Applying RMA for Scheduling Field Device Components[1]

Peng Liang, Gabriela Arévalo, Stéphane Ducasse,
Michele Lanza, Nathanael Schaerli, Roel Wuyts
and Oscar Nierstrasz
Software Composition Group
Post: IAM, Neubrückstrasse 10,
University of Bern,
CH-3012 Bern,
Switzerland
Fax: +41 031.631.3355
Email: {lpeng,arevalo,ducasse,lanza,schaerli,wuyts,oscar}@iam.unibe.ch
Web: www.iam.unibe.ch/~scg

Abstract. PECOS is a collaborative project between industrial and research partners that seeks to enable component-based technology for a class of embedded systems known as "field devices". Results so far include a *component model for field devices* and a *composition language* for specifying connections between software components. Here we investigate the application of Rate Monotonic Analysis (RMA) to the problem of generating real-time schedules for compositions of field device components.

1. ECOOP 2002 Workshop Reader

1. Introduction to PECOS

The goal of PECOS is to enable CBSD for embedded systems by providing an environment that supports the specification, composition, configuration checking, and deployment of embedded systems built from software components. ABB's Instruments business unit develops a large number of different *field devices*, such as temperature, pressure, and flow sensors, actuators, and positioners. A field device is a reactive, embedded system. Field devices make use of sensors to continuously gather data, such as temperature, pressure or rate of flow.

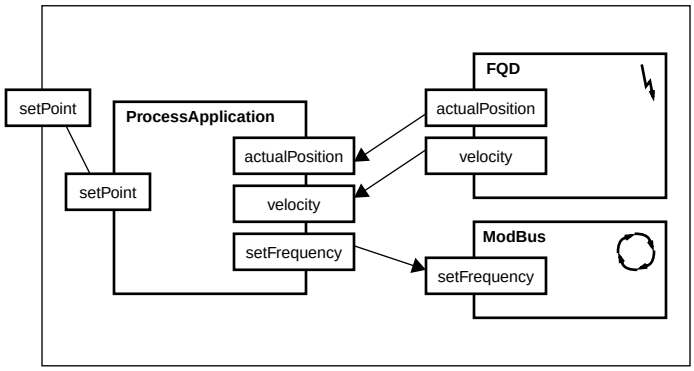

Figure 1 FQD Control loop example

In order to validate CBSD for embedded systems, the PECOS project is developing the hardware and software for a demonstration field device. Part of the PECOS case study is concerned with setting a valve at a specific position between *open* and *closed*. Figure 1 illustrates three connected PECOS components that collaborate to set the valve position. A control loop is used to continuously monitor and adjust the valve.

- The ModBus component 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 ProcessApplication component. ModBus outputs this value over a serial line to the frequency converter using the ModBus protocol (hence its name). The ModBus component runs in its own thread, because it blocks waiting for a (slow) response from the frequency converter.
- The FQD (Fast Quadrature Decoder [4]) component is responsible for capturing events from the motor. This component abstracts from a micro-controller module that does FQD in hardware. It provides the ProcessApplication with both the velocity and the position of the valve.
- The component ProcessApplication obtains the desired position of the valve (setPoint) and reads the current state of the valve from the FQD component. This information is then used to compute a frequency for the motor. Once the motor has opened the valve sufficiently, ascertained by the next reading from the FQD, the motor must be slowed or stopped. This repeated adjustment and monitoring constituted the control loop.

This example illustrates several key points concerning the field device domain.

- *Cyclic behaviour:* each component is responsible for a single task, which is repeatedly executed.
- *Information flow through ports:* components communicate by means of shared data. The interface of a component consists of a set of shared data ports.
- *Threading:* some components are passive, while others have their own thread of control.
- *Separate scheduler:* control flow is separately specified by a scheduler for the composite component.

2. A Component Model for Field Devices

The PECOS field device component model [2] defines a vocabulary of *components*, *ports* and *connectors* and the *rules* governing their composition.

A *component* is a computational element with a *name*, a number of *property bundles* and *ports*, and a *behaviour*.

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

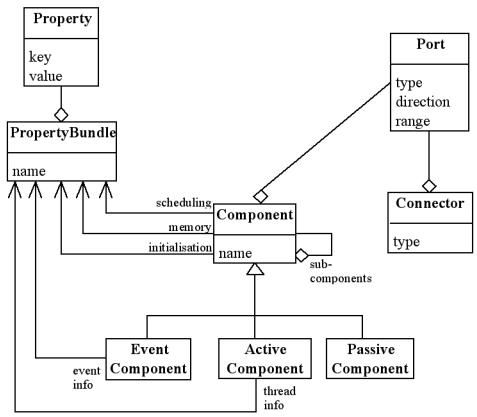

Figure 2 The PECOS Component Model

A *port* is a shared variable that allows a component to communicate with other components; *connected* ports represent the *same* shared variable. A port specifies: a *name*, which has to be unique within the component; a *type*, characterizing the data that it holds; a *range* of values (defined by a minimum and maximum value) that can be passed on this port; and a *direction* ("in", "out" or "inout") indicating whether the component reads, writes, or reads and writes the data.

A *connector* specifies 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 [2].

The *behaviour* of a component consists of a procedure that reads and writes data available at its ports, and may produce effects in the physical world.

A *composite component* contains a number of connected subcomponents, the ports of which form the *internal ports* of the composite. A composite component also has *external ports*, which are the only ones that are externally visible. The external ports are *connected* to appropriate internal ports. The subcomponents are not visible outside the composite that contains them.

The field device domain requires three kinds of components.

- *Active components* have their own thread of control; they are used to model ongoing or longer-lived activities that do not complete in a short time-cycle.
- *Passive components* do not have their own thread of control.
- *Event components* are those whose behaviour is triggered by an event.

In the example of figure 1, FQD is an event component, ProcessApplication is a passive component and ModBus is an active component. The composition will be modelled as a composite component.

FQD has "out" ports actualPosition and velocity, connected to "in" ports of the same name belonging to ProcessApplication. The in port setPoint belonging to ProcessApplication is shared with the composite component that encapsulates this composition. It is not yet connected to a compatible "out" port. Finally, the "out" port setFrequency is connected to the "in" port of the same name belonging to ModBus.

3. RMA for Scheduling Verification

We can model the run-time behaviour of a composition of components as a Petri net [2]. We would like to verify whether such compositions meet their real-time deadlines. To tackle this problem, we (i) encode the scheduling constraints in the property bundles of components, (ii) generate a schedule from the specified composition, and (iii) apply RMA to verify the constraints.

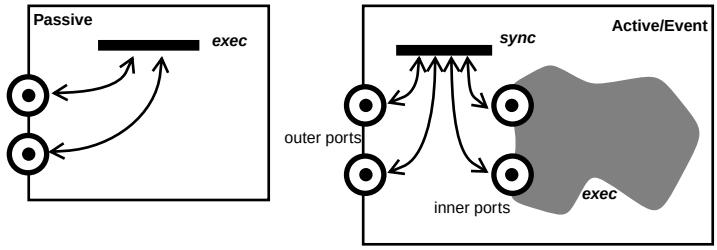

Figure 3 Passive vs Active components

3.1 Timing bundles

In a composition, each passive component is modelled as a single *exec* transition that reads or writes to its data ports (figure 3). To verify a schedule for a composition of components, certain scheduling information must be associated with each subcomponent; this includes the worst-case execution time (wcet) of the subcomponent and the desired cycletime, deadline, priority etc. This information is expressed with a *timing property bundle*:

```
timingBundle (wcet: w <Milliseconds>, [cycletime: c <Milliseconds>],
[deadline: d <Milliseconds>], [priority: p <Priority>]).
```

Passive components are scheduled by the *active parent* that encloses them.

In contrast to passive components, active components (figure 3) have both an *exec* transition for their behaviour, and a *sync* transition to safely synchronize their data ports with their surrounding environment [2]. For an active subcomponent, therefore, the timing bundle must separately characterize both the *sync* and *exec* parts.:

```
timingBundle(
sync( wcet: w <Milliseconds>, [cycletime: c <Milliseconds>], [deadline:
d <Milliseconds>], [priority: p <Priority>]),
exec( wcet: w <Milliseconds>, [cycletime: c <Milliseconds>], [deadline:
d <Milliseconds>], [priority: p <Priority>])).
```

The timing bundle for a composite component specifies the order in which its subcomponents have to be scheduled. Since a component can be active or passive, it also specifies the active or passive information as discussed with the respective leaf components:

```
timingBundle(...active or passive information...,
                order: ({componentname} <String>)+).
```

Finally there is also one timing bundle for the field device itself, so that overall information, default values for optional parts in the timing bundles of components and the order of the top-level components can be set:

```
timingBundle(cycletime: c <Milliseconds>, defaultPriority: p
<Priority>, order: ({componentname} <String>)+).
```

The timing bundles for the example would be expressed as follows:

ProcessApplication	`timingBundle(wcet:10).`

ModBus	`timingBundle(`
	`sync(wcet:5),`
	`exec(wcet:20,cycletime:500,priority:1)).`

FQD	`timingBundle(`
	`sync(wcet:10),`
	`exec(wcet:15,cycletime:30,priority:3)).`

Field device	`timingBundle(cycletime:60,defaultpriority:2, order:` `FQD, ProcessApplication, ModBus).`

3.2 Rate Monotonic Analysis

Rate Monotonic Analysis (RMA) [5] consists of a number of simple, practical techniques to generate or verify schedules for a set of real-time tasks. RMA provides different algorithms depending on whether the tasks are (i) periodic and independent, (ii) mixed periodic and aperiodic, or (iii) interacting. For a brief discussion, see the corresponding PECOS deliverable [1].

RMA algorithms assign a fixed priority to each task and assign higher priorities to tasks with shorter periods. The basic RMA algorithms assume that tasks are both periodic and independent. For PECOS, however, we must deal with tasks that are both aperiodic (event components) and interacting (*sync* methods of active 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. Suppose that there are two tasks (T_1 and T_2), where the priority of T_1 is lower than the priority of T_2, and during their execution both T_1 and T_2 need access to a shared resource that is locked by a semaphore. Whenever T_1 executes and uses the semaphore to lock the shared resource, the higher priority task T_2 has to wait for T_1 to finish using the shared resource. Hence the higher priority task is blocked by a lower priority task. This situation is called *priority inversion.*

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.

To remedy this situation RMA uses real-time synchronization protocols (such as the *priority ceiling protocol* or the *highest locker protocol),* that have two important properties that allow schedule verification:

1. freedom from mutual deadlock, and
2. bounded priority inversion, where at most one lower priority task can block a higher priority task.

We exploit the following standard RMA results that take blocking times into account (the theorems that do not do this are known as Theorem 1 and Theorem 2). The first theorem that we present (theorem 3) makes a crude approximation that can easily be used manually to quickly check whether a set of tasks might meet its deadlines. It is, however, very conservative.

Theorem 3 A set of *n* periodic tasks using the priority ceiling protocol can be scheduled by the Rate Monotonic algorithm, for all task phasings, if the following condition is satisfied:

$$\frac{C_1}{T_1} + \ldots + \frac{C_n}{T_n} + max\left(\frac{B_1}{T_1} + \ldots + \frac{B_{n-1}}{T_{n-1}}\right) \leq U(n)$$

where C_i , T_i and B_i, are, respectively, the execution time, the period and the worst-case blocking time of task t_i, and

$$U(n) = n \times (2^{1/n} - 1)$$

Note that $U(1) \leq 1$ and that U(n) quickly converges to 0.69314...

Theorem 3 is very pessimistic since the worse-case task set is contrived and unlikely to be encountered in practice. And when the proper real-time synchronization protocol is used (as seen in 3.2), a less conservative formula can be used, that is known as Theorem 4.

Theorem 4 A set of *n* periodic tasks using the priority ceiling protocol will always meet its deadlines, for all task phasings, if and only if

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

where C_j, T_j, and B_i are defined as in Theorem 3, and

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

3.2.1 Using RMA on PECOS Component Models

Mapping components to tasks is pretty straightforward. With every passive component P, we associate a (periodic) task that has a worst-case execution time, period and deadline as defined by P's timing bundle. With every active or event component A, we associate two tasks: a periodic task T_{sync} for the sync part and an aperiodic task T_{exec} for the exec part. 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.

Remember that the execution thread in an active component uses its own private data store that gets synchronized with the surrounding data store when the sync is run. This means that this local data store is a shared resource between T_{sync} and T_{exec}, and that we need to take blocking into account. Hence we will use Theorem 4, so we have to determine the maximum blocking time for each task.

To determine the blocking times, we first consider that the only shared resources that are involved are the private data stores between exec and sync tasks of active or event components. There are no other shared resources that we need to take into account. The maximum time that a shared resource is locked is given by the worst-case execution time of the sync task associated with that resource. This follows from the fact that the only purpose of the sync is to move data back and forth between the surrounding world and the local data store.

Since we consider using the priority ceiling protocol, we know that a task may be blocked, at most, for the duration of the longest critical section protected by the resource it uses. Hence, the maximum blocking time for the tasks for passive components will be 0 (since they do not use shared resources). The maximum blocking time for the exec and sync tasks used for active and event components is the worst-case execution time of the sync task.

3.2.2 Example RMA analysis

In the example, there is a set of periodic tasks (*exec* part of passive components and *sync* part of active/event components) and aperiodic tasks (*exec* part of certain active/event components, such as FQD) that use shared resources. Since the tasks include both periodic and aperiodic tasks, we first have to fit the aperiodic tasks into the periodic framework. Then we can apply Theorem 4, since shared resources are involved and the scheduler uses the *priority inheritance protocol*.

The difficult part of applying RMA on a component model is determining the specification parts of the servers that are needed to model the execution part of active and events components. In the mapping given, we noticed that the task for the execution part of the ModBus component is not critical (its priority is set lower than the average priority). Hence we chose to use a sporadic server task with a long deadline of 500 (task T5). The FQD, on the other hand, is more critical, since the user requires a good response time. Therefore we use a sporadic server with a deadline of 30 (task T1). We also order the tasks from the one with the highest priority to the lowest priority.The resulting table is shown below:

FQD(exec part)	T1	C_1=15, T_1=30, B_1=10
FQD(sync part)	T2	C_2=10, T_2=60, B_2=10
ProcessApplication	T3	C_3=10, T_3=60, B_3=0
ModBus(sync part)	T4	C_4=5, T_4=60, B_4=5
ModBus(exec part)	T5	C_5=20, T_5=500, B_5=5

We can then apply Theorem 4 to determine whether the deadline for each task in this task set can be met. Since there are 5 tasks that interest us, we apply the theorem for values of i ranging from 1 to 5. For each value of i, we have to find at least one possible pair of (k,l) that make the equation true. We do not show all the possible values here, but will stop when we satisfy the theorem for a value of i.

$i=1,k=1,l=1$: $C_1 + B_1 = 20 + 10 = 30$ (satisfied)

$i=2,k=1,l=1$: $C_1 + C_2 + B_2 = 20 + 10 + 10 = 40 > 30$

$i=2,k=1,l=2$: $2C_1 + C_2 + B_2 = 40 + 10 + 10 = 60$ (satisfied)

$i=3,k=1,l=1$: $C_1 + C_2 + C_3 + B_3 = 15 + 10 + 10 + 0 = 35 > 30$

$i=3,k=1,l=2$: $2C_1 + C_2 + C_3 + B_3 = 30 + 10 + 10 + 0 = 50 < 60$ (satisfied)

$i=4,k=1,l=1$: $C_1 + C_2 + C_3 + C_4 + B_4 = 15 + 10 + 10 + 5 + 5 = 45 > 30$

$i=4,k=1,l=1$: $2C_1 + C_2 + C_3 + C_4 + B_4 = 30 + 10 + 10 + 5 + 5 = 60$ (satisfied)

$i=5,k=1,l=1$: $C_1 + C_2 + C_3 + C_4 + C_5 + B_5 = 15 + 10 + 10 + 5 + 20 + 5 = 65 > 30$

...

$i=5,k=2,l=5$: $10C_1 + 5C_2 + 5C_3 + 5C_4 + C_5 + B_5$

$= 150 + 50 + 50 + 25 + 20 + 5 = 300$ (satisfied)

So, since for every possible value of i we can satisfy Theorem 4, the 5 tasks all meet their deadlines.

4. Open Questions and Future Work

Using the proven RMA technology in the context of CBSE for real-time systems seem to be an interesting way to verify the schedulability of such systems. In this paper we show a mapping that expresses components in our particular component model to tasks, such that we can apply the RMA theorems. This approach looks promising, but we are in the moment of refining it and applying it on more real-world examples.

Two points will decide whether the approach is really useful in a practical context. First of all the mapping of active components (that have an aperiodic part to them) into periodic tasks. This mapping is not trivial, but a large body of existing mappings in the context of RMA exists. A second problem lies in the very assumptions that RMA makes to apply Theorem 4: the OS or the implementation has to support more difficult protocols (such as priority ceiling protocol). When this is not the case, only Theorem 3 can be used, which is more conservative and will sometimes yields false negatives.

5. References

[1] Gabriela Arévalo, Stéphane Ducasse, Oscar Nierstrasz, Peng Liang, Roel Wuyts, "Verifying timing, memory consumption and scheduling of components," PECOS Deliverable D2.2.6-2, www.pecos-project.org

[2] Oscar Nierstrasz, Gabriela Arévalo, Stéphane Ducasse, Roel Wuyts, Andrew Black, Peter Müller, Christian Zeidler, Thomas Genssler, and Reinier van den Born, "A Component Model for Field Devices", in Proceedings of Component Deployment 2002, Berlin, to appear.

[3] Benedikt Schulz, Thomas Genssler, Alexander Christoph, Michael Winter, "Requirements for the Composition Environment", PECOS Deliverable D3.1, www.pecos-project.org

[4] Semiconductor Motorola Programming Note, Fast Quadrature Decode TPU Function (FQD), TPUPN02/D.

[5] Sha, Klein and Goodenough, J. Rate Monotonic Analysis for Real-Time Systems. Foundations of Real-Time Computing: Scheduling and Resource Management. Boston, MA: Kluwer Academic Publishers, pp. 129-155, 1991.