S_m. The datastore can thus be accessed simultaneously by A_{exec} and at most one element of S. To see which blocking can occur, S is split in two subsets: L is the subset of S where the elements have a priority that is lower than the priority of A_{exec}; H is the subset of elements that have a higher priority. With this division made, worst-case blocking times can be given for those elements that do not explicity provide values:

- Elements in L are the ones that can block A_{exec}. The worst-case blocking time of each element in L that does not specify an explicit value is either the blocking time of exec (when it is given) or the execution time of exec.

- Elements in H are the ones that can be blocked by A_{exec}. The worst-case blocking time of A_{exec} is the maximum of the worst-case execution times and explicit blocking times of the elements in H.

In the example, the blocking time for PA.exec is 0 since it is a passive component. ModBus.exec has the lowest priority in the example, so its blocking time is also 0. For the FQD component, no blocking times are specified. The worst-case blocking time for the FQD.exec is 10 (the worst-case execution time of the FQD.sync), and is 0 for

Behavior	wcet	period	blocking
FQD.exec	$C_1 = 15$	$T_1 = 30$	$B_1 = 10$
FQD.sync	$C_2 = 10$	$T_2 = 60$	$B_2 = 0$
PA.exec	$C_3 = 10$	$T_3 = 60$	$B_3 = 0$
ModBus.sync	$C_4 = 5$	$T_4 = 60$	$B_4 = 5$
ModBus.exec	$C_5 = 20$	$T_5 = 500$	$B_5 = 0$

Table 2
The result of mapping the component example to input for doing a RMA analysis

FQD.sync. Then the tasks are ordered from highest to lowest priority, resulting in Table 2. This table is the actual input for the RMA analysis.

Applying the theorem.

Theorem 4 can now be applied on the tasks obtained from the component model, since every tasks has a worst-case execution time, period and blocking time. The result of the analysis indicates whether the component composition given can meet its overall deadlines.

In the example the input for theorem is given by Table 2. The results of fitting the numbers from the table in the theorem are not shown here, but the result is that the equation holds. This assures us that globally there is enough time to run this component composition.

5. Schedule Generation and Verification

RMA assumes a set of tasks that run concurrently. However, as explained in Section 3.3.2, not all tasks run concurrently and hence checking the global timing properties is not enough. The model associates a thread with each active and event component, in which each exec and sync procedure has to be run (depending on the nesting of the components), and in what order (through the schedules specified in composite components). But whether it is possible to fit these behaviors sequentially in each thread needs to be checked by a schedule verification tool. Since this is essentially a resource allocation problem, constraint solving is an appropriate solution.

This section first explains constraint solving, and then shows how to map the component model to a constraint network.

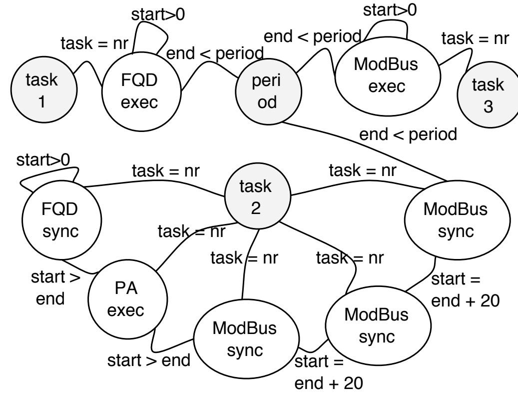

Figure 8. Constraint network for the example. The circles are variables for the activities (white) and for device information (grey). The connections are constraints on those variables.

5.1. Constraint Solving

The classical definition of a constraint is simply a relation between variables that should be maintained at all times [13, 4]. There are different mechanisms to express and solve constraints, a discussion of which falls outside the scope of this paper. Generally speaking constraint solving can be divided in two categories. First of all there are approaches based on logic programming, in particular Constraint Logic Programming (CLP) [13]. Second there is a number of (numerical) incremental constraint solvers that have primarily been applied in the context of graphical user interfaces [8, 4]. After experimenting with both we decided to go with a CLP approach because it allows us to generate all possible schedules, whereas incremental constraint solving techniques only give a single possible schedule.

5.2. Using CLP to Generate and Verify Schedules

To use CLP to verify or generate a schedule, a component model needs to be translated to a constraint network that can be solved. This section

explains the mapping from the component model to a constraint network. Applying the mapping to the example yields a network as shown in Figure 8.

With every sync or exec procedure one or more *activities* are associated. How many activities are needed depends on the period given for the behavior and the overall time for the schedule since the period specifies the interval at which the component expects to be executed. For example, when a component specifies that its execution behavior has a period of 20 milliseconds, and the schedule is 60 milliseconds long, the execution behavior has to be executed three times, separated by 20 milliseconds. So this execution behavior is mapped to three activities, with constraints that express the distance between them.

Then the activities need to be assigned to the appropriate tasks. Each activity therefore has a task number, and those task numbers are constrained by traversing the component tree. Whenever an active or event component is encountered, the activity for the exec procedure is assigned to a new task, together with the activities for the subcomponents (until one of those subcomponents is an active or event component again). The sync procedure inherits the task of the parent of the component.

Finally the order of components is codified as constraints on the appropriate activities. Any other information on starting times of components is also taken into account. For example, specifiying that a certain execution behavior component should start at 20 milliseconds is mapped to a constraint that specifies that the start of the activity corresponding to that execution behavior should be 20.

We used ECLiPSe, a constraint logic programming language, to implement the mapping, resulting in a constraint network as shown in Figure 8. Note that each of the variables in this network has a finite domain. The domain for the task variables is between 1 and the number of tasks needed. The domain for the *start*, *end* and *worst-case execution* times in each of the activity variables is between 0 and the period of the device. To solve this network we employ a technique known as *edge_finding* [6], a general constraint satisfaction technique directly available in ECLiPSe as a library. This technique derives stronger bounds on the starting times of the activities, while making sure that they do not overlap. The result of solving the network is either a schedule for each task in the component model or a failure. A failure indicates that there is at least one task for which no schedule could be found. When at least one solution can be found, a sequential schedule *for each thread separately* is found. One of these solutions for our example device is shown in Figure 9. Finding one static schedule for the whole device is not possible because field devices need active and event components. For simple devices, only passive components can be used inside the field device, and then the solutions of the schedule verification yield the static schedule for the overall device.

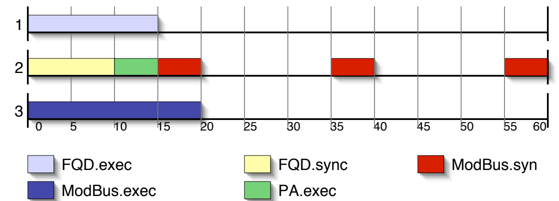

Figure 9. Graphical depiction of a result from running the schedule verification tool. It shows the sequential ordering of behaviors for the three tasks in the model.

Note that since a CLP language is used, any incomplete information is regarded as a logic variable that needs to be solved. For example, the constraint network can be used to find the maximum worst-case execution times that are possible for one or more components in a composition. Or schedules can be seen for different orders for subcomponents, so that the developer can select the most appropriate one. Hence the schedule verification tool can also be used to generate schedules or to have an interactive way of building schedules.

6. Discussion and Related Work

According to Szyperski [25], component-based software development now promises to "deliver reusable, off-the shelf software components for incorporation into large applications". Components are typically defined as entities that encapsulate some internal representation with one or more interfaces. This enables binary composition of components, usage of multiple implementation languages, easier distribution of components and even dynamic reconfiguration of applications at runtime [24, 17].

All these approaches, however, need some runtime infrastructure to work. For example, to support distribution, centralized or distributed naming services have to be used. Unfortunately, the context of field devices at this time does not allow for such runtime infrastructure. The PECOS component model therefore focuses on static composition, and assumes only a runtime environment that supports synchronous passing of data according to a blackboard-like architecture.

There are also similarities between the PECOS approach and architectural description languages (ADLs). An ADL is a language for modelling a software system's conceptual architecture in terms of components, connectors, and configurations [9]. An ADL typically embodies a formal semantic theory that allows certain analyses to be performed. Some ADLs focuss on supporting distributed applications (such as *Darwin/Regis* [16], *Rapide* [14]). Others focuss on describing the semantics of the internals of components in some formalism to do certain verifications (for example *Wright* [2]).

In the context of PECOS formal approaches (like Timed Petri nets [26], real-time object-oriented modeling [21], real-time UML [22] synchronous languages like Esterel [5] and Lustre [10], StateCharts [11], or Piccola [1]) that describe the behaviour of components in order to verify runtime and synchronization aspects were not applicable. This was motivated by the fact that the approach had to be usable by the current developers that had no experience with formal techniques. Instead the runtime aspects of the components are measured, a technique with which the developers have a lot of experience and have already well-proven hard- and software solutions.

The main similarities between the PECOS component model and general-purpose component models and ADLs reduce to:

- the behavior of components is black-box, and completely encapsulated in the components,
- the interface of a component, is completely separate from its implementation, and from its interconnection with other components,
- the model is language-independent, though composition of heterogeneous components is not supported.

The main differences can be summed up as:

- components have a single interface that consists of the data that it requires/provides. There is no interface for behavior, since the model is fine grained and a component encapsulates a single piece of behavior.
- model elements have property bundles that are used to attach meta-information to components.
- the model enables timing verification with rate monotonic analysis and schedule verification and generation.

7. Conclusion

Modern development techniques have been difficult to apply to embedded software due to the mismatch between the severe constraints posed by the target devices, and the generous assumptions required by the techniques. The PECOS project has developed a viable approach to enable CBSD for a class of embedded systems known as field devices. Rather than attempting to adapt existing component models to the domain of field devices, a new, pragmatic component model was developed that reflects best practice in terms of component concepts. With memory being a scarce resource and computing power being severely limited, the model is data-centric

and eliminates concurrency and synchronization wherever possible. However, it supports passive, active and event components and enables timing analysis and schedule verification to ensure that component compositions can be deployed successfully.

Although a full validation of the PECOS approach was not possible within the scope of the project, the ideas have been convincingly tested in the context of the field device demonstrator, and plans are underway to apply the model to further case studies.

Results of the project are available from `www.pecos-project.org`.

Acknowledgment. Thanks to Andrew Black for many useful discussions during his visit at the Software Composition Group lab.

REFERENCES

1. F. Achermann and O. Nierstrasz. Applications = Components + Scripts – A Tour of Piccola. In *Software Architectures and Component Technology*. Kluwer, 2001.
2. R. J. Allen. *A Formal Approach to Software Architecture*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, 1997.
3. G. Arévalo, S. Ducasse, O. Nierstrasz, P. Liang, and R. Wuyts. Verifying timing, memory consumption and scheduling of components. PECOS deliverable D2.2.6-3, University of Bern, 2002.
4. G. J. Badros and A. Borning. The cassowary linear arithmetic constraint solving algorithm: Interface and implementation, 1998. University of Washington, TR 98-06-04.
5. G. Berry. *The foundations of Esterel*. MIT Press, 2000.
6. J. Carlier and E. Pinson. A practical use of jackson's preemptive schedule for solving the job-shop problem. *Annals of Operations Research*, (26), 1990.
7. Pecos Consortium. *PECOS in a Nutshell. http://www.pecos-project.org/public_documents/pecosHandbook.pdf*. 2002.
8. B. Freeman-Benson, J. Maloney, and A. Borning. An incremental constraint solver. *Comminications of the ACM*, 33(1):55–63, 1990.
9. D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural description of component-based systems. In *Foundations of Component-Based Systems*. Cambridge Press, 2000.
10. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. In *Proceedings of the IEEE*, volume 79, September 1991.
11. D. Harel. On visual formalisms. *CACM*, 31(5):514–530, May 1988.
12. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
13. J. Jaffar and M. Maher. Constraint logic programming : a survey. *The Journal of Logic Programming*, (19,20):503–581, 1994.
14. J. J. Kenney. *Executable Formal Models of Distributed Transaction Systems based on Event Processing*. PhD thesis, Stanford University, 1995.
15. M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, 1993.
16. J. Magee, N. Dulay, and J. Kramer. Regis: A constructive development environment for distributed programs. *IEE/IOP/BCS Distributed Systems Engineering*, 1(5), 1994.
17. R. Monson-Haefel. *Enterprise JavaBeans*. O'Reilly, 1999.
18. Motorola. Fast quadrature decode tpu function (FQD). TPUPN02/D, 2002.
19. O. Nierstrasz, G. Arevalo, S. Ducasse, R. Wuyts, A. Black, P. Müller, C. Zeidler, T. Genssler, and R. Born. A component model for field devices. In *Proceedings of Working Conference on Component Deployment*. ACM, 2002.
20. J. L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3), 1977.
21. B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994.
22. B. Selic and J. Rumbaugh. Using UML for modeling complex real-time systems, 1998.
23. Sha, Klein, and Goodenough. *Rate Mono-*

tonic Analysis for Real-Time Systems. Foundations of Real-Time Computing: Scheduling and Resource Management. Kluwer Academic Publishers, 1991.

24. J. Siegel. *CORBA Fundamentals and Programming*. John Wiley & Sons, 1996.
25. Clemens A. Szyperski. *Component Software*. Addison Wesley, 1998.
26. J. Wang. *Timed Petri Nets*. Kluwer Academic Publishers, 1998.