Software/Hardware Engineering with the Parallel Object-Oriented Specification Language

B.D. Theelen[1], O. Florescu[1], M.C.W. Geilen[1], J. Huang[1], P.H.A. van der Putten[1], J.P.M. Voeten[1,2]
[1]Eindhoven University of Technology, Department of Electrical Engineering; [2]Embedded Systems Institute
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{b.d.theelen, o.florescu, m.c.w.geilen, j.huang, p.h.a.v.d.putten, j.p.m.voeten}@tue.nl

Abstract

The complexity of designing hardware/software systems motivates research on frameworks that structure and automate the design process. Such design methodologies reduce the risk of expensive design–implementation iterations by assisting designers in constructing models. Software/Hardware Engineering (SHE) is a general-purpose system-level design methodology that supports analysing both functional correctness and performance properties. SHE combines the Unified Modelling Language with the Parallel Object-Oriented Specification Language to specify models. The designer is assisted in constructing models using these languages and applying the analysis techniques with various guidelines and modelling patterns. A key feature of SHE is its foundation on formal methods, which ensures that the obtained analysis results are unambiguous. SHE also includes guidelines and techniques for automatic synthesis of real-time control software. This is again based on formal methods to ensure that properties in a model (including real-time properties) are preserved by the software realisation. Finally, to enable an effective and efficient application of the modelling languages as well as the analysis and synthesis techniques, SHE is accompanied with a set of user-friendly tools. This paper gives an overview of SHE, thereby briefly touching upon the underlying mathematical foundation of the analysis and synthesis techniques as well as upon some open issues that require further research.

1. Introduction

Designing hardware/software systems requires dealing with their increasing complexity within ever-shortening design times. Commonly, the design process involves considering alternative ideas for realising the required functionality. Early in the design process, the choice for a specific alternative may have a deep impact on for example the performance of the final implementation. To assist the designer in taking well-founded design decisions, system-level design methodologies can be applied. These are frameworks for structuring the earliest phases of the design process in order to find a feasible design within minimal time. They focus on constructing models that allow analysing functional and/or non-functional properties before actually realising the system in hardware and software. In addition, design methodologies structure the realisation process by assisting in refining models (possibly partly automated) towards a synthesisable description of hardware and software.

Design methodologies are built around four cornerstones: formalisms (languages), techniques, guidelines and tools. Formalisms for constructing models are called *modelling languages*, whereas formalisms for capturing the properties of interest are known as *property specification languages*. Well-known examples of modelling languages are UML [32] and SystemC [13]. Temporal logics [18] and PSL [8] are examples of property specification languages. *Techniques* are mathematically founded ways to transform or analyse models. Examples include techniques for model refinement, execution/simulation of models, functional verification and performance analysis. To effectively and efficiently apply the formalisms and techniques, design methodologies structure the design process in several steps, where each step is supported by *guidelines*. Such guidelines capture previous experience ('rules of thumb') with how to apply the provided formalisms and techniques. A special kind of guidelines are *modelling patterns*, which refer to templates for modelling typical aspects of systems with the provided modelling language (reusable model components). Finally, and most importantly from the designers viewpoint are the *tools*, which should enable effective and efficient application of the formalisms, techniques and guidelines. To allow the designer to focuss on the difficult task of constructing abstract yet adequate models or on refining models, tools should hide (mathematical) details of the execution, analysis, refinement and synthesis techniques as much as possible.

Choosing modelling languages for a design process is essential for developing a design methodology. Models should be abstract, adequate, succinct, easy to understand and executable. This implies that the used modelling languages must be sufficiently expressive. Next to the ability to express the key characteristics of a system in an abstract way, another aspect of the expressive power is orthogonality of the language primitives [31]. *Formal* modelling languages, for which the semantics of the primitives is defined mathematically [30], have an essential advantage here. Examples of formal modelling languages are LOTOS [4], ESTEREL [5] and CCS [25]. Another key advantage of formal modelling languages is the ability to define rigorous frameworks that allow for unambiguous (and automated) execution, analysis, refinement and synthesis of models. Informal languages like SystemC rely for example on compilers to define their semantics, which implies that using different (versions of a) compiler(s) may lead to different functional and non-functional properties [34].

Formal modelling languages have also disadvantages. A practical one is that designers of hardware/software systems are often uncomfortable with mathematical notation. Hence, modelling languages (and property specification languages) should have an intuitive syntax such that models are as easy to understand as back-of-the enveloppe sketches and plain texts. A more serious challenge for modelling languages in general is that designing a system usually starts with an informal and incomplete specification of the desired functionality and requirements. Design methodologies should therefore include a structured way to bridge the gap between this initial informal specification and the construction of formal models.

This paper presents an overview of the general-purpose system-level design methodology called Software/Hardware Engineering (SHE), which was originally introduced in [31]. SHE structures the design process to assist a designer in constructing and/or refining models using the informal modelling language UML and the formal Parallel Object-Oriented Specification Language (POOSL) [31, 12, 3]. Based on the devel-

oped models, SHE supports both functional verification and performance analysis with both exhaustive and simulation-based techniques. Furthermore, sufficiently refined models of real-time control software can be automatically synthesised. This is based on a correct-by-construction approach, which ensures that properties in the model (including real-time properties) are preserved by the software realisation. The SHE methodology is accompanied with three tools to ease the construction, validation, simulation, analysis and synthesis of models. The key novelty of SHE lies in the combination of several analysis and synthesis techniques into one system-level design methodology based on formal methods. The contribution of this paper is a concise overview of all aspects of SHE. Next to discussing the most relevant concepts forming the basis of SHE, we indicate the main features of the rigorous frameworks for execution, analysis and synthesis of models. We also identify open issues that require further research.

The remainder of this paper is organised as follows. Section 2 presents how SHE structures the design process in several steps. It also indicates in which steps the languages UML and POOSL and the various techniques are applied. Section 3 elaborates on the modelling languages UML and POOSL to indicate how SHE bridges the gap between informal specifications and formal models. Section 4 is devoted to the analysis of models covering both functional verification and performance analysis. The synthesis of real-time control software with SHE is discussed in Section 5 and Section 6 briefly reviews the tools for SHE. Section 7 refers to some industrial case studies to illustrate the applicability of SHE in different application domains. Section 8 summarises the conclusions.

2. Design Flow

The SHE methodology distinguishes a modelling and analysis phase from a realisation phase. The modelling and analysis phase covers the construction and refinement of models for the purpose of analysis and design space exploration, whereas the realisation phase involves converting a model into a synthesisable description of hardware and/or software.

Figure 1 depicts the framework for exploring design alternatives with SHE [36], which can be applied at different (possibly successive) levels of abstraction. This modelling and analysis flow consists of three phases; formulation, formalisation and evaluation. Designing a system or refining a model starts with brainstorm-sessions on concepts for realising the requested functionality. In addition, a set of (refined) requirements that are to be satisfied by the final implementation is identified. The *formulation* stage is concerned with structuring this creative process and with documenting the results. SHE uses Object-Oriented Analysis (OOA) and Object-Oriented Design (OOD) techniques in combination with several guidelines (including various modelling patterns) for applying them in the context of hardware/software systems [31]. They assist in formulating the concepts for realising the required functionality in an informal model expressed in UML. In addition, the requirements are formulated as questions or required properties. The latter process actually reflects determining the design issues that must be addressed. One may use plain texts or the UML profile for Schedulability, Performance and Time [29] to annotate the UML diagrams for the design. The result of the formulation stage is a structured (but informal and non-executable) specification of the design concepts and requirements using schematic diagrams and plain texts, which together compose the deliverable at milestone A.

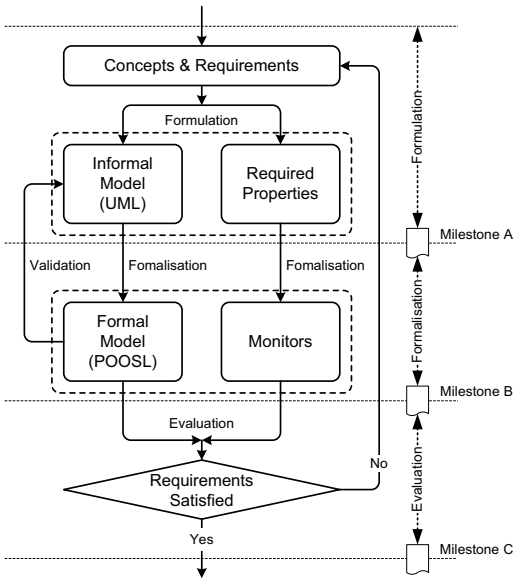


Figure 1. Modelling and analysis flow

The *formalisation* stage is concerned with developing formal representations of the informal model and required properties to enable rigorous automated analysis in the evaluation stage. The informal UML model is formalised into a formal executable model expressed in POOSL. Next to providing several guidelines (including many modelling patterns), SHE facilitates this formalisation process by using a specialised UML profile in the formulation stage. Similarly, the required properties are specified by monitors. For exhaustive evaluation, such monitors are constructed separately from the formal model. Real-time temporal logics like MTL [18] or MITL [1] are used to specify qualitative and quantitative real-time correctness properties (metrics that are expressible as a reachability property). In addition, temporal reward functions [41] are used to specify delay-type performance metrics (such as long-run time-averages and variances). In case of run-time verification or simulation-based analysis, SHE uses POOSL for extending the formal model of a system with monitors. Several guidelines are provided to ensure that such 'reflexive' monitors are non-intrusive. The mathematical link between for example performance monitors expressed in POOSL and the formalism of temporal reward functions from [41] is established in [36]. The term validation in Figure 1 refers to the process of checking that the formal model and monitors properly reflect the informal model and required properties. Hence, validation covers the difficult check whether the formal model represents the yet unknown implementation in a way that is adequate for evaluating whether the devised design concepts lead to satisfying the requirements. The formalisation stage is completed with the deliverable at milestone B, documenting the validated formal model and monitors.

In the *evaluation* stage, the properties expressed by the monitors are actually checked against the formal model. Verification of qualitative and quantitative functional correctness properties is based on the model checking techniques in [12], whereas the evaluation of long-run performance metrics relies on Markov chain based analysis techniques as documented in [36]. The mathematical link between the semantics of POOSL models and these analysis techniques ensures unambiguous results, both in case of exhaustive and simulation-based anal-

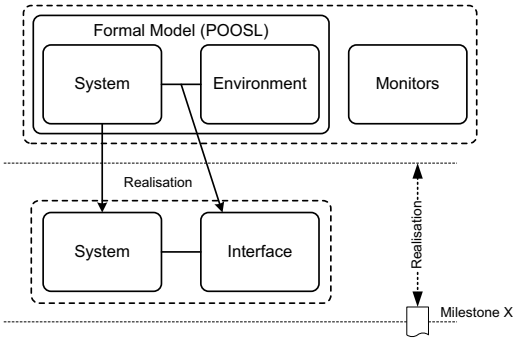


Figure 2. Synthesis flow

ysis. They also allow for highly automation of the analysis in tools. Based on the analysis results, a designer can conclude whether the requirements are satisfied. If so, the deliverable at milestone C documents the analysis results as a starting point for more detailed design. Otherwise, deliverable C contains the reasons for not satisfying the requirements, which will give rise to reconsidering the previously devised design concepts (and possibly also the requirements).

The flow in Figure 1 is rather general and needs further detailing to match specific application domains by means of additional guidelines and/or modelling patterns. For example, multi-media systems are commonly designed according to the Y-chart approach of [17], which separates a model of the multi-media programs from a model of the multi-processor platform on which these are executed. Such a separation is however undesirable for modelling for example telecommunication networks. SHE considers approaches like the Y-chart as more detailed guidelines for constructing models. They can easily be combined with SHE as shown in [42] for the Y-chart. We also developed tools that automatically generate POOSL models from propriety XML formats specifying a system for a certain application domain based on modelling patterns (see for example [37]). An aspect of modelling and analysing design alternatives that received little attention in SHE concerns efficient approaches for searching the design space. Our research focussed on guidelines that assist in constructing models of individual design alternatives. The University of Limerick proposed a design-space exploration approach based on evolutionary algorithms for designing a network processor using SHE in [27]. The generality of the modelling and analysis flow leaves room for exploiting specialised approaches like the mentioned ones within the context of SHE.

After developing a POOSL model that is refined to a level that describes (control) software in full functional detail, SHE proposes realising the final implementation according to the flow in Figure 2. SHE advocates the use of formal methods based correct-by-construction approaches to support the realisation process. Such approaches imply an improved reliability of the implementation but also reduce the design time compared to traditional approaches by minimising verification and test efforts. Currently, SHE supports only synthesis of real-time control software on single processor platforms using the techniques in [15]. These techniques preserve MTL and MITL properties satisfied by the formal model during the realisation of such software under certain conditions as discussed in Section 5.2. Synthesis of distributed software and software with data-intensive computations is subject of current research. Hardware synthesis is a topic of future research.

The basic idea behind synthesising control software is to execute a POOSL model in its real-time environment while interacting with this environment. POOSL models are closed in the sense that they typically include a representation of the system's environment. SHE provides guidelines to discard this environment model and the monitors that may have been added previously for the purpose of simulation-based analysis. More importantly, SHE includes guidelines to develop the interfaces between the realisation of the actual system and its environment [16], which includes several modelling patterns representing device drivers for software systems. The deliverable at milestone X concerns the realised system accompanied with documentation of all interfaces to its environment.

3. Modelling Languages of SHE

The development of SHE started almost two decades ago with an investigation on how the object-oriented way of thinking could be exploited for hardware/software co-design. This research resulted in developing a consistent set of informal diagram types and the accompanying formal modelling language POOSL as well as guidelines on how to apply them for specifying the static structure and dynamic behaviour of hardware/software systems. With the introduction of UML 2.0 [2], we realised however that several UML diagram types coincide with the original diagram types of SHE. Therefore, we introduced a UML profile for SHE in [36] that (slightly) adapts the UML notation to enable using it in the context of SHE as summarised in Section 3.1. POOSL was originally defined in [31] as an object-oriented extension of CCS that allowed expressing more complex behaviour than just synchronous communication between asynchronous concurrent processes. Meanwhile, POOSL has been extended with time in [12] and probabilities in [3] to become a very expressive formal modelling language that is presented in Section 3.2.

3.1. UML Profile for SHE

An important aspect of the UML profile is that SHE distinguishes three types of classes originating from the idea of modelling active and passive components separately as also suggested in [29]. *Data objects* model the passive components of a system representing information that is generated, communicated, processed and consumed by active components. Such active components may take the initiative to perform certain behaviour, and they may do so concurrently. SHE uses *processes* to represent the elementary active components of a system, while *clusters* model groups of active components in a hierarchical fashion. Data objects, processes and clusters are specified in data classes, process classes and cluster classes respectively, for which the UML profile requires modifications to the traditional class symbol.

Figure 3 illustrates the data, process and cluster class symbols. Data objects are similar to objects in traditional object-oriented programming languages such as Smalltalk and Java. The data class symbol is therefore equal to the traditional one, where the stereotypes match with naming conventions in SHE. The attributes of data objects are called instance variables and their (sequential) behaviour is specified with (data) methods. Process and cluster classes require a class symbol that deviates from the traditional one. The attributes of process classes include instance variables and instantiation parameters, where the latter allow to parameterise the behaviour of a process when it is instantiated. Processes perform their behaviour asynchronously concurrent, where the

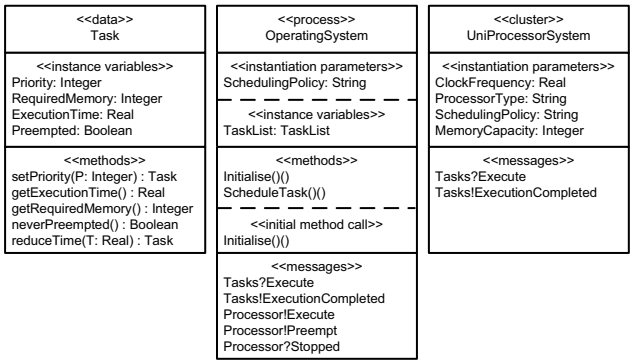


Figure 3. Data, process and cluster classes

behaviour of each process class is described with sequentially or asynchronous concurrently executed (process) methods. The start behaviour of a process is specified by means of the initial method call. The bottom compartment extends the traditional class symbol to allow expressing that in hardware/software systems, exchanging messages between processes is not always directly related to invoking certain behaviour (for example, a receiver might not be ready to participate in the communication or the message implies termination of the receiver's current behaviour). Notice that for example SystemC-based approaches do not make such distinction, which complicates succinctly modelling for example back-pressure. Clusters, which originate from aggregation relationships between active components, do not extend the behaviour of their constituents. The behaviour of clusters is defined by composite structure diagrams, while messages communicated through the cluster boundaries are listed in the bottom compartment of the cluster class symbol. Figure 4 exemplifies how a cluster class is defined with a structure diagram.

Next to the modified class symbols, the UML profile includes stereotypes to extend or specialise the semantics of UML symbols for diagrams that express the dynamic behaviour of hardware/software systems. Many of these stereotypes match with primitives of POOSL, which smoothes the formalisation of informal models. Some of the stereotypes are exemplified in the activity diagrams of Figure 5. SHE distinguishes two types of decisions, which are indicated by the `<<selection>>` and `<<choice>>` stereotypes. A `<<choice>>` refers to a deterministic decision between mutual-exclusive options, which matches with the traditional **`if`**-statement. Conversely, a `<<selection>>` denotes a non-deterministic decision between alternative activities of a process. Guarding conditions are optional in this case, where these guards do not necessarily have to be mutually exclusive. The actual decision for a `<<selection>>` is postponed until one or more of the alternative activities can be performed. Two examples of activities that may be blocked are those preceded by guards and the reception and sending of messages. The latter two refer to the concept of synchronous communication between processes and/or clusters. Other stereotypes specialise the communication symbol used in communication and sequence diagrams for various other types of communication (such as buffered asynchronous communication, interruptible communication, etc) as defined in [31]. Several modelling patterns of SHE assist in refining these communication types to use synchronous communication only (as supported by POOSL). SHE also allows the use of `<<atomic>>` activity regions when specifying the behaviour of processes, for which the constituent actions are considered as one indivisible activity that is free from interference of concurrent activities. Such `<<atomic>>` activity regions may only include activities that refer to manipulating data objects, possibly preceded by one communication action as shown in Figure 5.

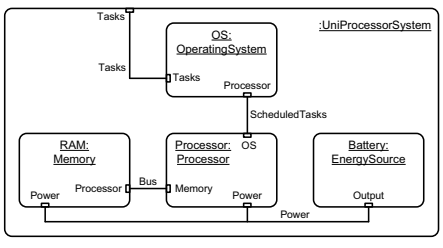


Figure 4. Defining the behaviour of clusters

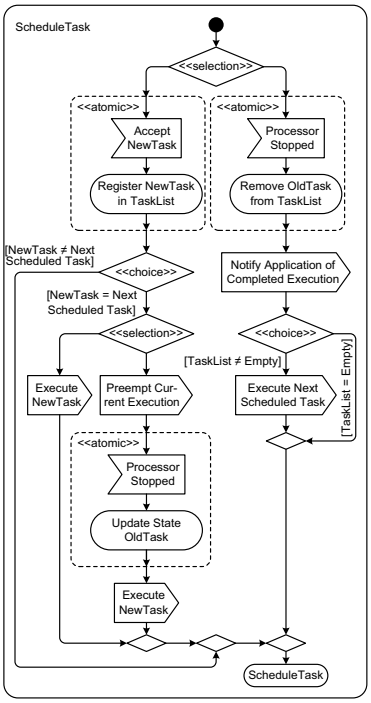
As indicated before, SHE considers two levels of concurrency. The first level reflects that active components in hardware/software systems perform their behaviour concurrently, whereas the second level originates from the potential of having concurrent behaviour within active components. SHE allows using the fork (join) construct in activity diagrams to reflect dynamic creation (and termination) of concurrent activities within processes. Another feature of SHE shown in Figure 5 is that it distinguishes two types of exceptions. An `<<abort>>` exception refers to terminating the current activity without finishing it after handling the exception. In case of an `<<interrupt>>` exception, the normal behaviour is resumed after the exceptional behaviour has been completed.

Besides the features of the UML profile illustrated in Figure 5, several other stereotypes have been defined. An example is `<<probability>>`, which allows annotating transitions in activity and statechart diagrams with a probability on making the transition. These include the transitions from a `<<choice>>` decision, which enables expressing probabilistic choices. These stochastic features of SHE are useful for modelling unpredictable behaviour in an abstract way towards probabilistic verification or performance analysis.

While the part of the UML profile reflecting the static structure of hardware/software systems has reached maturity, the part for specifying the dynamic behaviour is still under development. Further research is needed to formalise the SHE concepts for dynamic behaviour more clearly in the UML profile. Related to this is the need for more research on (automatically) transforming the UML specification of the dynamic behaviour into a formal POOSL model. We also intend to develop a specialised tool for drawing diagrams according to the UML profile since such a tool is not yet available.

3.2. POOSL

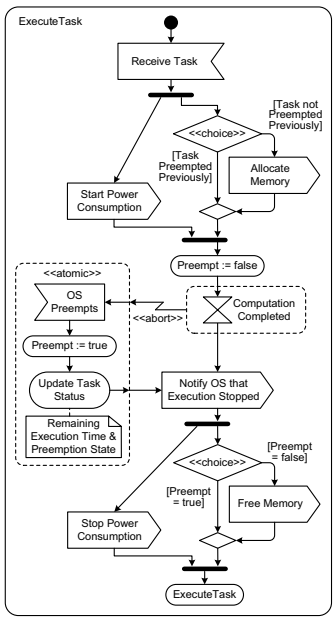
SHE uses POOSL for constructing formal models of hardware/software systems. POOSL follows the key concepts of SHE such as the distinction between data, process and cluster classes. POOSL has several primitives to define these classes, including one for specifying specialisation/generalisation relationships between data classes and between process classes by means of inheritance. For the definition of cluster classes as expressed in composite structure diagrams like the one in Figure 4, POOSL uses a CCS-like syntax. The (atomic) behaviour of data objects is specified with primitives that match those in traditional sequential object-oriented programming languages. Most interesting are the primitives for specifying the behaviour of processes listed in Table 1. Figure 5 exempli-




```
ScheduleTask() () |NewTask, OldTask: Task|

sel
   Tasks?Execute(NewTask)
      {TaskList register(NewTask)};
   if TaskList nextTask == NewTask then
      sel
         Processor!Execute(NewTask)
      or
         Processor!Preempt;
         Processor?Stopped(OldTask)
            {TaskList updateState(OldTask)};
         Processor!Execute(NewTask)
      les
   fi
or
   Processor?Stopped(OldTask)
      {TaskList remove(OldTask)};
   Tasks!ExecutionCompleted(OldTask);
   if TaskList notEmpty then
      Processor!Execute(TaskList nextTask)
   fi
les;
ScheduleTask() ().
```

Preemptable scheduling of tasks by an `OperatingSystem`




```
ExecuteTask() () |StartTime: Real,
Preempt: Boolean, Task: Task|

OS?Execute(Task){StartTime := currentTime};
par
   Power!StartConsumption(ProcessorPower)
and
   if Task neverPreempted then
      Memory!Allocate(Task getRequiredMemory)
   fi
rap;
Preempt := false;
abort
   delay Task getExecutionTime
with
   OS?Preempt{Preempt := true;
      Task reduceTime(currentTime - StartTime)};
OS!Stopped(Task);
par
   Power!StopConsumption(ProcessorPower)
and
   if Preempt == false then
      Memory!Free(Task getRequiredMemory)
   fi
rap;
ExecuteTask() ().
```

Preemptive execution of a task by a `Processor`

Figure 5. Specifying the behaviour of processes

fies the use of some of these statements for formalising the behaviour expressed in activity diagrams. Each process method consists of a combination of statements, where any S in Table 1 can refer to any of the listed statements. The meaning of any such a combination is unambiguously defined by a combination of the semantical rules for the constituent statements.

The first statement in Table 1 enables to name activities or activity regions. They denote methods that can be called from other activities of the same process. Such methods can be parameterised. The exhibited behaviour and outputs $v_1, \ldots, v_j$ may depend on the values of input parameters $E_1, \ldots, E_i$ when calling the method. Methods without output parameters can be called in a tail-recursive fashion as shown in Figure 5. A statement may reflect an expression denoting the manipulation of data objects. Optional braces can be used to group expressions into one indivisible atomic action. Activities can be combined sequentially and concurrently. Figure 5 illustrates for example how the **`par`**-statement is used to formalise the fork constructs in the right-hand activity diagram. The primitives for synchronous rendez-vous communication use an extended CCS-like syntax. The send statement

Statement S	Description
$m\,(E_1, \ldots, E_i)\,(v_1, \ldots, v_j)$	Method Call
E	Data Expression
$S_1; \ldots; S_n$	Sequential Composition
`par` S_1 **`and`** $\ldots$ **`and`** S_n **`rap`**	Parallel Composition
$E_p!m\,(E_1, \ldots, E_i)\,\{E_a\}$	Message Send
$E_p?m\,(v_1, \ldots, v_i \mid E_c)\,\{E_a\}$	(Conditional) Message Reception
$[E]\,S$	Guarded Execution
`sel` S_1 **`or`** $\ldots$ **`or`** S_n **`les`**	Non-deterministic Selection
`if` E_c **`then`** S_1 **`else`** S_2 **`fi`**	Deterministic Choice
`while` E_c **`do`** S **`od`**	Loop
`abort` S_1 **`with`** S_2	Abort
`interrupt` S_1 **`with`** S_2	Interrupt
`skip`	Empty Behaviour
`delay` E_t	Time Synchronisation

Table 1. POOSL statements for processes

denotes sending a message m through a port E_p with parameters $E_1, \ldots, E_i$. The optional expression E_a between braces is evaluated atomically after the message is sent. The complementary reception statement specifies receiving a message m from a port E_p, where the parameters of a matching send statement are bound to the variables $v_1, \ldots, v_i$. However, such synchronisation can only occur if the optional reception condition E_c (which can depend on the data objects to receive) is satisfied. The receive statement may also be followed by an atomically evaluated expression. Figure 5 shows how atomicity braces formalise `<<atomic>>` activity regions.

Statement $[E]\,S$ specifies that performing S is postponed until E becomes valid. The **`sel`**-statement denotes the non-deterministic decision as expressed in activity diagrams by the `<<selection>>` stereotype. Next in the list are the traditional deterministic choice and loop constructs. The abort and interrupt statements formalise the two different types of exceptions that SHE distinguishes, which is illustrated for the `<<abort>>` exception in Figure 5. Finally, **`skip`** denotes empty behaviour and **`delay`** specifies that time advances. SHE requires to specify any duration explicitly (in both the informal and formal models) since all actions are timeless. We furthermore remark that the time expression E_t determines the amount of (integer or real) time that should be advanced, which may be in accordance with a probability distribution.

The formal semantics of POOSL is given in [3]. Data objects are based on a probabilistic extension of traditional object-oriented programming languages and supports encapsulation, single inheritance (with method overriding) and polymorphism, which is all formalised by a denotational semantics. The structural operational semantics of processes and clusters is based on a probabilistic real-time extension of CCS. The probabilistic interpretation of the statements in Table 1 originates from using data objects in the expressions E, which denote random variables for discrete or continuous dis-

tributions [36]. SHE provides library classes for common distributions. The semantics of processes supports encapsulation (of data objects) and single inheritance (with method overriding), while clusters encapsulate processes and other clusters.

The formal semantics of processes and clusters is based on the two-phase execution model of [28]. It describes that the configuration (state) of a model can either change by asynchronously performing actions or by synchronous consumption of time. Time can only advance if no actions can be performed (action urgency). The semantics of any POOSL model (i.e., any process, cluster or complete model) defines a *timed probabilistic labelled transition system* of the form $(\mathcal{C}, C_s, \mathcal{A}, \{\xrightarrow{a} \subseteq \mathcal{C} \times \mathcal{D}(\mathcal{C}) \mid a \in \mathcal{A}\}, \mathcal{T}, \{\xrightarrow{t} \subseteq \mathcal{C} \times \mathcal{C} \mid t \in \mathcal{T}^+\})$. It consists of a set $\mathcal{C}$ of configurations, an initial configuration $C_s \in \mathcal{C}$, a set $\mathcal{A}$ of actions (of various types), a time domain $\mathcal{T}$ (which can be discrete or dense) and two sets of labelled transition relations. The elements of the latter two sets are defined based on the semantical rules for the statements in Table 1. The set $\{\xrightarrow{a} \subseteq \mathcal{C} \times \mathcal{D}(\mathcal{C}) \mid a \in \mathcal{A}\}$ denotes the action transitions for a model, where $\mathcal{D}(\mathcal{C})$ is the set of probability distribution functions over $\mathcal{C}$. When residing in configuration C, relation $C \xrightarrow{a} \boldsymbol{\pi}$ with $\boldsymbol{\pi} \in \mathcal{D}(\mathcal{C})$ holds if action $a \in A$ can be performed, after which the model transits to configuration $C' \in \mathcal{C}$ with probability $\boldsymbol{\pi}(C')$. In case several actions can be performed, the choice of which action is actually executed is made non-deterministically. Subsequently, a probabilistic choice determines the resulting configuration. The set $\{\xrightarrow{t} \subseteq \mathcal{C} \times \mathcal{C} \mid t \in \mathcal{T}^+\}$ denotes the time transitions for a model. When residing in configuration C, relation $C \xrightarrow{t} C'$ holds if the time can increase with a positive amount $t \in \mathcal{T}^+$ of time units. A model only has time transitions for the maximal amount of time that it is willing to wait before continuing with performing action transitions, where it is implicitly understood that it is also willing to wait for a shorter time [12]. As a result, there is at most one t and C' for each C such that $C \xrightarrow{t} C'$ holds. Time transitions are therefore considered to be deterministic (taken with probability 1).

The transition system defined by a POOSL model resembles a variant of a probabilistic timed automaton as defined in [33] and generally has a countable number of configurations and a countable number of transitions from a configuration. This transition system forms the basis of the rigorous frameworks for execution, formal verification, performance analysis and software synthesis as discussed in Sections 4 and 5.

4. Model Analysis

4.1. Execution

Execution facilitates a smooth formalisation of informal models by allowing validation of the dynamic behaviour. It also forms the basis for simulation-based analysis. Executing a POOSL model boils down to exploring a path through the transition system defined by its semantics but without explicitly constructing this transition system. Such execution is based on the framework depicted in Figure 6, which is very different from traditional approaches relying on a host operating system or those used in the case of for example SystemC [26]. Instead of maintaining a flattened representation of the model (the transition system for the complete model), the execution framework relies on representations of the transition systems defined by all processes individually. Each configuration $(\mathcal{B}, \mathcal{I}) \in \mathcal{C}$ of the transition system defined by a POOSL model includes a specification of the behaviour $\mathcal{B}$ that is to

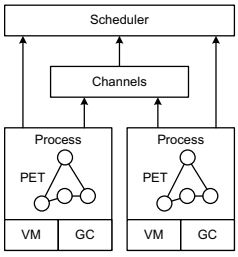


Figure 6. Executing POOSL models

be executed in the context of information $\mathcal{I}$. Behaviour $\mathcal{B}$ indicates the statements describing the (future) behaviour of an executing model, while information $\mathcal{I}$ captures the values that are assigned to variables. The execution framework manages the configuration of a process by representing the (future) behaviour $\mathcal{B}$ of the process in a so-called *process execution tree* (PET) [12, 3]. A PET is basically a data structure representing the composition of statements that specify the (future) behaviour of a process. The PET is dynamically updated in accordance with the semantical rules for the statements when making an execution step (a transition in the transition system). The context information $\mathcal{I}$ is updated by a virtual machine (VM) that executes any data expressions within a process, while a garbage collector (GC) automatically deletes data objects that have become unreachable or obsolete during such execution. Depending on the current configuration (that is, the next statements to execute), a process issues requests for performing action and time transitions. The requests for performing actions other than communication actions as well as for performing time transitions are directly submitted to a central scheduler. Requests for send and reception actions are submitted to a representation of the involved channel, which combines complementary requests into single communication requests that are forwarded to the central scheduler. The central scheduler always handles requests for performing actions before granting any requests for time transitions (action urgency). In case requests for several action transitions can be granted, the scheduler resolves such non-determinism in a probabilistic way using a uniform distribution over the set of possible action transitions. The scheduler grants time transitions in accordance with the concept of maximal progress, which ensures that time increases with the minimal amount to potentially enable action transitions again.

The framework of Figure 6 has proven to be very efficient, where run times scale for example linearly with the number of concurrent activities [3]. This efficiency results from using several locality-exploiting techniques that minimise the number of requests sent to the central scheduler and for minimising the complexity of the PETs [3]. Notice that the framework of Figure 6 matches with the two forms of synchronisation between processes: at the level of the channels for the synchronous communication and at the level of the central scheduler for progress in time. These forms of synchronisation complicate distributed execution of POOSL models. Future research includes an investigation on how the synchronisation overhead for distributed execution can be minimised.

4.2. Formal Verification

SHE uses the framework depicted in Figure 7 to support formal verification of correctness properties expressed in real-time temporal logics like MTL and MITL. These logics extend the operators of the Linear Temporal Logic (LTL) [24]

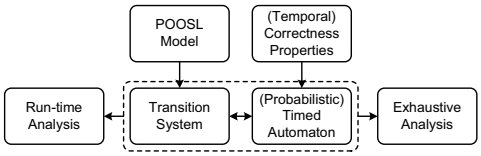


Figure 7. Framework for formal verification

with an interval during which a particular property should hold. An example of an MITL formula is $\Box(p \Rightarrow \Diamond_{\leq 5} q)$, which states that every event p is followed within 5 time units by an event q. Evaluating such properties with SHE follows the approach of linear-time temporal logic verification, where typically the logical negation of a required property is converted into a (probabilistic timed) automaton by means of a so-called tableau construction [11]. Subsequently, automata theoretic techniques are used to test whether the property is satisfied. As indicated in Section 4.1, an execution of a POOSL model is a path through the timed probabilistic labelled transition system defined by its semantics. Each of these possible paths constitutes a single linear execution trace that can be accepted or rejected by the automaton. In case the automaton accepts such a trace, then the model can exhibit behaviour with an undesirable property and hence the required property does not hold for all execution traces.

Formal verification can be accomplished by exhaustive or by run-time analysis. Exhaustive analysis involves constructing an abstract representation of the POOSL model in verification tools like SPIN [14], UPPAAL [22] or PRISM [20]. Depending on the used verification tool, one may include or abstract from timing and/or probabilistic behaviour to yield simpler representations. On the other hand, to keep the automaton implied by the property of interest sufficiently small, SHE considers the subset $\text{MITL}_{\leq}$ of MITL [12], where time intervals are of the shape $[0, d)$ or $[0, d]$ with d an arbitrary positive real number or $d = \infty$. Although this approach implies exact results based on taking all possible execution traces into account, the abstractions that are made need to be reviewed very carefully. False positives or negatives may occur in the verification results. Currently, SHE only includes a few guidelines for constructing a SPIN, UPPAAL or PRISM specification from a POOSL model [12]. More research is required to enable developing tools that (largely) automate this process.

As an alternative to exhaustive analysis, SHE enables run-time verification of the POOSL model by means of simulation (using the framework in Figure 6) or by executing the realisation as discussed in Section 5. During the execution, typical behaviour is exhibited that can automatically be checked for violation of the required property. Since only a single execution trace is considered, such run-time verification may not cover all possible behaviours and hence, rare situations causing a violation may easily escape from being detected. For such cases, the exhaustive approach is more effective. Conversely, not every violation of a required property can be detected from a single execution trace. In particular, liveness properties can never be refuted by a finite execution. To support run-time verification, the traditional automata theoretic techniques have been adapted to cope with the incrementally growing finite prefix of infinite execution traces that is constructed during the execution of a POOSL model [11]. To avoid backtracking in an ongoing execution, the automaton that is used as a monitor must be deterministic. [12] describes how this approach works for a specific class of properties that can be expressed as a reachability or safety property. The basic idea for answering whether a property is satisfied for a finite prefix of an infinite execution trace is to recognise good and bad prefixes. These are defined as finite traces for which all extensions to infinite traces satisfy a required property and for which there exists an extension to an infinite trace which satisfies an undesired property, respectively [19].

4.3. Performance Analysis

SHE supports analysing worst/best-case and average-case performance metrics based on the framework in Figure 8. The timed probabilistic labelled transition system defined by a POOSL model can be interpreted as a Markov decision process [41, 36], which expresses both the non-deterministic and probabilistic choices in the model. In case non-determinism is not resolved, a performance metric gives rise to a collection of results since different policies for doing so may imply different results for that metric. Only after resolving non-determinism, performance metrics are guaranteed to be given by a single result. Although exhaustive approaches exist that can determine the bounds for a metric when non-determinism is not resolved, their applicability is limited by the state-space explosion problem. Conversely, developing simulation-based analysis techniques that are capable of taking non-deterministic options into account is still a challenge for future research. Tools for executing models expressed in other modelling languages that do not exclude non-determinism like SystemC and SDL [21] often resolve it implicitly, without knowing the exact policy for doing so. Section 4.1 explained that the execution of POOSL models involves resolving non-determinism in a uniform probabilistic way, which is an arbitrary yet fair approach. To ensure comparable results for both exhaustive and simulation-based performance analysis, the framework in Figure 8 follows the same approach.

After resolving the non-determinism in a POOSL model, the original transition system defined by its semantics as described in Section 3.2 transforms into a timed probabilistic labelled transition system of the form $(\mathcal{C}, C_s, \mathcal{A}, \{\stackrel{a,p}{\Longrightarrow} \subseteq \mathcal{C} \times \mathcal{C} \mid a \in \mathcal{A}, p \in [0,1]\}, \mathcal{T}, \{\stackrel{t}{\Longrightarrow} \subseteq \mathcal{C} \times \mathcal{C} \mid t \in \mathcal{T}^+\})$, where $\mathcal{C}$, C_s, $\mathcal{A}$ and $\mathcal{T}$ are as in the original transition system. Sets $\{\stackrel{a,p}{\Longrightarrow} \subseteq \mathcal{C} \times \mathcal{C} \mid a \in \mathcal{A}, p \in [0,1]\}$ and $\{\stackrel{t}{\Longrightarrow} \subseteq \mathcal{C} \times \mathcal{C} \mid t \in \mathcal{T}^+\}$ indicate the action transitions and time transitions respectively *after* resolving non-determinism. When residing in configuration C, relation $C \stackrel{a,p}{\Longrightarrow} C'$ holds if configuration C' can be entered with probability p after performing action a. We remark that this implies that there exists at most one action transition $\stackrel{a,p}{\Longrightarrow}$ leading from C to C' [41, 36]. The relations $\stackrel{a,p}{\Longrightarrow}$ are straightforwardly determined from the relations $\stackrel{a}{\longrightarrow}$ of the original transition system and the policy for resolving non-determinism. Conversely, when residing in con-

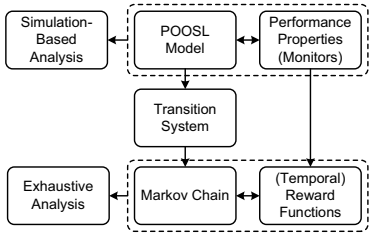


Figure 8. Performance analysis framework

figuration C, relation $C \stackrel{t}{\Longrightarrow} C'$ holds if configuration C' can be entered after advancing the time with t units. This is the case if $C \stackrel{t}{\longrightarrow} C'$ and no actions can be performed from C.

By shifting the action and time labels for the transitions of the new transition system into its configurations, a discrete-time Markov chain is obtained [41, 36]. From the states of this Markov chain, one can deduct information about the *actual* occurrence of actions and progress of time (from the transition system, it can only be concluded that an action or time transition *can* occur). Hence, action and time information can be encoded as reward values, which is necessary for evaluating performance metrics depending on such information. The state space $\mathcal{S}$ of the Markov chain is given as a subset of the set $\mathcal{C} \times (\mathcal{A} \cup \mathcal{T}^+ \cup \{-\})$, where $-$ denotes that no action or time transition could be performed (which holds only for the initial state). In general, $\mathcal{C} \times (\mathcal{A} \cup \mathcal{T}^+ \cup \{-\})$ is uncountable due to the cardinality of time domain $\mathcal{T}$. However, since $\mathcal{C}$ is countable and both $\{\stackrel{a,p}{\Longrightarrow} \subseteq \mathcal{C} \times \mathcal{C} \mid a \in \mathcal{A}, p \in [0,1]\}$ and $\{\stackrel{t}{\Longrightarrow} \subseteq \mathcal{C} \times \mathcal{C} \mid t \in \mathcal{T}^+\}$ are countable, $\mathcal{S}$ is countable as well. In case $|\mathcal{C}|$ is finite, then $|\mathcal{S}|$ equals the number of action and time transitions with different label/target-configuration combinations in the transition system plus one (initial state).

Performance metrics are specified with (temporal) reward functions, which retrieve information included in the states of a Markov chain [39]. Next to the occurred action or time transition, such reward functions may denote variables in the context $\mathcal{I}$ of a state. Analysis of worst/best case metrics boils down to identifying the maximum/minimum value for reward functions. Average-case metrics often concern combinations of several reward functions for which each possible value must be taken into account according to their occurrence probabilities. Next to evaluating traditional probabilistic and expected reachability properties and long-run sample averages, SHE allows analysing long-run averages that can only be expressed as an accumulation of reward values over a sequence of states. Using temporal rewards [41], one can express for example the difference in time between two subsequent visits of a state in which a certain event occurred (possibly with intermediate visits to other states). Such expressive power is required for denoting delay-type metrics like throughput and jitter. To express reflexive monitors for simulation-based analysis of delay-type metrics, POOSL offers the keyword **`currentTime`**, which can be seen as a special temporal reward function returning the accumulated time of all time transitions performed during an execution [36].

Similarly as for formal verification, SHE supports both exhaustive and simulation-based performance analysis. Figure 8 suggests that exhaustive analysis requires constructing the Markov chain that is implicitly defined by a POOSL model. This is only feasible in case the state space is finite. Subsequently determining worst/best-case results is straightforward, while computing average-case metrics relies on the ergodic theorem for Markov chains [41, 36, 39]. Although such exhaustive analysis gives exact performance results, the conditions for which POOSL models imply finite-state ergodic Markov chains have only been established for certain types of models such as the one in [37]. Next to following the traditional way out of just assuming ergodicity, a tool for constructing the Markov chain is currently also unavailable.

To circumvent the state-space explosion that is inherent to concurrent systems, SHE proposes to use simulation-based analysis as an alternative to exhaustive analysis. The basic idea is to execute a POOSL model (according to the framework of Figure 6), which implicitly corresponds to generating a path through the underlying Markov chain. Monitors that extend the model reflect the (temporal) reward functions expressing the performance metrics under evaluation [36]. Notice that this is similar to the traditional approach in industrial practice and that the accuracy of the estimation results compared to the true performance is not guaranteed in this case. SHE uses automatic termination of a simulation when estimation results have become sufficiently accurate, whereas the traditional approach is to just run a simulation for an arbitrary amount of time while having no concrete information about the accuracy of results. Unfortunately, it is not possible to draw conclusions about how close estimation results obtained for worst/best-case metrics are to the true worst/best-case. However, one can use confidence intervals derived based on the central limit theorem for Markov chains [7] to determine a statistical bound on the accuracy of average-case metrics [36]. SHE provides a library of data classes representing performance monitors that allow accuracy analysis for the most common types of performance metrics in hardware/software systems based on an algebra of confidence intervals [36].

5. Software Synthesis

5.1. Model Refinement

Models intended for analysing correctness and/or performance properties are usually not synthesisable into a realisation of the design because they abstract from too many implementation details [40]. Refinements like replacing nondeterministic and probabilistic abstractions with deterministic counterparts may be needed to bridge abstraction levels. Next to organising the refinement process [16], SHE includes guidelines for applying structural refinements such as the distribution of concurrent activities within one process over different processes and behaviour-preserving transformations that refine the composite structure of a system [31]. After including sufficient implementation details, SHE allows to (automatically) derive C++ code for real-time control software.

For synthesising software systems, SHE involves removing any monitors that might have been added to a POOSL model for the purpose of simulation-based analysis (see Section 2). In addition, the interactions between the model of the software system and the model of its environment must be refined into interfaces with other software or into device drives that interact with the hardware platform. Processes that represent images of such environmental systems are refined to use data classes for which the behaviour is specified with so-called primitive methods [3, 15]. These primitive methods are implemented as C++ functions that can be invoked from the C++ code that is derived from the POOSL model of the software system. Compiling these two parts together with a C++ implementation of the execution framework in Figure 4.1 gives the executable that realises the software system.

5.2. Predictable Code Generation

Realising a model of real-time software must preserve the properties of interest. Typically, these properties relate to the behaviour observed at its interfaces with the environment. A key problem of traditional software synthesis approaches is the mismatch between the timing semantics of the modelling language and that of the implementation language [15]. This emerges for example when the models uses timeless actions that do take time in reality. As a consequence, the interface

```
1  while action or time step possible do
2     while observable action possible do
3        perform an observable action;
4     if unobservable action possible then
5        perform an unobservable action;
6     else if time can advance then
7        advance time with minimal amount;
8        synchronise with physical time;
```

Figure 9. Scheduler for software synthesis

behaviour may not match with that exhibited in the model. For example, the TAU2 tool [35] realises SDL models by executing timing expressions based on asynchronous timer mechanisms provided by the host operating system. All expressions referring to some amount of time in the model will refer to *at least* that amount of time in the realisation, which can clearly lead to incorrect timing behaviour [15]. SHE overcomes this problem by limiting the time deviation between a model and its realisation such that properties of the realised software system can be predicted from the properties of the model.

Recall that executing a POOSL model resembles traversing a timed probabilistic labelled transition system by selecting and performing an enabled action in successively visited configurations. While it takes 0 time units in the model to perform such actions, they do take a certain amount of time in the realisation. We can denote the sequence of actions and the time instant at which they are performed (as also given by **`currentTime`**) for an execution as the trace $\sigma = (a_0, t_0), (a_1, t_1), \ldots, (a_n, t_n)$, where $a_n \in \mathcal{A}$ for all $n \geq 0$. If $a_0, a_1, \ldots, a_n$ are those actions in $\mathcal{A}$ that are observable at the interface with the environment, then the interface behaviour is said to satisfy an MTL or MITL property p in case p holds for all σ (see also Section 4.2). Synthesising software with SHE results in a trace $\hat{\sigma} = (a_0, \hat{t}_0), (a_1, \hat{t}_1), \ldots, (a_n, \hat{t}_n)$, where each $\hat{t}_n$ denotes the physical time instant at which the corresponding action a_n is actually observed. The observable timing deviation between the model and its realisation is defined as $\Delta(\sigma, \hat{\sigma}) = \sup |t_n - \hat{t}_n|$ for all $n \geq 0$. [15] proves that if σ satisfies p, then $\hat{\sigma}$ satisfies a $2\Delta(\sigma, \hat{\sigma})$ weakening of p. This means that the lower and upper bounds of the intervals in the MTL or MITL formula specifying p are relaxed with $2\Delta(\sigma, \hat{\sigma})$. [43] suggests to identify the most relaxed realisation of the model that still satisfies the original property. Instead, SHE proposes to quantitatively predict the properties of the realisation from that of the model based on $\Delta(\sigma, \hat{\sigma})$. The development of a technique to find a tight bound for the observable timing deviation is ongoing research. In [10], we propose to minimise $\Delta(\sigma, \hat{\sigma})$ by giving priority to performing observable actions over unobservable ones. Figure 9 illustrates how the scheduler for software synthesis implements this approach. The algorithm ensures that the realised software has exactly the same observable functional behaviour at the interface as the model, where the ordering of actions at different time instants remains preserved. Line 7 realises the concept of action urgency, while line 8 specifies advancing the virtual model time until it equals the physical time. Together, they ensure that the timing behaviour at the interface is as close as possible to that of an execution trace for the model.

6. Tool Support

SHE is accompanied with three major (academic) tools (of which two are freely downloadable from [44]): SHESim, Rotalumis and Rotalumis-RT. SHESim is a graphical tool for incremental construction of POOSL models in the formalisation stage [12]. As illustrated with the snapshot in Figure 10, it allows specifying data and process classes with an intuitive graphical user interface that matches the aspects in their class symbols of Figure 3. Cluster classes are defined by drawing structure diagrams like the one in Figure 4. Next to constructing POOSL models, SHESim supports simulating them based on the framework of Figure 6. During such simulation, SHESim allows to inspect data objects, processes and clusters for validation purposes. As opposed to debugging facilities in other tools, the inspection capabilities of SHESim provide feedback to the designer at the model level. Sequence and communication diagrams can for example be generated for validating the interactions between processes and clusters. Inspecting processes results in highlighting the POOSL statements that are to be executed, which allows validated their behaviour against for example activity or statechart diagrams. We also plan to extend SHESim with facilities for entering/producing UML specifications according to the profile for SHE as used in the formulation and formalisation stages. In contrast to SHESim, Rotalumis is a command-line tool for fast simulation-based analysis of large POOSL models in the evaluation stage [3]. Where SHESim executes a POOSL model in an interpretive way, Rotalumis compiles it into an intermediate byte code that is executed on a virtual machine implemented in C++. Compared to SHESim, this improves the simulation speed by a factor of about 100 (depending on the ratio between data expressions and process statements). Finally, Rotalumis-RT refers to the C++ code that represents the execution framework for realising real-time control software [3]. Rotalumis-RT already implements synchronisation with physical time, but the implementation of distinguishing observable and non-observable actions as in Figure 9 is still ongoing work. Where SHESim and Rotalumis have reached a level of maturity that allows their application for industrial systems, Rotalumis-RT also needs some user-interface related extensions to smoothen its applicability in industrial practice.

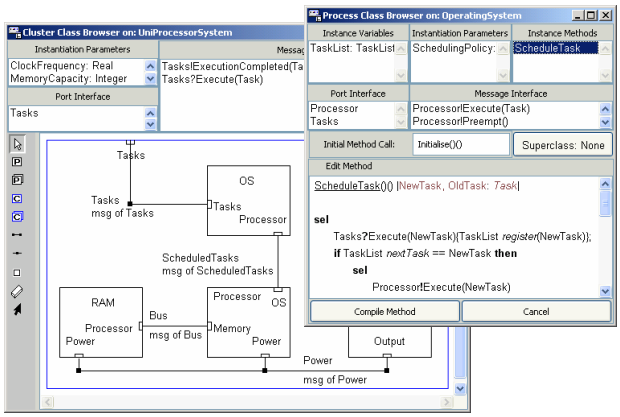


Figure 10. Snapshot of SHESim

7. Industrial Case Studies

SHE has been applied in many academic and industrial case studies. In the area of telecommunication systems, for example, a protection switching protocol for optical networks was verified in cooperation with Lucent Technologies to reveal a fatal error [23]. In cooperation with Alcatel Bell, the performance of an Internet router comprising over 10^6 concurrent activities was evaluated for different settings of parameters like the capacity of the included buffers [36]. This

case study not only confirmed the suitability of the proposed routing algorithm, it furthermore showed that the tools offered by SHE are very competitive to the commercial tool OPNET [6] (in terms of simulation speed). A similar observation arose in comparison with SystemC-based tools after performing a case study in cooperation with IBM Research Laboratory Zürich on identifying performance bottlenecks in a network processor [38]. The University of Limerick performed a design-space exploration case study for a network processor by Intel [27]. In the area of multi-media systems, [42] describes for example a case study performed in cooperation with Philips Research Laboratories Eindhoven on evaluating which mappings of a Picture-in-Picture application on a multi-processor system satisfied the performance requirements. [9] reports on a case study performed together with Chess Information Technology on evaluating the performance of alternative hardware architectures for executing an in-car navigation application by Siemens VDO. The latter two case studies not only provided valuable feedback to the designers of the considered systems, they also showed the feasibility of specialising the modelling and design flow of Figure 1 for a certain application domain. In the area of high-tech systems, a case study in cooperation with ASML identified a crucial performance bottleneck in newly developed distributed control software for their wafer stepper system and also lead to a solution that satisfied the requirements. In cooperation with Océ Technologies, a new hardware design of a low-level control system in the next generation of printers has been investigated to confirm its feasibility [9]. As a final example, [16] describes a case study performed together with the University of Twente on real-time control software synthesis for part of a molding machine by Stork. The diversity of the mentioned case studies shows the general-purpose character of the SHE methodology and its applicability in an industrial context.

8. Conclusions

Methodologies for designing industrial systems should provide an adequate and coherent combination of expressive formalisms, techniques, guidelines and tools. Although using formal languages to found rigorous frameworks for refining, executing, analysing and synthesising models has essential advantages, designing a system usually starts with informal discussions on how to realise the desired functionality such that the requirements are expected to be satisfied. A design methodology should therefore assist in bridging the gap between the informal specification of such conceptual ideas and their corresponding representation in formal models. SHE is an example of a general-purpose system-level design methodology that integrates formal methods and industrial practice based on UML and POOSL. SHE provides guidelines for constructing models with these formalisms and offers rigorous frameworks for refining, executing, analysing and synthesising such models based on the formal semantics of POOSL. The analysis capabilities cover both formal verification of functional correctness and performance analysis according to exhaustive and simulation-based techniques. Synthesis of real-time control software is supported based on a correct-by-construction approach. Although SHE has proven to be suitable for designing industrial systems in various application domains, more research is needed to complete SHE.

References

[1] R. Alur. *Techniques for Automatic Verification of Real-Time Systems*. PhD Thesis. Stanford University, 1991.

[2] M. Björkander and C. Kobryn. Architecting Systems with UML 2.0. *IEEE Software*, vol 20 (4), pp 57–61, 2003.

[3] L.J. van Bokhoven. *Constructive Tool Design for Formal Languages; From Semantics to Executing Models*. PhD Thesis. Eindhoven University of Technology, 2002.

[4] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, vol 14 (1), pp 25–59, 1987.

[5] F. Boussinot and R. de Simone. The Esterel Language. *Proceedings of the IEEE*, vol 79, pp 1293–1304, 1991.

[6] X. Chang. Network Simulations with OPNET. *Proceedings of WSC*, pp 307–314, 1999.

[7] M.A. Cranes and A.J. Lemoine. An Introduction to the Regenerative Method for Simulation Analysis. *Lecture Notes in Control and Information Sciences*, vol 4, 1977.

[8] C. Eisner and D. Fisman. *A Practical Introduction to PSL*. Springer, 2006.

[9] O. Florescu, J.P.M. Voeten, M. Verhoef and H. Corporaal. Reusing Real-Time Systems Design Experience through Modeling Patterns. *Proceedings of FDL*, pp 19–22, 2006.

[10] O. Florescu, J. Huang, J.P.M. Voeten and H. Corporaal. Strengthening Property Preservation in Concurrent Real-Time Systems. *Proceedings of RTCSA*, pp 106–109, 2006.

[11] M.C.W. Geilen. On the Construction of Monitors for Temporal Logic Properties. *Electronic Notes in Theoretical Computer Science*, vol 55 (2), 2001.

[12] M.C.W. Geilen. *Formal Techniques for Verification of Complex Real-Time Systems*. PhD Thesis. Eindhoven University of Technology, 2002.

[13] T. Grötker, S. Liao, G. Martin and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[14] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[15] J. Huang. *Predictability in Real-Time System Design*. PhD Thesis. Eindhoven University of Technology, 2005.

[16] J. Huang, J.P.M. Voeten, M. Groothuis, J. Broenink and H. Corporaal. A Model-Driven Design Approach for Mechatronic Systems. *Proceedings of ACSD*, 2007.

[17] K. Keutzer, S. Malik, R. Newton, J. Rabaey and A. Sangiovanni-Vincentelli. System Level Design: Orthogonalization of Concerns and Platform-Based Design. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, vol 19 (12), pp 1523–1543, 2000.

[18] R. Koymans. Specifying Real-Time Properties with Metric Temporal Logic. *Real-Time Systems*, vol 2 (4), pp 255–299, 1990.

[19] O. Kupferman and M.Y. Vardi. Model Checking of Safety Properties. *Proceedings of CAV*, pp 172–183, 1999.

[20] M. Kwiatkowska, G. Norman, R. Segala, and J. Sproston. Automatic Verification of Real-Time Systems with Discrete Probability Distributions. *Theoretical Computer Science*, vol 282, pp 101-150, 2002.

[21] ITU-T. *Recommendation Z.100: Specification and Description Language (SDL)*. November 1999.

[22] K.G. Larsen, P. Pettersson and W. Yi. UPPAAL in a Nutshell. *Journal on Software Tools for Technology Transfer*, vol 1 (1-2), pp 134–152, 1997.

[23] G. Lopez. *Modélisation, Simulation et Vérification d'un Protocole de Telecommunication*. MSc Thesis. Eindhoven University of Technology, 1998.

[24] Z. Manna and A. Pnuli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer Verlag, 1992.

[25] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[26] W. Müller, J. Ruf, D. Hoffman, J. Gerlach, T. Kropf and W. Rosenstiehl. The Simulation Semantics of SystemC. *Proceedings of DATE*, pp 64–70, 2001.

[27] L. Noonan and C. Flanagan. Utilising Evolutionary Approaches and Object Oriented Techniques for Design Space Exploration. *Proceedings of DSD*, pp 346–352, 2006.

[28] X. Nicollin and J. Sifakis. An Overview and Synthesis on Timed Process Algebras. *Proceedings of CAV*, pp 376–398, 1991.

[29] OMG. *UML Profile for Schedulability, Performance and Time Specification*. OMG Adopted Specification ptc/02-03-02. Object Management Group, 2002.

[30] B. Partee, A. ter Meulen and R. Wall. *Mathematical Methods in Linguistic*. Kluwer Academic Publishers, 1990.

[31] P.H.A. van der Putten and J.P.M. Voeten. *Specification of Reactive Hardware/Software Systems*. PhD Thesis. Eindhoven University of Technology, 1997.

[32] J. Rumbaugh, I. Jacobson and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[33] R. Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems*. PhD Thesis. MIT, 1995.

[34] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1997.

[35] Telelogic TAU Generation 2. See `http://taug2.com`. 2002.

[36] B.D. Theelen. *Performance Modelling for System-Level Design*. PhD Thesis. Eindhoven University of Technology, 2004.

[37] B.D. Theelen, M.C.W. Geilen, T. Basten, J.P.M. Voeten, S.V. Gheorghita and S. Stuijk. A Scenario-Aware Data Flow Model for Combined Long-Run Average and Worst-Case Performance Analysis. *Proceedings of MEMOCODE*, pp 185-194, 2006.

[38] B.D. Theelen, J.P.M. Voeten and R.D.J. Kramer. Performance Modelling of a Network Processor using POOSL. *Journal of Computer Networks*, vol 41 (5), pp 667–684, 2003.

[39] H.C. Tijms. *Stochastic Models; An Algorithmic Approach*. John Wiley & Sons, 1994.

[40] M. Verhappen, J.P.M. Voeten and P.H.A. van der Putten. Traversing the Fundamental System-Level Design Gap using Modelling Patterns. *Proceedings of FDL*, 2003.

[41] J.P.M. Voeten. Performance Evaluation with Temporal Rewards. *Performance Evaluation*, vol 50 (2/3), pp 189–218, 2002.

[42] F.N. van Wijk, J.P.M. Voeten and A.J.W.M. ten Berg. An Abstract Modeling Approach Towards System-Level Design-Space Exploration. *System Specification and Design Languages*, chapter 22, pp. 267–282, 2003.

[43] M.D. Wulf, L. Doyen and J.F. Raskin. Systematic Implementation of Real-Time Models. LNCS 3582, pp 139–156, 2005.

[44] SHE/POOSL website, see `http://www.es.ele.tue.nl/poosl`