Quality-Oriented Design Space Exploration for Component-Based Architectures

Egor Bondarev[a], Michel Chaudron[a], Peter de With[a,b] and Arjen Klomp[b]

[a]Eindhoven University of Technology, P.O. Box 513, 5600 MB, Eindhoven, The Netherlands

[b]LogicaCMG Nederland, P.O. Box 7089, 5605 JB, Eindhoven, The Netherlands

ABSTRACT

In this paper we present the DeSiX design space exploration methodology for software component-based systems on multiprocessor architectures. The proposed technique adopts multi-dimensional quality attribute analysis in the early design phases and is based on (1) various types of models for software components, processing nodes, memories and bus links, (2) scenarios of system critical execution, allowing the designer to focus only on relevant static and dynamic system configurations, (3) simulation of tasks automatically reconstructed for each scenario and (4) Pareto curves for identification of optimal architecture alternatives. The feasibility of our methodology is shown by a detailed case study of a car radio navigation system, where te starting point is five candidate system architectures, representing different cost versus performance sensitivity versus reliability tradeoffs.

Keywords: Design Space Exploration, Software Quality, Component-Based Software Architecture, Performance Prediction, Quality Attribute Trade-off, Multi-Criteria Design

1. INTRODUCTION

A major challenge in system development is finding the right balance between the different quality requirements that a system has to meet. Time-to market constraints require that design decisions are taken as early as possible in the development process. A recent trend is the assembly of systems out of existing building blocks (which can be both software and hardware components) as this has the potential of reducing development time as well as development cost.

To solve this problem, the architect should be able to easily construct models of architectural alternatives and assess their quality properties. Essentially he needs a means to efficiently explore the design space. This requires the ability to assess the quality properties of component-based architectural models. Some work in this direction

Further author information:

Egor Bondarev: E-mail: e.bondarev@tue.nl, Telephone: +31 40 247 2480

has been done by approaches towards the *predictable assembly* problem[1] ,[2] ,[3] . However, these approaches focus on the prediction of a single quality attribute, whereas assessment of multiple quality attributes is needed in order to motivate design trade-offs. A related work on multi-dimensional design space exploration is given in Section 6.

In this paper we propose the so-called DeSiX (**De**sign, **Si**mulate, e**X**plore) methodology that uses a component-based architecture as the skeletal structure, onto which multiple analysis methods for different quality properties are attached. A distinguishing feature of our approach is that the analysis is based on the evaluation of a number of key scenarios. The use of scenarios enables efficient analysis while also enabling the architect to trade modelling effort (modelling multiple scenario's improves the 'coverage' of the behaviour of the system) against confidence in the reliability of the results.

The main contribution issues of the proposed methodology are the following: (A) It is adopted for *software component-based technologies* that allows wide reuse of the software, and increases development speed. (B) *Multi-objective quality attributes* can be taken into consideration at the same time. Any type of attribute can be taken for analysis, because a component can be supplied with non-limited number of model types (resource, behaviour, reliability, safety, cost models). (C) The methodology supports design of *multiprocessor systems*. It allows mapping software components onto hardware nodes, thereby increasing design flexibility and efficiency. (D) DeSiX enables assessment of the quality attributes based not only on the static architecture, but also on the predicted *dynamic behaviour* of a complete SW/HW system. (E) It adopts a multi-objective Pareto analysis[18] for design space exploration, thereby helping the designer to easily cope with contradictory requirements.

In this paper we apply our methodology to a design case study of a Car Radio Navigation (CRN) system. Our analysis includes timeliness, resource utilization (CPU, bandwidth) and reliability. Ongoing work includes the analysis technique extension for other system properties; including reliability and cost.

Structure of the paper is as follows. In Section 2 we summarize the Robocop component model used as a middleware in the CRN system. In Section 3 we outline the design space exploration process by explaining its main steps. The scenario simulation approach, which is a core part of the DeSiX methodology, is explained in Section 4. The practical aspects of the methodology are illustrated through application to the Car Navigation case study, which is described in Section 5. We conclude the paper with a discussion of related work in Section 6 and conclusions in Section 7.

2. COMPONENT-BASED SOFTWARE ARCHITECTURE

As we are focusing on design space exploration techniques for component-based architecture[1] , let us first give a short outline of the used component-based framework. Within the international Space4U project*, we have developed a Robocop Component-Based Architecture (CBA)[21] . The Robocop CBA complies with the

*Space4U is part of the ITEA research program funded by the European Union.

common COTS standard for component-based development. This architecture is developed for middleware in consumer devices, with an emphasis on robustness and reliability. The Robocop CBA is similar to CORBA[15] and Koala[16] , but enables more efficient quality attributes realization via *modelling* techniques. For example, a component designer can supply a *component resource model* along with the component's executable code. A designer composes an application from a number of components and can predict the application resource consumption, using the set of such component resource models.

The Robocop CBA is highly efficient, particularly for embedded systems and Systems-on-Chip. Firstly, it allows decomposing the processing software into functional blocks and then mapping these blocks on the architecture in an optimal way. Secondly, the supporting Robocop Run-Time Environment has a built-in Quality-of-Service framework, that enables run-time monitoring and enforcement of the system resources. Finally, the freedom of defining the necessary types of models allows addressing not only the processor usage, but also other important attributes (e.g. memory, bus load, reliability and robustness).

The primary goal of the component paradigm is that systems can be built from a set of existing components. These components may have been developed by the organization that constructs the system or by some third-party on the market. This approach allows (A) to avoid the costly build-from-scratch development cycle and (B) to flexibly select components (from among a set of similar ones) that provide the best fit in terms of quality attributes.

Let us now define the Robocop component model in more detail. A Robocop component is a set M of possibly related models, as depicted in Fig. 1(a). 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*, *reliability model*, and *behaviour model*.

A component offers functionality through a set of 'services' P (see Fig. 1(b)). Services are an executable entities, which are the Robocop equivalents of *public classes* in object-oriented (OO) programming. More formally, the arbitrary executable model m is specified in Fig. 1(c).

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*. The binding between service instances in the application is made via a pair of provides-requires interfaces.

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.

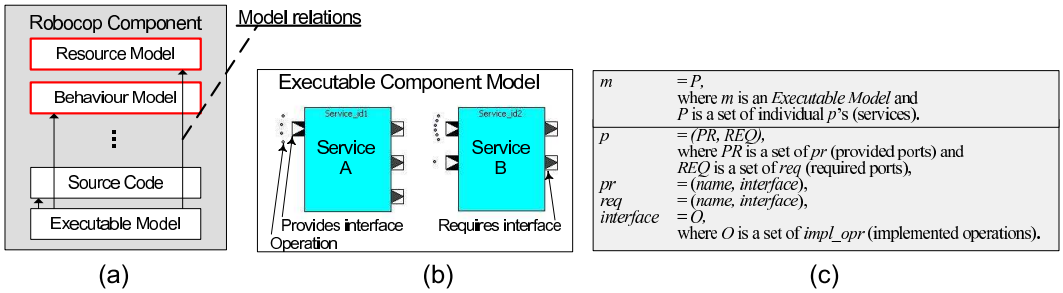

Figure 1. (a) Robocop component model, (b) example of executable component, (c) specification of a component executable model.

The Robocop CBA implies no implementation-level constraints. The architecture has no limitations on programming languages and platforms. A service can implement any number of threads. Besides this, both synchronous and asynchronous communication are possible.

3. DESIGN SPACE EXPLORATION METHODOLOGY

In this section we present the DeSiX design space exploration methodology and mention its main assumptions.

At the system design phase, we assume that the components are already available (no need to build from scratch). The component development process stays out of the scope of this paper. For the detailed information on this topic see the Robocop tutorial[21] . The second assumption is that the necessary component models (resource, reliability, behaviour models) are specified by a component developer and provided in the shipped component package. However, the construction rules (meta-model) for each model are specified by the Robocop framework. The meta-models are described below in this paper.

The DeSiX methodology comprises several phases (see Fig. 2), that can be iteratively repeated till the best design solution is found. In the above-mentioned figure, these phases are represented in the core circle. The periphery circle describes tooling support for easy adaptation of the DeSiX methodology in the design place. For this purpose, we have developed a toolkit called Robocop Real-Time Integration Environment (RTIE). The toolkit guides the designer through the DeSiX phases, automates the complex simulation and compilation operations, and gives him necessary graphical means for the component composition and design alternative analysis tasks.

Let us outline the methodology phases. A detailed description is given in the following subsections.

Software specification. After requirements analysis, the designer selects available software components (from the RTIE Repository), whose services *will* satisfy defined functional requirements and *may* satisfy extra-functional requirements. By means of the RTIE Graphical Composer, the designer instantiates the services and binds them together, thereby specifying the so-called software component composition. A composition example is depicted in the Fig. 3 and represents an MPEG-4 coding application from the case-study discussed in[20] . In

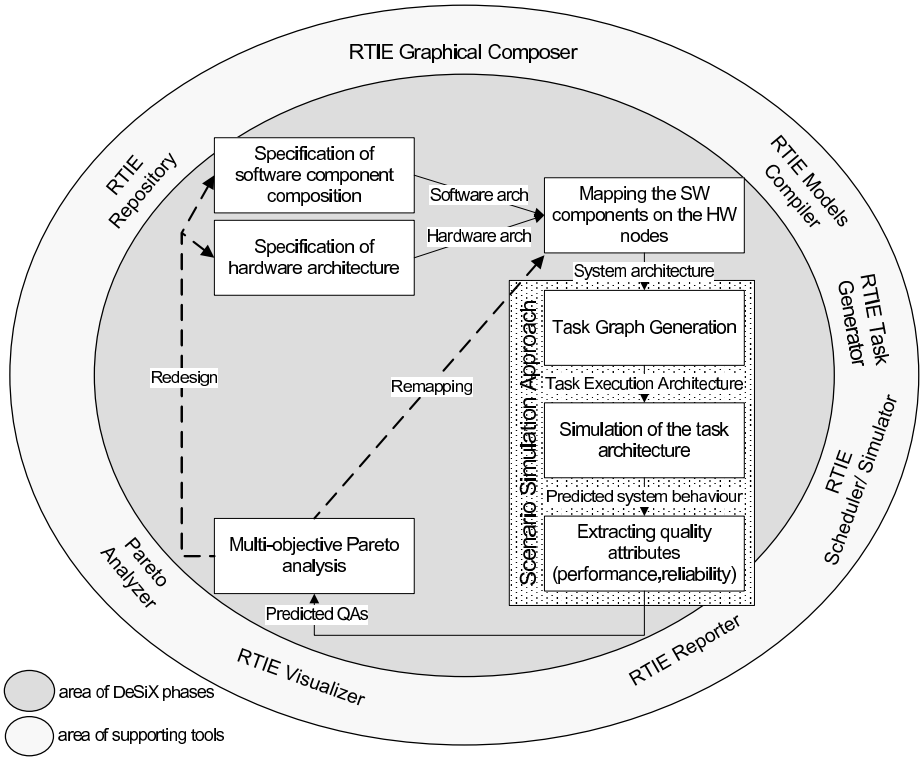

Figure 2. Main phases and supporting tools of the DeSiX methodology.

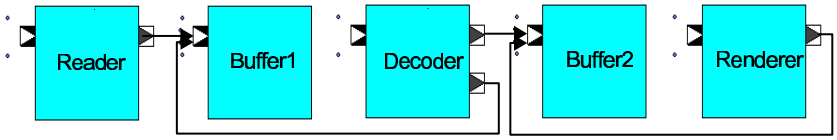

Figure 3. Example composition of software components into an MPEG-4 coding application.

this example, there are 5 services instantiated and bound together in order to read a bit stream, decompress it and render the decoded video on a screen. The buffers are used to store the intermediate data between the main processing steps. The process flow in a component-based application is defined by self-triggering (active) services or by flow triggers implemented on a higher application level. In this example, there are three active services: Reader, Decoder and Renderer. Each 40 ms Reader reads the bit stream and stores data in Buffer1; Decoder gets data from Buffer1, decodes it and stores a frame in Buffer2; Renderer gets a frame from Buffer2 and displays it on a screen. Note that each component contributing his service to an application should encapsulate a set of models (*resource*, *behaviour*, *reliability*, etc) in its distribution package.

Hardware specification. The hardware (HW) architecture specification can be done in parallel with software specification. The DeSiX methodology does not constrain the HW architecture to a specific topology, number of processing nodes, type of memory, buses and scheduling policy. However, it requires modelling of the

HW elements. The modelling aspects are (but not restricted to): processing speed, memory addressing type, bus bandwidth, reliability, cost per element, etc. The meta-models are defined by the DeSiX methodology. The designer selects the hardware elements from the RTIE Repository and combines them on the hardware graphics plane. An example of the HW architecture is depicted in Fig. 4.

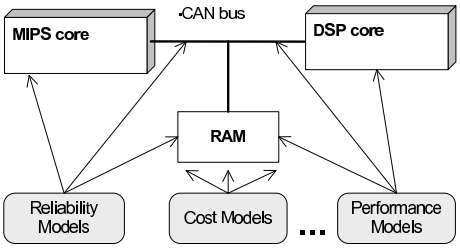

Figure 4. Example hardware architecture specification.

Mapping. Once the software composition and hardware architecture are specified, the mapping of the software components on the hardware cores can be performed. The mapping dictates on which processing core each component should be executed. Various mapping alternatives are possible in this stage. Each of the alternatives represents a static system architecture. The mapping example is given in Fig. 5.

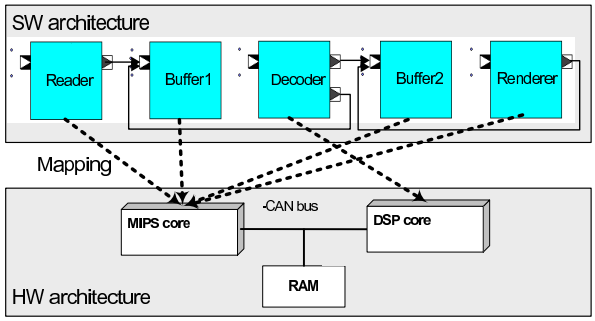

Figure 5. SW/HW mapping phase.

Here, the Decoder service is mapped on the DSP processor because it provides computational expensive operations. The rest of the services are mapped on the general-purpose MIPS core. Note, that this mapping imposes Decoder-to-Buffer communication via the bus. The next phases are required to synthesize the dynamic system architecture or, in other words, behaviour of the system.

Execution Architecture. A system *cost* attribute can be found analytically given a static architecture and cost values of its constituent parts. However, for the prediction of other important system attributes (performance, timeliness, power consumption and reliability) behavioral characteristics of a system are needed. In our approach, we obtain these characteristics through the *scenario simulation method*[19] . This method assembles a model of the execution architecture by composing models of the behavior of individual software components,

models of resource use and models of resource management policies (such as scheduling). The scenario simulation method is described in more details in the next section. As an output, the method provides the following data: (a) specification of the tasks (processes) running in the system for the designed architecture, (b) data and communication dependencies between the tasks, (c) processing nodes, components and their operations involved in each task (see Fig. 6.A), (d) execution timeline for the tasks acquired from simulation (see Fig. 6.B) and (e) hardware resource consumption (performance) of the designed architecture. These models key characteristics of the dynamic view on the system architecture.

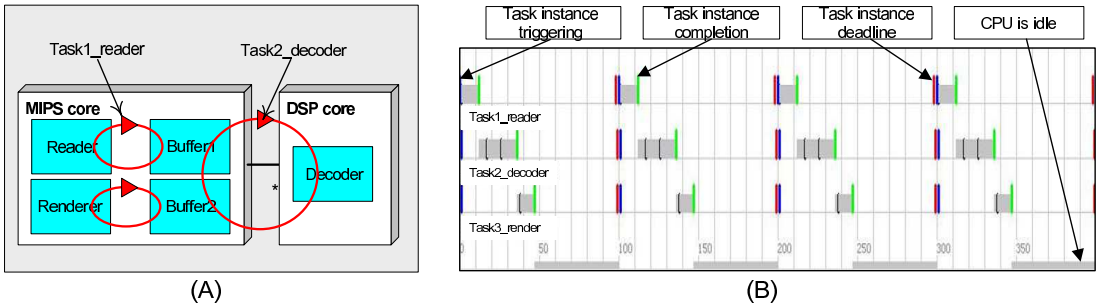

Figure 6. (A) Example of a task distribution over processing nodes and software components, (B) Example of a task execution timeline

Reliability Analysis. The model of execution architecture constructed in the previous step is subsequently used as the skeleton for modelling and analysis of reliability properties of the architecture. As depicted in Fig. 7, the reliability models of the constituent software and hardware elements are mapped on the predicted execution timeline. A reliability model of an element specifies a probability of success (opposite to probability of failure) of that element. A component reliability model describes probability of success (no erroneous response) of each operation implemented by the component. The probability success value can be extracted in the component development stage by endurance or regression tests. The execution timeline (see Fig. 6.b) shows which operations were executed and how many times per defined simulation period each operation was executed. Thereby, aggregating the probabilities of success for the operations, nodes, memories and busses, we can calculate an architecture's reliability for certain period.

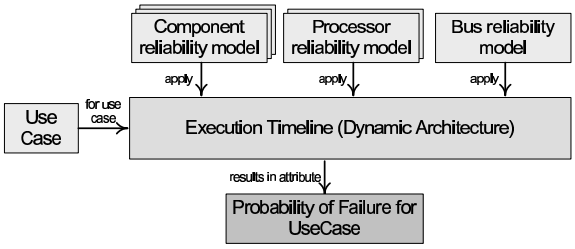

Figure 7. Acquiring reliability per use case from the dynamic architecture and corresponding reliability models

The operational semantics for system reliability calculation is presented in Appendix I.

Pareto multi-objective analysis. Inserting the number of specified architectures into the RTIE Pareto analysis tool, we strive for finding a dominant architecture solution, or a number of optimal non-dominant solutions. Pareto analysis for multi-objective optimization is a powerful means for resolving conflicting objectives.[18] The multi-objective optimization problem does not yield a unique solution. Instead, it yields a set of solutions that are Pareto optimal, which form the Pareto frontier design space. In Fig. 8 we depict a Pareto diagram for finding optimal architectures with two-dimensional objective: *system cost* and *critical task latency*. Each architecture solution is placed on the diagram according to its attribute values. Pareto curve passes through the optimal non-dominant architectures A, C and E. The rest of the solutions (above/right from the curve) are not optimal. Obviously, the Architecture B is worse than the Architecture C because it more expensive and slower. The same holds for the Architecture D. The further analysis can be done for the optimal architectures. If, for instance, the timing requirement for the critical task is 29 ms, then the Architecture E can be rejected as a risky solution. If the system cost is a dominant attribute, than we may consider Architecture C or E as preferred optimal solutions. The three- and four-dimensional optimization is more challenging for visualization, but conceptually the same principle holds.

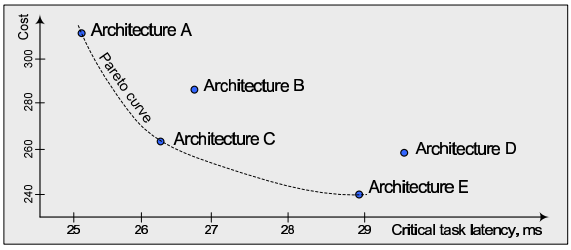

Figure 8. Two-dimensional objective design space exploration diagram, using the Pareto curve.

4. SCENARIO SIMULATION APPROACH

As depicted in Fig. 2, the scenario simulation approach is an integral part of the DeSiX methodology. The approach enables early predictions of performance and behavioural properties of a designed architecture. The output from the simulation serves as a skeleton for further analyses of attributes that depend on the timing of internal system actions (performance, security, reliability, etc).

The approach is based on three concepts:

- *Models* of behaviour and resource usage of the system's component
- *Identification of Key Execution Scenarios* of the complete system; such as situations in which the resources are potentially overloaded,

- *Simulation* of the scenarios which results in timing behaviour of the architecture.

The main assumption for the approach is that each component has accompanying *resource-* and *behaviour* models, and that each hardware part has a *performance model*. The *resource model* contains processing, bandwidth and memory requirements of each operation implemented by the component. The *behaviour model* specifies for each implemented operation a sequence of external calls to operations of other interfaces. The external call is a (synchronous or asynchronous) invocation of other interface's operation made inside the implemented operation. Besides this, the behaviour model may specify thread triggers, if they are implemented by the services of the component. The *performance model* of a hardware part specifies its processing capabilities. For a processing core it is a frequency rate and scheduling policy; for memory it is a memory size and addressing type; for networking means - bus size in bits, frequency and scheduling policy.

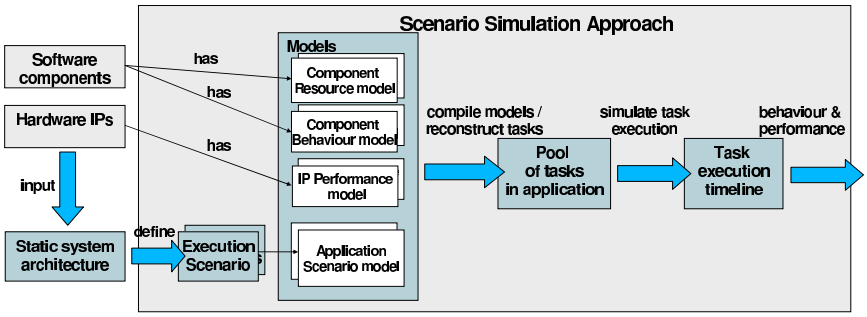

Figure 9. Workflow phases of the scenario simulation approach.

The workflow of our approach (see Fig. 9) is based on the three following phases.

4.1. Scenario Identification

For a static system architecture, the designer defines a set of resource-critical scenarios[†] and for each of them specifies an *application scenario model*. In the scenario, the designer may specify stimuli (events or thread triggers) that influence the system behaviour. For a stimulus, the designer may define the burst rate, minimal interarrival time, period, deadline, offset, jitter, task priority, and so on. In Fig. 10, the designer introduces three stimuli (periodic triggers) that calls reading, decoding and rendering operations each 40 ms. By defining the stimuli, the designer specifies autonomous behaviour of the system, or emulates an environmental influence (interrupts, network calls) to the system. Later, the code generation tool may implement these stimuli on the application level. Finally, for each critical scenario, a developer initializes (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 in the parameter values, or in a burst rate of a certain event.

[†]Critical scenarios are the application execution configurations that may introduce processor, memory or bus overload.

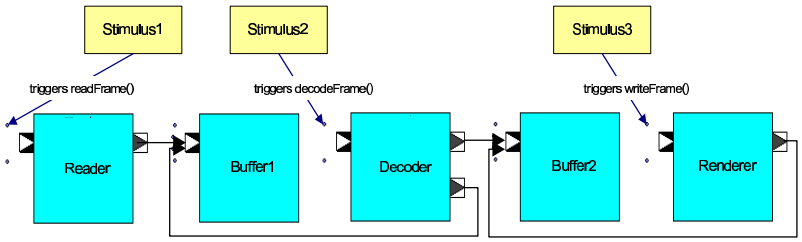

Figure 10. Structure of a critical execution scenario.

The scenario-based approach reduces the designer efforts, allowing him to analyse only scenarios of concern, but not all possible system configurations and execution profiles in one view.

4.2. Generation of Scenario Tasks

The *application scenario*, *resource*, *behaviour* and *performance models* are jointly compiled by the RTIE Compiler. 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 and components. For instance, the *task periodicity* may be specified in an application scenario model, whereas the information about the *operation call sequence* comprising the task is spread over relevant component behaviour models. The compiler combines these two types of data in the task information containing period, jitter, offset, deadline and operation sequence call graph.

The task generation concepts for the decoding example works as follows. The behaviour model of the Decoder component specifies the operation call sequence of the operation decode(): getFrame(), storeFrame() (see Fig. 11.a). Afterwards, the compiler gathers from related behavior models the behaviour of the latter two operations. The Buffer's operation getFrame() calls one operation belonging to other interfaces: ILogData.logEvent()(see Fig. 11.b).

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. 11.c). Thus, the complete reconstructed sequence of the operations executed in the task is as depicted in Fig. 11.d.

A resource consumption property of each operation in this sequence is specified in the *claim* primitive in the related component *resource model*. Knowing this data, we can calculate total resource consumption of the task. For example, the CPU time used by the task (execution time) on a reference processing core is the sum of CPU times used by the operations composing the task. In Fig. 11.d, 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 stimulus properties that are specified in the *scenario model*.

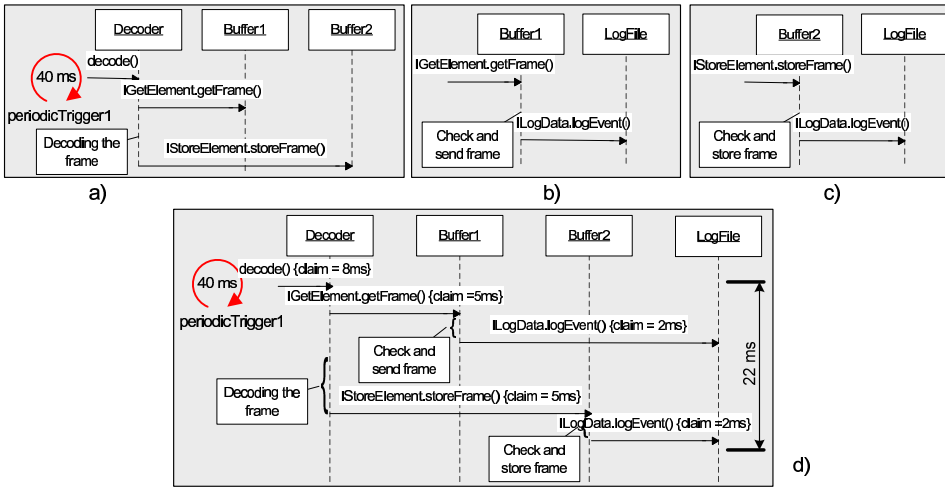

Figure 11. a) sequence of operation calls (behaviour) of decode() operation; b) behaviour of operation getFrame(); c) behaviour of operation storeFrame(); d) decoding task generated from the models.

4.3. Scenario Simulation and Performance Analysis

An application developer deploys one of the virtual schedulers (from RTIE tool) to simulate the execution of the generated task pool, thereby simulating the execution of the defined scenario. The simulation scheduling policy of the application execution should be compliant with the scheduling policy of the target operating system. The resulting data from the scheduler is a *task execution timeline*. This timeline allows extracting the behaviour, real-time, memory- and bus-related performance properties of an application (system). The timeline example is given in Fig. 6.b. From the timeline it is known which operation is executed on each node at every moment.

The scenario simulation approach is specified in our previous publication[19] in more detail. The following section describes a case study validating the proposed DeSiX methodology.

5. CASE STUDY: DISTRIBUTED CAR RADIO NAVIGATION SYSTEM

The case study for a distributed in-car radio navigation system was selected from the case study for Modular Performance Analysis given in[17] . The system has three major logical/functional blocks (see Fig. 12).

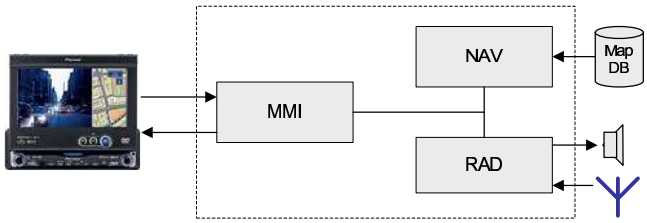

Figure 12. Overview of the functional blocks of the car radio navigation system.

- "The man-machine interface (MMI) that takes care of all interactions with the end-user, such as handling key inputs and graphical display output.
- The navigation functionality (NAV) responsible for destination entry, route planning and turn-by-turn route guidance giving the driver both audible and visual advices. The navigation functionality relies on the availability of a map database and positioning information (for example tacho signal and GPS, both not shown here).
- The radio functionality (RAD) responsible for basic tuner and volume control as well as handling of traffic information services such as RDS-TMC (Traffic Message Channel). TMC is broadcasted along with the audio signal of radio channels."

Key issues of the case study. The general goal of the case study is to validate the proposed design space exploration methodology. Technically, we are going to address the following goal: *given* a set of functional and real-time requirements, as well as a set of software components and hardware blocks, to *find* a set of optimal architecture solutions, *taking into account* four quality objectives - critical task latencies, performance sensitivity, system reliability and cost.

The given system requirements are portrayed in Table 1.

Table 1. Requirements for a car radio navigation system.

Req. ID	**Requirement description**
F1	System shall be able to gradually (scale = 32 grads) change the sound volume
RT1	The response time of the operation F1 to the user is less than 200ms (per grade)
F2	System shall be able to find and retrieve an address specified by user
RT2	The response time of the operation F2 to the user is less than 200ms
F3	The system should be able to receive and handle TMC message
RT3	The response time to the user of the operation F3 is less than 1000ms

Available software components and their models. Assume, we have a set of available components from the RTIE Repository (actually, we have prepared them for a case study): MMI, RAD, and NAV components, each carrying identically named service. The services, their provides/requires ports and operations are given in Fig. 13.

The MMI service provides IGUIControl interface and requires to be bound to IParameters and IDatabase interfaces. The IGUIControl interface provides access to three implemented operations: setVolume (handles set volume rotary button request from the user), setAddress (handles set address keyboard request from the

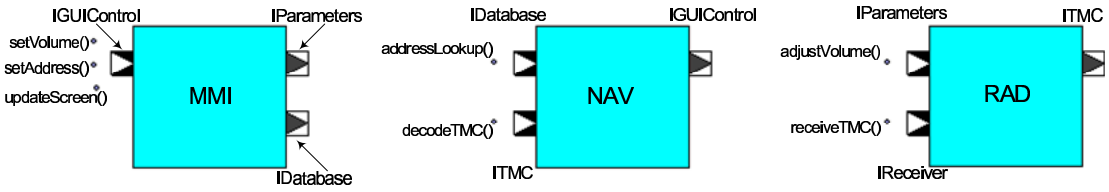

Figure 13. Services used for the case study.

user) and updateScreen (updates the GUI display of the device). The NAV service provides IDatabase, ITMC interfaces and requires operations from the IGUIControl interface. The IDatabase interface gives access to addressLookup() operation, which queries the address in the database and finds a path to this address. The ITMC interface provides an access to decodeTMC() operation. The RAD service provides IParameters, IReceiver interfaces and requires ITMC interface. The two operations implemented by this service are adjustVolume() and receiveTMC().

Each component was accompanied with their corresponding *resource*, *behaviour* models (see simplified version in Fig. 14), *reliability* and *cost models*. The resource model specifies resource requirements per individual component operation, while the behaviour model also describes the underlying calls to other operations per component operation as well as thread triggers (if existing) implemented by this operation. The resource usage data per operation has been extracted by testing and profiling of each individual component. The operation behaviour data has been generated from the component source code. Reading the RAD model, we can see that the operation adjustVolume() calls once synchronously IGUIControl.updateScreen() operation. The maximum CPU claim of the operation adjustVolume() equals to 1E5 cycles. The CPU claim numbers are acquired by profiling on a reference RISC processor. Note that the CPU claim of called IGUIControl.updateScreen() operation is specified in the MMI resource model - 5E5 cycles. The model also shows the bus usage of the adjustVolume() operation: 4 bytes. That means the operation sends outside (in this case as an argument of updateScreen() operation) 4 bytes of data.

Software architecture specification. Following the DeSiX methodology, we composed a service (component) assembly (see Fig. 15) from the available services. The three services were instantiated and bound together via pairs of their provides/requires interfaces. This assembly satisfies the three above-mentioned functional requirements F1, F2 and F3.

Hardware specification and mapping. The next phase is to define a hardware architecture and map the software components on it. From the MPA case study[17] , we took five alternative architectures with different mapping schemas (see Fig. 16). Note that the capacity of the processing cores and communication infrastructure is realistic - they were taken from the datasheets of several commercially available automotive CPUs. The multi-objective design space analysis will be carried out against these five solutions.

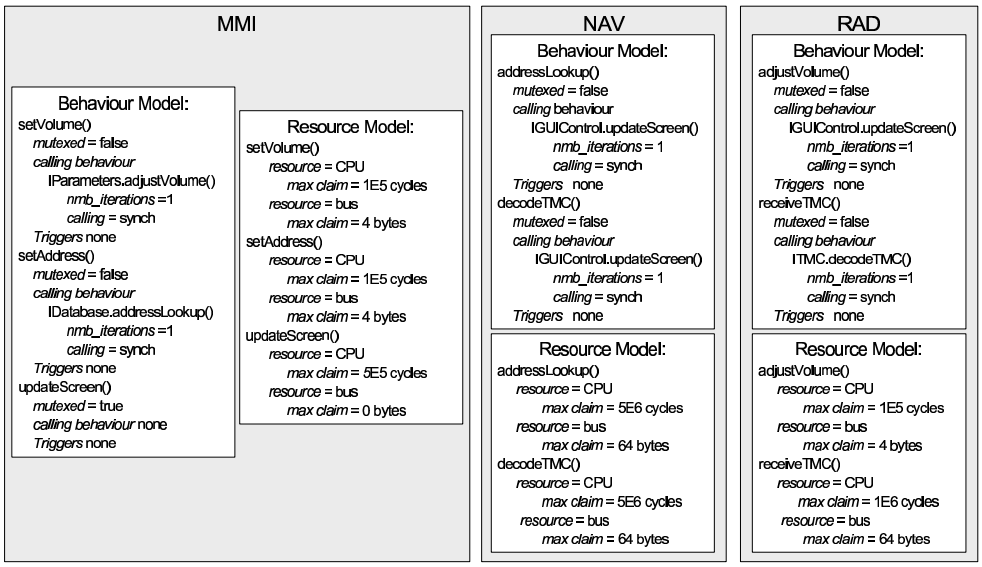

Figure 14. Behaviour and resource models of the selected components.

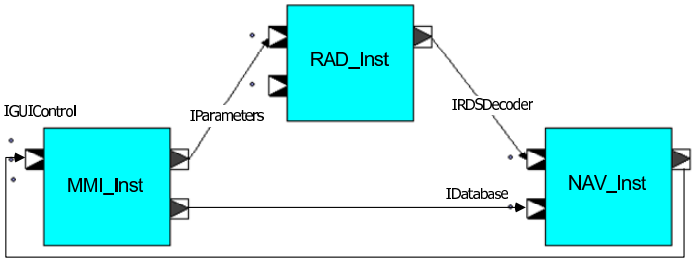

Figure 15. The component assembly of the car navigation system.

Having defined the system architecture we can proceed to the scenario simulation approach workflow to find the behaviour and performance attributes of the system.

5.1. Scenarios and Simulation

For our case study, we have selected three distinctive scenarios (defined in[17]) in order to efficiently access the architecture against the six defined requirements. These scenarios impose the highest possible load on the hardware resources in order to evaluate the real-time requirements RT1, RT2 and RT3.

"Change Volume" scenario: The user turns the rotary button and expects instantaneous audible and visual feedback from the system. The maximum rotation speed of the button is 1 sec from lowest to highest position. For emulating this user activity, we introduced (by RTIE graphical means) the VolumeStimulus task trigger (see Fig. 17.a). The parameters of the trigger are defined in the following way: the event period is set to 1/32 sec (the volume button scale contains 32 grades). The deadline for the task is set to 200 ms, according to the requirement R1. The trigger and component assembly resemble a model of the scenario (model structure is

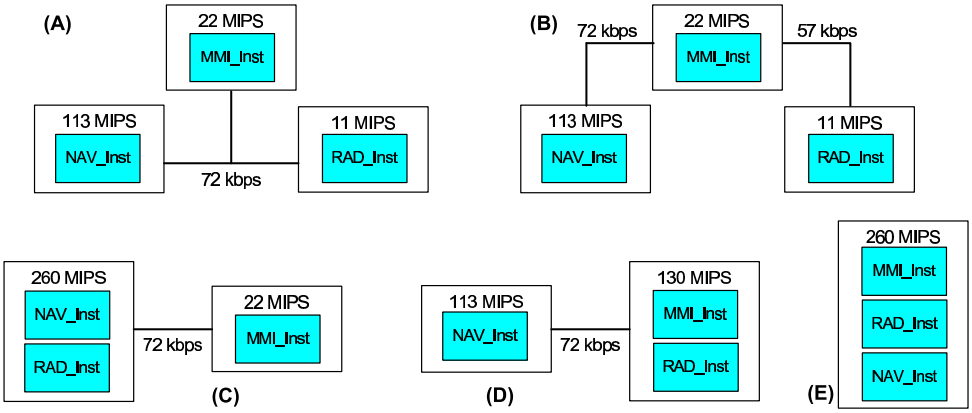

Figure 16. Alternative system architectures to explore.

not discussed here).

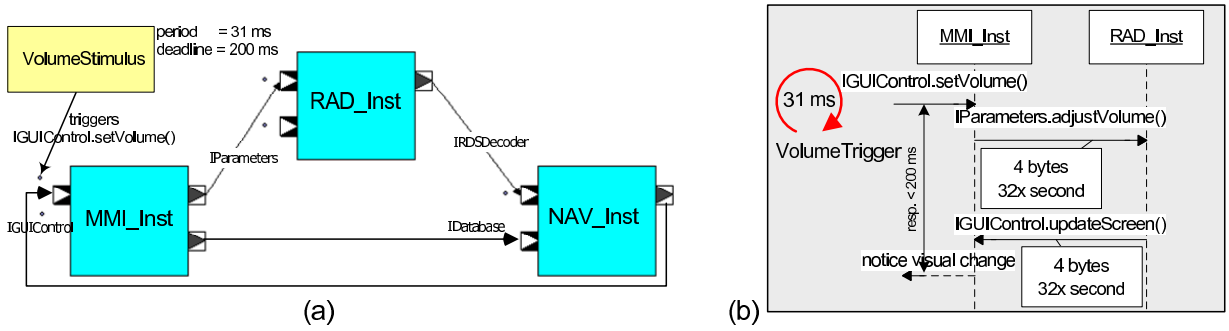

Figure 17. (a) Graphical representation of "Change Volume" scenario; (b) Generated task of the scenario.

Applying the task generation tool (generation concept is shown in Fig. 11) on this scenario, we extracted from the behaviour models of participating components the message sequence diagram of operation calls involved in the task execution. The obtained task is shown in Fig. 17.b. The task is executed periodically (with a period of = 31 ms) and passes through two service instances: MMI_Inst and RAD_Inst.

"Address Lookup" scenario. Destination entry is supported by a smart "typewriter" style interface. The display shows the alphabet and the user selects the first letter of a city (or street). By turning a knob the user can move from letter to letter; by pressing it the user will select the currently highlighted letter. The map database is searched for each letter that is selected and only those letters in the on-screen alphabet are enabled that are potential next letters in the list. We assumed that the worst-case rate of the letter selection is 1 times per second. This user activity was emulated with a LookupStimulus trigger. The stimulus period was set to 1000 ms. The deadline for the address lookup task is 200 ms, according to real-time requirement RT2. Fig. 18.a depicts the model of the address lookup scenario.

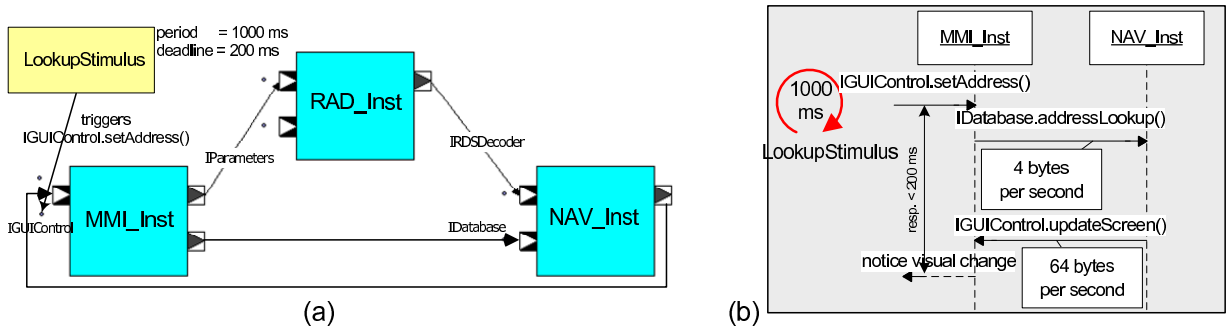

Figure 18. (a) Graphical representation of "Address Lookup" scenario; (b) Generated task of the scenario.

The task generation procedure outputs the message sequence diagram of operation calls involved in the task execution. The obtained task is shown in Fig. 18.b. The task is executed periodically (1000 ms) and passes through two service instances: MMI_Inst and NAV_Inst. The task deadline is 200 ms.

"TMC Message Handling" scenario. RDS TMC is a digital traffic information enables automatic replanning of the planned route if a traffic jam occurs ahead. TMC messages are broadcasted by radio stations together with stereo audio sound. Traffic information messages are received by the RAD service (in a worst case 1 time per 3 seconds). RDS TMC messages are encoded; only problem location identifiers and message types are transmitted. The map database of the NAV service contains two look-up tables that allow the receiver to translate these identifiers into map locations and human readable RDS TMC message texts. We introduced TMCStimulus trigger emulating the TMC messages from a radio station. The period is set to 3000 ms. The deadline for the TMC handling task is set to 1000ms, according to the real-time requirement RT3. Fig. 19.a depicts the model of the TMC handling scenario.

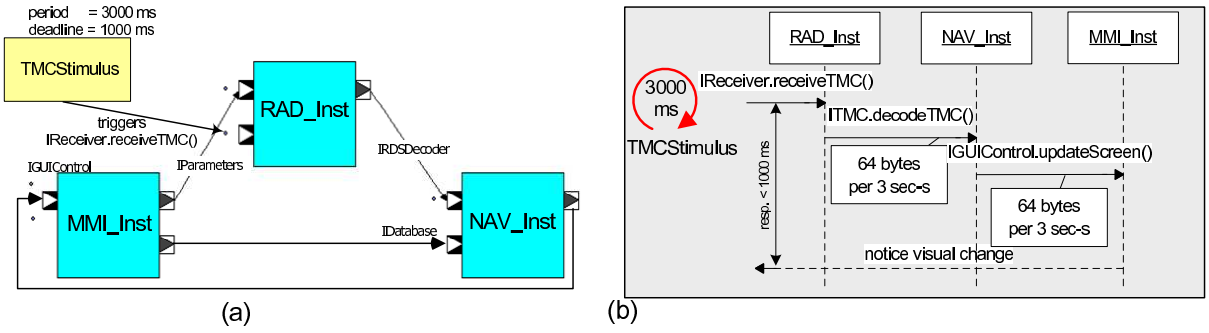

Figure 19. (a) Graphical representation of "TMC Handling" scenario; (b) Generated task of the scenario.

The task generation procedure outputs the message sequence diagram of operation calls involved in the task execution. The obtained task is shown in Fig. 19.b. The task is executed periodically (3000 ms) and passes through three service instances: RAD_Inst, MMI_Inst and NAV_Inst. The task deadline is 1000 ms. Note that

the radio only receives the TMC messages, decoding is performed on the navigation subsystem because the map database is needed for that. The fully decoded and relevant messages are forwarded to the user.

The scenarios sketched above have an interesting property: they can occur in parallel. TMC messages must be processed while the user changes the volume or enters a destination address at the same time. Therefore, we combine these three scenarios in two in order to get really worst-case load on the system resources during simulation. Consequently, a *ScenarioA* is defined as a combination of the SetVolume and TMCHandling scenarios, and a *ScenarioB* combines the AddressLookup and TMCHandling scenarios. From the processing points of view, both new scenarios have two tasks executing in parallel.

Scenario simulation. Following the DeSiX methodology, we simulated (with RTIE Rate Monotonic Scheduler) the execution of the two defined scenarios on each of the five system architectures (see Fig. 16). Before simulation the tool performs preprocessing of the computation and communication time data. For each of the processing node, the execution times of all operations to be executed on the node are calculated from the *component resource* and *node performance* models (execution time =*CPU claim* value * processor speed). The communication time of the operation calls made through processor boundaries, is calculated by dividing the *bus claim* value of an operation on bus bandwidth value, defined in a performance model of the bus. Additionally, the task synchronization constraints are considered during the simulation (this issue stays out of the paper scope).

The scenario simulation results in (a)predicted system behaviour, (b)resource consumption of a system for each scenario and task best-case, average and worst-case latencies. First, we analyzed the predicted worst-case task latencies against the real-time requirements RT1, RT2 and RT3 for each of the five architecture solutions. Table 2 portrays the comparison of the analysis data.

Table 2. Verifying the real-time requirements against the predicted task latencies for the five architectures.

Req. ID	Requirement Value	Arch. A	Arch. B	Arch. C	Arch. D	Arch. E
RT1	200 ms	37.55 ms	37.55 ms	30.52 ms	9.18 ms	3.58 ms
RT2	200 ms	86.51 ms	86.51 ms	61.49 ms	63.79 ms	21.05 ms
RT3	1000 ms	375.05 ms	395.05 ms	101.71 ms	114.12 ms	46.02 ms

Performance sensitivity. Analyzing Table 2 data, we can conclude that all five architectures satisfy the given real-time requirements. Architectures A and B can be considered as fast enough, architecture E is the fastest solution. Because all solutions satisfy the real-time requirements it is more interesting to analyze the *sensitivity* of the architectures to changes in the input event rates (arrival periode of the three stimuli). For that we increased the data rate of the three stimuli by 5% (i.e. VolumeStimulus to 33.6 events/s, LookupStimulus to 1.05 events/s and TMCStimulus to 0.35 events/s). Then we re-simulated the adjusted scenarios and obtained new task latencies. Table 3 represents the increase of the latency of the TMC handling task in percentage to

the normal latency per architecture. For instance, end-to-end delay of the TMC message handling task for architecture A increases by 57.6%. This increase helped to identify the bottleneck in the Architecture A. The processor 22_MIPS has been overloaded by the execution of IGUIControl.updateScreen operation of MMI_Inst service instance.

Table 3. Latency increase of the TMC handling task in percentage to the normal latency (performance sensitivity) per architecture.

Task name	**Latency Increase Arch. A**	**Arch. B**	**Arch. C**	**Arch. D**	**Arch. E**
TMC handling	57.6%	51.1%	3.2%	3.1%	0.0%

Reliability. As we mentioned in Section 4.3, the found execution timeline shows which operation is executed on each node at every moment in time. As depicted in Fig. 7, applying given reliability models of components and hardware IPs (processing node, bus, memory) on the timeline, we can assess the total system reliability for the simulation time. The hardware IP reliability model contains probability of success of the IP per working period of one hour. The software component reliability model contains data on the probability of success of each implemented operation (probability of correct response of the operation). The system probability of success is calculated as a product of (A) cumulative hardware IP blocks probability of success for the simulation period; and (B) cumulative probabilities of success of the operations invoked during the simulation period. The found reliability for each architecture is given in Table 4.

Table 4. Calculated reliability values for each architecture.

Attribute	**Arch. A**	**Arch. B**	**Arch. C**	**Arch. D**	**Arch. E**
Reliability	96.94%	95.53%	97.93%	97.93%	98.35%
Probability of failure (100%-Rel)	3.06%	4.47%	2.07%	2.07%	1.65%

The reliability data for the component and hardware IP models has been chosen hypothetically. This data was not measured on actual software or hardware components. However, the data is used to illustrate the method if suitable data is available. We correlated the probability of success of an operation with the operation source code complexity.

Interpreting the found reliability for each architecture, one may notice that Architecture B has low reliability value. This is mainly due to the hardware topology, where the communication between processors 113_MIPS and 11_MIPS is done via processor 22_MIPS, which imposes relatively complex data and control paths from one processor to another. The Architecture E has high reliability because all service instances are deployed on the single 260_MIPS processor and no physical connection is involved in the control and data paths.

System cost. The system cost attribute was calculated as a cumulative cost of the system hardware and software components (integration cost was not included). The software component cost has been defined with correlation to the component source code complexity. In a real-life case the cost of a third-party component is defined by the component producer. The cost of the hardware parts has been calculated from the available market prices divided by factor of five (for the design comparison purpose the value of the factor is not critical). The total calculated cost for each architecture is given in Table 5.

Table 5. Calculated cost values for each architecture.

Attribute	Arch. A	Arch. B	Arch. C	Arch. D	Arch. E
Cost, euro	290	305	380	335	340

The obtained cost values can be interpreted as follows. The Architecture C is most expensive because it combines the fast 260_MIPS processor together with bus link and 22_MIPS processor. The Architecture A is least expensive because it contains three relatively cheap (slow) processors and one bus link.

5.2. Analysis of Architecture Alternatives

The performance sensitivity, reliability and cost attributes were selected as main objectives for the design space exploration. Using the RTIE Pareto analysis tool we obtained several two-dimensional Pareto graphs. Two of them, *sensitivity vs. cost* and *reliability vs. cost* are depicted in Fig. 20). The principles of the graph and Pareto curve construction are explained in Section 3.

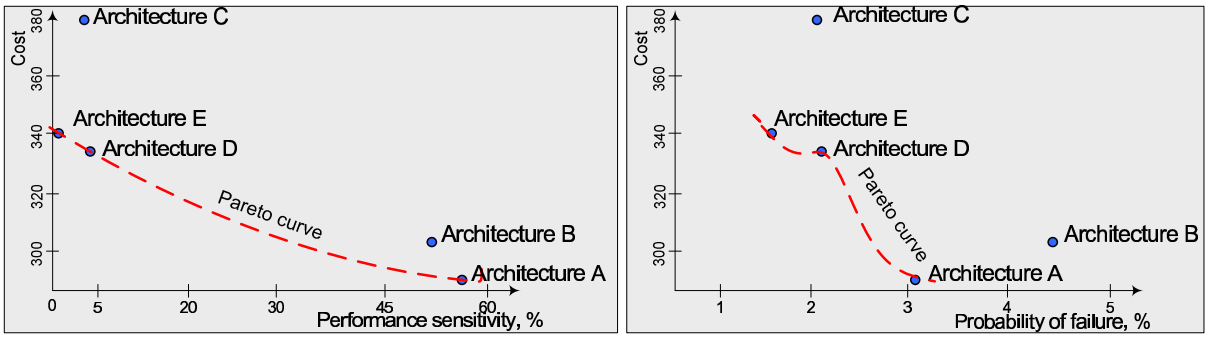

Figure 20. Multidimensional (cost-sensitivity-reliability) exploration of the five architecture alternatives.

With respect to the cost-sensitivity trade-off, the optimal architecture alternatives are E, D and A. The alternatives C and B are non-optimal. The choice among the three alternative architectures depends on a weighting function (priority) for cost and sensitivity attributes. If the cost is a most important aspect - the Architecture A should be chosen. If the performance sensitivity is a critical factor, then Architecture A is not the best candidate.

With respect to the cost-reliability trade-off, again the optimal alternatives are E, D and A, though D does not lie on the hypothetical ideal Pareto curve. If the cost weighting function is higher than reliability weighting function - Architecture A can be adopted for further development. Concluding, Architectures A and E can be considered as preferable architectures, based on this multi-dimensional Pareto analysis.

6. RELATED WORK

The multi-dimensional design space exploration is a challenging topic that has been extensively addressed in system architecture research. Principal ideas based on evolutionary algorithms are presented in[4] . Specific approaches are used for different application domains. The "Spade" technique[5] provides a methodology for architecture exploration of heterogeneous signal processing systems. A framework based on real-time calculus[6] is proposed for design space exploration of network processor architectures. Related work on the simulation-based design exploration of system-on-chip communication architectures is presented in[7] and[8] .

For a software component-based system domain, few techniques exists: micro-architecture modelling[9] and scenario-based static performance evaluation[10] . The above-mentioned approaches are focused mostly on the performance-cost-power trade-off. The research on the early assessment of the reliability quality attribute is rarely linked with the design space exploration topic. An exception is the technique[11] based on symbolic search and multi-granular simulation. Scenario-based approaches for early assessment of component-based software architectures are described in[12] and[13] . A recent method based on Markov models for reliability estimation of component-based systems is given in[14] . None of the above-mentioned approaches address the multi-objective design space exploration for software component-based multiprocessor systems.

7. CONCLUSION

In this paper, we have presented a DeSiX design space exploration methodology for software component-based systems on multiprocessor architectures. For validation purpose, we have exploited this technique on a case study of a car radio navigation system. The case study revealed that the DeSiX methodology enables: (a) quality attribute predictions at the early design phases and (b) further analysis of the proposed design solutions, which is based on the found attributes.

The main contribution issues of the proposed methodology are: (A) It is adopted for *software component-based technologies* that allow wide reuse of the software, and increases development speed. (B) *Multi-objective quality attributes* can be taken into consideration at the same time. Any type of attribute can be taken for analysis, because a component can be supplied with non-limited number of model types (resource, behaviour, reliability, safety, cost models). (C) The methodology supports design of *multiprocessor systems*. It allows mapping software components onto hardware nodes, thereby increasing design flexibility and efficiency. (D) DeSiX enables assessment of the quality attributes based not only on the static architecture, but also on the

predicted *dynamic behaviour* of a complete SW/HW system. (E) It adopts the multi-objective Pareto analysis for design space exploration, thereby helping the designer to easily cope with contradictory requirements.

The accuracy of performance attribute prediction has been previously validated by a case study on a Linux-based MPEG-4 player[20] . The prediction accuracy on the general performance proved to be higher than 90%. In all case studies, the modelling effort required from application designer was fairly small - in the order of hours. The most of the modelling work goes to the component developer, because he should provide the component models. Thereby, the application developer may relatively easily model a system out of 100 components (scalability), because necessary models are already supplied within these components. The process enables early identification of the bottlenecks of individual alternatives and leads to selection of optimal solutions. In this paper we address strictly performance and reliability attributes, however the proposed DeSiX process enables targeting other important quality attributes (QAs), like security and availability. This extensibility is realized by open component model structure, in which new model types can be easily added.

There are certain limitations of the process. To compose a real-time application out of the Robocop components a designer can only select the components that are performance-aware - containing resource and behaviour models. Moreover, the QAs that are system-wide, like safety and security, cannot be easily localized and modeled at the component level. At present, we investigate feasible composition approaches for such QAs.

For future research, we plan to introduce analytical techniques for obtaining the quality attributes that would speed up the design space exploration process. Beside this, we aim at defining a method that could allow to find the low-level reliability data (of a software operation, processing node or bus link) in a rigorous and mature way. Finally, we strive for a validation case-study on an industrial system, that could give indispensable points for improvement.

8. ACKNOWLEDGEMENTS

We would like to thank Marcel Verhoef for providing the case study on the CRN system as an input for validation of our design space exploration process. Also, warm words of gratitude go to Jozef Hooman and Harold Weffers for their valuable feedback during the review process of this report.

REFERENCES

1. I. Crnkovic and M. Larsson. *Building Reliable Component-based Software Systems*, Artech House, 2002, ISBN 1-580-53327-2
2. K.C. Wallnau, "Volume III: A Technology for Predictable Assembly from Certifiable Components", *CMU/ESI-2003-TR-009 report*, April 2003.
3. S.A. Hissam, *et al.*, "Packaging Predictable Assembly with Prediction-Enabled Component Technology", *CMU/ESI-2001-TR-024 report*, November 2001.
4. K. Deb. *Multi-objective optimization using evolutionary algorithms*. John Wiley, Chichester, 2001.

5. P. Lieverse, P. van der Wolf, K. Vissers, "A Methodology for Architecture Exploration of Heterogeneous Signal Processing Systems", *Journ. VLSI Signal Proc. Signal, Image and Video Proc.*, vol. 29, pp. 197-207, 2001.
6. L. Thiele, S. Chakraborty, M. Gries, and S. Kunzli. "A framework for evaluating design tradeoffs in packet processing architectures", *In Proc. 39th DAC conference*, New Orleans, USA, 2002. ACM Press.
7. K. Lahiri, A. Raghunathan, and S. Dey. "System level performance analysis for designing on-chip communication architectures". *IEEE Trans. on Computer Aided-Design of Integrated Circuits and Systems*, 20(6):768-783, 2001.
8. M. Grunewald, *et al.* "A framework for design space exploration of resource efficient network processing on multi-processor SoCs". *In Procs. 3rd Workshop on Network Processors and Applications*, Madrid, Spain, February, 2004.
9. M. Vachharajani. *Microarchitecture Modeling for Design-space Exploration*. PhD thes., Princeton University, 2004.
10. J. T. Russell,"Architecture-level performance evaluation of component-based embedded systems", *In Proc. 40th DAC conference*, Anaheim, USA , 2003. ACM Press.
11. S. Mohanty *et al.*, "Rapid Design Space Exploration of Heterogeneous Embedded Systems using Symbolic Search and Multi-Granular Simulation", *In Procs. LCTES'02-SCOPES'02*, June, 2002, Berlin, Germany.
12. S. M. Yacoub *et al.*, "Scenario-Based Reliability Analysis of Component-Based Software", *In Procs. 10th Int. Symp. on Software Reliability Engineering* November, 1999, Boca Raton, Florida.
13. S. Krishnamurthy, A.P. Mathur, "On The Estimation Of Reliability Of A Software System Using Reliabilities Of Its Components", *In Procs. ISSRE 97* November 1997, Albuquerque, USA.
14. R. Roshandel *et al.* "Toward Architecture-Based Reliability Estimation", *In Procs. WADS conference*, UK, 2004.
15. T. Mowbray and R. Zahavi, *Essential Corba*, John Wiley and Sons, New York, 1995.
16. R. van Ommering *et al.*, "The Koala component model for consumer electronics software", *IEEE Trans. Computer*, 33 (3), 78-85, Mar. 2002.
17. E. Wandeler, L. Thiele, M. Verhoef, "System Architecture Evaluation Using Modular Performance Analysis - A Case Study", *Proc. 1th ISOLA Symposium*, 2004.
18. Mattson, C. A., and Messac, A., "A Non-Deterministic Approach to Concept Selection Using s-Pareto Frontiers", *Proceedings of ASME DETC 2002*, Montreal, Canada, September 2002, 2002.
19. E. Bondarev, J. Muskens, P.H.N. de With and M.R.V. Chaudron, "Predicting Real-Time Properties of Component Assemblies: a Scenario-Simulation Approach", *Proc. 30th Euromicro Conf., CBSE Track*, pp. 40-47, September 2004.
20. 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
21. "Robocop: Robust Open Component Based Software Architecture", http://www.hitech-projects.com/euprojects/robocop/deliverables.htm

APPENDIX I. Formula for Reliability Calculation

om_{ij}	Number of invocations of an operation *i* in scenario *j* (taken from the simulation timeline)
r_i	Success probability of an operation *i*, when it executes once (given by component developer)
rm_i	Reliability of a processor *n* where an operation *i* is executed (given in the reliability model of the processor)
$r_{ij} = (r_i \times rm_i)^{om_{ij}}$	Probability of success for all executions of an operation *i* in a scenario *j*

(A)

$\lvert Interact(m,n,j)\rvert$	Number of physical (via bus) interactions that components *m* and *n* have in a scenario *j* (taken from the timeline)
ψ_l	Success probability of one interaction over a bus *l* (given in the reliability model of a bus *l*)
$\psi_{mnj} = \psi_l^{\lvert Interact(m,n,j)\rvert}$	Probability of success for all interactions between *m* and *n* in a scenario *j*

(B)

N_j	Number of operations in a scenario *j* (taken from the timeline)
P_j	Probability of a scenario *j* to be executed in a use case *u* (normally = 1)
K_u	Number of scenarios in a use case *u* (normally = 1)
$\theta_u = \sum_{j=1}^{K_u} P_j \left[\prod_{i=1}^{N_j} (r_i \times rc_i \times rm_i)^{om_{ij}} \bullet \prod_{(m,n)} \psi_l^{\lvert Interact(m,n,j)\rvert} \right]$	Success probability of a use case *u* when it is executed

(C)

λ_u	Arrival rate of a use case *u*. Execution times of *u* in a unit time, e.g. an hour
$\gamma_{uT} = \theta_u^{\lambda_u T}$	Probability of success of a use case *u* in the period of *[0..T]*

(D)

Figure 21.

Formula for reliability calculation of: (A) all executions of an operation in a scenario; (B) all interactions in a scenario; (C) a use-case (scenario) when it is executed once; (D) a use-case (scenario) over certain period.