Verifying Timing, Memory Consumption and Scheduling of Software Components

Gabriela Arévalo, Stéphane Ducasse, Peng Liang
Oscar Nierstrasz, Roel Wuyts

May, 2002
PECOS-D2.2.6

Software Composition Group
Post: IAM, Neubrückstrasse 10,
University of Bern,
CH-3012 Bern,
Switzerland

Abstract
This report firstly describes the format of the timing bundles, which are the places in the model where timing information is added to components. Then it introduces two techniques that can be used together to verify schedules: constraint solving and Rate Monotonic Analysis (RMA). We present each technique, show how to express the component model and show a concrete example. The report also outlines the verification of memory consumption.

This research has been partially funded by European CORDIS Community Research project (EU), PECOS: Pervasive Component Systems (IST-1999-20398).
# Content

1. Introduction................................................................................................................... 3
2. Timing Bundles................................................................................................................. 3
3. Verification of Schedules............................................................................................ 5
4. Numerical Constraint Solving....................................................................................... 6
5. Rate Monotonic Analysis.............................................................................................. 8
6. References.................................................................................................................. 16
1 Introduction

This report describes the verification of the non-functional requirements timing and memory consumption. The content of this report is as follows. Section 2 briefly recapitulates Component Model-V and describes the format of the timing bundles, which are used to describe the information regarding timing. Section 3 discusses schedule verification in general, and what we want to address in Pecos. Sections 4, 5 and 6 discuss techniques to do schedule verification. Section 7 presents how property bundles support memory consumption verification.

2 Timing Bundles

In this section we look at what information needs to be attached to components to verify their schedules. The Pecos Component Model-V leaves open placeholders for this information in the form of the timing bundles. This section motivates and describes what kind of information we put in the timing bundles of various component types. In the next section we show how this information is used by the various techniques under consideration.

2.1 Component Model-V

The scheduling verification described in this report is meant to be used in the context of the Pecos model. Therefore we briefly review the model before looking at the verification itself. More specifically we focus on the static aspects of the model. For information regarding the semantics, and more detailed information and examples, please refer to report PECOS-D2.2.8 which describes the model.

Remember that the PECOS Component Model-V was developed in the context of field devices. Therefore, its main usage was to perform:

- Structural checking (checking of well-formedness of components and structural checks on the composition of components), and
- Checking of non-functional requirements.

The non-functional requirement that is identified as being the most important is timing/scheduling. This report focuses on the timing and scheduling issues, and more specifically on the verification of schedules.

The Field Device Component Model-V consists of components, ports and connectors. It enforces an execution model where components can run concurrently, but where they can only communicate data on their ports during synchronization events determined by a scheduler. The core entity in the model is the component. Every component can contain one or more connected sub-components, has three property bundles (scheduling, memory and initialization), and a set of ports. Ports denote data that is available to share with other components. Connectors are used to model actual data sharing between specific ports on specific components. There are different types of components: passive, active, and event.

We now have a detailed look at the contents of the timing bundle.

2.2 Timing Bundle Contents

The information regarding timing that can be used to verify timing and scheduling is put in the timing bundles of components. In this section we discuss the contents of the timing bundle for the different component types and the device itself. We describe the contents of the timing bundles in a regular expression-like syntax using the following notation:

- labels (to describe the values) are put in ordinary text
- values are put in italics
- values are typed. The type follows the value and is put between < and >
- We consider three possible types:
  - Milliseconds: a positive number that expresses a period in milliseconds.
  - Priority: a strictly positive number.
  - String.
Appending an arity symbol to any expression makes sequences. We use the following arity indicators:
- \* for 0 or more
- \+ for 1 or more
- +X for X or more

Items between { and } denote references to other items. For example, \{componentName\} is a placeholder where the name of a component needs to be specified.

Items between [ and ] are optional.

2.2.1 Passive leaf component

\[
\text{timingBundle( wcet: } w <\text{Milliseconds}> ,
\{\text{cycletime: } c <\text{Milliseconds}>\},
\{\text{deadline: } d <\text{Milliseconds}>\})
\]

Figure 1: Timing bundle for a passive leaf component

The content of the timing bundle is given by Figure 1. It consists of the following items:
- worst-case execution time (wcet): this field gives the maximum execution time of the behavior defined by the component.
- cycletime: this field specifies either the cycletime (the interval at which the component expects to be executed – also known as period) or nothing. When it is not given, it indicates that the cycletime of the device is used when the component is composed.
- deadline: this field specifies either the deadline time (a window in which the task has to be executed; the window starts when the behavior of the component is executed and ends at the tasks deadline) or nothing. When nothing is specified, this field is equal to the cycletime of the component.

For example, the bundle for a passive component that specifies a worst-case execution time of 10 ms and a cycletime of 30 ms is given as follows:

\[
\text{timingBundle( wcet: 10,}
\text{cycletime: 30).}
\]

2.2.2 Active/Event leaf component

\[
\text{timingBundle( sync( wcet: } w <\text{Milliseconds}> ,
\{\text{cycletime: } c <\text{Milliseconds}>\},
\{\text{deadline: } d <\text{Milliseconds}>\},
\{\text{blocking: } l <\text{Milliseconds}>\}),
\text{exec( wcet: } w <\text{Milliseconds}> ,
\{\text{cycletime: } c <\text{Milliseconds}>\},
\{\text{deadline: } d <\text{Milliseconds}>\},
\{\text{priority: } p <\text{Priority}>\},
\{\text{blocking: } l <\text{Milliseconds}>\}))
\]

Figure 2: Timing bundle for an active or event component

The content of the timing bundle is shown in Figure 2. It basically consists of two timing bundles: one for the synchronization action of the component and one for the execute action. For the description of these items we refer to the explanation for passive components. The only additions are a priority and a blocking time:
- Priority: it may be necessary to add priorities to decide at runtime which component will be run. For example, a component involved in the user-interface may take priority to execute over other components. This field allows specifying this priority, when needed. When the field is left open, the default priority of the device is used.
- Blocking time: This parameter indicates the maximum time that the execution or the sync needs to lock the internal data store associated with an active component. Please note that
when it is not specified, there is no default value that is assigned. As we see later on in this report, techniques like RMA should be used to provide figures.

For example, an active component where the sync action has an execution time of 2 ms, and the execution part has a worst-case execution time of 10 ms, and a priority of 2, is specified as:

```
timingBundle( sync( wcet: 2),
            exec( wcet: w 10,
                   priority: 2)).
```

### 2.2.3 Composite Components

… passive or active bundle …,

[order: {name} <String> [( [maxDelay <Milliseconds>], {name} <String>)+ ]]

Figure 3: Timing bundle for a composite component

Composite components, whether they are active or passive, can specify an order in which their subcomponents have to be executed. This order is partial. When the field device is composed, all the partial orders have to be taken into account to define a total order for the executing components of the device.

As can be seen in Figure 3, the order is given as a sequence terminated with the last component to run, or not at all. The sequence consists of component names that can be followed by a maximum delay time. The maximum delay time specifies that the following component should start within a window defined by the end of the previous task and the maximum delay time. When it is not given, it means that the following component starts somewhere after the previous one, without real constraint on when exactly that is.

For example, suppose that we have a composite component Z containing four subcomponents (A, B, C and D), and that we want to specify that B should be run before D. The maximum delay between B and D is 30 ms. We then write the following order expression in the timing bundle of Z:

```
order: (B, 30, D)
```

Note that, since the order is partial we do not need to specify where A and C will run. We only require that B be run before D, and will start within 30 milliseconds of the end of D.

```
timingBundle(   cycletime: c <Number>,
                defaultPriority: d <Number>,
                [order: {name} <String> [, maxDelay <Milliseconds>]+ {name} <String>]).
```

Figure 4: Timing bundle for a field device

### 2.2.4 Field Device

The field device itself also needs to specify the information shown in Figure 4: the cycletime for the field device, the default priority that is used for components that did not specify values for their priorities, and a partial order of subcomponents, as with a composite component.

### 3 Verification of Schedules

The model does not specify anything regarding the scheduling of components, what scheduler can be used and how schedules can be checked to see if they are actually feasible. It only assumes that there is a scheduler. For example, when the model is formalized with Petri nets, part of the Petri net model is a scheduler net that specifies the order in which the behavior of the components is executed.

Regarding scheduling verification, there are several questions that deserve an answer:

1. What scheduling algorithm can be used, and how does this affect schedule verification?
2. How can we check whether a field device meets the deadlines of all the components that it contains?
3. How can we determine the order in which the behavior defined by the components has to be run?

Note that in order to answer these questions we rely solely on the model and the information provided by the timing bundles. How to fill in the timing bundles falls outside the scope of this report. The basic idea is to build a testbed and use the Run-time environment (RTE) to get the timing information for a component.

In the following sections we explain the two techniques that we combine to answer these questions: numerical constraint solving, and Rate Monotonic Analysis (RMA\(^1\)). We can already say that to verify a schedule of a field device, we need to combine both constraints and RMA. The former makes sure that the tasks can fit a particular schedule (e.g., that for the order given it is indeed possible to find a distribution). The latter makes sure that the timing requirements are met. In the following section we explain each technique and then we show how they can be used together in a practical setting.

4 Numerical Constraint Solving

The first technique we investigate is numerical constraint solving. In this section we give an overview about constraint solving in general and numerical constraint solving in particular, and describe how we express the timing information in a constraint logic programming language. Using the constraint logic programming language schedules can be generated and verified using the structure and timing information in the model. Note that it is not a good technique to do general timing analysis (as we will see, RMA is much better suited for that).

4.1 Constraint Solving

The classical definition of a constraint is simply a relation between variables (the constraint variables) that should be maintained at all times [JL87,Cohe90,BB98]. There are different mechanisms to express and solve constraints, a discussion of which falls outside the scope of this report. In general we can say that there are two broad categories. First of all there are approaches based on logic programming, in particular Constraint Logic Programming (CLP) [JL87,Cohe90]. Second there is a number of (numerical) incremental constraint solvers that have primarily been applied in the context of graphical user interfaces [FBMB89,BB98].

Before we continue, we want to introduce some terminology regarding constraint solving. Constraints are grouped into constraint networks. Every variable that is constrained (plays a role in a constraint) has a domain associated with it. The domain contains the possible values for the constraint variable in the constraint network. A value is possible if it makes the constraint hold in the given constraint network (we say that the constraint is satisfied). When a constraint network is constructed, it can be solved.

The main idea is to use constraints to generate or verify a schedule for components since this is essentially a resource allocation problem: a number of non-overlapping intervals need to be distributed taking some relations between the intervals into account. Both CLP and incremental constraint solvers can be used in this context, and we have experiment with both. The incremental constraint solvers, however, have the drawback that they can not generate schedules. They can be used to verify schedules, though, and have the advantage that preferences can be put on constraints and that they can be checked incrementally.

However, since we want be able to both verify and generate schedules (using the information from the model as input), we also look at CLP based approaches. This section therefore discusses this CLP-based approach, and the implementation we did in the CLP language ECLiPSe (not to be confused with the open source development environment ECLIPSE used to integrate all the Pecos tools).

4.2 ECLiPSe

ECLiPSe is a general purpose constraint programming platform that was initiated at ECRC in Munich in conjunction with Bull, ICL and Siemens and that is now funded by Parc Technologies Limited (a spin-off of Imperial College). It has been used to build and deliver large scale commercial applications e.g. network planning, scheduling and resource allocation.

\(^1\) We also performed experiments with time Petri Nets, but the results were inconclusive.
In Pecos we use ECLiPSe to input a component model and timing bundles, and optionally a schedule, and generate or verify a schedule. Consider for example the following simple model that shows a field device consisting of three components (including one composite component PA), with the timing bundles attached to each component:

- NV (Active)
  - sync( wcet : 10 ),
  - exec( wcet: 30,
          cycletime: 1000)

- PA (Passive)
  - wcet: 11,
  - order: QA, QB

- HMI (Event)
  - sync( wcet : 2,
            cycletime: 20 ),
  - exec( wcet: 10 )

- QA (Passive)
  - wcet: 2,
  - cycletime: 16

- QB (Passive)
  - wcet: 4,
  - cycletime: 30

Then we want ECLiPSe to generate us the following schedule, that shows that there are three tasks needed for this field device, and shows for each tasks the order in which to execute the behavior of components:

- activity(QA, 0, 2, 2, 1)
- activity(PA, 2, 11, 13, 1)
- activity(HMI-sync, 14, 2, 16, 1)
- activity(QA-2, 16, 2, 18, 1)
- activity(QB, 20, 4, 24, 1)
- activity(QA-3, 32, 2, 34, 1)
- activity(HMI-sync-2, 34, 2, 36, 1)
- activity(NV-sync, 36, 10, 46, 1)
- activity(QA-4, 48, 2, 50, 1)
- activity(QB-2, 50, 4, 54, 1)
- activity(HMI-sync-3, 54, 2, 56, 1)
- activity(NV-exec, 0, 30, 30, 2)
- activity(HMI-exec, 0, 10, 10, 3)

To attain this goal, we expressed the problem in a constraint form, where the answer was a schedule. More specifically, we want the solver to find start and end times for activities in such a way that:
- the activities within a task do not overlap,
- the cycle times and priorities are taken into account, and
- the partial order of the activities as specified in composite components is satisfied.

When no single distribution of activities can be found that satisfies these criteria, then no schedule can be found. If a distribution is given (in the form of a schedule), then it is checked to see that it satisfies these requirements. When nothing is given, the constraint solver gives all possible distributions that satisfy the criteria. Every single solution is a possible schedule.

---

2 This will be integrated in the Pecos tool chain, so that the developer only sees Coco models and Coco schedules, while the generation of verification is done by ECLiPSe under the hood.
We do not want to discuss the ECLiPSe implementation we made to verify and generate schedules, since this is outside the scope of this report. Instead we will highlight some key points when expressing the model under the form of constraints.

Basically, the implementation needs only the hierarchical component structure and the timing information for each component. The component structure is traversed recursively. For an active component, we add as many activities to the task of the parent as needed to satisfy the cycle time. For every active component we add a task (for the exec and maybe its subcomponents), activities for its sync behavior (depending on the sync cycle time given) and activities for the exec behavior (again as much as needed to satisfy the cycle time). Then the children are recursively enumerated.

Note that while enumerating the component structure we do nothing but adding constraint variables to the network (for starting times, ending times and tasks for activities), and constraints between them. For example, in the example above we specified that the QB component has a cycle time of 30. This means that two activities have to be added for this component when it is used in a device with a period of 60 (as in the example). The constraint between these activities is that their start times have a difference of 20. Likewise we add constraints to express that the start + wcet = end, and to express the partial orders. When the device is completely enumerated, we have a constraint network that we then ask the constraint solver to solve. The result is a schedule as shown: a number of tasks, with for each task a number of non-overlapping activities that satisfy the given constraints.

4.3 Conclusion

A constraint solver is used to verify or generate schedules, taking partial orders and other constraints in account. The result of using the constraint solver is a number of tasks, where each tasks contains an ordered set of components with indicative (static) starting times. This assures that the component model given can indeed be scheduled, in the sense that a non-overlapping distribution of activities can be found that satisfies the order given.

On the other hand, the timing analysis that is performed by the constraint solver is extremely rudimentary. It merely checks whether the total sum of worst-case execution times does not exceed the total time that is available. The RMA timing analysis that is also discussed in this report is a far more sophisticated approach, and should be used for verifying whether the component behavior will meet their deadlines. This is explained in the next section.

5 Rate Monotonic Analysis

Rate monotonic analysis is a mathematical technique to prove that a certain set of tasks can meet their deadlines. In this section we give a general overview of this technique, then show how we can apply RMA analysis on Pecos component models. We illustrate this with a concrete example.

5.1 Introduction

In developing real-time systems, response times are important. Rate Monotonic Analysis (RMA) is a simple, practical, mathematical technique to ensure that all timing requirements will be met. RMA originated with Rate Monotonic Scheduling (RMS) theory [Liu73]. It provides a collection of quantitative methods and algorithms to specify, understand, analyze, and predict the timing behavior of real-time software systems, thus improving the system dependability and evolution.

Rate Monotonic Scheduling (RMS) theory first appeared in 1973, when Liu and Layland wrote a paper on a scheduling theory that provided mathematical conditions for assessing the schedulability of a set of tasks. These results could be used only if all tasks were periodic, preemtible, and independent. The term "Rate Monotonic" originated as a name for the optimal task priority assignment in which higher priorities are assigned to tasks that execute at higher rates (shorter period). An RMS algorithm guarantees that all the task deadlines are satisfied if and only if certain mathematical conditions hold for this set of tasks.

During the 1980s the limitations of the original RMS theory were overcome and the theory was generalized to the point of being more practicable for a large range of realistic situations encountered in the design and analysis of real-time systems [Sha91a]. RMA can be used for prediction, and verification of a set of:

- Independent periodic tasks,
- Both aperiodic and periodic tasks, and
• Interacting tasks (or synchronised tasks)

The next section gives an overview about how RMA is used in real-time systems, and provides a series of theorems for assessing the schedulability of tasks. Then we show how these theorems can be applied in the context of the PECOS project by demonstrating a concrete example from the PECOS case study.

5.2 Overview of RMA algorithms

This section describes the RMA algorithms that can be used to check whether a certain set of tasks can meet its deadlines. We start with the simple case of independent periodic tasks, then include aperiodic tasks by mapping them to periodic tasks, and finally look at interacting tasks. We then give an example where we apply the theorems. In the next section we see how to apply RMA in the context of Pecos.

5.2.1 Independent periodic tasks

<table>
<thead>
<tr>
<th>RMA Symbol</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>worst-case execution time</td>
</tr>
<tr>
<td>T</td>
<td>Period</td>
</tr>
<tr>
<td>B</td>
<td>Maximum blocking time</td>
</tr>
</tbody>
</table>

Table 1: RMA symbols

Throughout RMA certain symbols are used for the tasks that are modelled. We have enumerated them in Table 1.

A set of independent periodic tasks is said to be schedulable (all task deadlines are met) by the RMA algorithm, which gives each task a fixed priority and assigns higher priorities to tasks with shorter periods, if the condition of Theorem 1 is met (sufficient condition).

**Theorem 1:** A set of \( n \) independent periodic tasks scheduled by the rate monotonic algorithm will always meet its deadlines, for all task phasings, if

\[
\frac{C_1}{T_1} + \cdots + \frac{C_n}{T_n} \leq U(n)
\]

where \( C_i \) and \( T_i \) are the execution time and period of task \( t_i \) respectively and

\[
U(n) = n \left( \frac{1}{2^n} - 1 \right)
\]

Theorem 1 is very conservative, meaning that it will declare some schedules as not feasible when actually they are feasible. Therefore this algorithm was further refined to one that is less conservative while still guaranteeing schedule feasibility.

**Theorem 2:** A set of \( n \) independent periodic tasks scheduled by the rate monotonic algorithm will always meet its deadlines, for all task phasings, if and only if

\[
\forall i, 1 \leq i \leq n, \min_{R_i} \left( \sum_{j=1}^{l} \frac{C_j}{T_j} \right) \left \lfloor \frac{IT_k}{T_j} \right \rfloor \leq 1
\]

where \( C_j \) and \( T_j \) are the execution time and period of task \( t_j \) respectively and

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

5.2.2 Periodic and aperiodic tasks

Most applications do not consist solely of periodic tasks, but also contain some aperiodic tasks, for example to handle input events. RMA guarantees the response time for an aperiodic task. To analyse
systems that contain aperiodic tasks, aperiodic events have to be cast into the periodic framework. This can be done either by exploiting natural limits on the aperiodic arrival pattern (such as a known minimum interarrival interval), or by using aperiodic server algorithms (such as the sporadic server algorithm). Detailed information can be found in Chapter 5, “Handling Aperiodic Events” page 5-61 [Klein 93], or in other RMA handbooks.

### 5.2.3 Interacting tasks (or synchronized tasks)

Previous sections discuss tasks that are completely independent. In this section we discuss tasks that share resources, and where synchronization is needed to protect these resources. Real time systems must be built in a way that ensures that high priority tasks are minimally delayed by lower priority tasks when both are contending for the same resources. However, when shared resources come into play, situations can occur where a task has to wait for the finishing of a lower priority task. For example, suppose that there are two tasks (T₁ and T₂), that the priority of T₁ is lower than the priority of T₂, and that during their execution both T₁ and T₂ need access to a shared resource that be locked by a semaphore. In this setup, whenever T₁ executes and uses the semaphore to lock the shared resource, the higher priority task T₂ has to wait for T₁ 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.

In order to take blocking into account with Theorems 1 and 2, the blocking needs to be known. Hence the scheduler should ensure that unbounded priority inversion does not occur. Note that the **Priority Inheritance Protocol (PIP)**, that is widely used, does not guarantee bounded blocking. More sophisticated real-time synchronization protocols need to be used, like the **Priority Ceiling Protocol (PCP)** or **Highest Locker Protocol (HLP)**.

The **priority ceiling protocol** is a real-time synchronization protocol with 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.

To achieve the priority ceiling protocol, we define the **priority ceiling** of a binary semaphore S to be the highest priority of all tasks that may lock S. When a task T attempts to execute one of its critical sections, it will be suspended unless its priority is higher than the priority ceilings of all semaphores currently locked by tasks other than T. If task T is unable to enter its critical section for this reason, the task that holds the lock on the semaphore with the highest priority ceiling is said to be blocking T and inherits the priority of T. As long as a task T is not attempting to enter one of its critical sections, it will pre-empt every task that has a lower priority [Sha90,BR99].

The differences between the highest locker protocol and the priority ceiling protocol are that a task currently locking a resource cannot be pre-empted by a task potentially concurrent for the same resource. The result is a protocol that is much simpler to implement. More information can be found in [BR99].

When PCP or HLP are used, we can refine the existing theorems 1 and 2 to take the blocking times into account.

**Theorem 3**: A set of \( n \) periodic tasks using an appropriate real-time synchronization protocol (like PCP or HLP) can be scheduled by the Rate Monotonic algorithm, for all task phasings, if the following condition is satisfied:

\[
\frac{C_i}{T_i} + \ldots + \frac{C_n}{T_n} + \max \left( \frac{B_i}{T_i} + \ldots + \frac{B_{n-1}}{T_{n-1}} \right) \leq U(n)
\]

where \( C_i \), \( T_i \), and \( R_i \) are defined as in Theorem 2, and \( B_i \) is the worst-case blocking time for \( T_i \).

**Theorem 4**: A set of \( n \) periodic tasks using an appropriate real-time synchronization protocol (like PCP or HLP) will always meet its deadlines, for all task phasings, if and only if

\[
\forall i, 1 \leq i \leq n, \min \sum_{(k,j) \in R_i} C_j \left\lfloor \frac{I_{T_k}}{T_j} \right\rfloor + C_i + B_i \leq I_{T_k}
\]
where $C_j$, $T_j$ and $R_i$ are defined as in Theorem 2, and $B_i$ is the worst-case blocking time for $t_i$.

### 5.3 Applying RMA Analysis on Pecos Component Models

In this section we map Pecos component models to the information needed by RMA. The goal is to be able to use RMA techniques to make sure that all components can meet their deadlines.

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 task $T_{\text{sync}}$ for the sync part and a task $T_{\text{exec}}$ for the exec part. The worst-case execution time, period and deadline of $T_{\text{sync}}$ and $T_{\text{exec}}$ are given by the sync and exec part of the timing bundle of $A$.

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, we choose arbitrary priorities so that the tasks can be ordered according their priority. So when mapping the model to RMA this has to be taken into account: the timing bundles have to be processed to make sure that these conditions hold. When the priorities are given, we can start to think about the blocking times.

Remember that the execution thread in an active or event component uses its own private datastore that gets synchronized with the surrounding data store when the sync is run. Remember also that active and event components are responsible for a thread in which they sequentially execute the behavior from their sync behavior and for behaviors defined by their subcomponents (if any). Hence, all behavior scheduled by an active or an event component uses the datastore in that active component. From this explanation follows immediately that the datastore is shared by the exec and the behavior defined in that component, but that at all times at most two tasks can attempt to access the datastore simultaneously (the exec and one behavior in the active component). The result is that unbounded blocking cannot occur, since at most two tasks are involved (at most one can block the other). The general drawback of RMA that it needs a more sophisticated task switching protocol (like priority ceiling protocol or highest locker protocol) is thus not needed when analyzing a Pecos model. This means that we can use a simpler task switching protocol (priority inheritance protocol, in our case), that is readily supported by most real-time operating systems and is less costly in terms of performance and overhead.

The only question we have left to answer is what the maximum blocking times are for those tasks that do not explicitly specify them. This depends on the priorities of the involved behavior (note that the priorities can not be equal, as explained before). Assume the following setup: $A$ is a composite active or event component that has $n$ subcomponents. The sync behavior of $A$ is called $A_{\text{sync}}$. The exec behavior is called $A_{\text{exec}}$. Every subcomponent has behavior that is scheduled by $A$. We call this behavior $S_1$ to $S_m$, where $m > n$ (since some of the subcomponents can be composite components). Now call $S$ the set consisting of $A_{\text{sync}}$ and $S_1$ to $S_m$. The datastore can thus be accessed simultaneously by $A_{\text{exec}}$ and one element of $S$. We are now interested in the blocking that can occur when this happens. Therefore we split $S$ in two subsets: $L$ is the subset $S$ where the elements have a priority that is lower than the priority of $A_{\text{exec}}$; $H$ is the subset of elements that have a higher priority. With this division we can now set the worst-case blocking times for those elements that do not explicitly provide values:

- Elements in $L$ are the ones that can block $A_{\text{exec}}$. We then set the worst-case blocking time of each element in $L$ that does not specify an explicit value to be either the blocking time of $\text{exec}$ (when it is given) or the execution time of $\text{exec}$.
- Elements in $H$ are the ones that can be blocked by $A_{\text{exec}}$. We then set the worst-case blocking time of $A_{\text{exec}}$ to the maximum of worst-case execution times and explicit blocking times of the elements in $H$.

When applying theorem 4, 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).

---

3 The behaviour of active sub components, the syncs of active components and this recursively downwards.
5.4 Example of RMA Analysis

Consider the following example field device that is concerned with setting a valve at a specific position between open and closed. Figure 5 illustrates three connected Pecos components that collaborate to set the valve position. A control loop is used to continuously monitor and adjust the valve. Three components play a role in this simple example:

- The ModBus component provides a software interface 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 constitutes the control loop.

<table>
<thead>
<tr>
<th>ProcessApplication</th>
<th>timingBundle( wcet: 10 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>ModBus</td>
<td>timingBundle( sync( wcet: 5, blocking: 5), exec( wcet: 20, cycletime: 500, priority: 1) ).</td>
</tr>
<tr>
<td>FQD</td>
<td>timingBundle( sync( wcet: 10), exec( wcet: 15, cycletime: 30, priority: 3) ).</td>
</tr>
<tr>
<td>Field device</td>
<td>timingBundle( cycletime: 60, defaultPriority: 2, order: [FQD, 20, ProcessApplication, 10, ModBus]).</td>
</tr>
</tbody>
</table>

Table 2: Timing Bundles for RMA analysis example

The timing bundles for this example are given in Table 2.

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 we assume a scheduler that uses the priority inheritance protocol.

The difficult part of applying RMA lies in determining the specification parts of the servers that are needed to model the execution part of active and events components. In the given mapping, 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).

Then we need to assign the maximum blocking times. The ModBus active component has a sync and an exec task, which share a common resource (the internal data store, as described earlier). Since no explicit blocking time is specified for exec in the timing bundles, we set it to 0 (since it has the lower priority of the two). The maximum-blocking time for sync was given in the timing bundle (it is 5). For the FQD component, no blocking times are specified. Hence we give the worst-case blocking time for the exec task the value of 10 (the worst-case execution time of the sync task), and for the sync task the value of 0.

When we then order the tasks from the one with the highest priority to the lowest priority, we obtain the following table that is used for the actual analysis:

<table>
<thead>
<tr>
<th>Component</th>
<th>Task</th>
<th>C</th>
<th>T</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>FQD (exec part)</td>
<td>T1</td>
<td>15</td>
<td>30</td>
<td>10</td>
</tr>
<tr>
<td>FQD (sync part)</td>
<td>T2</td>
<td>10</td>
<td>60</td>
<td>0</td>
</tr>
<tr>
<td>ProcessApplication</td>
<td>T3</td>
<td>10</td>
<td>60</td>
<td>0</td>
</tr>
<tr>
<td>ModBus (sync part)</td>
<td>T4</td>
<td>5</td>
<td>60</td>
<td>5</td>
</tr>
<tr>
<td>ModBus (exec part)</td>
<td>T5</td>
<td>20</td>
<td>500</td>
<td>0</td>
</tr>
</tbody>
</table>

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.

\[
i = 1, k = 1, l = 1: C_1 + B_1 = 20 + 10 = 30 (ok)
\]
\[
i = 2, k = 1, l = 1: C_1 + C_2 + B_2 = 20 + 10 + 0 = 30 (ok)
\]
\[
i = 3, k = 1, l = 2: C_1 + C_2 + C_3 + B_3 = 15 + 10 + 10 + 0 = 35 > 30
\]
\[
i = 4, k = 1, l = 1: C_1 + C_2 + C_3 + C_4 + B_4 = 15 + 10 + 10 + 5 + 5 = 45 > 30
\]
\[
i = 5, k = 1, l = 1: C_1 + C_2 + C_3 + C_4 + C_5 + B_5 = 15 + 10 + 10 + 5 + 20 + 0 = 60 > 30
\]
\[
\cdots
\]
\[
i = 5, k = 2, l = 5: 10C_1 + 5C_2 + 5C_3 + 5C_4 + C_5 + B_5 = 150 + 50 + 50 + 25 + 20 = 295 (ok)
\]

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

5.5 Conclusion

RMA analysis can be used to check whether tasks can meet their deadlines. The mapping from the model to RMA is not very complicated, and hence this technique seems to be really useful in the context of Pecos. It allows us to check whether the timing information specified for a certain set of components in a field device is feasible. Moreover, the constraints imposed by the model allow using RMA together with simple task switching protocols. However, we see some problems that might hinder the practical adoption of this technique:

1. RMA has an assumption that the scheduler decides at all time which task has to be run next. The RMA mathematical proofs of the Theorems rely on this assumption (and, for the more advanced theorems 2 and 4, even on the protocol used to determine the next task to be run). However, sometimes it might be needed to stipulate that a certain component has to be run at a specific time. This breaks the assumptions that are made, since it means that there is a moment where a certain task needs to be run and where the scheduler is not free to choose what task to run. We are currently looking for extensions or alternatives for RMA that allow incorporating this in the analysis, but we have not found any to date. The current solution, shown in the next section, is to combine RMA with the constraint solver.

2. Applying RMA is fairly straightforward, except for the mapping of aperiodic services to periodic services. In the case of Pecos, this means that the behavior in the execution parts of
active and event components needs to be represented by a periodic task. This could pose problems, although after discussing with domain experts it seems that the so-called sporadic server (a well-known RMA concept to model aperiodic tasks) is applicable to most of the cases. In that case we can make that the default mapping and assume that the developer does not overload the system with aperiodic events. If not, then other servers should be investigated by the responsible of the component in standard RMA literature to find an appropriate one.

6. Combining RMA and Constraint Verification

RMA is a very good technique to do a pure verification of the timing issues for a component model. The constraint solver is very good at making sure that schedules can actually be found that take the order and possibly explicit starting times (as well as other user-defined constraints) into account. Hence, to do a timing check of a component model both techniques are needed. In this section we give examples on applying both on some examples to clearly demonstrate the synergy of combining both approaches.

6.1 Example 1

Here we want to make clear that only using the constraint solver is not enough. Consider the following example of a simple device consisting of two components.

<table>
<thead>
<tr>
<th>Component</th>
<th>wcet</th>
<th>cycletime</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>A (Passive)</td>
<td>6</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>B (Active)</td>
<td></td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

sync( wcet: 4, cycletime: 10),
exec( wcet: 10, priority: 1)

When giving this to the constraint solver, it comes up with the following solution (it actually finds far more solutions, but only permutatioons of this one):

![Figure 6: Example 1 Checked In YASA – RMA tool](image)

This is correct: this solution indeed satisfies the constraints given. The problem is that this clearly will not run from a timing analysis point of view. The thread with A and B.sync consumes all the available time and does not leave any time for B.exec to be executed. But since the constraint solver does not perform a real timing analysis, this problem is not catched by it. Running an RMA analysis on this model clearly identifies the problem.

6.2 Example 2

This example shows that only making an RMA analysis is not enough. Consider the following example of a simple device consisting of two passive components.
An RMA analysis for this example finds that everything is schedulable, so we can be sure that the
timing requirements are met. However, when giving this to the constraint solver, it can not find a single
schedule for this example. The reason very simply is that because A very frequently uses small
executions of 2 milliseconds, it only leaves gaps of 6 milliseconds. Hence B, that needs a single
continues timeslice of 7 milliseconds, can never fit this schedule. This example shows how only doing
an RMA analysis is not enough to verify a model.

Note that the solution to the problem given here would be to make A an active component, since it
executes so frequently. Then B has more than time enough to execute its behavior. It will just be	imesliced from time to time by A.exec.

6.3 Example 3

We now illustrate a counter example which can be checked by YASA.

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>C1 = 2, T1 = 4, B1 = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>T2</td>
<td>C1 = 6, T2 = 32, B2 = 0</td>
</tr>
<tr>
<td>B</td>
<td>T3</td>
<td>C1 = 12, T3 = 32, B3 = 0</td>
</tr>
</tbody>
</table>

Table 3: Timing bundles of a counter example

There are three independent and periodic tasks A, B, C with timing bundles specified in Table 3
above.
Analysis result shows that task C deadline can not be met by a red airwards arrow in deadline checkpoint (32ms).

7. Memory Bundles and Memory Consumption Verification

As for timing information, Component Model-V specifies that information regarding memory consumption can be specified in the memory bundle. This section outlines the contents of these memory bundles, as well as how this information can be used to verify the memory consumption for components.

7.1 Memory Bundle Contents

When looking at the memory consumption that needs to be taken into account, then we first of all have to divide two kinds of data that can be used by components. First of all we have the ports that indicate data that is being shared by different components. In the RTE these values are put in tables (called datastores) that are referenced from the components. Then we also have data that is used internally in a component (in regular variables in the C++ or Java code implementing the actual behavior). From the memory verification point of view, there is an important difference between those two kinds of data: the size of the datastores can be extracted easily, while the internal variables have to be measured (or calculated) and specified by the programmer in the memory bundle.

More specifically, the memory bundle of a component consists of the following two pieces of information that have to be specified:

1. stacksize: the static size that is needed by the component on the stack.
2. dynamic allocation size: the amount that is needed by internal variables that are allocated at runtime.

7.2 Memory Verification

With the generated code for a component model and the memory bundles for each component, the total memory consumption of a field device can be calculated by summing up the sizes of the datastores with the stacksizes and the dynamic allocation sizes. This number then has to fit the memory of the device.

8. Conclusion

In this report we described the format of the timing bundles that are used to specify timing information in the model. Then we investigated two techniques (constraint solving and RMA) to verify schedules. RMA allows us to do a mathematical analysis that assures that the timing requirements are met. The constraint solver allows one to generate or check schedules. Combining these two techniques allows to check the timing and scheduling properties of a given component model. We describe the implementation of the schedule generation/verification tool in the constraint logic programming language ECLiPSe, and we show the RMA tool Yasa (YetAnother Schedule Analyzer).

Besides the timing and scheduling analysis we also outline the support for checking the memory consumption. We describe the contents of the memory property bundle, and the verification that should be done with this.

9. References


