Modelling of Input-Parameter Dependency for Performance Predictions of Component-Based Embedded Systems

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

The guaranty of meeting the timing constraints during the design phase of real-time component-based embedded software has not been realized. To satisfy real-time requirements, we need to understand behaviour and resource usage of a system over time. In this paper we address both aspects in detail by observing the influence of input data on the system behaviour and performance. We extend an existing scenario simulation approach that features the modelling of input parameter dependencies and simulating the execution of the models. The approach enables specification of the dependencies in the component models, as well as initialisation of the parameters in the application scenario model. This gives a component-based application designer an explorative possibility of going through all possible execution scenarios with different parameter initialisations, and finding the worst-case scenarios where the predicted performance does not satisfy the requirements. The identification of these scenarios is important because it avoids system redesign at the later stage. In addition, the conditional behaviour and resource usage modelling with respect to the input data provide more accurate prediction.

1. Introduction

Presently, industrial focus in embedded software is shifting from improving implementation techniques towards improving system design methods. The advanced design methods allow evaluation of the system functionality and performance already at very early production phases, which reduces technical risks and minimizes time-to-market.

Advanced design techniques become imperative especially for time-critical software-intensive embedded systems. The real-time requirements imposed on these systems, such as signal latency limitations can only be validated when the system is implemented. A performance test failure can cause a complete iteration in the development process, thus taking additional time and money. To avoid system redesign for ensuring performance properties, we concentrate on the accurate *prediction* of the system extra-functional properties at an early design phase.

We have adopted the so-called Robocop component-based architecture [3] for conducting our research on methods for predictable software design. As any component-based framework, it allows: (a) decomposition of large-system functionality into variable-scale composable blocks, which eases the system maintenance and evolution; (b) wide reuse of the existing components, which reduces development cost and time-to-market.

Application performance properties (CPU usage, memory and bus load, etc) vary within a broad range during the application execution. These variations may depend on many factors: execution platform, current system configuration and state of the application. However, the primary influential factor is *input parameter* data. The input parameter data frequently predefines *data flow* and *control flow* in the application. For instance in a video encoding application, one of the input parameters is the frame size. Depending on the current size of frames to encode, the encoder uses different amounts of processing resources, and even deploys different encoding paths (control flows). This covers the influence of only one key input parameter. Therefore,

in this paper we include input parameter dependency into our models with the aim to come to more accurate prediction of time-detailed performance of component-based systems.

SPE is one of the first approaches which provide a technique for evaluating the performance of software systems [17]. That approach can be enhanced if specialized for component based software engineering [18]. Recently, the UML Profile for Schedulability, Performance and Time [1] has been introduced by the OMG standardization group. In the domain of component-based systems there are several in-depth approaches [7] [8] [9], representing engineering practice to the prediction problem. A very promising 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. A solid method based on formal specification of non-functional properties of component-based software is presented in [12]. However, none of the above-mentioned approaches deals with input data dependencies, so that they all provide relatively lower accuracy of performance prediction. The influence of the input parameters was reported earlier in literature, but the approaches exist only for low level of coding paradigms [13] [14]. At a higher level of coding abstraction, so-called Parametric Performance Contracts have been introduced in [16]. The dependency on the input data is also addressed in [2]. The proposed method allows specifying performance contracts parameterized by input values of objects.

Our contribution in this paper is an extension of a scenario simulation approach for performance prediction [11] with modelling of input parameter dependent behaviour and resource usage for real-time applications. The extension gives a designer an explorative possibility of validating all possible execution scenarios, ranging from best-case to worst-case. The approach is targeted at the domain of *component-based* applications, where the problem of addressing the input data dependencies remains an unsolved problem.

This paper is structured in the following way. Section 2 refers to the Robocop component-based architecture, as a deployment framework for the proposed technique. Section 3 gives classification of input parameters in a component-based system. Sections 4 and 5 discuss the workflow of the proposed approach and give specifications of the models involved. Section 6 explains the task reconstruction algorithm as one of the phases of the workflow. In Section 7, we discuss the simulation and schedulability analysis phases of the workflow. Section 8 concludes with the benefits and drawbacks of the proposed approach.

2. Robocop Component Based Architecture

We have adopted the Robocop Component-Based Architecture (CBA) [3] for conducting our research on predictable software design, because it offers software reuse and speeds up the development. The Robocop architecture is developed for middle-ware in consumer devices, with the emphasis on robustness and reliability. The Robocop CBA is similar to CORBA [5], COM [4] and Koala [6] but enables more efficient realization of real-time and performance constraints via modelling techniques.

A Robocop component is a set of possibly related models. (see Figure 1). Each model provides a particular type of information about the component. Models may be in human-readable form (e.g. as documentation) or in binary form. One of the models is the executable model, which contains the executable component. Other examples of models are: the resource model ($rm \in RM$) and the behaviour model ($bm \in BM$). The Robocop component model is open in the sense that new types of models can be added.

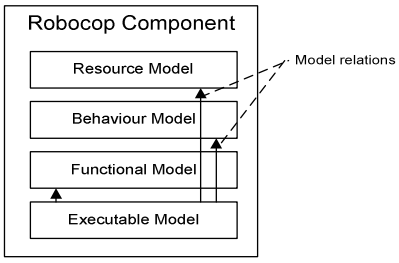

Figure 1. Example Robocop component model

A Robocop component can be specified in a formal notation:

$RC = \wp(M) \times \wp(M \times M \times T)$;
$M = EM \cup BM \cup RM \cup \ldots$;
$T = MODELTYPE \times MODELTYPE \times NAME$;
we assume there is a function φ,
$\varphi : M \rightarrow MODELTYPE$;
where RC is a set of Robocop components, $\wp(M)$ is a power set of model types, and T is a set of relations between the models.

An executable component offers functionality through a set of 'services' belonging to þ(*S*) (see Figure 2). Services are static entities that are the Robocop equivalent of public classes in object-oriented (OO) programming languages. Services are instantiated at run-time. The resulting entity is called a 'service instance', which is the Robocop equivalent of an object in the OO paradigm.

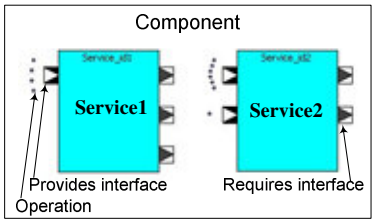

Figure 2. Example of executable component

A Robocop service *s* may define several interfaces (ports). We distinguish a set of 'provides' ports *PR* and a set of 'requires' ports *REQ*. The former defines interfaces *I* 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 *I* is defined as a set of implemented operations *O*. A set of services *S* is formally specified by:

$$
\begin{aligned}
S &= \text{þ}(PR) \times \text{þ}(REQ) ; \\
PR,REQ &= NAME \times I ; \\
NAME &= STRING ; \\
I &= \text{þ}(O) ; \\
O &= \text{set of operations } o.
\end{aligned}
$$

3. Classification of Input Parameters

Behaviour and resource usage of any system depend on its current state, instruction set and input parameters for these instructions. Prior to modelling the input parameter dependencies, we have analysed what kind of parameters normally influence system operation. Within the component-based paradigm, we distinguish two types of parameters for a component: (a) external or environmental parameters, and (b) internal or operation arguments.

External parameters are normally application-level attributes. For example, in a video decoding application, the external parameter for all rendering components (reader, decoder, renderer) can be the 'number of pixels per picture'. This attribute is predetermined by the format of the actual video sequence, thus, by the application. The external parameters can be classified into two types: configuration and dataflow-based (see Figure 3). Configuration parameters provide the means for external application control and result in direct settings for the component state and behaviour. An example is the setting of a 'perceived quality profile' for a decoding component. Dataflow-based parameters ('number of pixels per picture') influence the component behaviour and resource usage in an indirect way.

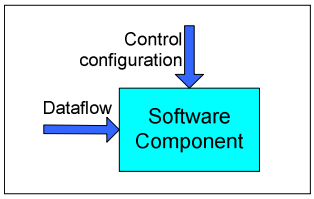

Figure 3. Inputs for external parameters

Internal component parameters are arguments of an operation *o*, implemented by service *s*. Specifying internal parameter dependencies gives higher prediction accuracy for performance in comparison with external parameters, because they are inside of the actual code and lead to more fine-grained control. Note that in some cases external and internal parameters have overlapping influence and cannot be easily classified. For example, an operation `Decoder.decodeFrame(int numberOfPixels)` can exist, where the 'number of pixels' parameter is also an internal parameter of a component.

4. Scenario Simulation Approach

In the component-based software development, a real-time application developer should satisfy given real-time, performance and functional requirements, when he builds his application on the basis of available components. The scenario simulation approach enables early predictions of the performance properties of a designed application, which help to reason about its quality attributes at early stages of development.

The approach is based on three concepts:

1. *models* of the system component's behaviour and resource usage,
2. *execution scenarios* of the complete system, in which the resources are potentially overloaded,
3. *simulation* of these scenarios, resulting in timing behaviour of the designed system.

In order to support the smooth interaction and deployment of the above points, we have developed a

new tool, called Real-Time Integration Environment (RTIE).

The workflow of our approach is described in [11] in detail. In this section we focus on the aspects related to the input parameter dependencies, its modelling and initialisation.

The implementation of our approach is based on four phases (see Figure 4).

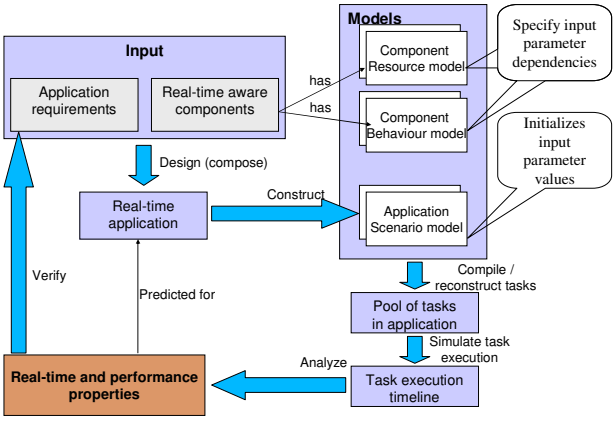

Figure 4. Workflow phases of the scenario simulation approach

Phase 1. A *component* developer specifies the *behaviour* and *resource models* of a real-time aware component. These models should be supplied along with the executables of the component. Within this task, a component developer identifies the external and internal input parameters that influence the component behaviour and resource usage. He finds out (empirically or analytically) the cost functions $Resource\_Use_{operation} = f(parameter)$ and $Behaviour_{operation} = g(parameter)$ for each of the operations, resource types and parameters. Finally, he inserts the cost functions in the component behaviour and resource models. The model structures are specified in Section 5.

Phase 2. An *application* (system) developer graphically composes a real-time application from the set of available components using the RTIE tool. He defines resource-critical scenarios and for each of them specifies an *application scenario model*. Critical scenarios are the application execution configurations that may introduce processor, memory or bus overload. Finally, for each critical scenario, a developer initialises (gives a value to) all input parameters of the constituent components and stores the value into the corresponding scenario model. Consequently, the result of this phase is a set of critical execution scenarios, which sometimes may differ only in the parameter values.

Phase 3. The *application scenario, component resource* and *component behaviour models* are jointly compiled by the RTIE tool. The objective of the compilation is to reconstruct (generate) the tasks running in the application. Prior to compilation, the task-related data is spread over different models. For instance, the task periodicity may be specified in an *application scenario model*, whereas the operation call sequence comprising the task is specified in relevant *component behaviour models*. The compiler reconstructs all necessary properties of the tasks, like deadline, period, priority and operation call sequence.

Phase 4. An application developer schedules the reconstructed task pool (by the RTIE tool), simulating the execution of the defined scenario. The simulation scheduling policy of the application execution should be compliant with the scheduling policy of the operating system. The resulting data from the scheduler is a *task execution timeline*. This timeline allows extracting the real-time, memory- and bus-related performance properties of an application. A comparison between the predicted data with the application requirements, allows us to quantitatively assess the design of the application. If any of the requirements are not satisfied, a developer may optimise the composition or find other design alternatives and repeat the workflow.

5. Model description

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.

5.1. Resource Model

The Resource Model specifies the resource usage for all the operations implemented by services of an Executable Component (see Figure 5). The resource usage properties of an operation may result from either average-case or worst-case analysis. These properties are calculated only for the operation body itself, and they exclude resource consumption properties of called operations. This technique allows the calculation of the resource consumption of any sequence of operation calls. The resource model is specified for a particular reference platform.

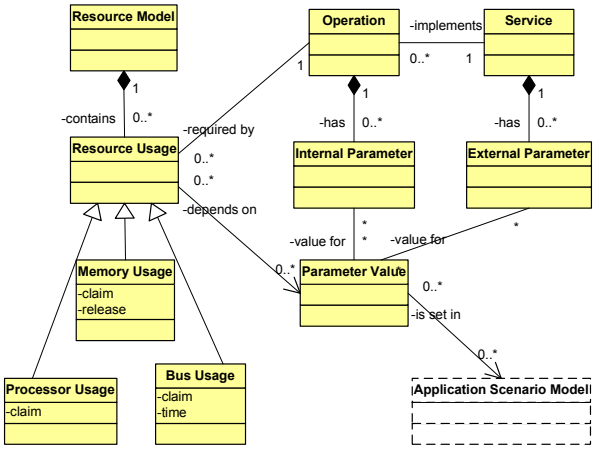

Figure 5. Resource Model Representation

The resource usage can be specified for any type of resource. We propose to model scarce major hardware resources: processor, memory and bus (network). The predicted resource consumption is specified as a (*claim{bytes}, release{bytes}*) tuple for memory; and (*claim{bytes}, time{ms}*) tuple for network-type resources. For processing resources, the consumption is specified as a single (*claim{ms}*).

The resource usage depends on the value of external and internal input parameters. These dependencies are specified as a cost function. Let us briefly define this in more detail. We specify a cost function as the product of the operations and the input parameter influencing the resource usage of these operations. For each resource type $r \in R$, the model provides a function that maps a product of the following elements:

(1) operation implementation,
(2) vector of values of external parameters and
(3) vector of values of internal parameters,

into a sequence of natural numbers that represents the resource usage.

More formally, a cost function ($O_{\text{imp}} \times VV_{EP} \times VV_{IP} \rightarrow$ Nat*) gives the cost ($c \rightarrow$ Nat*) for an operation ($o_{\text{imp}} \in O_{\text{imp}}$). Per resource r there can be multiple cost functions $\beta(O_{\text{imp}} \times VV_{EP} \times VV_{IP} \rightarrow$ Nat*). The resource model results in a set of used resources:

$$R = \{ r_i , \forall i \mid r_i = \beta(O_{imp} \times VV_{EP} \times VV_{IP} \rightarrow \text{Nat*})\};$$

where the parameters are explained by

O_{imp} – operation set implemented by the component,
VV_{EP} is a vector of values of external parameters,
VV_{IP} is a vector of values of internal parameters,
Nat* is a sequence of natural numbers (for instance, *claim* and *release*).

The parameter values are initialized later by application developer in the scenario model.

5.2. Behaviour Model

The behaviour model specifies the behaviour of all operations O_{imp} implemented by services of an executable component. A part of the model is shown in Figure 6.

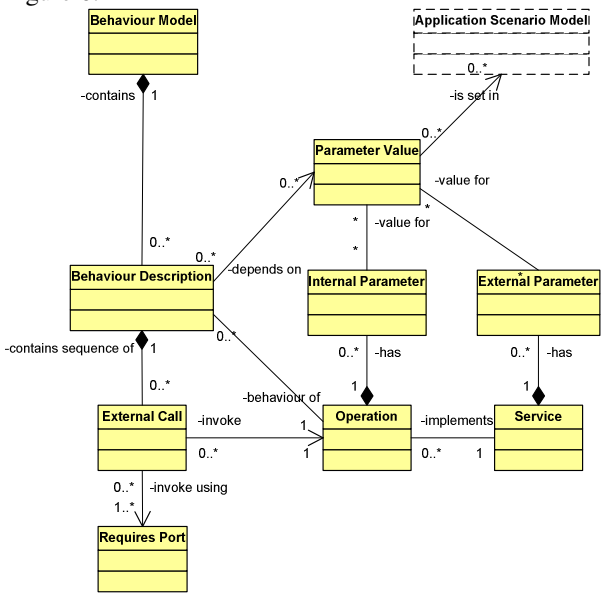

Figure 6. Behaviour Model Representation

Conceptually, the operation behaviour is specified as a sequence of external operation calls (to other interfaces) made inside the implemented operation. For example in Figure 7, the implemented operation `Controller.updatePosition()` has a behaviour described by the following call sequence: {`IData.refreshData()`, `IControl.move()`}. The `IData` and `IControl` are the interfaces provided by Sensor and Actuator services, respectively.

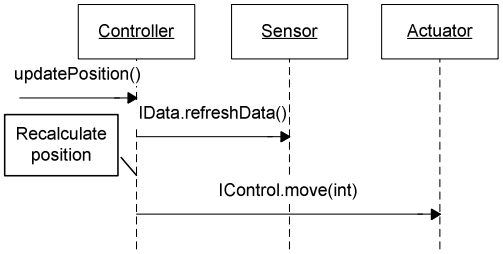

Figure 7. Sequence of operation calls (*behaviour*) of updatePosition() operation.

The operation behaviour may depend on the input parameters. In this case, the component developer can model these dependencies by again specifying cost functions. For each operation $o_{imp} \in O_{\text{imp}}$ implemented by the component, the model may provide a function that maps a threefold product of:

1) operation implementation,
2) vector of values of external parameters and
3) vector of values of internal parameters,

into a sequence of external operation calls with a set of value vectors of arguments for each external operation. More formally, this leads to

$$BM = (O_{imp} \times VV_{EP} \times VV_{IP}) \rightarrow (O \times VV_{IP})^*;$$

where

O_{imp} is a set of operations implemented by the component,

O is a set of external operations (provided by other interfaces),

$(O \times VV_{IP})^*$ is a sequence of external operation calls made by each implemented operation $o_{imp} \in O_{\text{imp}}$,

VV_{IP} is a set of value vectors of internal parameters (operation arguments).

The values of the parameters must be initialised later in an application scenario model.

Note that also task triggers can be modelled within a behaviour model. The reader is referred to [11] for a description of this option.

5.3. Application Scenario Model

The *application scenario model* (*SM*) specifies the application structure and behaviour for a critical or commonly used execution scenario. Several *SM*s can be built for an application, depending on a number of scenarios that need to be validated. An application developer is responsible for the *scenario models* construction.

A scenario model consists of a *composition* description, a set of parameter initialisation *PI*, and a set of application-level task triggers *T*. Composition is specified by *SI* (set of service instances *si*) and *B* (set of *bindings* between the *si*). A binding $b \in B$ includes information about the bound service instances $si \in SI$, and in/out ports of their interfaces $pr \in PR$, $req \in REQ$. Task trigger $t \in T$ points to a first called operation $o_{imp} \in O_{imp}$ and contains a set of attributes *ATTR* of the trigger (period, offset, jitter, inter-arrival time, etc).

The scenario model should initialise all parameters specified in the resource and behaviour models of the components used in this scenario. Thus, parameter initialisation *pi* is a given value v_{EP} (v_{IP}) residing within the vector of values VV_{EP} (VV_{IP}). The scenario model *SM* is defined by

$$SM = composition \times PI \times T;$$

where

$composition = SI \times B;$
$SI = S \times NAME;$
$B = SI \times PR \times SI \times REQ;$
$PI = (V_{EP} \in VV_{EP}) \text{ x } (V_{IP} \in VV_{IP});$
$T = O_{imp} \times ATTR.$

Once the scenario models are ready, an application developer proceeds to the task reconstruction and simulation phases of the workflow.

6. Task Reconstruction

In the task reconstruction phase, an application developer brings together the *application scenario model* and combined *behaviour-resource models* of the components used in the application. At this stage, this set of models can be compiled by RTIE tool. The goal of the compilation is to identify and reconstruct a set of tasks with their parameters such that the application executes in a particular scenario.

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

The formal definition of the task reconstruction algorithm can be specified as follows. Assume that $I.o(V_{IP})$ is the initial operation invocation of a task *t*. We have omitted brackets for simplicity of notation. The operation *o* is invoked with the initialised values for the set of input arguments V_{IP} (a set is indicated by a capital). The subsequent invocations made by the operation *o* can be found via the behaviour model of a component (service) implementing this operation. The function $\text{Expand}(I.o(V_{IP}))$ yields a term that represents the call-graph for an invocation $I.o(V_{IP})$. This resulting term includes sub-terms for each individual operation invocation. For the operation *o* invocation the complete call-graph can be specified as:

$$Expand(\ I.o(V_{IP})\ ) = I.o(V_{IP}) \oplus (\ Expand(I.o_1(V_{IP1})) \otimes \ldots \otimes Expand(I.o_n(V_{IP\,n}))\ );$$

where ⊕ = 'with the following added' ,

⊗ = 'sequence'

$o_1 \ldots o_n$ is a sequence of called operations of operation *o* specified in a behaviour model.

The result of the above is a tree of operations, where the sequence of operations is found when we traverse the tree.

The input parameter dependent call-graph tree represents a task in a system. The task invocation periodicity, jitter, offset is known from the models. The resource claims of the task comprising operations are known (they may also be parameter dependent). Thus, for a processing resource, we can calculate the task execution time as a sum of claims of all operations involved in the task call-graph.

For the controller example in Figure 7, the task reconstruction works as follows: the related *behaviour model* specifies the operation call sequence of the operation `updatePosition(): {refreshData(), move()}`. Afterwards, the compiler gathers from related behaviour models the operation call sequence of the latter two operations. The operation `refreshData()` calls one operation belonging to other interfaces: `ILogData.logEvent()` (see Figure 8).

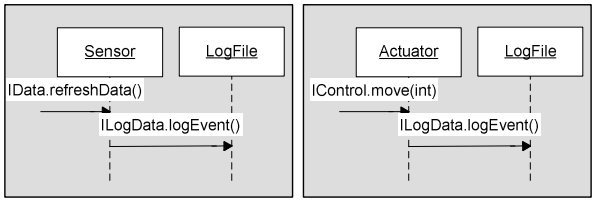

Figure 8. refreshData() and move() behaviour.

Assume that operation `ILogData.logEvent()` is a leaf operation. The operation `move()` also calls this leaf operation: `ILogData.logEvent()` (see Figure 8). Thus, the complete reconstructed sequence of the operations executed in the task is as shown in Figure 9.

A resource consumption property of each operation in this sequence is specified in the claim primitive of the related component resource model (see Section 5.1). Knowing this data, we can calculate the 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: 50ms + 12ms + 5ms + 80ms + 5ms = 152ms.

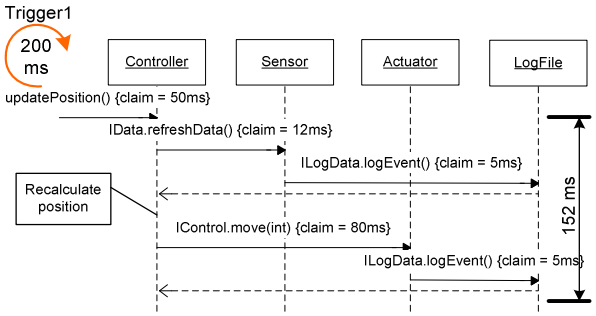

Figure 9. Task reconstructed from the models.

The other task parameters (period, offset, and deadline) and precedence are obtained from the corresponding task trigger (Trigger1) properties specified in the models.

7. Simulation and Schedulability Analysis

When the set of tasks in the scenario is identified, an execution of this set is simulated by the RTIE virtual scheduler. The simulation results for a processing resource are represented as a task execution timeline (see Figure 10).

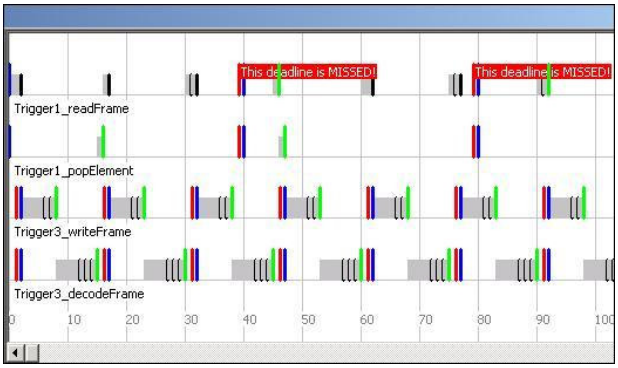

Figure 10. Task execution timeline of scenario

The schedulability analysis of the simulation data gives the predicted timing and performance properties of an application. The response time, blocking time, number of missed deadlines can be found for each task. Beside this, the processor, memory and bus bandwidth utilization bounds can be analysed per application scenario. When playing with a range of values for specified input parameters, a designer can find resource overloading execution scenarios. This may lead to a system (application) redesign. In any case, the predicted performance properties are to be validated with respect to the application requirements.

8. Conclusion

In this paper, we have proposed an extension to an already existing scenario simulation approach for the prediction of system timing and performance properties at early stages of development. The extension enables handling a design of real-time systems, of which the resource consumption is heavily depending on the input data parameters. The extension introduces a concept for modelling of two dependency types: (1) input parameter influencing the system resource usage, and (2) input parameters influencing the behaviour of operations in the system. The prediction approach should be employed in the domain of component-based software development. Therefore, two model categories are introduced: a component model specified during development of a component (behaviour and resource models) and a scenario model described during the composition of a system. In the component model, the behaviour and resource usage are specified as cost functions from the input parameters. In the scenario model, these input parameters are initialised, resulting in the instances of exact behaviour and resource usage per scenario. Furthermore, each specified execution scenario can be simulated by our recently developed RTIE tool, giving a predicted performance of a system.

We have validated the approach by a real-world case study based on an object-oriented MPEG-4 decoding system [15]. Initial experiments without any method calibration give already a prediction accuracy on the performance within 10-30%. At the time of writing, more experiments are conducted.

Besides exploring the input data dependencies for the prediction accuracy, our proposal has a number of additional benefits. The use of scenarios reduces modelling complexity and avoids a state-space explosion. The proposed technique can be applied to any other component-based architecture (CORBA, Koala). It can be used for different application domains and for various architectural styles. For example, it works for 'blackboard', 'client-server' and 'pipe-line' architectures. Finally, it is our opinion that the proposed approach may lead to higher accuracy when incorporating task synchronization constraints and distinguishing synchronous and asynchronous communication. This aspect requires more research in the future.

The method has some limitations that need further study. It provides no techniques for specifying the component resource consumption for multiple platforms. Besides this, we assume that a designed system has no multiple processors and networks.

References

[1] OMG Group. UML Profile for Schedulability, Performance and Time.

[2] Krone, J., Ogden, W.F., Sitaraman, M., *Modular verification of performance constraints*. Technical Report RSRG-03-04, Clemson University (2003)

[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, New York, 1995.

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

[7] I. Crnkovic, *et al*., "Anatomy of a research project in predictable assembly". Proc. *5th ICSE Workshop on CBSE*. ACM, May, 2002.

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

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

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

[11] E. Bondarev, J. Muskens, P. de With and M. Chaudron, "Predicting Real-Time Properties of Component Assemblies: a Scenario-Simulation Approach", *Proc. 30th EUROMICRO conf., CBSE Track,* IEEE Computer Science, Sept. 2004.

[12] S. Zschaler, "Towards a Semantic Framework for Non-functional Specifications of Component-Based Systems", *Proc. 30th EUROMICRO Conf.*, Rennes, France, Aug./Sep. 2004, pages 92-99, IEEE Computer Science, Sept. 2004.

[13] F. Wolf, "Intervals in software execution cost analysis", *Proc. 13th Int. Symp. on System synthesis* 2000, ISBN 1080-1082, 2000.

[14] H. Gautama, A. J. C. van Gemund, *Performance Prediction of Data-Dependent Task Parallel Programs*, Lecture Notes in Computer Science, vol. 2150, 2001

[15] E. Bondarev, M. Pastrnak, P. de With and M. Chaudron, "On Predictable Software Design of Real-Time MPEG-4 Video Applications", *SPIE Proc. of VCIP' 2005 conference*. Beijing, China. July, 2005.

[16] Viktoria Firus at al., Parametric Performance Contracts for QML-specified Software Components. *In Procs. FESCA workshop 2005*. Edinburgh, April 2005

[17] Smith, C.U., Williams, L.G., *Performance Solutions: a practical guide to creating responsive, scalable software.* Addison-Wesley (2002)

[18] Bertolino, A., Mirandola, R., CB-SPE Tool: Putting Component-Based Performance Engineering into Practice. *Proc. 7th International Symposium on CBSE*, Edinburgh, UK. Vol. 3054 of LNCS, Springer (2004) 233–248