Predicting Real-Time Properties of Component Assemblies: a Scenario-Simulation Approach

Egor Bondarev[1,2], Johan Muskens[1], Peter de With[2], Michel Chaudron[1], Johan Lukkien[1]

System Architecture and Networking[1] and Video, Coding and Architectures[2] groups
Eindhoven University of Technology, P.O. Box 513,
5600 MB Eindhoven, The Netherlands
E.Bondarev@tue.nl

Abstract

This paper addresses the problem of predicting timing properties of multi-tasking component assemblies during the design phase. For real-time applications, it is of vital importance to guarantee that the timing requirements of an assembly will be met. We propose a simulation-based approach for predicting the real-time behaviour of an assembly based on models of its constituent components.

Our approach extends the scenario-based method in [2] by offering a system model that is tailored to the domain of real-time applications. Contributions of this paper include the possibility to handle the following features: mutual exclusion, combinations of aperiodic and periodic tasks and synchronization constraints. The analytical approach we used in previous work cannot handle these features. Therefore, we introduce the simulation-based approach.

Our simulator provides data about dynamic resource consumption and real-time properties like response time, blocking time and number of missed deadlines per task. We have validated our approach using a video-decoder application.

1. Introduction

High-volume embedded appliances such as PDAs, mobile phones, set-top boxes and DVD-players have a high evolution rate. Strong competition in this domain makes a rapid time to market as well as low development cost vital. Typical applications for these systems have real-time constraints, e.g., multimedia applications. In combination with the scarce resources these devices have, this leads to a major challenge for product designers, even more while the functions of these devices are always expanding.

For decreasing time to market and supporting this evolution an open, component-based software architecture for a middleware layer has been proposed (in ITEA project Robocop [3]). The research described in this paper was conducted in ITEA project Space4U that uses this architecture as a starting point. It is aimed at improving the Robocop architecture by developing a framework that allows prediction of real-time properties and resource consumption of an application based on the components from which it is composed.

Component-based technology complicates the prediction of resource consumption and timing properties of an application. In component-based systems, the actual behaviour and resource consumption is determined by a collection of internally developed and third-party components. Thus, the prediction task becomes twofold. First, find and express the component's extra-functional properties. Second, combine these properties in order to predict the behaviour of the composition of the constituent components. In the sequel, we will denote an application also as an *assembly*, because it makes use of the underlying components.

The challenge of predictable component assembly is of significant interest because of the rapid development of component-based software. Therefore, we present a brief literature overview. Some approaches give an engineering practice [7-9] to the problem. A very interesting technique that allows design-time estimations of real-time properties of component-based systems is presented in [10]. In this technique, many possible types of software constructions are taken into account, like synchronous and asynchronous communication, as well as synchronization constraints.

Recent work on the prediction of performance for evolving architectures is described in [11]. This approach is based on collecting the component performance data on different platforms and interpolating it for new components or platforms. Real-time frameworks have been introduced in the object-oriented development field. Methods have emerged that enable execution of UML-like specifications, notably Room [12] and Rhapsody [13]. The PRIMA-UML methodology [14] applies queuing networks and extends UML with a real-time performance model for system performance validation. The scenario-based approach [15] involves estimating *static* resource consumption of a component assembly.

In contrast, through our scenario simulation approach we address a *dynamic* instead of static resource consumption, thereby giving more accuracy in the prediction of the assembly behaviour. With respect to *task synchronization,* we adopt the use of synchronization constraints for further adding accuracy in the prediction. The approach still requires little effort from an application developer, because the introduction of application scenarios narrows the state-space and behavior of an application that the developer should model and simulate.

This paper is structured as follows. Section 2 presents the Robocop component model, as a starting point. Section 3 gives the definitions of the timing properties, tasks and synchronization constraints. Section 4 outlines the proposed approach. In Section 5 we discuss the workflow of the approach. Section 6 specifies models needed for simulation and effective prediction. Section 7 describes the simulation and schedulability analysis part of the prediction technique. Section 8 concludes with the benefits and drawbacks of the proposed prediction approach.

2. Robocop component model

In this section we discuss the Robocop component model. The model is inspired by existing component-based architectures, such as COM [4], CORBA [5], and Koala [6]. A Robocop component is a set of possibly related models *M* (see Figure 1). Each individual model *m* provides a particular type of information about the component. Models can be represented in human-readable form (e.g. documentation) or in binary code. One of the models is the *executable model* that contains the executable component. Other examples of models are: *resource model*, *functional model*, and *behavior model*.
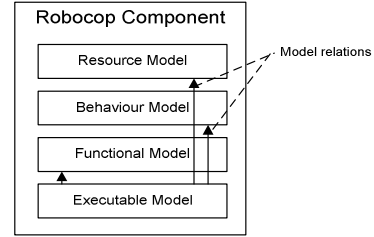

Figure 1. Example Robocop component model

A component offers functionality through a set of 'services' *P* (see Figure 2). Services are static entities, which are the Robocop equivalents of *public classes* in object-oriented (OO) programming languages. More formally, we can specify an arbitrary executable model *m* by:

m   = *P,*
      where *m* is an *Executable Model* and
      P is a set of *p* (services).

Services are instantiated at run-time, using a service manager. The resulting entity is called 'service instance', which is a Robocop equivalent of an *object* in OO programming languages.
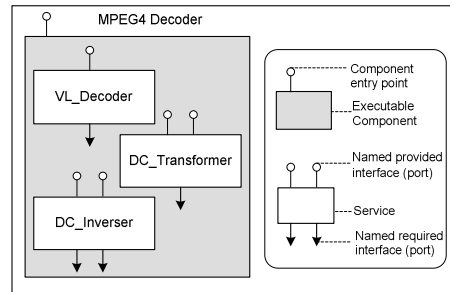

Figure 2. Example of executable component representation.

A Robocop service may define several interfaces (ports). We distinguish a set of 'provides' ports *PR* and a set of 'requires' ports *REQ*. The former defines interfaces that are offered by the service, while the latter defines interfaces that the service needs from other services in order to operate properly. An interface is defined as a set of implemented operations *impl_opr*. A service *p* being part of the above-mentioned executable model is specified by:

p   = *(PR, REQ),*
      where *PR* is a set of *pr* (provided ports) and
      REQ is a set of *req* (required ports),
pr   = (*name, interface*),
req   = (*name, interface*),
interface = *O,*
      where *O* is a set of *impl_opr* (impl'ted operations).

Please note that a *Robocop service* is an equivalent to a *component* in COM or CORBA, i.e. a service is a subject of composition, and it has input and output ports. A *Robocop component* is a deployable container that packages these services. Therefore, in the Robocop context the term *composition* means a composition of services.

In Robocop, as well as in other component models, service interfaces and service implementations are separated to support "plug-compatibility". This allows different services that implement the same interface to be replaced. As a consequence, the actual service implementations to which a service will be bound do not need to be known at the time of designing that service. This implies that resource consumption cannot be completely determined for an operation, until an application binds the required interfaces of the instance of the service to provided interfaces.

The Robocop architecture implies no implementation-level constraints. A service can implement any number of threads. Beside this, synchronous and asynchronous communication types are possible.

3. Timing properties, tasks and synchronization

This section defines the basic terms used in the paper, e.g. timing property, task and task synchronization constraints. There is a clear difference between the component and application timing properties. The *component timing properties* are independent from system run-time execution and scheduling. In most of the cases, these properties are: worst-case, mean-case and best-case execution times per operation.

The *application timing properties*, instead, are closely coupled to run-time instances, tasks and scheduling algorithms used in the system. We concentrate on the following timing properties: response and blocking times of a task as well as the number of missed deadlines of a task.

The response time of a task is not just a sum of execution times of the operations comprising the task. Usually, the response time is composed of the execution time, blocking time and preemption time of the task. Therefore, for the assembly timing property, the task synchronization and scheduling aspects should be considered.

In out context, the *task* is an event-triggered sequence of executed operations. The operations composing a sequence may be implemented by different services. The operations within the sequence

may be called synchronously and asynchronously. The tasks my have *synchronization constraints* between them, e.g. precedence, rendezvous and mutual exclusion. Usually, the system resource sharing imposes these constraints.

4. Scenario simulation approach overview

Our approach proposes to combine the behaviour and resource consumption models of used components with an application model constructed for possible critical execution scenarios. The *component behaviour model* describes behaviour of each operation implemented by the component. The *component resource model* describes resource usage of an operation implemented by the component. The *application scenario model* defines the static structure, internal and external events of the application for critical scenarios. *Critical scenarios* are the scenarios that may introduce CPU or memory overload. The resulting set of models serves as an input for virtual scheduling (simulation). The simulation output data shows execution behaviour of the assembly tasks and timing properties of those tasks, i.e. predicted execution timeline, latency and resource utilization of each task.

Summarizing, the key features of our approach are:
a) Predictions on timing properties are made by simulation at an early stage of development.
b) It avoids combinatorial complexity of full state-space analysis of a system by usage of scenarios.
c) It takes task synchronization and scheduling aspects into account.

The following section describes *how* the approach should be implemented.

5. Predictable composition workflow

5.1. Workflow and its assumptions

The main objective of an assembly developer is, given a set of available components and requirements for an assembly, to embed the components in the assembly satisfying the given requirements.

Addressing this objective, the proposed composition workflow contains principal steps for the assembly developer. These steps lead to prediction of resource consumption, timing properties of an assembly and a validation of the extra-functional requirements (see Figure 3). These results are obtained already in the design phase, prior to running the assembly on the target device.
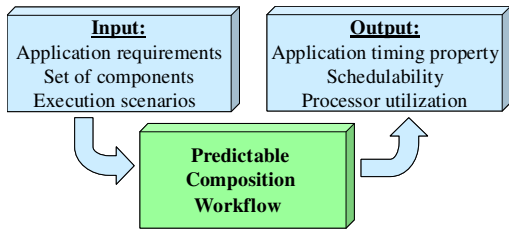
Figure 3. Conceptual view on the prediction-enabling composition workflow.

The workflow works is based on two *assumptions*:
a) Resource usage property and behaviour of the constituent components are specified and available in the corresponding component models.
b) An application developer is able to find out critical scenarios of the application.

The remainder of this section defines consecutive steps of the workflow.

5.2. Primary steps of the workflow

Component selection

When having a set of available components, a developer selects and composes them into an assembly, in order to meet the functional and extra-functional requirements. According to the first assumption, the selected components should have a *resource model* and *behaviour model*. The models are described in a modeling language that has all necessary primitives for describing resource consumption and behaviour of operations implemented by the component. These two models are used to accompany an *application scenario model* that is constructed in the next step and, thus, complete the mosaic of the application behaviour. The description of the models and further reference to Figure 4 are given in the following text and Section 6.
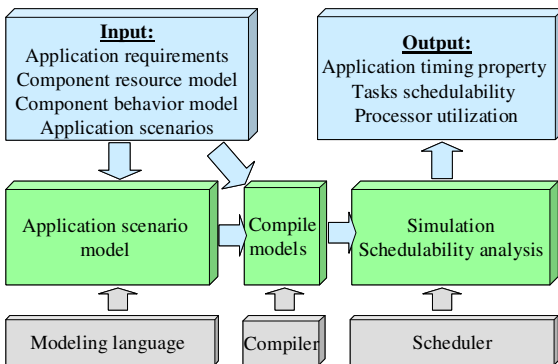
Figure 4. Main steps in the predictable RT composition workflow.

Application scenario model construction

For each critical or commonly used scenario, a developer constructs an *application scenario model*. The application scenario model consists of two parts: (a) description of service instances and bindings between them, particular for the selected scenario, and (b) description of the application-level events and active threads that trigger execution of operations of the service instances. Afterwards, the application scenario model is added to the model set for further compilation. Multiple scenario models are possible, depending on the number of found critical scenarios.

Compilation of the models

The *application scenario*, *component resource* and *component behaviour models* are compiled for further schedulability analysis. After this step, the model data is prepared for simulation (virtual scheduling).

Simulation and schedulability analysis

An application developer applies a scheduler to the compiled model set in order to simulate the execution of the scenario specified in the related model. The simulation data (CPU timeline of tasks execution) is used for schedulability analysis. The analysis helps to reason about application timing properties like response time, latency of critical tasks, overall schedulability and processor utilization bounds.

The predicted timing properties are checked against the real-time requirements of an application. For example, worst-case response time of a critical task is checked against the deadline mentioned in the requirements. If one of the requirements is not met, a developer optimizes the composition and repeats the workflow until all requirements are satisfied.

6. Models

The purpose of this section is to specify the models introduced in the previous section. It is emphasized here that the models are not a goal by themselves, but are required for obtaining the resource consumption and timing properties.

A real-time system model should be precise and detailed enough to provide data for schedulability analysis. On the other hand, the model should be kept simple for construction and understanding. This trade off brings the challenge of carefully selecting the description data relevant for the model, and afterwards representing that data, while avoiding complications.

According to Figure 4, we propose to model application scenarios. This allows decomposing each type of application behavior into a separate simple

scenario model. Thus, we can reduce the complexity of the complete behavioral model of the application and partly avoid exploration of all application states.

The following sub-sections specify the above-mentioned models in detail.

6.1. Component resource model

The *component resource model* (RM) is one of the models of the Robocop component model. RM specifies the predicted resource consumption for all the operations *impl_opr* implemented by services of an executable component. Resources (r) can be memory, CPU, etc. The predicted resource consumption is specified as a (claim, release) tuple for non-processing resources, like memory. For processing resources, like the CPU, the consumption is specified as a single claim.

m = *RM,*
where *m* is a *Resource Model* and
RM is a set of *rm* (resource usage of an operation).
rm = (*impl_opr*, *resource*, *consumption*),
for operation *impl_opr*.
resource = $r \in$ {*memory*, *cpu*, *bus*, …}.
consumption = *claim,*
in case *resource* is *cpu*.
consumption = (*claim*, *release*),
in case *resource* is *memory*.
consumption = (*claim*, *time*),
in case *resource* is *bus*.

A component developer defines the resource consumption properties of an operation by worst-case analysis. These properties are calculated only for the operation body itself, excluding resource consumption properties of called operations. This approach allows calculating resource consumption of any sequences of operation calls. In this paper, we do not address platform and parametric variations of the operation resource consumption. The resource model should be specified for a particular reference platform.

6.2. Component behaviour model

The *component behaviour model* (BM) also belongs to the Robocop component model. BM specifies the behaviour of all operations *impl_opr* implemented by services of an executable component. A semi-formal specification of the model is as follows.

m = *BM,*
where *m* is a *Behaviour Model* and
BM is a set of *bm* (behaviour of an operation).
bm = (*impl_opr*, *mutexed*, *behaviour*, *T*),
where *impl_opr* is the implemented operation and
behaviour is the operation behaviour description,

T is a set of *t* (task triggers the operation is associated with),
mutexed shows if the operation is mutexed.
mutexed = $\in$ {*true*, *false*}.
behaviour = (*called_opr1*, *called_opr2*, …*called_oprn*, *CS*),
where *called_opr1*, …*called_oprn* is a sequence of called operations and
CS is a set of *cs* (critical sections).
called_opr = (*opr*, *nmb_iterations*, *calling_type*),
where *opr* is the called operation and
nmb_iterations - number of times the operation is called,
calling_type $\in$ {*synchronous*, *asynchronous*}.
cs = (*called_opr1*, *called_opr2*, …*called_oprn*).
t = (*periodicity*, *param*, *PRECED*),
where *periodicity* $\in$ {*periodic*, *sporadic*, *aperiodic*},
PRECED is a set of *preced* (preceding task triggers),
*param* includes various parameters of *t*.
param = (*period*, *interarrival_time*, *priority*, *deadline*, *offset*, *jitter*).
preced = (*t*, *ratio*),
where *t* is a task trigger that precedes the specified task trigger.
ratio = *nmb_jobs_of_current_task* / *nmb_jobs_of_preceding_task*.

Firstly, for each operation *impl_opr* implemented by an executable component, a component developer defines its mutual exclusion property. If an operation is *mutexed*, at most one thread can enter the operation at the same time. Secondly, operation *behaviour* describes a sequence of operation calls to other interfaces made inside the implemented operation. For example in Figure 5, the implemented operation `Decoder.decode()` has a behaviour described by the following call sequence: `IGetElement.getFrame()`, `IStoreElement.storeFrame()`. The `IGetElement` and `IStoreElement` are the interfaces provided by ReadBuffer and WriteBuffer services correspondingly.
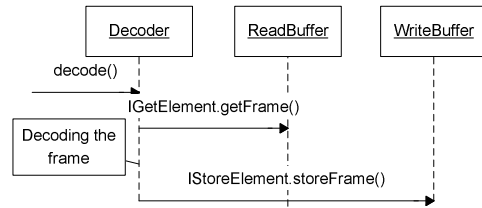
Figure 5. Sequence of operation calls (*behaviour*) of decode() operation.

For each called operation *called_opr* in the sequence, the number of iterations *nmb_iterations* and calling type *calling_type* are specified. Additionally, a set of critical sections *CS* can be specified if necessary in *behaviour*. Critical section *cs* points out the

operation of which the execution cannot be pre-empted. Please note that each *called_opr* must belong to one of the required interfaces for the service.

Finally, a component developer must define the operation autonomous behaviour *T*. We consider that an operation has autonomous behaviour if there is at least one task trigger *t* implemented by the operation. One of the examples of the task trigger is an iterative thread, triggered periodically by a timer. In the decoder example, the `decode()` operation can implement an iterative thread, which is triggered by the system timer each 20 ms. Thus, the whole calling sequence repeats each 20 ms. In the model, the task trigger properties can be specified, including *periodicity*, *period*, *deadline*, *offset*, precedence constraints *preced*, etc.

Concluding, these two models describe component resource consumption and behaviour properties independent of the application context where the component is going to be used.

6.3. Application scenario model

The *application scenario model* (SM) specifies application structure and behaviour for a critical or commonly used execution scenario. Several SMs can be built for an application, depending on a number of interesting scenarios. An application developer is in charge of the *scenario models* construction. The semi-formal structure of the model is presented below.

SM = (*appl*, *structure*, *E, T, depend*),
  where *E* is a set of *e* (event coming from outside of the *appl*),
  T is a set of *t* (task trigger the *appl* implements),
  depend is a set of components used in the *appl*.
structure = (*SI, B*),
  where *SI* is a set of *si* (service instances) and *B* is a set of *b* (bindings).
b = (*from, from port, to, to port*).
from, *to* = service instance.
from port, *to port* = port name (named interface).
e and *t* = (*opr, periodicity, param, PRECED*),
  where *opr* is an operation triggered by the *e* or *t*,
  periodicity Є {*periodic, sporadic, aperiodic*},
  PRECED is a set of *preced* (preceding *e* or *t*),
  param is number of parameters of *e* or *t*.
param = (*period, interarrival_time, priority, deadline, offset, jitter*).
preced = (*e* or *t, ratio*),
  where *e* or *t* is event or trigger which precedes the current one.
ratio = *nmb_current_events/nmb_preceding_events*.

Firstly, an application developer specifies an application *structure* for a scenario. The *structure* is represented by a tuple containing *SI* (set of service instances *si*) and *B* (set of *bindings* between the *si*). A *binding* includes information about the bound service instances *from*, and *to*, and in/out ports of the instances *from port*, *to port*. In Figure 6, dashed lines represent the bindings.
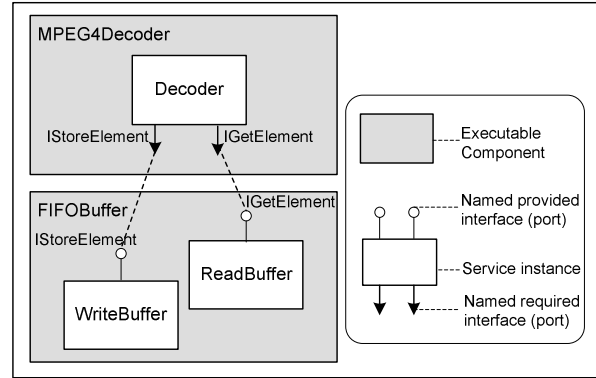


Figure 6. Example of application structure.

Secondly, the model defines the components (*depend*) used in the application. This data links the scenario model with the *behaviour* and *resource models* of the corresponding components.

Finally, the application scenario model specifies sets *E* and *T* of events *e* and in-application task triggers *t*, respectively. We define an as event any influence coming from *outside* to an application that changes the current application state. Hardware interrupt, timer or signal from an external sensor can trigger the event. Normally, this influence is expressed as a call of one of the operations of the application component.

Conceptually, an in-application task trigger is also an event, but it comes from *inside* the application. In other words, this task trigger is implemented by the application. Please recall that we also have a task trigger notion in the *component behavior model*. That task trigger differs by being implemented inside a component. The two types of task triggers are separated into different models, because an in-component task trigger must be specified by a component developer and an in-application task trigger must be specified by an application developer.

The application task trigger calls one of the operations of the application components, thereby starting the task action sequence. Therefore, the *e* and *t* must be associated with the operation called first (*opr*). In Figure 7, an application periodic task trigger calls `decode()` operation each 40 ms. Thus, in the *scenario model* the trigger should be associated with this operation.

For each event *e* as well the in-application task trigger *t*, its *periodicity*, parameters *param* and precedence constraints *preced* are specified.
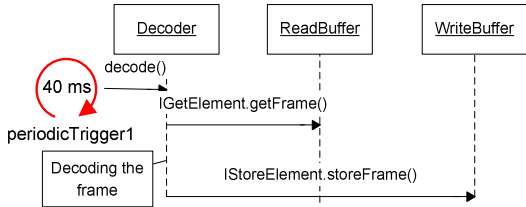
Figure 7. Task triggered by in-application trigger.

When the scenario models are ready, an application developer proceeds to the simulation phase.

7. Simulation and schedulability analysis

In the Space4U project, we have developed a Robocop Integration Environment (RIE) tool that does compilation of the above-mentioned models, simulation of an application scenario and visualization of the simulation data.

In the simulation and schedulability analysis phase, an application developer brings together the *application scenario model* and combined *behaviour-resource models* of the components deployed in the application. At this stage this stack of models can be compiled by RIE. The conceptual goal of the compilation is to identify and reconstruct a set of tasks that the application executes for a particular scenario.

The task-set reconstruction uses only the data from the three above-mentioned models. These models contain all events; in-application and in-component task triggers, as well as operation call sequences that define a flow of control for the tasks.

For the decoder example, the task reconstruction works as follows: the related *behaviour model* specifies the operation call sequence of the operation `decode()`: `getFrame()`, `storeFrame()` (see Figure 5). Afterwards, the compiler gathers from related *behavior models* the behaviour of these two operations. The operation `getFrame()` calls one operation belonging to other interfaces: `ILogData.logEvent()` (see Figure 8).
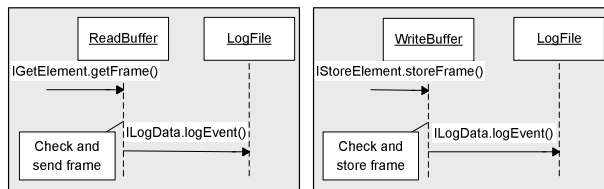
Figure 8. getFrame() and storeFrame() behaviour.

If an operation has an empty operation call sequence (does not call operations belonging to other interfaces),

it is considered as a leaf and the task generation proceeds to the next branch. Let us assume that operation `ILogData.logEvent()` is such a leaf. The next operation `storeFrame()` then also calls this leaf operation: `ILogData.logEvent()` (see Figure 8). Thus, the complete reconstructed sequence of the operations executed in the task is as depicted in Figure 9.
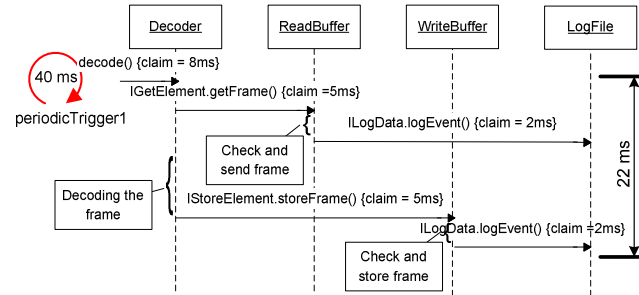

Figure 9. Task generated from the models.

A resource consumption property of each operation in this sequence is specified in the *claim* primitive in the related *component resource model* (see Section 6.1). Knowing this data, we can calculate total resource consumption of the task. For example, the CPU time used by the task (execution time) is the sum of CPU times used by the operations composing the task. In Figure 9, the total execution time of the task amounts to: 8ms + 5ms + 2ms + 5ms + 2ms = 22ms. The other task parameters (period, offset, and deadline) and precedence are obtained from corresponding task trigger properties that are specified in models of the previous section.

Synchronization constraints for each task are also extracted from the models. The task precedence has been already mentioned. *Mutexed* and critical section *cs,* which are properties of an operation, as well as a task precedence *preced* specified in the *component behavior model,* all define synchronization constraints of tasks. If a mutexed operation of the same service instance is used by two different tasks, then only one of the tasks can execute the operation at the same time.

An execution of the reconstructed tasks of the scenario is simulated by a virtual scheduler. Its scheduling algorithm should conform to the algorithm of an operating system used for deployment of this application. It can be round robin, RMA, EDF, etc. During the simulation, the specified synchronization constraints are taken into account. Therefore, the virtual scheduler should incorporate deadline prevention algorithms.

The simulation results are represented as a task execution timeline (see Figure 10).
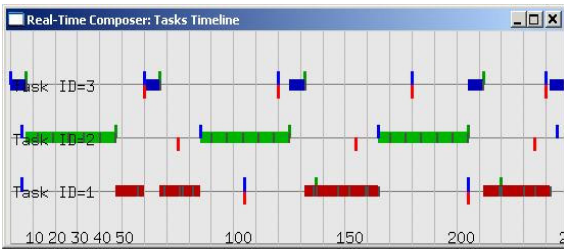
Figure 10. Task timeline execution of scenario

The schedulability analysis of the simulation data gives us the timing properties of an application. The response time, blocking time, number of missed deadlines can be found for each task. Beside this, the processor utilization bound can be analyzed per application. The predicted properties can be validated with respect to the application requirements.

8. Conclusions

We have extended the scenario-based approach for predicting resource usage of component-based systems in [2] with the specifications of *task synchronization*, *component behaviour model* and *application scenario model*. This allows simulation of the real-time task execution per application scenario and handling of synchronization constraints. Based on the simulation results, a developer can derive the behavior and dynamic resource consumption of an application per scenario. Afterwards, a developer uses this data for prediction of the real-time properties of an application. The method was validated through the Robocop Integration Environment tool that automates complex operations and guides a developer through the component composition process.

The proposed prediction approach has a number of benefits. Firstly, it is general and can be applied in different application domains and for various architectural styles. For example, it works for 'blackboard' and 'client-server' architectures. Secondly, the approach allows prediction of dynamically changing resource consumption. Thirdly, the approach is more accurate by incorporating task synchronization constraints and distinguishing synchronous and asynchronous communication. Fourthly, the method is compositional, meaning that the resource usage data of an application can be based on data from its constituent components. Finally, the use of scenarios decreases modeling complexity.

The approach also has some assumptions and limitations that need further study. Firstly, it assumes that resource consumption is constant per operation, whereas it actually may depend on parameter values

passed to operations and/or application state. Secondly, the method is restricted to the Robocop component model, which has a notion of 'requires interfaces', whereas other architectures, such as CORBA do not provide this. Finally, the scenario-based approach demands finding of critical scenarios, which are not always easy to identify.

At the current stage, we have applied the prediction approach to simple applications. In the future we intend to perform case studies for real-world applications.

References

[1] Ivica Crnkovic and Magnus Larsson. *Building Reliable Component-based Software Systems*, Artech House, 2002, ISBN 1-580-53327-2

[2] Johan Muskens and Michel Chaudron. Prediction of Run-time Consumption in Multi-task Component-Based Systems. In *Proceedings of 7th ICSE Symposium on Component Based Software Engineering*. May, 2004.

[3] Robocop public homepage. [http://www.extra.research.philips.com/euprojects/robocop/]

[4] D. Box. *Essential COM*. Object Technology Series. Addison-Wesley, 1997.

[5] T. Mowbray and R. Zahavi. *Essential Corba*. John Wiley and Sons, 1995.

[6] R. van Ommering et al., The Koala component model for consumer electronics software. *IEEE Computer*, 33 (3): 78-85, Mar. 2002.

[7] I. Crnkovic, et al., Anatomy of a research project in predictable assembly. In *5th ICSE Workshop on Component Based Software Engineering*. ACM, May, 2002.

[8] Kurt C. Wallnau. *Volume III: A Technology for Predictable Assembly from Certifiable Components*. April 2003, CMU/ESI-2003-TR-009

[9] Scott A. Hissam, et al., Packaging Predictable Assembly with Prediction-Enabled Component Technology. November 2001, CMU/ESI-2001-TR-024

[10] Scott Hissam et al., *Predictable Assembly of Substation Automation Systems: An Experiment Report*. September 2002, CMU/SEI 2002-TR-031

[11] A. V. Fioukov et al., Estimation of static memory consumption for systems built from source code components. In *Proc. 28th EUROMICRO conference, Component-Based Software Engineering Track*. IEEE Computer Society Press, Sept. 2002.

[12] B. Selic, et al. *Real-Time Object-Oriented Modeling*, Wiley, 1995, ISBN 0471599174.

[13] B.P. Douglass, *Doing Hard Time. Developing Real-time Systems with UML, Objects, Frameworks* and Patterns, Addison Wesley 1999, ISBN 0-201-49837-5.

[14] Vittorio Cortellessa, Rafaella Mirandola. *PRIAM-UML: a performance validation incremental methodology on early UML diagrams*. Elsevier Science B.V., 02/2002.

[15] M. de Jonge, J. Muskens and M. Chaudron. Scenario-based prediction of run-time resource consumption in component-based systems. In *Proceedings of 6th ICSE Workshop on CBSE*. ACM. June, 2003.