Predicting Real-Time Properties of Component-Based Applications

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

[1] System Architecture and Networking group,
[2] Signal Processing Systems group/LogicaCMG,
Eindhoven University of Technology, P.O. Box 513,
5600 MB Eindhoven, The Netherlands
E.Bondarev@tue.nl

Abstract. In this paper we discuss the prediction of timing properties of a multi-tasking component-based application already during the design phase. At this stage, it is of vital importance to guarantee that the timing requirements (e.g. response times) of a real-time application will be satisfied. This can be achieved by predicting the real-time behaviour of a component-based application. In this paper, we extend an existing scenario-based approach [2] with the possibility to model the behaviour of an application and the behavior of the underlying components. This extension allows an application developer to predict the real-time properties of a component assembly before its actual implementation. The modelling involves the specification of synchronization constraints for tasks and the simulation of application behaviour. A concluding case-study of video encoder development reveals that the approach is not only feasible but also brings a contribution to the design of the software-intensive multimedia real-time systems.

1 Introduction

Software-intensive embedded systems usually have two closely coupled properties: limited resources and real-time constraints for execution of the applications. The limitation of resources, such as memory size, bus bandwidth and processing power, complicates the satisfaction of the real-time constraints. It is evident that these guarantees are of great importance for e.g. multimedia devices.

For high-volume embedded appliances, such as PDAs and mobile phones, etc, an open, component-based framework for a middleware layer in the software architecture has been proposed. This framework, known as Robocop [3], was used as a reference for specifying a follow-up ITEA research project, called Space4U. Our research aims at improving the Robocop component-based architecture, by enabling the predictions of application real-time properties.

During the design phase, an application is evaluated to fit on a target system. For an *a-priori* evaluation, this requires prediction of the resource usage of an application. Early prediction of resource usage and timing properties of an application at the design stage increases system robustness and reduces cost and problems in product development.

Component-based technology complicates the prediction of resource usage and timing properties of an application. In component-based systems, the actual behaviour and resource usage are determined by an ensemble of internally and also *externally* developed components. Thus, the prediction task becomes twofold: (1) find and express the *component's* extra-functional properties, and (2) combine these properties 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 predicting real-time properties of a component assembly is of significant interest because of the rapid development of component-based technologies in the embedded systems domain. Some approaches represent an engineering practice [7-9] to the problem. We used [8] as a guideline for our work. 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. 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. We concentrate on similar methods, but now in the component-based development field. The scenario-based approach proposed in [15] involves estimating static resource usage of a component assembly.

In contrast with this, through our scenario simulation approach, we address a *dynamic* or *time dependent* instead of *static* resource usage, thereby giving more accuracy in the prediction of the assembly behaviour. With respect to *task synchronization*, we adopt the use of synchronization constraints (precedence, critical sections, mutexes) 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 behaviour of an application that the developer should model and simulate. A practical case-study revealed that, besides proving feasibility, the important problem of task parallel execution and synchronization comes to the foreground. This prevents that these usually hidden problems remain unsolved.

The paper is structured as follows. Section 2 addresses various aspects of timing properties of a component assembly. Section 3 presents the Robocop component model, as a fundament for the new extensions. Section 4 discusses the workflow of the approach and gives specifications of the required models. Section 5 clarifies the proposed approach with an encoder application case-study. Section 6 concludes with the benefits and disadvantages of the proposed prediction-enabling technique.

2 Timing and Parallel Execution Aspects

This section defines 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. In the real-time application domain, 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 pre-emption time of the task. Therefore, for the assembly timing property, the task synchronization and scheduling aspects should be considered.

In literature, several definitions of tasks are used. In our 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 synchronous or asynchronous. The tasks may have *synchronization constraints* between them, e.g. precedence, rendezvous and mutual exclusion. Usually, the system resource sharing imposes the synchronization constraints.

All of the above aspects have been used in this paper in a regular way, confining to the presented definitions.

3 Robocop Component Model

The Robocop component model is inspired by existing component-based architectures, such as COM [4], CORBA [5], and Koala [6]. A Robocop component is a set M of possibly related models, as depicted in Fig. 1.

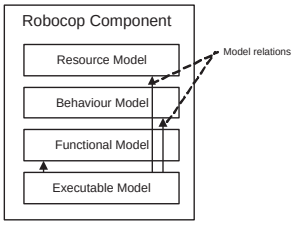


Fig. 1. Example Robocop component model

Each individual model m provides a particular type of information about the component. Models can be represented in readable form (e.g. documentation)

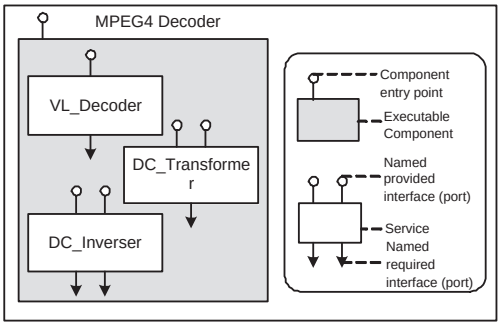
or in binary code. One of the models is the *executable model* that contains the executable component. Other examples are: *resource model*, *functional model*, and *behaviour model*.

A component offers functionality through a set of 'services' P (see Fig. 2 and Fig. 3(a)). Services are static entities, which are the Robocop equivalents of *public classes* in object-oriented (OO) programming. 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).

Fig. 2. Specification of executable model

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. 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 in Fig. 3(b).



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).

(b)

Fig. 3. (a) Example of executable component, (b) Specification of Robocop service

Please note that a *Robocop service* is 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* stands for a composition of services.

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

4 Scenario Simulation Approach

4.1 Workflow of the Approach

The main task of a component-based application developer is, given a set of available components and requirements for an application, to select the right components and compose them into an application that will satisfy the given requirements.

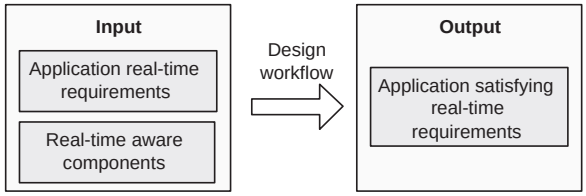


Fig. 4. Conceptual view on the prediction-enabling composition workflow

In the domain of real-time application, a developer needs to focus on satisfying extra-functional requirements like response time, or memory usage (Fig. 4).

The proposed prediction-enabling composition workflow contains principal steps for a real-time application developer. These steps lead a developer from the application requirements and set of available components to an application with the required timing properties. The timing properties are obtained already in the design phase, prior to running the assembly on the target system. The workflow is based on the two following *assumptions*.

1. Resource usage property and behaviour of the constituent components are specified and available in the corresponding component models.
2. An application developer is able to find out critical scenarios of the application.

The main steps of the workflow are depicted in Fig. 5. The remainder of this section defines the consecutive steps of the workflow.

Component composition. A developer selects and composes a set of available components into an application. According to the above assumption (1), selected components should be *real-time aware*, e.g. have both a *resource model* and a *behaviour model*. These two models are to be written by a component developer and 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 is given in Section 4.2.

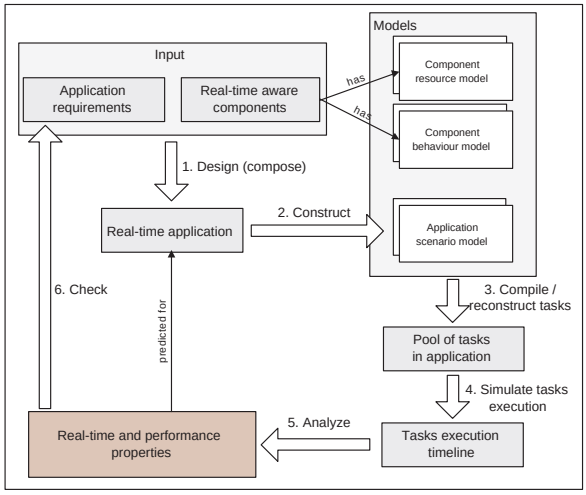


Fig. 5. Main steps in the predictable RT-composition workflow

Construction of application scenario model. For each critical or commonly used scenario, a developer constructs (with the help of a GUI-based tool) an *application scenario model* (see Fig. 5). 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.

Compilation of models. The *application scenario, component resource* and *component behaviour models* are jointly compiled. The goal 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.

Simulation of tasks execution. An application developer applies a scheduler to the reconstructed task pool, simulating the execution of defined scenario. The scheduling algorithm may vary depending on the algorithm of the operating system, on which the application is supposed to execute. The scheduler should implement prevention of unbounded priority inversion, because the models define various types of synchronization constraints. Resulting data from the scheduler is the *task execution timeline*. This timeline is a subject for schedulability and performance analysis.

Schedulability analysis. The analysis of the task execution timeline helps to reason about application timing properties like response time, latency of critical tasks, overall schedulability and processor utilization bounds. Many other possible application properties can be derived: rate of missed deadlines, blocking time, worst and best-case response time per task. This step results in predicted *real-time and performance properties* of the designed application.

Checking properties against requirements. The predicted timing properties are checked against the real-time requirements of an application (see Fig. 5). For example, worst-case response time of a critical task is verified with its deadline specified in the requirements. If any of the requirements are not met, a developer optimises the composition and repeats the workflow.

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

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

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* { *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*.

Fig. 6. Specification of component resource model

A component developer defines the resource usage properties of an operation by worst-case analysis. These properties are calculated only for the operation body itself, excluding resource usage properties of called operations. This approach allows calculating resource usage of any sequences of operation calls. In

this paper, we do not address platform and parametric variations of the operation resource usage. The resource model should be specified for a particular reference platform.

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	= { *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 { *synch*, *asynch*}.
cs	= (*called_opr1*, *called_opr2*, …*called_oprn*).
t	= (*periodicity*, *param*, *PRECED*),
	where *periodicity* { *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.*

Fig. 7. Specification of component behaviour model

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 Fig. 8, 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, respectively.

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*

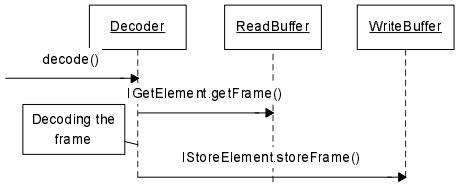


Fig. 8. Sequence of operation calls (behaviour) of decode() operation

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

Finally, a component developer should 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 usage and behaviour properties independent of the application context where the component is going to be used.

Application Scenario Model. According to Fig. 5, we propose to model *scenarios* in an application. This allows decomposing each type of application behaviour into a separate simple scenario model. Thus, we can reduce the complexity of the complete behavioural model of the application and partly avoid exploration of all application states.

The application scenario model (*SM*) specifies 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 interesting scenarios. An application developer is in charge of the scenario models construction. The semi-formal structure of the model is presented below (Fig. 9).

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 Fig. 10, dashed lines represent the *bindings*.

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 *event* as any influence

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* { *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.*

Fig. 9. Specification of application scenario model

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.

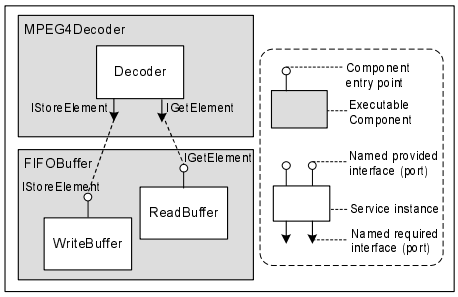


Fig. 10. Example of application structure

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 developer. 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 should be specified by a component developer and an in-application task trigger should 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 Fig. 11, 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 and in-application task trigger t, its *periodicity*, parameters *param* and precedence constraints *preced* can be specified.

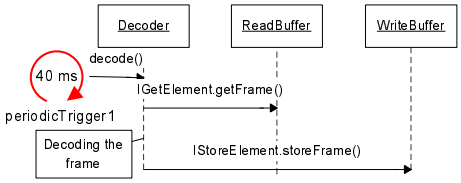


Fig. 11. Task triggered by in-application trigger

When the scenario models are ready, a developer proceeds to the next phase.

4.3 Model Compilation 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 model compilation phase, an application developer brings together the *application scenario model*, *behaviour* and *resource models* of the components deployed in the application. At this stage, this set of models can be compiled by the RIE. The conceptual goal of the compilation is to identify and reconstruct a set of tasks that the application will execute 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 Fig. 8). Afterwards, the compiler gathers from related *behaviour models* the behaviour of these two operations. The operation getFrame() calls one operation belonging to other interfaces: ILogData.logEvent()(see Fig. 12).

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 Fig. 8). Thus, the complete reconstructed sequence of the operations executed in the task is as depicted in Fig. 13.

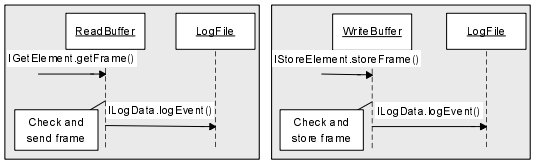


Fig. 12. The getFrame() and storeFrame() behaviour

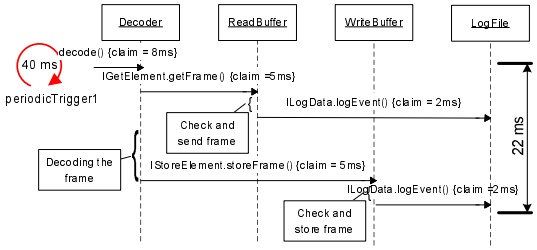


Fig. 13. Task generated from the models

A resource-usage property of each operation in this sequence is specified in the *claim* primitive in the related *component resource model* (see Section 4.2). Knowing this data, we can calculate total resource usage 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 Fig. 13, 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 the associated 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 behaviour model*, all define *synchronization constraints* of tasks.

An execution of the reconstructed tasks of the scenario is simulated by a virtual scheduler. During the simulation, these synchronization constraints are taken into account. The scheduling algorithm should be the same as an algorithm of the targeting operating system. The simulation results are represented as a task execution timeline (see Fig. 14).

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

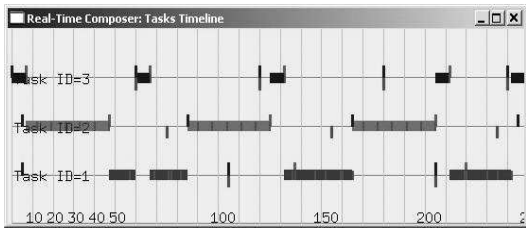


Fig. 14. Task timeline execution in scenario

5 Video Encoder Case-study

The objectives of the video encoder case-study are to show practical aspects of the approach utilization and give further clarification. The example starts with requirements, goes through the prediction-enabling composition workflow and ends with predicted timing properties of the application.

5.1 Requirements

Taking into account that we do not focus on functional requirements, the required functionality can be expressed in one sentence: *the application shall encode on-the-fly the audio and video signals in MPEG-4 format and subsequently multiplex the compressed signals into one stream* (REQ1). The extra-functional requirement for the TV-like application: *the number of skipped frames during the encoding on-the-fly should be NULL* (REQ2). This implies that we do not allow missed deadlines for audio and video encoding tasks (real-time application).

5.2 Component Selection

After the requirements elicitation, the process of the component-based application development continues with component selection. Because our application has a real-time nature, we should select only real-time aware components (resource and behaviour models in their distribution package).

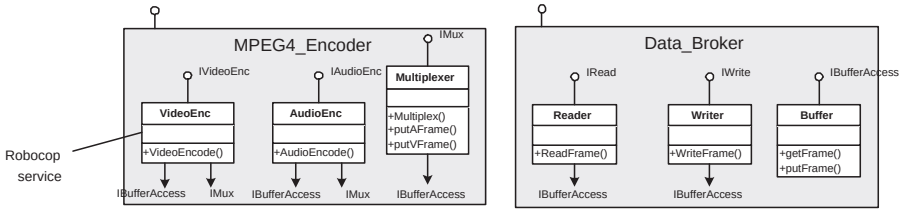


Fig. 15. Selected components with services, interfaces and operations

We selected two real-time aware components that bring the required functionality: MPEG4_Encoder and Data_Broker having three service each as indicated in Fig. 15. Each service has *provides* and *requires* interfaces. For instance, the VideoEnc service provides IVideoEnc interface and requires IBufferAccess and IMux interfaces. The IVideoEnc interface encapsulates the VideoEncode() operation. All public operations are also represented in Fig. 15.

The corresponding resource and behaviour models (see Fig. 16) are constructed according to the rules defined in Section 4.2. The behaviour (resource) model specifies behavioural (resource usage) aspects of all public operations of the component. Note that there are no task-triggering operations specified in services of both components (fields for task triggers T are empty). It means that all operations are passive (have no autonomous behaviour) and should be controlled by application-level events and task triggers.

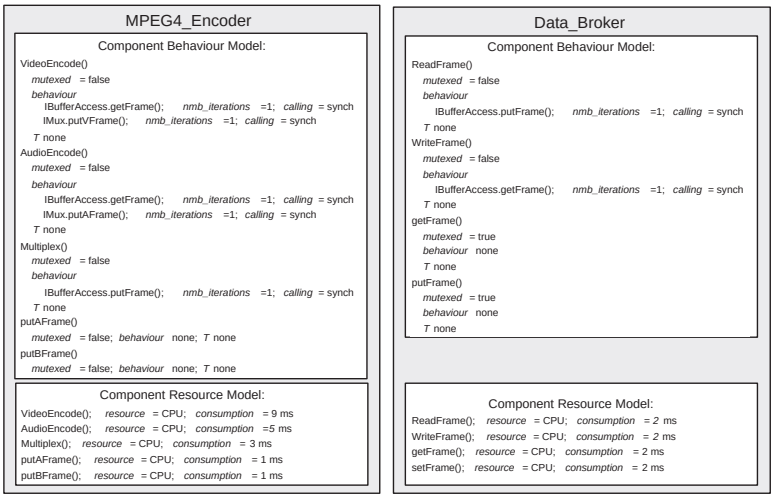


Fig. 16. *Behaviour* and *resource models* of the components from Fig. 15

5.3 Composing the Encoder Application

The design (composition) phase, which is the first stage of the workflow, consists of three steps: service instantiation, service instances binding and design of application level events and task triggers.

The *service instantiation* is basically a process of defining a structure of an application depending on required functionality. Our encoder should read, encode AV streams and multiplex them in one MPEG-4 stream. Finally, this stream should be stored. Therefore, the encoder should have at least the following service instances: audio-, video- readers, audio-, video- encoders, multiplexer, and writer. Data communication between the instances can be realized by a set

of buffers. This structure (service instantiation) is depicted in Fig. 17. As can be noticed, the Reader service is instantiated twice (aReader, vReader) and the Buffer service has three instances (vBuffer, aBuffer, mBuffer).

The second step is *binding the service instances*. *Requires interfaces* are connected to *provides interfaces* of the same type, thus defining data and control flows in the application. Fig. 17 depicts the service instance bindings by the connecting arrows.

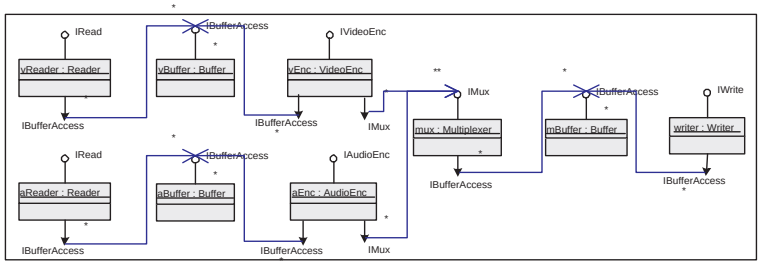


Fig. 17. Binding the encoder service instances

In the third step, a developer identifies necessary application-level *events* and *task triggers*. In component-based systems, an application-level *task trigger* can be implemented in the code of application as a separate thread that wakes up periodically (by timer signals) and invokes one of the component operations. In its turn, an *event* is usually implied by hardware platforms (i.e. interrupts). A developer needs not to implement events, but should take them into account during the design phase.

The services composing the encoder application have no autonomous operations with task triggers inside (all services are passive). In order to make the application alive, we designed six task triggers executing on the application level (see Fig. 18). Each of the task triggers periodically invokes one of the operations, thereby creating a separate thread of control. For example, Trigger1 invokes the IRead.readFrame() operation of the vReader service instance. This operation reads one video frame from a file and stores the frame in vBuffer. All triggers are designed to fire with periodicity of 40 ms, since this is common video streaming rate. We defined the deadlines for the triggered tasks to be equal to their periods (40 ms). We specified no precedence constraints for the tasks. Having this information we can construct an *application scenario model*.

5.4 Constructing the Application Scenario Model

The construction of a scenario model starts with identification of relevant scenarios. The relevant scenario can be either a common execution scenario or a critical scenario. In the encoder case, the common execution scenario (e.g. en-

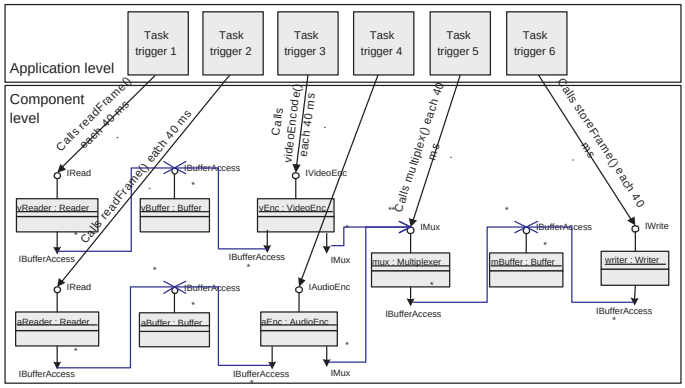


Fig. 18. Application level task triggers

coding mode) is relevant to consider, because it implies high resource usage and correlates with REQ2 (see Section 5.1).

Scenario Model "encoding":

structure
  SI
    vReader; aReader; vBuffer; aBuffer; vEnc; aEnc; mux; mBuffer; writer
  B
    vReader; IBufferAccess; vBuffer; IBufferAccess
    aReader; IBufferAccess; aBuffer; IBufferAccess
    vEnc; IBufferAccess; vBuffer; IBufferAccess
    aEnc; IBufferAccess; aBuffer; IBufferAccess
    vEnc; IMux; mux; IMux
    aEnc; IMux; mux; Imux
    mux; IBufferAccess; mBuffer; IBufferAccess
    writer; IBufferAccess; mBuffer; IBufferAccess
E none

T
  Trigger1
    opr = IRead.readFrame()
    periodicity = periodic; *period* = 40; *deadline* = 40
    preced none
  Trigger3
    opr = IVideoEnc.videoEncode()
    periodicity = periodic; *period* = 40; *deadline* = 40
  ...
  Trigger5
    opr = IMux.multiplex()
    periodicity = periodic; *period* = 40; *deadline* = 40
depend
  MPEG4_Encoder; Data_Broker

Fig. 19. Application scenario model for encoding scenario

The application scenario model (defined in Section 4.2) requires data about service instances, bindings, events and task triggers for the selected scenario. This data is already known from the above-mentioned design steps, so that we only need to represent this data in the scenario model format. A major part of this scenario model is portait by Fig. 19.

After all related data is inserted in the application scenario model, we give a flow to the RIE for the models compilation and simulation of the compiled tasks.

5.5 Models Compilation

The RIE compiler reconstructs the tasks in the application scenario (reconstruction process is explained in Section 4.3). Here we graphically represent the result of the task reconstruction (see Fig. 20). The tasks are circular lines with arrows

showing the control flow directions. For example, video encoding task is triggered by Trigger3 who calls operation VideoEncode(). This operation first calls getFrame() operation of vBuffer, then encodes the received frame and finally calls putVFrame() operation of Mux service instance. This task repeats each 40 ms. The sequence diagram for the task is similar to the structure from Fig. 11.

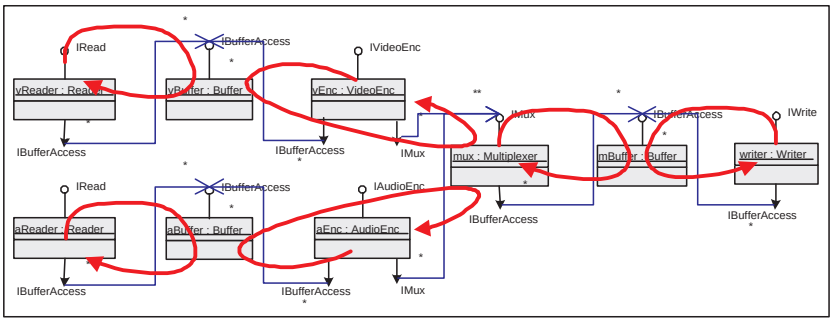


Fig. 20. Reconstructed tasks after models compilation

5.6 Simulation of Tasks Execution

An execution of the reconstructed tasks can be further simulated by the RIE scheduler. The current algorithm used in the RIE scheduler is rate monotonic with bounded priority inversion. The virtual scheduling of the encoder tasks results in the execution timeline depicted in Fig. 21. The three bold vertical lines show: completion, deadlines and triggering moments of each task instance.

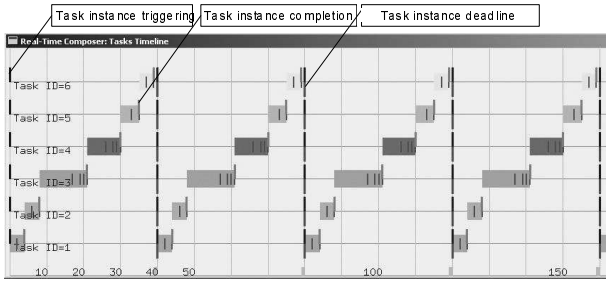


Fig. 21. Part of the task execution timeline for encoder application

5.7 Schedulability Analysis

The schedulability analysis leads to the requirements validation. Our extra-functional REQ2 demands no missing deadlines of the audio and video encoding tasks (see Section 5.1). According to the generated tasks execution timeline (Fig. 21), video encoding (TaskID = 3) and audio encoding (TaskID = 4) tasks meet all deadlines for a simulation period of 10 seconds. Note that this is only true under the condition that the assigned CPU budget is 100 %.

This step ends with the conclusion that the designed application meets its real-time requirements and we can now proceed to the implementation phase.

6 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 behaviour 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 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 usage. 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 modelling complexity.

The proposal also has some assumptions and limitations that need further study. Firstly, it assumes that resource usage 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 COM, do not have this notion. Finally, it provides no techniques for specifying the component resource model for different platforms. Extending the relatively simple case in this paper, we are currently validating the approach on a more complex MPEG-4 codec software.

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. To be 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. Developping 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.