Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 712329, 18 pages
doi:10.1155/2008/712329

Research Article

Combining UML2 Application and SystemC Platform Modelling for Performance Evaluation of Real-Time Embedded Systems

Jari Kreku,[1] Mika Hoppari,[1] Tuomo Kestilä,[1] Yang Qu,[2] Juha-Pekka Soininen,[1] Per Andersson,[3] and Kari Tiensyrjä[1]

[1] *Communication Platforms, Technical Research Centre of Finland (VTT), Kaitoväylä 1, FI-90571 Oulu, Finland*
[2] *Nokia Device R&D, Elektroniikkatie 13, FI-90571 Oulu, Finland*
[3] *Department of Computer Science, Faculty of Engineering, Lund University, P.O. Box 118, SE-221 00 Lund, Sweden*

Correspondence should be addressed to Jari Kreku, jari.kreku@vtt.fi

Received 1 October 2007; Revised 19 February 2008; Accepted 16 June 2008

Recommended by Eugenio Villar

Future mobile devices will be based on heterogeneous multiprocessing platforms accommodating several stand-alone applications. The network-on-chip communication and device networking combine the design challenges of conventional distributed systems and resource constrained real-time embedded systems. Interoperable design space exploration for both the application and platform development is required. Application designer needs abstract platform models to rapidly check the feasibility of a new feature or application. Platform designer needs abstract application models for defining platform computation and communication capacities. We propose a layered UML application/workload and SystemC platform modelling approach that allow application and platform to be modelled at several levels of abstraction, which enables early performance evaluation of the resulting system. The overall approach has been experimented with a mobile video player case study, while different load extraction methods have been validated by applying them to MPEG-4 encoder, Quake2 3D game, and MP3 decoder case studies previously.

Copyright © 2008 Jari Kreku et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

1. INTRODUCTION

Future handheld mobile multimedia terminals will merge features of several currently independent mobile devices, for example, phone, music player, television, movie player, desktop, and Internet tablet. They accommodate a large number of on-terminal and/or downloadable applications that offer the user with services, whose contents may be provided by servers anywhere on Earth. The sets and types of applications running on the terminal are dependent on the context of the user. To deliver requested services to the user, some of the applications run sequentially and independently, while many others execute concurrently and interact with each others.

The digital processing architectures of terminals will evolve from current system-on-chips (SoCs) and multi-processor-SoCs with a few processor cores to massively parallel computers that consist mostly of heterogeneous subsystems, but may also contain homogeneous computing subsystems. The network-on-chip (NoC) communication paradigm will replace the bus-based communication to allow scalability, but it will increase uncertainties due to latencies in case large centralized or external memories are required.

The application and architecture development trends will increase the overall complexity of system development by orders of magnitude. The optimisation of performance, energy, and cost of the battery-powered devices while respecting the user expectations is of vital importance on one hand. On the other, the costs, risks, and time of system development require flexibility, which will be achieved through programmable/adaptable computing resources, configurable memory and communication architecture, and design methodology approach; and tools that span from application use cases until implementation design. Design methodology approaches for mobile devices have evolved from the application-specific integrated circuit (ASIC) style in 80's to platform-based design in late 90's and model-based design was introduced recently.

In the ASIC style, designers take a (architectural) specification, create a microarchitecture description, and synthesise/optimise it for speed (clock frequency), area (gate count), and power (e.g., modes and clock gating).

Platform-based design addresses the challenges of increasing design complexity of SoCs that consist typically of a few processor cores, hardware accelerators, memories, and I/O peripherals communicating through a shared bus. While the emphasis is on intellectual property (IP) design and integration, the function-architecture codesign and (micro-) architecture exploration already pave the way to the model-based approach that is the research direction today. To alleviate the scalability problems of the shared bus, the NoC architecture paradigm proposes communication centric approach for systems requiring multiple processors or integration of multiple SoCs.

Model-based approaches extend the separation of the application and execution platform modelling further. Usually the specify-explore-refine paradigm following the principles of the Y-chart model [1] is applied. In other words, a model of application is mapped onto a model of platform and the resulting allocated model is analysed. The computation and communication modelling are separated on both the application and platform sides. Recent trend is service-orientation, where the end user interactions and the associated applications are modelled in terms of services required from the underlying execution platform [2]. An obvious consequence is that the execution platform also needs to be modelled in terms of services it provides for the applications.

Both the application and platform designers are facing an abundant number of design alternatives and need systematic approaches for the exploration of the design space. For example, an application designer has to know early whether a new application or a feature is feasible on the target platform. A platform designer must be able to analyse the impacts of next generation applications on the platform even before the applications are implemented. Efficient methods and tools for early system-level performance analysis are necessary to avoid wrong decisions at the critical stage of system development.

The performance analysis models are required to capture both the characteristics of the application functionality and the architectural resources needed for the execution. Using models at a too low level of abstraction, for example, register-transfer-level (RTL) or instruction-set simulation (ISS) is not feasible: although giving accurate results, due to the vast amount of details needed the modelling effort is heavy and simulation times are long. Some high-level abstraction approaches like queuing networks (QNs) and its variants fail to exhibit the characteristics of the execution platforms.

In this paper, we present VTT_ABSOLUT2.2% (VTT ABstract inStruction wOrkLoad & execUtion plaTform UML2/SystemC2-based performance simulation)—it is a model-based approach for system-level design that is capable of performance evaluation of future real-time embedded systems and provides early information for development decisions. Applications are modelled in either unified modelling language (UML) or SystemC [3] domain as workloads consisting of load primitives. Platform models are cycle-approximate transaction-level SystemC models. Mapping

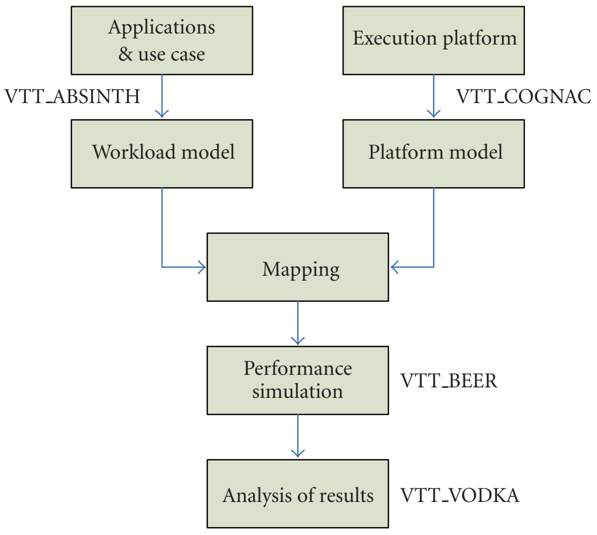


Figure 1: Y-chart model of plain VTT_ABSOLUT.

between UML application models and the SystemC platform models is based on automatic generation of simulation models for system-level performance evaluation. The workload models reflect accurately the control structures of the applications, but the computing and communication loads are abstractions derived either analytically from measured traces or using a source code-compilation approach called VTT_ABSINTH (VTT ABStract INstruction exTraction Helper) (Figure 1). The execution platform model is configured from a library of performance models using VTT_COGNAC (VTT COnfiguration GeNerator for Absolut performanCe simulation). The executable simulation model is based on the open source open SystemC initiative (OSCI) SystemC library, extended with configurable instrumentation and called VTT_BEER (VTT Binary pErformance EvaluatoR). Finally, the simulation results can be selected for analysis and viewed using VTT_VODKA (VTT Viewer of collecteD Key information for Analysis).

The tool support is based on a commercial UML2 tool, Telelogic Tau G2, and an open source SystemC simulation tool of OSCI.

The approach enables early performance evaluation, exhibits light modelling effort, allows fast exploration iteration, and reuses application and platform models. It also provides performance results that are accurate enough for system-level exploration.

2. RELATED WORK

Performance evaluation has been approached in many ways at different levels of refinement. Some other research approaches aiming at similar goals to ours by different modelling and simulation approaches are described briefly in the sequel.

SPADE [4] implements a trace-driven, system-level cosimulation of application and architecture. The application is described by Kahn process networks using YAPI [5].

Symbolic instruction traces generated by the application are interpreted by architecture models to reveal timing behaviour. Abstract, instruction-accurate performance models are used for describing architectures.

The Artemis work [6] extends the work described in [4] by introducing the concept of virtual processors and bounded buffers. One drawback of restricting the designer to using Kahn process networks is the inability to model time-dependent behaviour. In the developed Sesame modelling methodology, a designer first selects candidate architectures using analytical modelling and multiobjective optimization. The system-level simulation environment allows for architectural exploration at different levels of abstraction. The high-level and architecture-independent application specifications are maintained by applying dataflow graphs in its intermediate mapping layer. These dataflow graphs take care of the runtime transformation of coarse-grained application-level events into finer grained architecture-level events that drive the architecture model components.

The basic principle of the TAPES [7] performance evaluation approach is to abstract the involved functionalities by processing latencies and to cover only the interaction of the associated subfunctions on the architecture. These interactions are represented as inter-SoC-module transactions, without actually running the corresponding program code. This abstraction enables higher simulation speed than an annotated, fully-fledged functional model. Each subfunction is captured as a sequence of transactions, also referred to as trace. The binding decision for the subfunctions is considered by storing the corresponding trace in the respective architectural resource. A resource may contain several traces, one per each subfunction that is bound to it. The application is then simulated by forwarding packet references through the system and triggering the traces that are required for processing particular data packets in the respective SoC modules.

MESH [8] looks at resources (hardware building blocks), software, and schedulers/protocols as three abstraction levels that are modelled by software threads on the evaluation host. Hardware is represented by continuously activated, rate-based threads, whereas threads for software and schedulers have no guaranteed activation patterns. The software threads contain annotations describing the hardware requirements, so-called time budgets that are arbitrated by scheduler threads. Software time budgets are derived beforehand by estimation or profiling. The resolution of a time budget is a design parameter and can vary from single compute cycles to task-level periods. The advance of simulation time is driven by the periodic hardware threads. The scheduler threads synchronize the available time budgets with the requirements of the software threads.

SpecC [9] defines a methodology for system design including architecture exploration, communication synthesis, validation, and implementation. SpecC can be considered as a specification and modelling language that has a rich support for many system design phases. Similar properties can be found also in SystemC language that is more widely adopted in high-level system modelling. Especially, the transaction-level modelling using SystemC has been adopted for performance modelling and simulation [10], and OSCI is finalising the version 2.0 of its SystemC Transaction-level Modelling Standard [11].

Posadas et al. in [12] present a POSIX-based SystemC RTOS library for timing estimation at system level. The library is based on a general and systematic methodology that takes as input the original SystemC source code without any modification and provides the estimation parameters by simply including the library within a usual simulation. As a consequence, the same models of computation used during system design are preserved and all simulation conditions are maintained. The method exploits the advantages of dynamic analysis: easy management of unpredictable data-dependent conditions, and computational efficiency compared with other alternatives (ISS or RTL simulation, without the need for software (SW) generation and compilation and hardware (HW) synthesis).

Koski [13] is a UML-based automated SoC design methodology focusing on abstract modelling of application and architecture for early architecture exploration, methods to generate the models from the original design entry, system-level architecture exploration performing automatically allocation and mapping, tool chain supporting the defined methodology utilizing a graphical user interface, well-defined tool interfaces, a common intermediate format, and a simulation tool that combines abstract application and architecture models for cosimulation.

Ptolemy II [14] is a Java-based software framework developed as part of the Ptolemy project [15], supporting heterogeneous, concurrent modelling and design. Metropolis is a framework for platform-based design, which consists of an internal representation mechanism, the design methodology, and base tools for simulation and design imports [16].

MARTE [17] is a UML profile for model-driven development of real-time and embedded systems, which provides support for specification, design, and verification/validation. It is intended to replace the earlier UML schedulability, performance, and time (SPT) profile.

Our approach differs from the SPADE and Artemis as to the way the application is modelled and abstracted. The UML-based workload model mimics truly the control structures of the applications, but the leaf level load data is presented like traces. Also the execution platform is modelled rather at transaction than instruction level. The TAPES approach uses transaction-level SystemC simulation like ours, but describes the application functionality as traces that are stored in the architectural resources. The layering approach of the MESH is somewhat similar to ours, but the way of obtaining the timing information differs; in MESH it is obtained by profiling on a host, while the simulation gives the timing information in our case. The SpecC approach estimates timing information for the simulation, too. The Koski approach uses libraries and compilation/synthesis on field programmable logic arrays (FPGAs) to extract timing, although it allows setting timing requirements during application modelling. The UML part of our approach could have been based on the MARTE profile; however, the profile was not publicly available at the time of this work.

3. UML2-SYSTEMC-BASED PERFORMANCE MODELLING

3.1. Overview

The performance modelling and evaluation approach of VTT_ABSOLUT2.2% follows the Y-chart model as depicted in Figure 1. The basic principle of our performance evaluation approach is as follows [18]. The layered hierarchical workload models represent the computation and communication loads the applications cause on the platform when executed. The layered hierarchical platform models represent the computation and communication capacities the platform offers to the applications. The workload models are mapped onto the platform models and the resulting system model is simulated at transaction-level to obtain performance data.

The starting points for the performance modelling are the end-user requirements of the system. These are modelled as a service-oriented application model, which has a layered hierarchy. The top layer consists of system level services visible to the user that are composed of subservices and divided further to primitive services. The functional simulation of the model in Telelogic Tau UML2 tool provides sequence diagrams which are needed for verification and for building the workload model.

The purpose of workload modelling is to illustrate the load an application causes to an execution platform when executed. Workload models are nonfunctional in the sense that they do not perform the calculations or operations of the original application. Workload modelling enables performance evaluation already in the early phases of the design process, because the models do not require that the applications are finalized. Workload modelling also enhances simulation speed as the functionality is not simulated and models can typically be easily modified to quickly evaluate various use cases.

Platform modelling comprises the description of both hardware and platform software (middleware) components and interconnections that are needed for performance simulation. Like workload modelling, platform modelling considers hierarchical and repetitive structures to exploit topology and parallelism. The resulting models provide interfaces, through which the workload models use the resources and services provided by the platform [19].

The workload models are created with UML or SystemC, while the platform models are based on SystemC only. If the workload modelling is done with a UML tool, the models have to be transformed into SystemC. UML is a standard language used in software development and thus the possibility to use UML-based workload models enables reuse of existing UML application models. This ultimately reduces the effort required for workload modelling and makes the performance simulation approach more accessible in general. The use of UML is also beneficial because the models are visual and easier to understand for others besides the person who created them.

Figure 2 depicts the flow from UML2 application models to generated SystemC workload models [20]. The entire hierarchy of workload models—applications, processes, functions, and so forth—are collected in a class or package diagram, which presents the associations, dependences, and compositions of the workloads. Control inside the application, process and function workloads is described with state machine diagrams. Composite structure diagrams are used to connect the control implementation with the corresponding workload model. Section 6 presents examples of how the composite structure and state machine diagrams are used in our approach for implementing the workload models. All workload model layers, with the possible exception of the load primitive layer, are implemented in the UML model.

A skeleton model of the platform is manually created in the UML model. This facilitates mapping between the workload models with service requirements and the platform models with service provisions. The skeleton model describes the components and services available in the platform and thus enables the use of these services from the workloads. In the mapping phase, each workload entity is linked to a processor or other component, which is able to provide the services required by that entity. This is realized in the UML model using composite structure diagrams, for example.

Transformation to SystemC is triggered from the Telelogic Tau tool and it is based on the approach developed by Lund University, which is described in [21]. The generator produces SystemC code files, which include SystemC modules of classes and channels required for communication, and Makefiles for building the models. However, we build the SystemC workloads together with the platform model and thus do not use the generated Makefiles. The load primitive layer, if not modelled in the UML domain, is implemented in the SystemC workload refinement phase. This includes either writing read(), write(), and execute() service calls manually into the models or using a suitable automatic tool for the job.

The form of the UML workload models is quite flexible and only the following criteria have to be met:

(i) the platform model provides its services via functions and the SystemC workloads generated from the UML workload models must be able to use them. This can be achieved via signals in state machine diagrams, for example,

(ii) only diagrams supported by Lund University 's code generator can be used [21].

After mapping the workloads to the platform, the models can be combined for transaction-level performance simulation in SystemC. Based on the simulation results, we can analyse, for example, processor utilisation, bus or memory traffic, and execution time.

3.2. Workload model

Workload models are used for characterising the control flow and the loads of the data processing and communication of applications on the execution platform. Therefore the models can be created and simulated before the applications are finalized, enabling early performance evaluation. As opposed to most of the performance simulation approaches, the workload models do not contain timing information. It

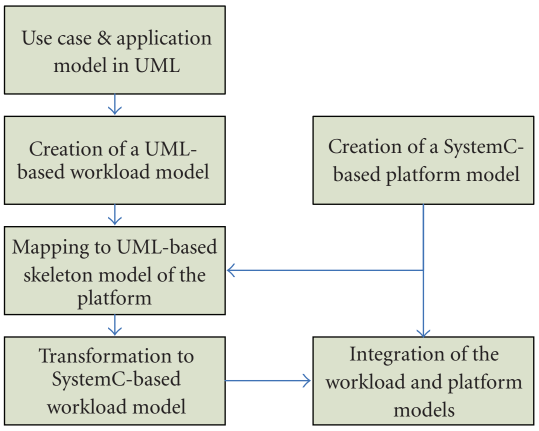

Use case & application model in UML
Creation of a UML-based workload model
Creation of a SystemC-based platform model
Mapping to UML-based skeleton model of the platform
Transformation to SystemC-based workload model
Integration of the workload and platform models

FIGURE 2: Transformation from UML2 application model to SystemC workload model.

is left to the platform model to find out how long it takes to process the workloads. This arrangement results in enhanced modelling and simulation speed. It is also easy to modify the models, which facilitates easier evaluation of various use cases with minor differences. For example, it is possible to parameterise the models so that the execution order of applications varies from one use case to another.

The workload models have a hierarchical structure, where top-level workload model W divides into application workloads A_i, $1 \leq i \leq n$ for different processing units of the physical architecture model (Figure 3):

$$W = \{C_a, A_1, A_2, \dots, A_n\}, \quad (1)$$

where C_a denotes the common control between the workloads, which takes care of the concurrent execution of loads mapped to different processors. n is the number of application workloads under the top-level workload.

Each application workload A_i is constructed of one or more processes P_i:

$$A_i = \{C_p, P_{1,} P_2, \dots, P_n\}, \quad (2)$$

where C_p corresponds to the control between the processes.

The structure of the main workload model and the application workloads is depicted in the UML diagram of Figure 4. The application and process control are shown as classes in the diagram; however, they may be implemented using, for example, standard C++ control structures in SystemC-based workload models.

The processes are comprised of function workloads i:

$$P_i = \{C_f, F_1, F_2, \dots, F_n\}, \quad (3)$$

where C_f is control and describes the relations of the functions, for example, branches and loops. The operating system models of the platform handle workload scheduling at the process level.

Function workload models are basically control flow graphs

$$F_i = (V, G), \quad (4)$$

where nodes $v_i \in V$ are basic blocks and arcs $g_i \in G$ are branches. Basic blocks are ordered sets of load primitives used for load characterization. Load primitives are abstract instructions *read* and *write* for modelling memory accesses and *execute* for modelling data processing.

Process and function workload models can also be statistical. In this case the model will describe the total number of different types of load primitives and the control is a statistical distribution for the primitives (Figure 5). This is beneficial in case the chosen load extraction method is not accurate enough so that functions and/or basic blocks could be modelled in detail. Less important, for example, background, workloads can also be modelled this way for reducing the modelling effort. Workload models using deterministic process models but statistical function models are more accurate than those using statistical process models. Models, which are deterministic down to basic block level, are of course the most accurate.

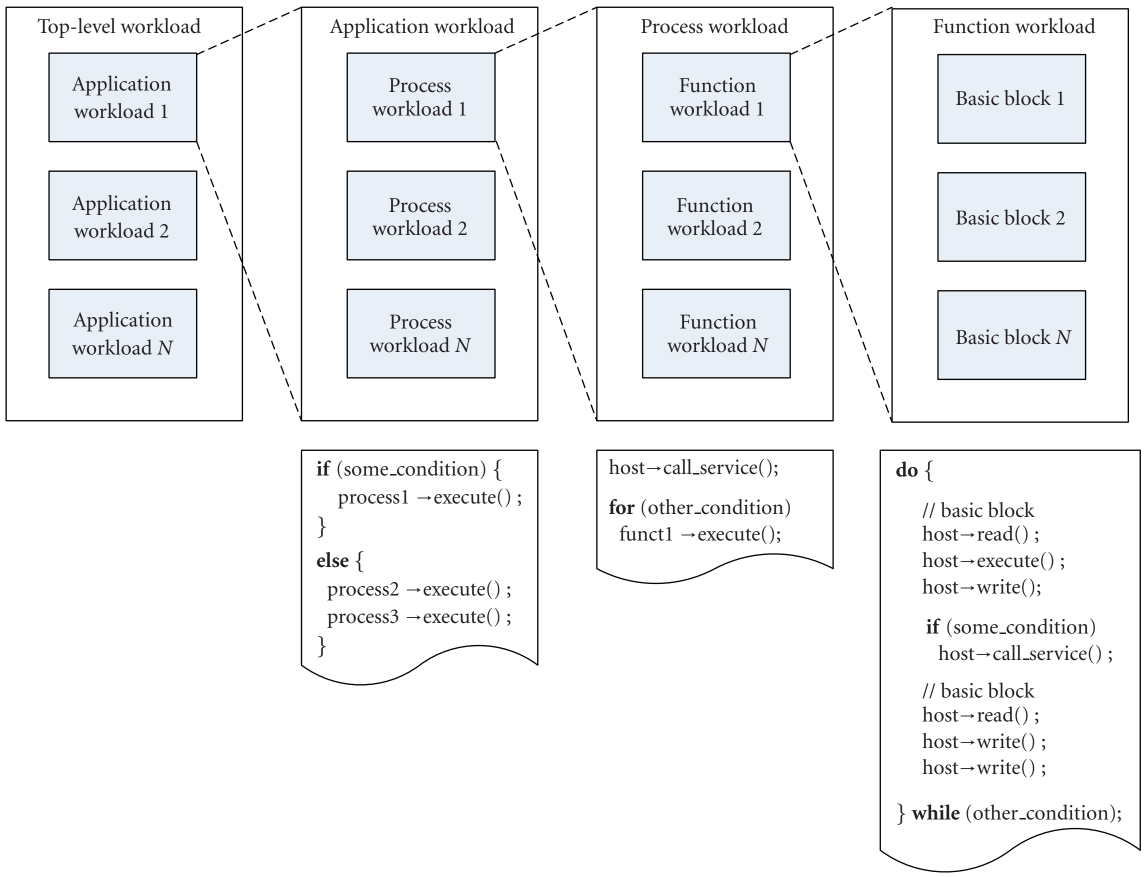
3.3. Execution platform model

The platform model is an abstracted hierarchical representation of the actual platform architecture. It contains cycle-approximate timing information along with structural and behavioural aspects. The platform model is composed of three layers: component layer, subsystem layer, and platform architecture layer (Figure 6). Each layer has its own services, which are abstraction views of the architecture models. They describe the platform behaviours and related attributes, for example, performance, but hide other details. Services in the subsystem and platform architecture layers can be invoked by workload models. High-level services are built on low-level services, and they can also use the services at the same level. Each service might have many different implementations. This makes the design space exploration process easier, because replacing components or platforms by others could be easily done as long as they implement the same services.

3.3.1. Component layer

This layer consists of processing (e.g., processors, DSPs, dedicated hardware, and reconfigurable logic), storage, and interconnection (e.g., bus and network structure) elements. An element must implement one or more types of component-layer services. For example, a network interface component should implement both master and slave services. In addition, some elements need to implement services that are not explicitly defined in component-layer services, for instance, a bus will support arbitration and a network will support routing.

The component-layer read, write, and execute services are the primitive services, based on which higher-level services are built. The processing elements in the component layer realize the low-level workload-platform interface, through which the load primitives are transferred from the workload side. The processing element models will then generate accesses to the interconnections and slaves as appropriate.



```
if (some_condition) {
   process1 →execute() ;
}
else {
  process2 →execute() ;
  process3 →execute() ;
}
```

```
host→call_service();

for (other_condition)
  funct1 →execute();
```

```
do {

   // basic block
   host→read() ;
   host→execute() ;
   host→write();

    if (some_condition)
      host→call_service() ;

   // basic block
   host→read() ;
   host→write() ;
   host→write() ;

} while (other_condition);
```

Figure 3: Workload models have a hierarchical structure.

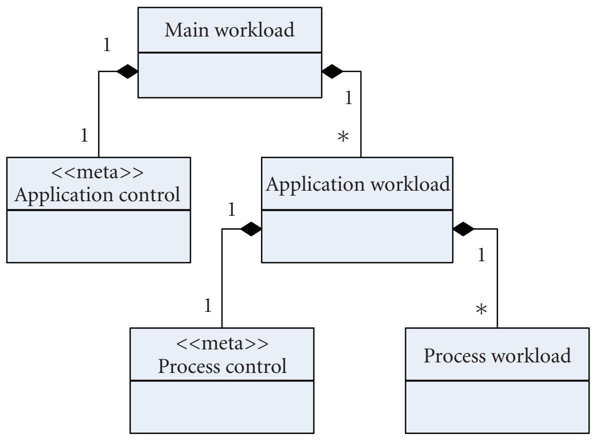


Figure 4: The top-level and application workloads consist of one or more lower level items and control.

All the component models contain cycle-approximate or cycle-accurate timing information. Specifically, the data path of processing units is not modelled in detail; instead the processor models have a cycles-per-instruction (CPI) value, which is used in estimating the execution time of the workloads. For example, the execution time for data processing instructions is the number of instructions to execute times CPI (Figure 7). Furthermore, caches and SDRAM page misses, for example, are modelled statistically since the workload models typically do not include accurate address information.

3.3.2. *Subsystem layer*

The subsystem layer is built on top of the component layer and describes the components of the system and how they are connected. The services used at this layer could include, for example, video preprocessing, decoding, and postprocessing for a video acceleration subsystem.

The model can be presented as a composition of structure diagrams that instantiates the elements taken from the library. The load of the application is executed on processing elements. The communication network connects the processing elements with each other. The processing elements are connected to the communication network via interfaces.

3.3.3. *Platform architecture layer*

The platform architecture layer is built on top of the subsystem layer by incorporating platform software and serves as the portals that link the workload models and the platforms in the mapping process. Platform-layer services

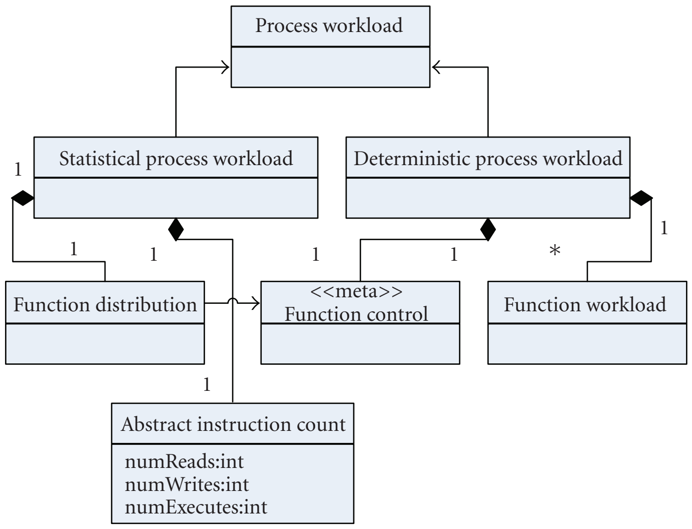


FIGURE 5: The process workloads can be either statistical or deterministic.

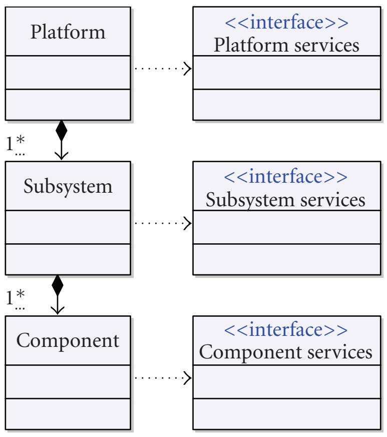


FIGURE 6: The execution platform model consists of platform, subsystem, and component layers.

consist of service declaration and instantiation information. The service declaration describes the functionalities that the platform can provide. Because a platform can provide the same service with quite different manners, the instantiation information describes how a service is instantiated in a platform.

The platform-layer services are divided into several categories with each category matching one application domain, for example, video processing, audio processing, and encryption/decryption. The OS system call services are in an individual domain, and as mentioned earlier they can also be invoked by other services at the same level. A number of platform-layer services are defined for each domain, and more could be added if necessary.

Application workloads typically call platform or subsystem level services, process workloads call subsystem services, and function workloads call component-level services. Ideally, all services required by the application are provided by the execution platform and there is a 1 : 1 mapping between the requirements and provisions. However, often this is not the case and the workloads need to use several lower-level services in combination to produce the desired effect.

TABLE 1: The low-level interface consists of functions intended for transferring load primitives between workload and platform models.

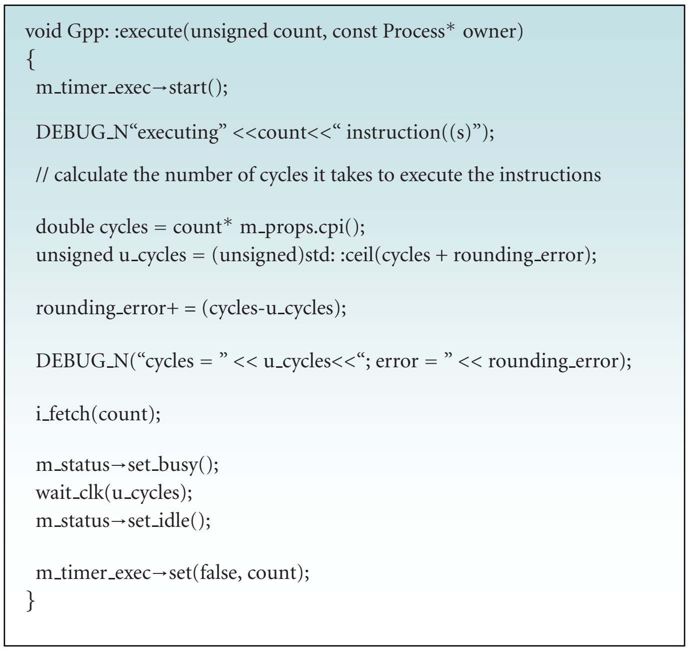
Interface function	Description
Read(A, W, B)	Read W words of B bits from address A
Write(A, W, B)	Write W words of B bits to address A
Execute(N)	Simulate N data processing instructions

3.4. Interface between workload and platform models

The platform model provides two interfaces for utilising its resources from the workload models. The low-level interface is intended for transferring load primitives and is depicted in Table 1. The functions of the low-level interface are blocking—in other words a load primitive level workload model is not able to issue further primitives before the previous primitives have been executed.

The high-level interface enables workload models to request services from the platform model (Table 2). These functions can be called from workload models between the function and application layers. The use_service() call is used to request the given service and it is nonblocking so that the workload model can continue while the service is being processed. Use_service() returns a unique service identifier, which can be given as a parameter to the blocking wait_service() call to wait until the requested service has been completed, if necessary.

In simple cases, the execution of workloads can be scheduled manually by hard-coding it to the models. However, typically the platform model includes one or more



```
void Gpp: :execute(unsigned count, const Process* owner)
{
  m_timer_exec→start();

  DEBUG_N“executing” <<count<<“ instruction((s)”);

  // calculate the number of cycles it takes to execute the instructions

  double cycles = count* m_props.cpi();
  unsigned u_cycles = (unsigned)std: :ceil(cycles + rounding_error);

  rounding_error+ = (cycles-u_cycles);

  DEBUG_N(“cycles = ” << u_cycles<<“; error = ” << rounding_error);

  i_fetch(count);

  m_status→set_busy();
  wait_clk(u_cycles);
  m_status→set_idle();

  m_timer_exec→set(false, count);
}
```

Figure 7: Code extract showing how processor models calculate the execution time for data processing instructions.

Table 2: The services of the platform model are exploited via the high-level interface.

Interface function	Return value	Description
Use_service(*name*, *attr*)	Service identifier *id*	Request service *name* using *attr* as parameters
Wait_service(*id*)	N/A	Wait until the completion of service *id*

operating system (OS) models, which control access to the processing unit models of the platform by scheduling the execution of process workload models (Figure 8). The OS model provides both low-level and high-level interfaces to the workloads and relays interface function calls to the processor or other models which realize those interfaces. The OS model will allow only these process workloads which have been scheduled for execution to call the interface functions. Rescheduling of process workloads is performed periodically according to the scheduling policy implemented in the model.

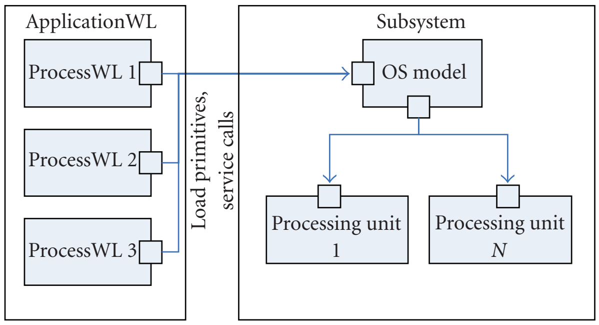


Figure 8: During simulation process workloads send load primitives and service calls to the platform model.

3.5. *Transformation from UML to SystemC*

Though it is possible to define a set of mapping rules from a pure UML model directly into SystemC code, it would give the engineer little influence on the mapping and most likely a less satisfactory result. Instead, we divide the mapping into three steps. All models are available and editable. This makes it possible for the engineer to have full control over the relevant details for the system under development and have the tool to manage all remaining details.

Step 1, vertical refinement transformation: In this step an initial UML description is refined to a UML description, which follows a UML profile for SystemC. This step will partly be carried out manually. To minimise the design effort, it is not required to tag the whole model. This means that a set of default values for the SystemC specific attributes of the UML profile have been defined. The default values provide a satisfactory mapping in the following transformation steps.

Step 2, vertical refinement transformation: in this step the model is transformed into a new UML description that only includes UML constructs with direct representations in SystemC, that is, classes, attributes, inheritance, and so on. Other constructs such as state machines are translated to the target language. During this step each state machine is transformed to a class with methods that implement the behaviour of the states and transitions. In the first version of the tool, the resulting model is an untimed functional model. The mapping rules for UML to SystemC are beyond the scope of this paper and interested readers are referred to [21, 22].

In addition to removing UML only concepts, all relations in the model are made explicit. When a class is made active in UML it implies that the class will have its own thread of execution. In SystemC this is realized using SC_THREAD or SC_METHOD which implies that the class is an instance of the SystemC class sc_module. For example, a generalisation relation to the SystemC class sc_module is added from all active UML classes.

Step 3, horizontal transformation: in this step the UML model resulting from step 2 is transformed into a corresponding SystemC code. This transformation is a one–to-one correspondence between the UML model and the resulting SystemC code. This step is implemented using the existing C++ code generator from Telelogic and thus reuses its support for scope rules, header-file inclusion and make file generation without any modifications. If the generated code is to be read by humans, it is desirable to use the common SystemC macros when applicable. This requires a slight customisation of the syntax of the generated code. This is done using an agent, a mechanism which makes it possible for third-party executables to interact with the C++ Code generator in Telelogic Tau G2. The agent generates SystemC like module declarations, instead of a C++ class declarations, SC_MODULE(MyModule){· · ·} instead of class MyModule:public sc_module{· · ·}.

4. LOAD EXTRACTION

The workload models capture the control behaviour of the applications in the hierarchically layered structures. On the other hand, they abstract the details of data processing and communication as loads. To obtain the load information, three different techniques are currently used: analytical, measurement-based and source code-based. These can be used separately or in combination depending on what kind of descriptions of application algorithms are available.

The selection of which extraction method to use depends on (i) what kind of information is available of the application(s), (ii) how accurate the resulting models need be, and (iii) how much effort one is willing to use. In general, you will want to use as accurate method as possible for best results. On the other hand, if you have only limited information available of the applications, you may be forced to use the analytical method. If you are modelling small background load with a minor impact on the overall performance, it makes sense to use one of the faster extraction methods instead of the source code-based approach.

4.1. Analytical load modelling

Analytical load modelling method is currently the simplest way to create workload models but also probably the least accurate. As the name suggests it is based on analysis of the number of operations from an algorithm description or other suitable source.

Analytical modelling [23] consists of three phases. First, the number of data processing instructions and amount of memory traffic are analysed from the algorithm description. In principle, after this point we could write a simple three line workload model that would consist of one read(), one write(), and one execute() call. However, the ordering and block size of the operations usually affects the performance notably. Therefore in the second phase the total operation numbers are divided into smaller blocks. In the best case, this can also be done based on the algorithm description. In the worst case we can only create a number of uniformly distributed blocks of read(); execute(); write().

Let us consider the case of an MPEG4 video colour conversion algorithm as an example. Once the video resolution and the amount of bits per pixel for input and output are known it is possible to calculate the number of memory reads and writes per frame. For VGA resolution, YUV422 input and YUV420 output we get at least $640 \times 480 \times 16$ bits of data reads and $640 \times 480 \times 12$ bits of writes. It is also possible to estimate the efficiency of the implementation with a simple factor $n \leq 1$. The word length W (in bits) and burst length N (in words) of the target processor should also be taken into account so that the number of reads R and writes W for F frames in this case are

$$\begin{aligned} R &= \left\lceil \frac{640\cdot 480\cdot 16\cdot F}{nWN} \right\rceil \\ W &= \left\lceil \frac{640\cdot 480\cdot 12\cdot F}{nWN} \right\rceil, \end{aligned} \tag{5}$$

respectively. Since the operations performed for each pixel are known it is straightforward to estimate also the number of data processing instructions E. In this case the algorithm calculates an average of each two chrominance data points, so we have $640 \times 480/2$ calculations per frame. One addition and one division or shift by two, or two instructions in total, are required per each average, resulting to the following formula:

$$E = \left\lceil \frac{640\cdot 480\cdot F}{n} \right\rceil. \tag{6}$$

In this case, the algorithm must be executed so that data is available for at least two pixels at any point. Thus we can divide the model in the second phase so that firstly the data for two pixels is read, then the two pixels are processed and, finally, the new data is written out. This sequence is repeated until all pixels of all frames are processed. In pseudocode, this results to workload model shown in Algorithm 1.

However, it should be noted that this is not the only feasible way to divide the algorithm into smaller blocks.

The analytical method typically produces the most compact workload models with the least effort. However, the quality of the workload models depends a great deal on the use case (algorithm) being modelled. So far, the analytical approach for load extraction has been partially used in the modelling of Quake 2 3D game [23] and MPEG4 video encoding and decoding on the OMAP2 platform.

```
for(loop through all frames) {
    for(loop until all pixels of a
        single frame are processed) {
      read(two_pixels);
      execute(two_pixels);
      write(two_pixels);
    }
}
```

Algorithm 1

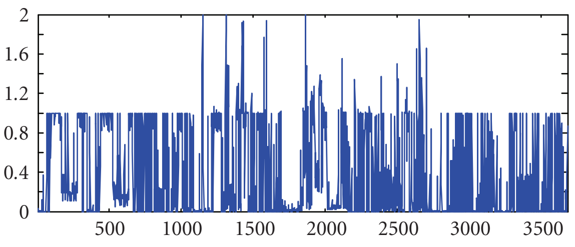


Figure 9: An example of a concatenated and merged utilisation curve.

4.2. *Measurement-based load generation*

The second method for creating the load models is to extract data from partial measurements or traces made of the use case [24]. This method can easily be automated, which allows rapid modelling of complex use cases with minimal manual work. It is also useful for performance analysis if the entire use case cannot be executed and monitored for some reason.

A method has been developed for generation of complex workload models from processor utilisation data. The complex use case means in this context a workload that cannot be measured with available system and that consists of several programs that can be measured or estimated alone. The first step of our approach is to measure processor utilisation data of these individual programs. The second step is to merge and concatenate the data according to the state sequence model of a complex use case. An example of this merging is shown in Figure 9. The merged utilisation curve is likely to have data points where the utilisation is more than 100%. This is not a problem because the value will only be used as a basis for estimating the number of load primitives for the workload model. The third step is to generate the workload models of the primitive samples of monitoring tool.

The workload models consist of read, write, and execute operations, whose amounts can be estimated for each sample point of the input utilisation data in the following way: let $X(n)$ be the combined trace data, which was composed by merging the measurement data, and N the number of samples in the combined trace. Furthermore, let t_{sp} be the sampling interval, C the number of cycles per second (i.e., processor clock frequency), c_{sp} the number of cycles per sample point, $C_{op}^{R,W,E}$ the number of cycles per read, write, or execute operation, $\rho^{R,W,E}$ the percentages of read, write, and execute operations, and $Y^{R,W,E}(n)$ the amount of read, write, or execute operations per sample point.

Now, the time, t^R, spent for read operations during one sample point can be calculated as

$$t^R = t_{sp}\rho^R. \tag{7}$$

During this time, the processor clock counts $t^R C$ cycles. This amount of cycles is sufficient to perform exactly $(t^R C)/c_{op}^R$ read operations. When the combined trace data gives a utilisation level of X for the given sample point, n, we can assume that the number of read operations, $Y^R(n)$, for that sample point is $(t^R C)/c_{op}^R X$, which can be rearranged as

$$Y^R(n) = C\frac{\rho^R}{c_{op}^R}t_{sp}X. \tag{8}$$

In general terms, this can be expressed as

$$\overline{Y}(n) = \begin{bmatrix} Y^R(n) \\ Y^W(n) \\ Y^E(n) \end{bmatrix} = C\left[\overline{B}t_{sp}X(n)\right], \quad n = 1,2,3,\dots,N$$
$$\overline{B} = \begin{bmatrix} \dfrac{\rho^R}{c_{op}^R} \\ \dfrac{\rho^W}{c_{op}^W} \\ \dfrac{\rho^E}{c_{op}^E} \end{bmatrix}. \tag{9}$$

In a shorter form this can also be expressed as

$$Y^{R,W,E}(n) = \left\lfloor \frac{c_{sp}}{c_{op}^{R,W,E}}\rho^{R,W,E}X(n)\right\rfloor \mid c_{sp} = Ct_{sp}, \quad n = 1,2,3,\dots,N. \tag{10}$$

The execution of these operations is then spread uniformly across the given sample point and, finally, workload models containing the information for each sample point are composed.

The measurement-based approach has been successfully applied to complex use cases consisting of GPRS connection, Bluetooth download, SMS/MMS messaging, and video capturing among others [24]. For comparison purposes the same applications were executed in a mobile phone prototype based on the same platform. The performance of the platform was monitored and the average difference between measured and simulated processor utilisation was about 19%.

4.3. *Source code-based load generation*

Source code-based workload generation requires estimating the amount of elementary operations by examining the function source codes line by line. This approach results in quite detailed and accurate workload models but requires a significantly high amount of manual work.

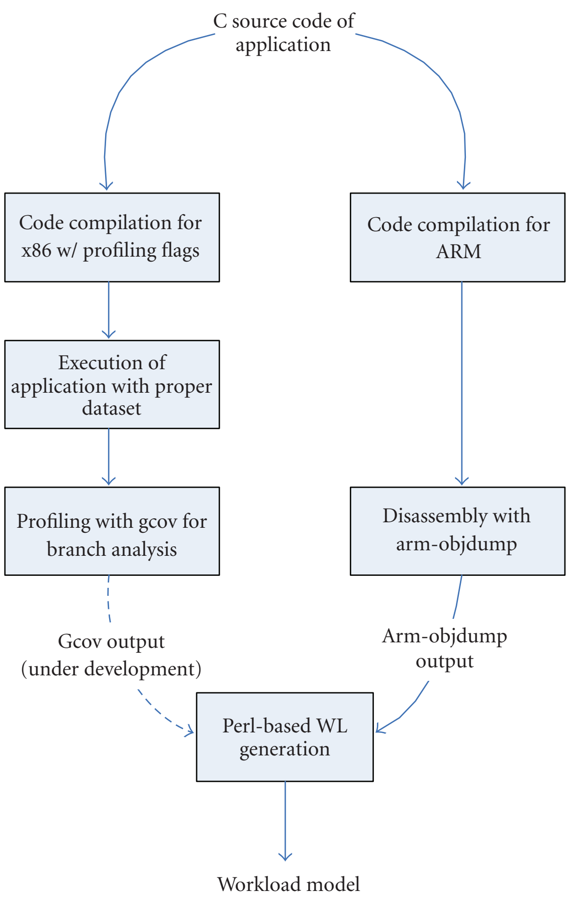

C source code of application
Code compilation for x86 w/ profiling flags
Code compilation for ARM
Execution of application with proper dataset
Profiling with gcov for branch analysis
Disassembly with arm-objdump
Gcov output (under development)
Arm-objdump output
Perl-based WL generation
Workload model

FIGURE 10: Semiautomatic method for C source code-based load information extraction.

A semiautomatic tool has been developed for extracting the load information from C language source code (Figure 10). The use of this tool requires that the source code is available and mature enough that it can be compiled and executed.

Firstly, the source code is compiled for the ARM instruction set with a GNU GCC cross-compiler. The resulting binary is disassembled with arm-objdump, which reveals the instructions the program would execute on an ARM processor. This arm-objdump output is then analysed by a perl script, which produces abstract instructions for the load primitive workload layer from the dump. Each load and store instruction in the dump will result to a corresponding read or write primitive and all the other instructions are replaced with execute primitives. It should be noted that the ARM compiler is used because it is easier to differentiate between memory access and other instructions with the ARM instruction set than with, for example, x86 instruction set. The extraction method, however, is not limited to ARM.

Secondly, the application is compiled for the host architecture—typically an x86-based PC-using the native GCC tool chain. Profiling flags are enabled during the build process and the resulting binary is executed with a suitable dataset. The profiling information reveals branch probabilities and the number of times loops have been executed. The function layer of the workload models can be constructed based on this information, if necessary. For example, consider a situation where the profile reveals that one code branch is executed with a 60% probability. We can add a test based on the random number generator that results in the corresponding load primitives being executed just as often. However, combining the profiling information and the load primitive information from the ARM disassembly is done manually at the moment, though an automatic tool is under development.

The source code-based approach has been used in the modelling of a 3D game [23], MP3 player, [24] and MPEG4 video encoder [19]. It has also been used partially in a case consisting of internet browsing over virtual network computing (VNC) session on a handheld device. Apart from the 3D game, these case examples have also been verified by executing the same applications in a real platform. The average difference between simulations and measurements across all those cases was about 10%.

5. PERFORMANCE SIMULATION

The combined system model is built in a GNU/Linux environment for performance simulation, using GNU compiler collection (GCC), CMake cross-platform build system, and OSCI SystemC library. Currently manually created CMakeFiles are used for configuring the build. The simulator is a command line program, which is executed in a terminal window. During the simulation it prints progress information to standard output and after the simulation is complete it displays the collected performance results.

A C++- and XML-based automatic configuration tool called VTT_COGNAC is being developed to simplify the process. With this tool also the platform model is sketched in the Tau UML2 tool from a component library (like the skeleton model of the platform currently) and the parameters of the components are set up in UML. The tool reads the XML output from Tau and generates the system model by combining the SystemC workload models generated from UML and the SystemC component models in the component library. The system simulator VTT_BEER then executes the simulation.

During the simulation of the system model the workloads send load primitives and service calls to the platform model (Figure 8). The platform model processes the primitives and service calls, advancing the simulation time while doing so. The simulation run will continue until the top-level workload model stops it when the use case has been completed.

The platform model is instrumented with counters, timers, and probes, which record the status of the components during the simulation. These performance probes are manually inserted in the component models where appropriate and are flexible so that they can be used to gather information about platform performance as needed.

Typically,

(i) status probes collect information about utilisation of components and scheduling of processes performed by the operating system models,

(ii) counters are used to calculate the number of load primitives, service calls, requests, and responses performed by the components,

(iii) timers keep track of the task switch times of the OS models and processing times of services.

Once the simulation is complete, the performance probes output the collected performance data to the standard output. A C++-based tool VTT_VODKA has been developed for viewing the data, for example, processor utilisation curves, graphically. The data can be analysed and feedback given to application or platform design. For example, if the utilisation of components is low, lowering the clock frequency can be proposed to platform designers for decreasing power consumption.

In a mobile video player case [20], the simulation speed was one tenth of real time. In other words, simulating the system for one second took about ten seconds in a Linux PC with a dual-core Intel Xeon processor, although the simulation was using only one of the cores. This is faster than cycle-accurate instruction set simulation, even if the models used in the case example were not optimised. It is also fast enough for performing early-phase performance evaluation and for simulating multiple, alternative use cases for architecture exploration.

6. MOBILE VIDEO PLAYER CASE STUDY

The performance modelling approach has been applied to the mobile video player case study. In this use case, a mobile terminal user wants to view a movie on the device. He selects a movie from a list of movies available on the mobile terminal. The execution platform of the mobile terminal is assumed to provide services for storing of movie files, for playing and displaying the selected movie, and for running the application. The use case does not focus on the human-terminal interface, but on the modelling of functionalities, services, communication, and so on that execute on the platform in fulfilling the user request.

Figure 11 displays an overall view of the workload models of the case study. PlayerApplication is the model of the software application that provides a graphical interface to the user. It responds to events coming from the user and sends messages accordingly to the VideoPlayer, ObjectServer, and DisplayServer. Since our method does not simulate application functionality, the user events are either collected to an input file or hard-coded to the model. The ObjectServer manages stored video files, provides the list of video clips available for viewing to the PlayerApplication, and streams video data to the VideoPlayer when viewing files. The VideoPlayer handles video preprocessing, decoding, and postprocessing. Finally, the DisplayServer gets the uncompressed video stream from the VideoPlayer and displays it on a screen as requested by the PlayerApplication.

Figure 12 presents the application workload model of the PlayerApplication as a composite structure diagram. It consists of process control and several process workloads, namely ObjectServer, VideoPlayer, and DisplayServer processes. The VideoPlayer process workload is further decomposed in Figure 13. In a similar manner it is built of function control and several function workloads.

Figure 14 shows the video decoding/displaying control sequence inside the VideoPlayer process workload as a state machine diagram. Firstly, a handle is obtained to the video file which has been selected, then segments of the file are requested in a loop until the last segment has been received. Each segment is decoded once it has been received.

The UML-based workload models were transformed to SystemC using the code generator developed at Lund University [21]. The SystemC-based version of the transition from wf_decode_cnf to wf_postprocess_cnf state in Figure 14 is presented in Figure 15 as an example.

The platform in the mobile video player case consists of four subsystems depicted as a block diagram in Figure 16:

(i) general purpose (GP) subsystem, which is used for executing an operating system and generic applications and services, for example,

(ii) image (IM) subsystem, which is intended for hardware-accelerated image processing and video playback and recording,

(iii) storage (ST) subsystem, which contains a repository for video clips and services for loading and storing them,

(iv) display (DP) subsystem, which takes care of displaying framebuffer data on a screen, including the viewed video files.

The subsystems are interconnected by a network using a ring topology with best-effort (BE) and guaranteed throughput (GT) routing approaches. GT is used for transferring streaming data and BE is used for transferring control messages between subsystems.

The general purpose subsystem has two ARM11 general purpose processors for executing the operating system and applications. There is also a subsystem-local SDRAM memory controller and memory to be used by the two processors. For communication with the other subsystems, the GP subsystem—like all the other subsystems—has a network interface.

Image subsystem is built around the video accelerator, which provides hardware accelerated compression and decompression, preprocessing and postprocessing services. The services provided by the subsystem are controlled by a simple ARM7 general purpose processor. There is also some SRAM memory for the ARM, a video accelerator and a DMA controller for offloading large data transfers between subsystems.

Storage and display subsystems are mostly similar to the image subsystem and they contain a simple ARM, a DMA controller and a network interface. However, instead of the video accelerator the storage subsystem has local memory for

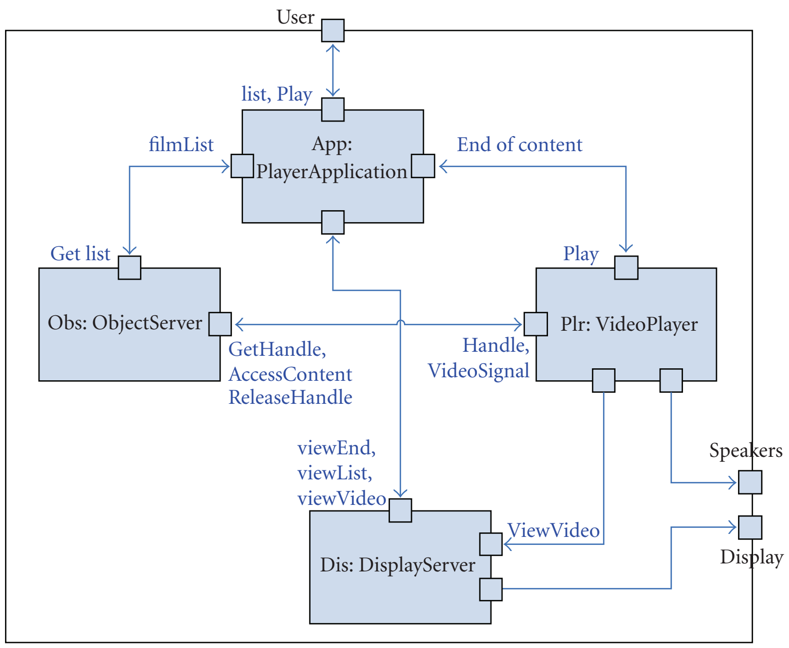


FIGURE 11: Interaction of the mobile video player workload models.

storage and metadata. The display subsystem has a graphics accelerator, local SRAM for graphics, and a display interface for the screen.

The platform model was created using SystemC and the communication interfaces were based on OCP TL3 on the platform layer. OCP TL2 was used on the subsystem and component layers. After the workload and platform modelling phases the workload models were mapped to the skeleton model of the platform (Figure 17). The flow of operation after the mapping is the following:

(i) the PlayerApplication, executed in one of the ARM11 processors of the GP subsystem, is launched and requests a list of movie files from the ObjectServer in the ST subsystem,

(ii) one of the movie files is selected, after which the PlayerApplication triggers the VideoPlayer in the IM subsystem,

(iii) the VideoPlayer requests the movie file from the ObjectServer, which initiates streaming of the file,

(iv) the VideoPlayer decodes the compressed stream and transfers video frames to the DisplayServer in the DP subsystem,

(v) the DisplayServer displays incoming video frames on the screen.

The video player case was simulated in the SystemC domain. The platform model was instrumented in order to collect information of the platform's performance during the simulation run. For example, the processor models record the time spent in idle, data processing, and memory accessing states for the calculation of processor utilisation. Subsystem models measure the average, minimum and maximum processing time for each service.

TABLE 3: Utilisation of the display subsystem components in the mobile video player use case.

Component	Idle	Busy
ARM7 microcontroller	98%	2%
SRAM controller	83%	17%
DMA controller	65%	35%
Display controller	74%	26%
Bus	66%	34%
Network interface	66%	34%

TABLE 4: Examples of average, minimum, and maximum processing times of services.

Service	Average	Min	Max
DMA (ST)	450 μs	1 μs	480 μs
Decoding (IM)	16 μs	14 μs	16 μs
DMA (DP)	1.4 ms	1 ms	1.6 ms

Table 3 presents the utilisation of each of the components in the display subsystem, which was obtained from the simulation. None of these components has been at the limit of their capacity: the DMA controller has been the most burdened of all and its average load has been only 35%. Furthermore, the load of the simple ARM7 microcontroller was under 3%. It is clearly possible to execute more demanding applications on this platform—at least from the display subsystem point of view. Another alternative is to reduce the clock frequency of these components to decrease the power consumption of the device.

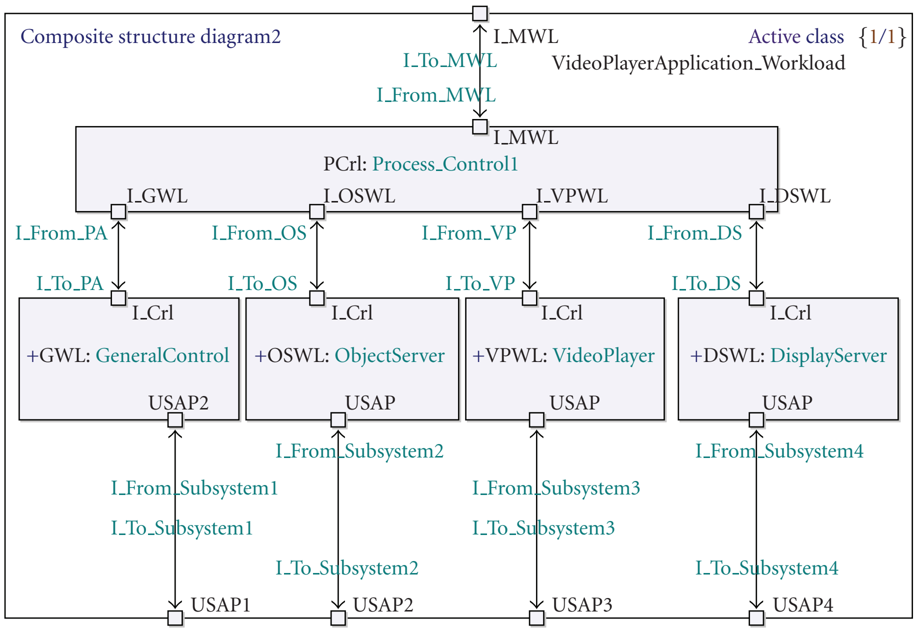


FIGURE 12: PlayerApplication application workload consists of process control and ObjectServer, VideoPlayer, and DisplayServer process workloads.

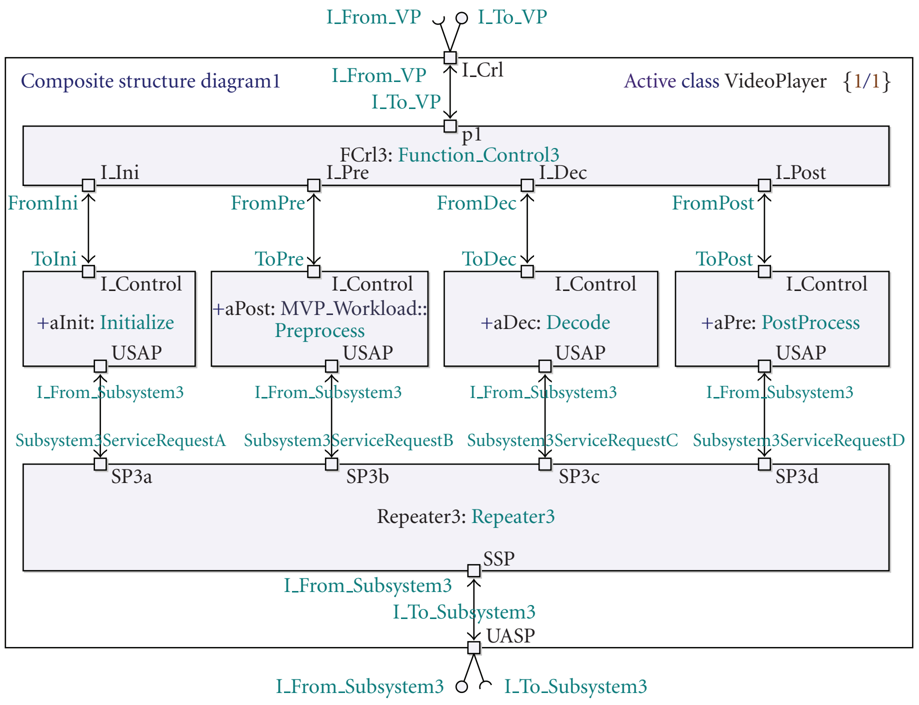


FIGURE 13: VideoPlayer process workload consists of function control and several functions workloads.

Another view on component utilisation is given in Figure 18, which depicts how the utilisation changes during the simulation with respect to time. The display subsystem is constantly updating the screen while the device is powered and those updates cause the lower, about 30% and 70% peaks to the utilisation curves of bus and memory components. In this simulation the video player application has been running only for a short period of time, inflicting the higher peaks.

Table 5 visualises the data reads and writes initiated or serviced by each component in the same subsystem. For example, 1.0 million read requests were processed by the bus resulting to about 4.2 million transferred 32-bit words.

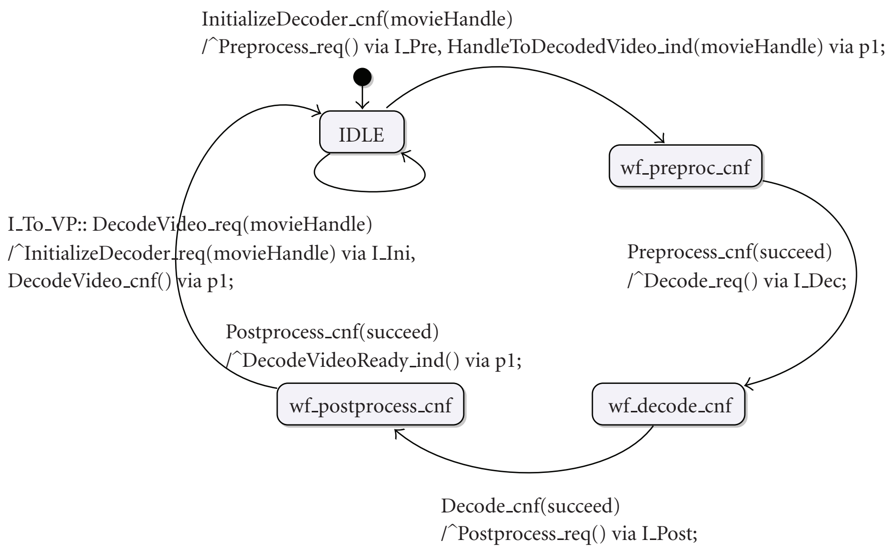

InitializeDecoder_cnf(movieHandle)
/^Preprocess_req() via I_Pre, HandleToDecodedVideo_ind(movieHandle) via p1;
IDLE
wf_preproc_cnf
I_To_VP:: DecodeVideo_req(movieHandle)
/^InitializeDecoder_req(movieHandle) via I_Ini,
DecodeVideo_cnf() via p1;
Preprocess_cnf(succeed)
/^Decode_req() via I_Dec;
Postprocess_cnf(succeed)
/^DecodeVideoReady_ind() via p1;
wf_postprocess_cnf
wf_decode_cnf
Decode_cnf(succeed)
/^Postprocess_req() via I_Post;

Figure 14: State machine example from the mobile video player UML model.

```
bool
MVP_Workload: :Function_Control3: :__11_trans_2
(UML_signal *pMsg)
{
  if (typeid(*pMsg) ==
      typeid(MVP_workload:: FromDec:: Decode_cnf_signal))
  {
    FromDec: :Decode_cnf_signal *pSig =
         dynamic_cast<MVP_workload:: FromDec::Decode_cnf_signal* > (pMsg);
    succeed_sm_11 = pSig → par0;
    {
      I_Post_port → Postprocess_req();
    }
    delete nextState_sm_11;
    nextState_sm_11 = new _sm_11_wf_postprocess_cnf(this);
    nextState_sm_11 → enter_action();
    return true;
  }
    return false;
}
```

Figure 15: An extract from the SystemC workload generated from the state machine diagram of Figure 14.

Table 5: Data traffic initiated or serviced by the display subsystem components.

Component	Reads	Read words	Writes	Written words
ARM7 microcontroller	1400	1400	1250	1250
SRAM controller	312 k	1245 k	635 k	2536 k
DMA controller	634 k	2534 k	634 k	2534 k
Display controller	311 k	1244 k	0	0
Bus	946 k	3780 k	635 k	2536 k
Network interface	634 k	2535 k	0	0

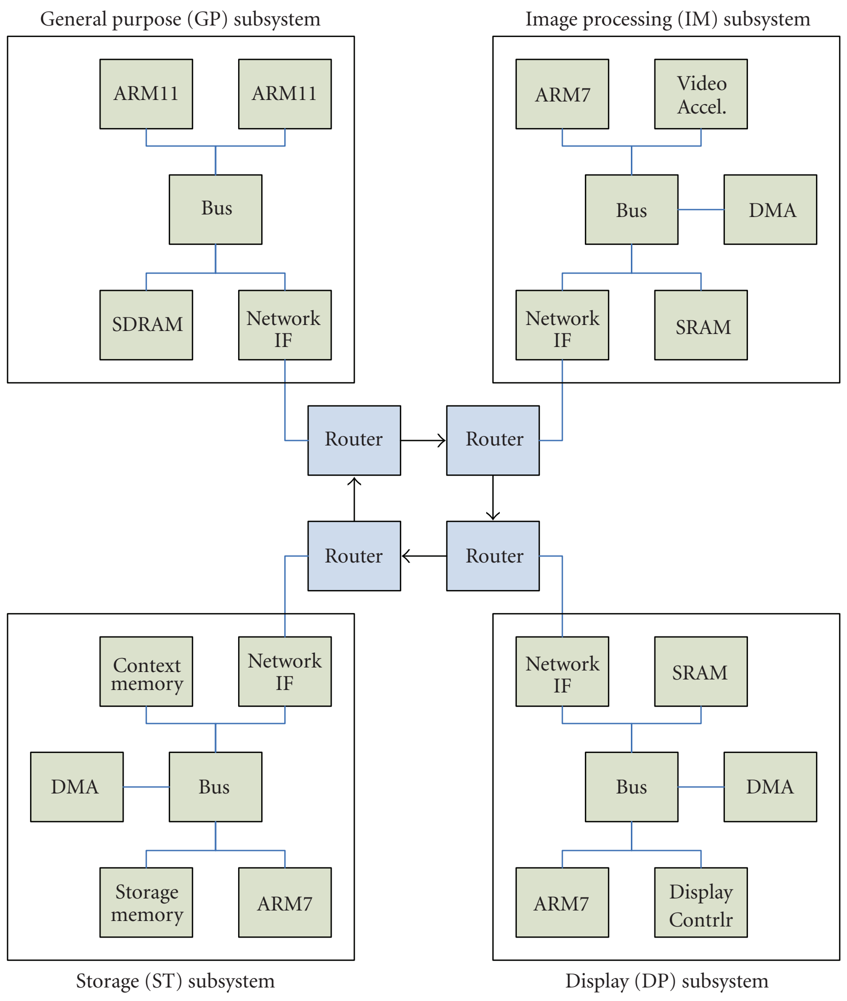


FIGURE 16: The execution platform of the mobile video player case consists of four subsystems.

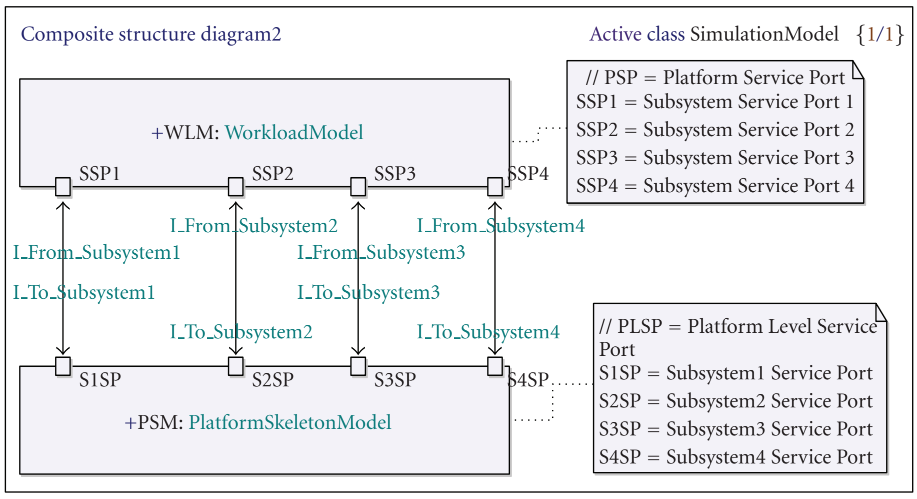

Composite structure diagram2
Active class SimulationModel {1/1}
+WLM: WorkloadModel
SSP1
SSP2
SSP3
SSP4
// PSP = Platform Service Port
SSP1 = Subsystem Service Port 1
SSP2 = Subsystem Service Port 2
SSP3 = Subsystem Service Port 3
SSP4 = Subsystem Service Port 4
I_From_Subsystem1
I_From_Subsystem2
I_From_Subsystem3
I_From_Subsystem4
I_To_Subsystem1
I_To_Subsystem2
I_To_Subsystem3
I_To_Subsystem4
S1SP
S2SP
S3SP
S4SP
+PSM: PlatformSkeletonModel
// PLSP = Platform Level Service Port
S1SP = Subsystem1 Service Port
S2SP = Subsystem2 Service Port
S3SP = Subsystem3 Service Port
S4SP = Subsystem4 Service Port

FIGURE 17: Mapping of the workload models to the platform skeleton model.

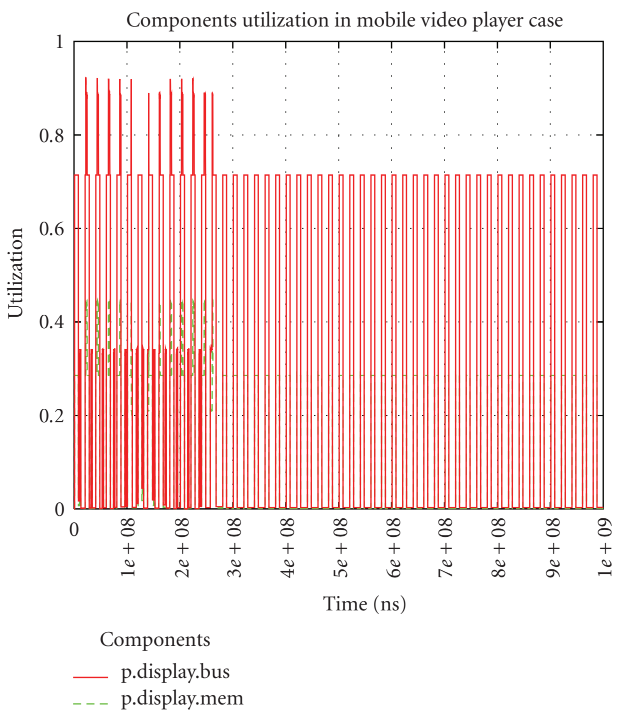


Figure 18: The red (upper) and green (lower) curves display the utilisation of bus and memory components in the display subsystem, respectively.

The same information as displayed in Tables 3 and 4 and Figure 18 was also collected from all the other subsystems, though they are not displayed here. Finally, Table 4 contains the processing times of, for instance, DMA transfer services from the subsystem layer.

The simulation results have not been validated with, for example, measurements since the execution platform is an invented platform intended to portray a future architecture. As such, there are no cycle-accurate simulators for the platform which could be used to obtain reference performance data. However, we have modelled MPEG4 video processing and the OMAP platform earlier and compared those simulation results to measurements from a real application in a real architecture [19]. In that case the average difference between simulations and measurements was about 12%. Furthermore, the accuracy of the simulation approach has been validated with other case examples in [23, 24].

7. CONCLUSIONS

A layered UML application/workload and SystemC platform modelling approach for performance modelling and evaluation was described. It allows application and platform to be modelled at several levels of abstraction to enable early performance evaluation of the resulting system. The approach applies a Y-chart-like specify-explore-refine paradigm. Applications are modelled in either UML or SystemC domain as workloads consisting of load primitives. Platform models are cycle-approximate transaction-level SystemC models. Mapping between UML application models and the SystemC platform models is based on automatic generation of simulation models for system-level performance evaluation.

The workload models reflect accurately the control structures of the applications. The layered hierarchical structure covers application, process, function, and basic block layers. Computation and communication are abstracted as loads that can be extracted using either analytical, measurement-based or source code-based methods, and support tools.

The platform model is an abstracted hierarchical representation of the actual platform architecture. It contains cycle-approximate timing information along with structural and behavioural aspects. The platform model is composed of three layers: component layer, subsystem layer, and platform architecture layer. Each layer has its own services, which are abstraction views of the architecture models. Services in the subsystem and platform architecture layers are invoked by workload models.

The tool support is based on a commercial UML2 tool, Telelogic Tau G2, and an open source SystemC simulation tool of OSCI. The overall performance modelling and evaluation method with support tools is called VTT_ABSOLUT2.2%. The source code-compilation method for load extraction is supported by VTT_ABSINTH. The execution platform model is configured from a library of performance models using VTT_COGNAC. The simulator is based on the open source OSCI SystemC simulator extended with configurable instrumentation and is called VTT_BEER. The simulation results can be selected for analysis and viewed using VTT_VODKA.

The overall approach has been experimented with a mobile video player case study. Unfortunately comparisons were not possible since no reference implementation was available. Different load extraction methods have been validated by applying them to MPEG-4 encoder, Quake2 3D game, and MP3 decoder case studies previously, where the average and maximum errors between simulated and monitored results have been about 15% and 25%, respectively.

The approach enables early performance evaluation, exhibits light modelling effort, allows fast exploration iteration, and reuses application and platform models. Furthermore, it provides performance results that are accurate enough for system-level exploration.

In the future, the approach will be expanded, so that power consumption or other criteria besides performance can be evaluated. As mentioned earlier, further tool support for automation of some steps of the approach is in progress. Future work will also include a real-scale case study to further validate the approach.

ACKNOWLEDGMENTS

This work is supported by Tekes (Finnish Funding Agency for Technology and Innovation), VTT under the EUREKA/ITEA contract 04006 MARTES, and partially by the European Community under the Grant agreement 215244 MOSART.

REFERENCES

[1] B. Kienhuis, E. Deprettere, K. Vissers, and P. van der Wolf, "Approach for quantitative analysis of application-specific dataflow architectures," in *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP '97)*, pp. 338–349, Zurich, Switzerland, July 1997.

[2] NoTA World Open Architecture Initiative, http://www.notaworld.org/.

[3] Open SystemC Initiative, "IEEE Standard SystemC Language Reference Manual," IEEE Computer Society, 2006. IEEE Std 1666–2005.

[4] P. Lieverse, P. van der Wolf, K. Vissers, and E. Deprettere, "A methodology for architecture exploration of heterogeneous signal processing systems," *The Journal of VLSI Signal Processing*, vol. 29, no. 3, pp. 197–207, 2001.

[5] E. A. de Kock, G. Essink, W. J. M. Smits, et al., "YAPI: application modeling for signal processing systems," in *Proceedings of the 37th Design Automation Conference (DAC '00)*, pp. 402–405, Los Angeles, Calif, USA, June 2000.

[6] A. D. Pimentel, C. Erbas, and S. Polstra, "A systematic approach to exploring embedded system architectures at multiple abstraction levels," *IEEE Transactions on Computers*, vol. 55, no. 2, pp. 99–111, 2006.

[7] T. Wild, A. Herkersdorf, and G.-Y. Lee, "TAPES—trace-based architecture performance evaluation with SystemC," *Design Automation for Embedded Systems*, vol. 10, no. 2-3, pp. 157–179, 2005.

[8] J. M. Paul, D. E. Thomas, and A. S. Cassidy, "High-level modeling and simulation of single-chip programmable heterogeneous multiprocessors," *ACM Transactions on Design Automation of Electronic Systems*, vol. 10, no. 3, pp. 431–461, 2005.

[9] D. Gajski, J. Zhu, R. Dömer, A. Gestlauer, and S. Zhao, *SpecC: Specification Language and Methodology*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 2000.

[10] F. Ghenassia, Ed., *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*, Springer, New York, NY, USA, 2005.

[11] TLM2 Whitepaper, http://www.systemc.org/members/download_files/check_file?agreement=tlm2_whitepaper.

[12] H. Posadas, F. Herrera, P. Sánchez, E. Villar, and F. Blasco, "System-level performance analysis in SystemC," in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE '04)*, vol. 1, pp. 378–383, Paris, France, February 2004.

[13] T. Kangas, P. Kukkala, H. Orsila, et al., "UML-based multiprocessor SoC design framework," *ACM Transactions on Embedded Computing Systems*, vol. 5, no. 2, pp. 281–320, 2006.

[14] Ptolemy II—heterogeneous modelling and design, http://ptolemy.berkeley.edu/ptolemyII.

[15] J. Eker, J. W. Janneck, E. A. Lee, et al., "Taming heterogeneity—the ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–143, 2003.

[16] A. L. Sangiovanni-Vincentelli, "Quo vadis SLD: reasoning about trends and challenges of system-level design," *Proceedings of the IEEE*, vol. 95, no. 3, pp. 467–506, 2007.

[17] UML Profile for MARTE Beta 1, http://www.omg.org/cgi-bin/doc?ptc/2007-08-04.

[18] J. Kreku, Y. Qu, J.-P. Soininen, and K. Tiensyrjä, "Layered UML workload and SystemC platform models for performance simulation," in *Proceedings of the International Forum on Specification and Design Languages (FDL '06)*, pp. 223–228, Darmstadt, Germany, September 2006.

[19] J. Kreku, M. Eteläperä, and J.-P. Soininen, "Exploitation of UML 2.0-based platform service model and SystemC workload simulation in MPEG-4 partitioning," in *Proceedings of International Symposium on System-on-Chip (SoC '05)*, pp. 167–170, Tampere, Finland, November 2005.

[20] J. Kreku, M. Hoppari, K. Tiensyrjä, and P. Andersson, "SystemC workload model generation from UML for performance simulation," in *Proceedings of the International Forum on Specification and Design Languages (FDL '07)*, Barcelona, Spain, September 2007.

[21] P. Andersson and M. Höst, "UML and SystemC-comparison and mapping rules for automatic code generation," in *Proceedings of the International Forum on Specification and Design Languages (FDL '07)*, Barcelona, Spain, September 2007.

[22] P. Andersson, M. Höst, and M. Bengtström, "UML to SystemC transformation in the MARTES project," in *Proceedings of the 32nd Euromicro Conference on Software Engineering and Advanced Applications (SEAA '06)*, Cavat/Dubrovnik, Croatia, August-September 2006.

[23] J. Kreku, T. Kauppi, and J.-P. Soininen, "Evaluation of platform architecture performance using abstract instruction-level workload models," in *Proceedings of International Symposium on System-on-Chip (SoC '04)*, pp. 43–48, Tampere, Finland, November 2004.

[24] J. Kreku, J. Penttilä, J.-P. Soininen, and J. Kangas, "Workload simulation method for evaluation of application feasibility in a mobile multiprocessor platform," in *Proceedings of the Euromicro Symposium on Digital System Design (DSD '04)*, pp. 532–539, Rennes, France, August-September 2004.